UNIX Network Programming Volume 2

Second Edition

Interprocess Communications

by W. Richard Stevens

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

ISBN 0-13-081081-9
90000
9 780130 810816

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 `forks`. 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.

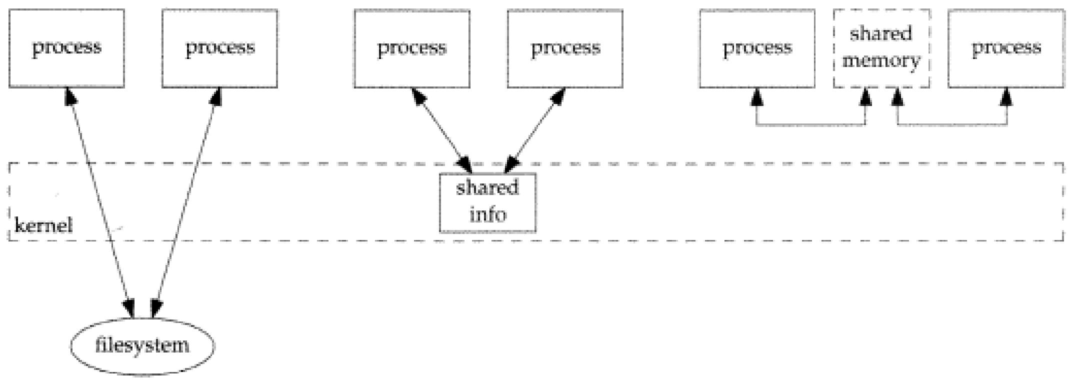

Figure 1.1 Three ways to share information between Unix processes.

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.

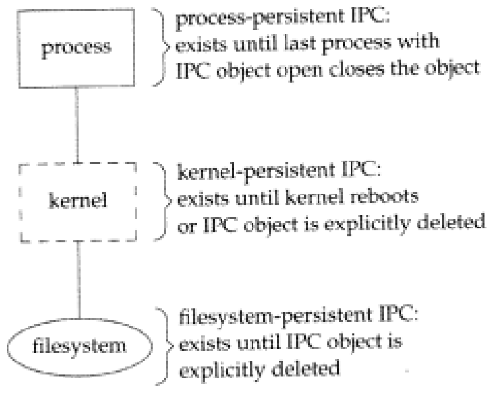
process
process-persistent IPC: exists until last process with IPC object open closes the object
kernel
kernel-persistent IPC: exists until kernel reboots or IPC object is explicitly deleted
filesystem
filesystem-persistent IPC: exists until IPC object is explicitly deleted

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 filesystem-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

```
void
Pthread_mutex_lock(pthread_mutex_t *mptr)
{
    int     n;

    if ( (n = pthread_mutex_lock(mptr)) == 0)
        return;
    errno = n;
    err_sys("pthread_mutex_lock error");
}
```

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 site.

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 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

```c
#include    "unpipc.h"

char *
px_ipc_name(const char *name)
{
    char    *dir, *dst, *slash;

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

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

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

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.

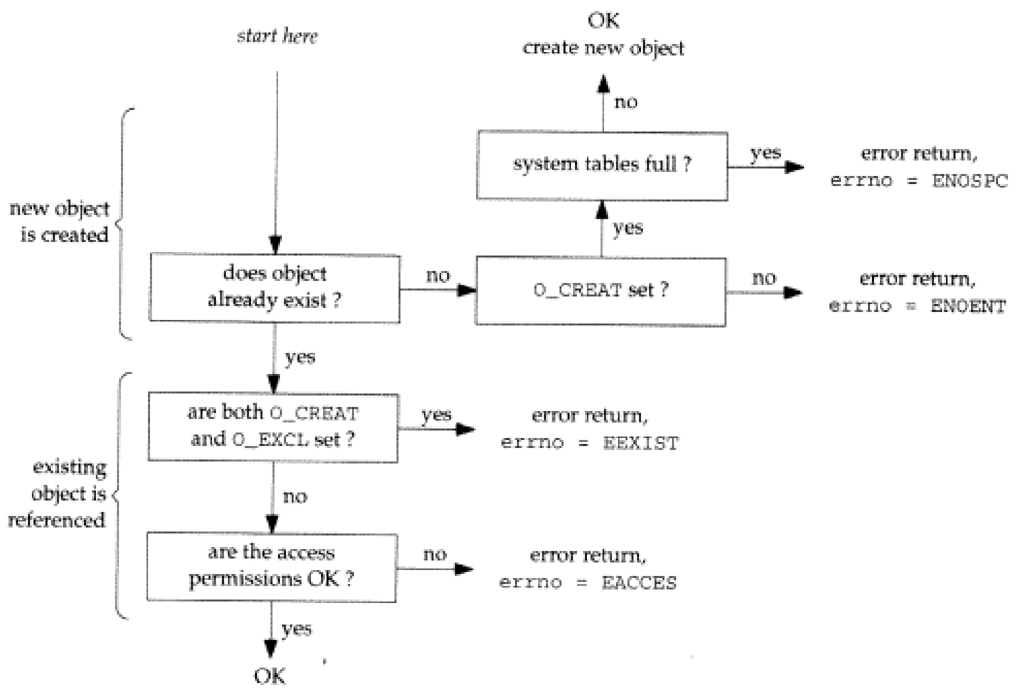

Figure 2.5 Logic for opening or creating an IPC object.

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* (which must not be 0).

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

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    struct stat stat;

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

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

    exit(0);
}
```

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 `get`*XXX* 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 `get`*XXX* 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.

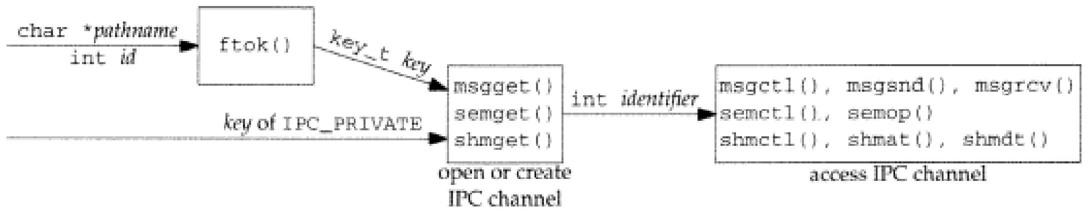

Figure 3.3 Generating IPC identifiers from IPC keys.

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.

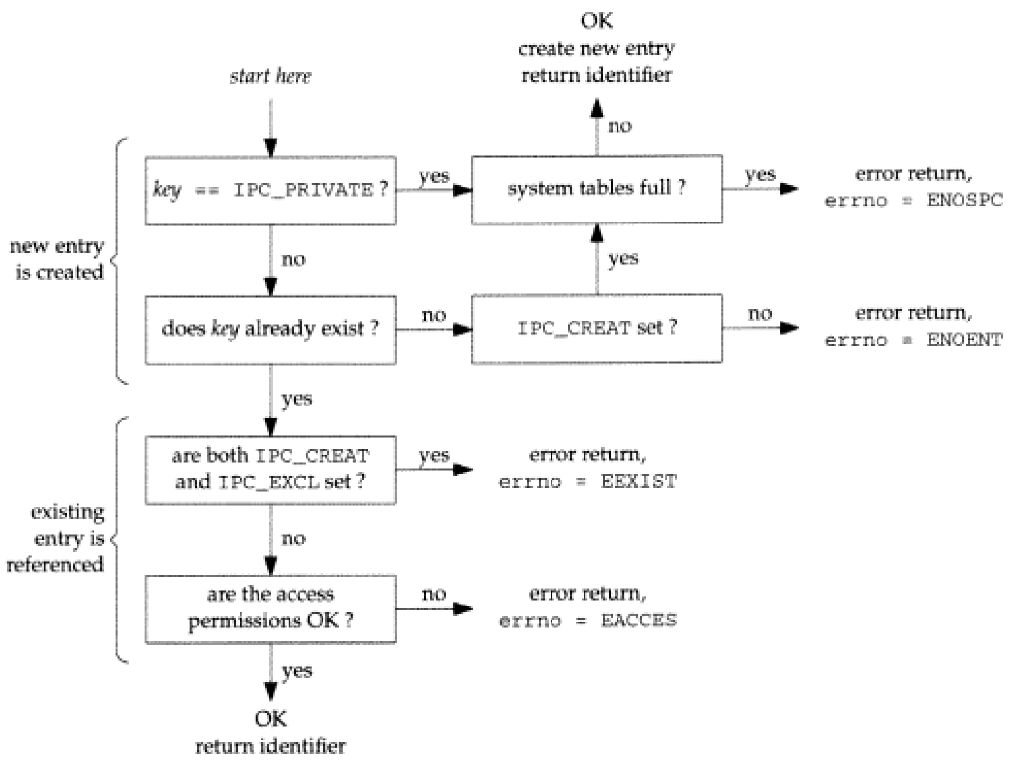
start here
key == IPC_PRIVATE ?
yes
system tables full ?
yes
error return, errno = ENOSPC
no
OK
create new entry
return identifier
no
does key already exist ?
no
IPC_CREAT set ?
no
error return, errno = ENOENT
yes
new entry is created
yes
are both IPC_CREAT and IPC_EXCL set ?
yes
error return, errno = EEXIST
no
are the access permissions OK ?
no
error return, errno = EACCES
yes
OK
return identifier
existing entry is referenced

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 `get`*XXX* 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 `ctl`*XXX* function for the IPC mechanism with a command of `IPC_SET`. The three `ctl`*XXX* 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 `get`*XXX* 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 `get`*XXX* 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 `get`*XXX* 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 `get`*XXX* 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 `get`*XXX* 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

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     i, msqid;

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

        Msgctl(msqid, IPC_RMID, NULL);
    }
    exit(0);
}
```

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.

```
set 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 *XXX*ctl 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 get*XXX* 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 `get`*XXX* 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

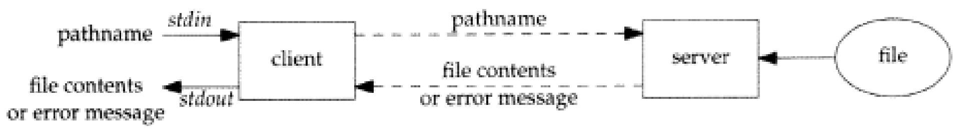

Figure 4.1 Client–server example.

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 `forks` to create a copy of itself, as shown in Figure 4.3.

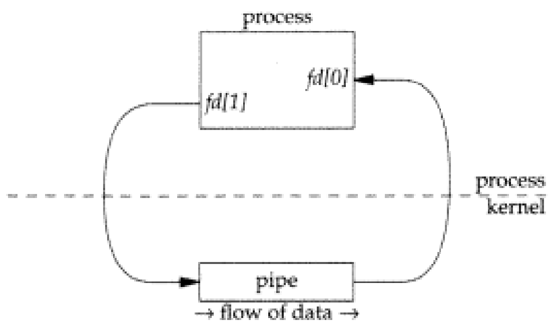

Figure 4.2 A pipe in a single process.

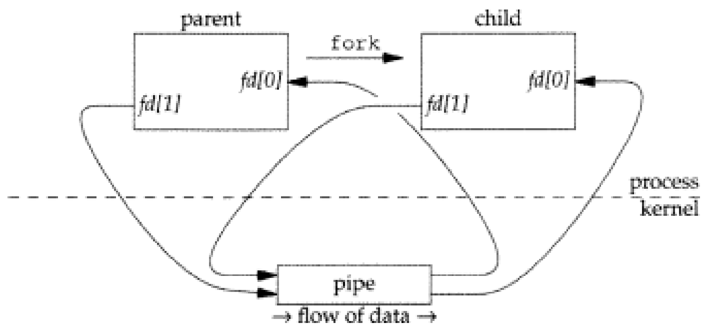

Figure 4.3 Pipe in a single process, immediately after `fork`.

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.

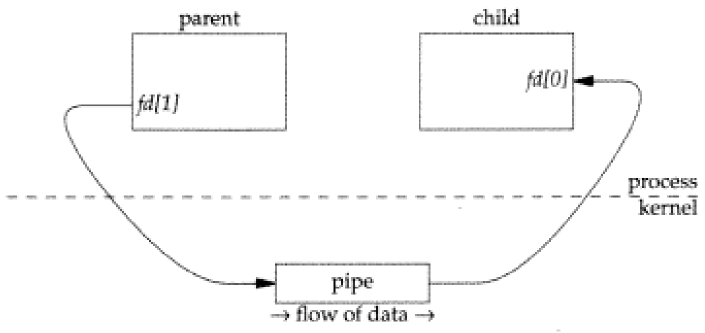

Figure 4.4 Pipe between two processes.

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.

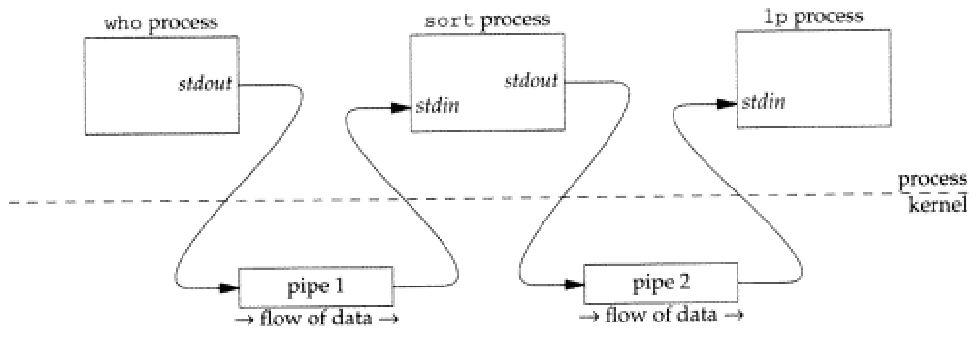

Figure 4.5 Pipes between three processes in a shell pipeline.

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.

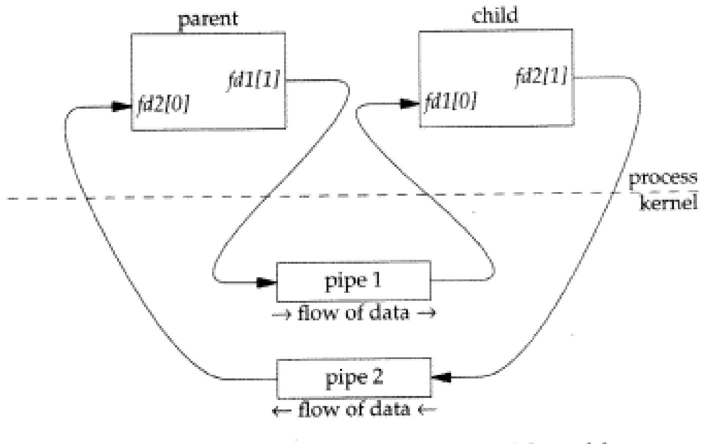

Figure 4.6 Two pipes to provide a bidirectional flow of data.

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.

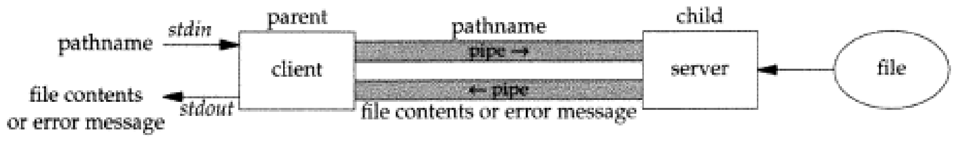

Figure 4.7 Implementation of Figure 4.1 using two pipes.

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

```
#include    "unpipc.h"

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

int
main(int argc, char **argv)
{
    int     pipe1[2], pipe2[2];
    pid_t   childpid;

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

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

        server(pipe1[0], pipe2[1]);
        exit(0);
    }
        /* parent */
    Close(pipe1[0]);
    Close(pipe2[1]);

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

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

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--;                  /* ignore 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.

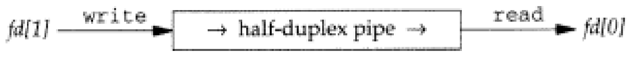

Figure 4.11 Half-duplex pipe.

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.

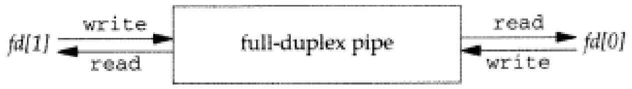

Figure 4.12 One possible (incorrect) implementation of a full-duplex pipe.

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.

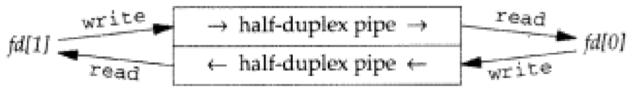

Figure 4.13 Actual implementation of a full-duplex pipe.

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

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     fd[2], n;
    char    c;
    pid_t   childpid;

    Pipe(fd);                   /* assumes a full-duplex pipe (e.g., SVR4) */
    if ( (childpid = Fork()) == 0) {      /* child */
        sleep(3);
        if ( (n = Read(fd[0], &c, 1)) != 1)
            err_quit("child: read returned %d", n);
        printf("child read %c\n", c);
        Write(fd[0], "c", 1);
        exit(0);
    }
        /* parent */
    Write(fd[1], "p", 1);
    if ( (n = Read(fd[1], &c, 1)) != 1)
        err_quit("parent: read returned %d", n);
    printf("parent read %c\n", c);
    exit(0);
}
```

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         buff[n - 1] = '\0';     /* 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

```
#include    "unpipc.h"

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

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.

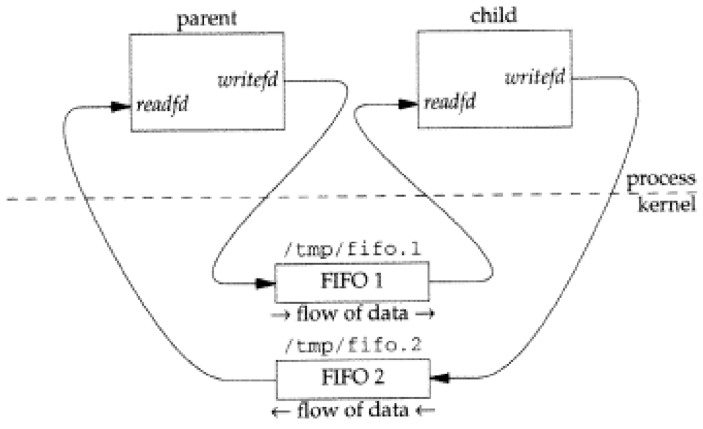

Figure 4.17 Client–server example using two FIFOs.

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

```c
#include    "fifo.h"

void    server(int, int);

int
main(int argc, char **argv)
{
    int     readfd, writefd;

        /* create two FIFOs; OK if they already exist */
    if ((mkfifo(FIFO1, FILE_MODE) < 0) && (errno != EEXIST))
        err_sys("can't create %s", FIFO1);
    if ((mkfifo(FIFO2, FILE_MODE) < 0) && (errno != EEXIST)) {
        unlink(FIFO1);
        err_sys("can't create %s", FIFO2);
    }
    readfd = Open(FIFO1, O_RDONLY, 0);
    writefd = Open(FIFO2, O_WRONLY, 0);

    server(readfd, writefd);
    exit(0);
}
```

pipe/server_main.c

Figure 4.18 Stand-alone server `main` function.

pipe/fifo.h.c

```c
#include    "unpipc.h"

#define FIFO1   "/tmp/fifo.1"
#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

```c
#include    "fifo.h"

void    client(int, int);

int
main(int argc, char **argv)
{
    int     readfd, writefd;

    writefd = Open(FIFO1, O_WRONLY, 0);
    readfd = Open(FIFO2, O_RDONLY, 0);

    client(readfd, writefd);

    Close(readfd);
    Close(writefd);

    Unlink(FIFO1);
    Unlink(FIFO2);
    exit(0);
}
```

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 `write`s 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

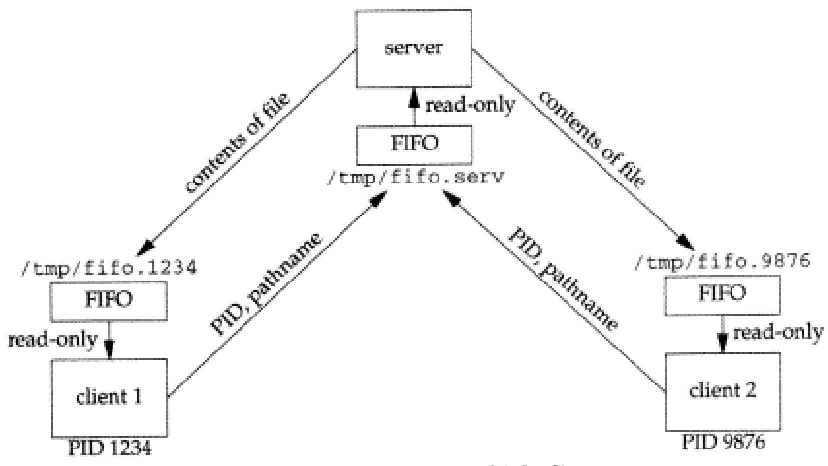

Figure 4.22 One server, multiple clients.

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

```c
#include    "fifo.h"

void    server(int, int);

int
main(int argc, char **argv)
{
    int     readfifo, writefifo, dummyfd, fd;
    char   *ptr, buff[MAXLINE + 1], fifoname[MAXLINE];
    pid_t   pid;
    ssize_t n;

        /* create server's well-known FIFO; OK if already exists */
    if ((mkfifo(SERV_FIFO, FILE_MODE) < 0) && (errno != EEXIST))
        err_sys("can't create %s", SERV_FIFO);

        /* open server's well-known FIFO for reading and writing */
    readfifo = Open(SERV_FIFO, O_RDONLY, 0);
    dummyfd = Open(SERV_FIFO, O_WRONLY, 0);     /* never used */

    while ( (n = Readline(readfifo, buff, MAXLINE)) > 0) {
        if (buff[n - 1] == '\n')
            n--;                /* delete newline from readline() */
        buff[n] = '\0';         /* null terminate pathname */

        if ( (ptr = strchr(buff, ' ')) == NULL) {
            err_msg("bogus request: %s", buff);
            continue;
        }
        *ptr++ = 0;             /* null terminate PID, ptr = pathname */
        pid = atol(buff);
        snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);
        if ( (writefifo = open(fifoname, O_WRONLY, 0)) < 0) {
            err_msg("cannot open: %s", fifoname);
            continue;
        }
        if ( (fd = open(ptr, O_RDONLY)) < 0) {
                /* error: must tell client */
            snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",
                     strerror(errno));
            n = strlen(ptr);
            Write(writefifo, ptr, n);
            Close(writefifo);

        } else {
                /* open succeeded: copy file to FIFO */
            while ( (n = Read(fd, buff, MAXLINE)) > 0)
                Write(writefifo, buff, n);
            Close(fd);
            Close(writefifo);
        }
    }
    exit(0);
}
```

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

```
#include    "fifo.h"

int
main(int argc, char **argv)
{
    int     readfifo, writefifo;
    size_t  len;
    ssize_t n;
    char    *ptr, fifoname[MAXLINE], buff[MAXLINE];
    pid_t   pid;

        /* create FIFO with our PID as part of name */
    pid = getpid();
    snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);
    if ((mkfifo(fifoname, FILE_MODE) < 0) && (errno != EEXIST))
        err_sys("can't create %s", fifoname);

        /* start buffer with pid and a blank */
    snprintf(buff, sizeof(buff), "%ld ", (long) pid);
    len = strlen(buff);
    ptr = buff + len;

        /* read pathname */
    Fgets(ptr, MAXLINE - len, stdin);
    len = strlen(buff);         /* fgets() guarantees null byte at end */

        /* open FIFO to server and write PID and pathname to FIFO */
    writefifo = Open(SERV_FIFO, O_WRONLY, 0);
    Write(writefifo, buff, len);

        /* now open our FIFO; blocks until server opens for writing */
    readfifo = Open(fifoname, O_RDONLY, 0);

        /* read from IPC, write to standard output */
    while ( (n = Read(readfifo, buff, MAXLINE)) > 0)
        Write(STDOUT_FILENO, buff, n);

    Close(readfifo);
    Unlink(fifoname);
    exit(0);
}
```

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 server 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

```
#include    "unpipc.h"

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

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

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

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

ssize_t mesg_send(int, struct mymesg *);
void    Mesg_send(int, struct mymesg *);
ssize_t mesg_recv(int, struct mymesg *);
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.

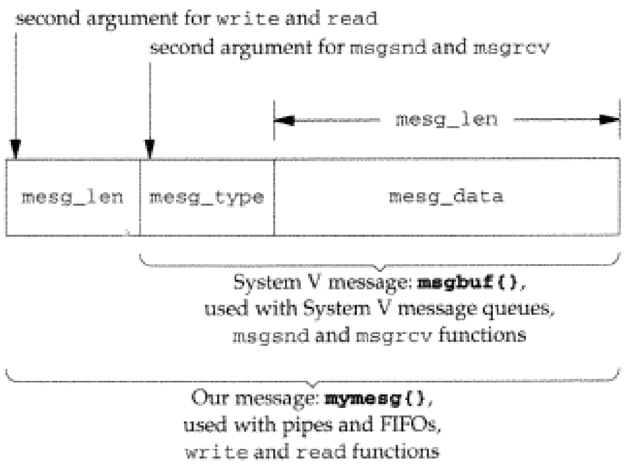
second argument for write and read
second argument for msgsnd and msgrcv
mesg_len
mesg_len
mesg_type
mesg_data
System V message: msgbuf{}, used with System V message queues, msgsnd and msgrcv functions
Our message: mymesg{}, used with pipes and FIFOs, write and read functions

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

```
#include    "mesg.h"

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

pipemesg/mesg_send.c

Figure 4.27 mesg_send function.

pipemesg/mesg_recv.c

```
#include    "mesg.h"

ssize_t
mesg_recv(int fd, struct mymesg *mptr)
{
    size_t  len;
    ssize_t n;

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

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

pipemesg/mesg_recv.c

Figure 4.28 mesg_recv function.

It now takes two reads 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
```
#include    "unpipc.h"

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

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

```
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 `opens` this FIFO for writing. How can we get around this, causing both `opens` 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 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.

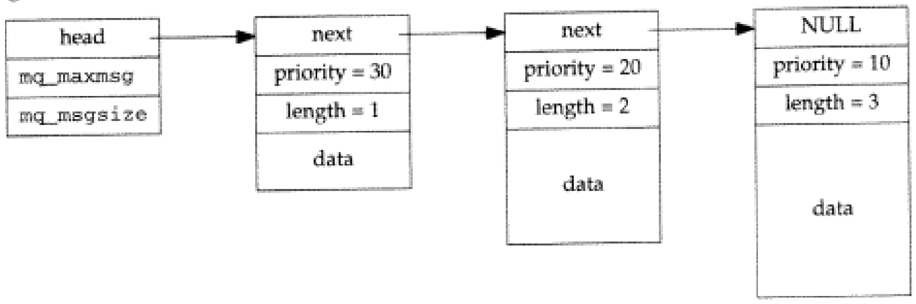

Figure 5.1 Possible arrangement of a Posix message queue containing three messages.

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

We describe the rules about the *name* argument in Section 2.2.

The *oflag* argument is one of O_RDONLY, O_WRONLY, or O_RDWR, and may be bitwise-ORed with O_CREAT, O_EXCL, and O_NONBLOCK. We describe all these flags in Section 2.3.

When a new queue is created (O_CREAT is specified and the message queue does not already exist), the *mode* and *attr* arguments are required. We describe the *mode* values in Figure 2.4. The *attr* argument lets us specify some attributes for the queue. If this argument is a null pointer, the default attributes apply. We discuss these attributes in Section 5.3.

The return value from mq_open is called a *message queue descriptor*, but it need not be (and probably is not) a small integer like a file descriptor or a socket descriptor. This value is used as the first argument to the remaining seven message queue functions.

> Solaris 2.6 defines mqd_t as a void* whereas Digital Unix 4.0B defines it as an int. In our sample implementation in Section 5.8, these descriptors are pointers to a structure. Calling these datatypes a descriptor is an unfortunate mistake.

An open message queue is closed by mq_close.

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);

                                        Returns: 0 if OK, -1 on error
```

The functionality is similar to the close of an open file: the calling process can no longer use the descriptor, but the message queue is not removed from the system. If the process terminates, all open message queues are closed, as if mq_close were called.

To remove a *name* that was used as an argument to mq_open from the system, mq_unlink must be called.

```
#include <mqueue.h>

int mq_unlink(const char *name);

                                        Returns: 0 if OK, -1 on error
```

Message queues have a reference count of how many times they are currently open (just like files), and this function is similar to the unlink function for a file: the *name* can be removed from the system while its reference count is greater than 0, but the destruction of the queue (versus removing its name from the system) does not take place until the last mq_close occurs.

Posix message queues have at least *kernel persistence* (recall Section 1.3). That is, they exist along with any messages written to the queue, even if no process currently has the queue open, until the queue is removed by calling mq_unlink and having the queue reference count reach 0.

We will see that if these message queues are implemented using memory-mapped files (Section 12.2), then they can have filesystem persistence, but this is not required and cannot be counted on.

Example: `mqcreate1` Program

Since Posix message queues have at least kernel persistence, we can write a set of small programs to manipulate these queues, providing an easy way to experiment with them. The program in Figure 5.2 creates a message queue whose name is specified as the command-line argument.

pxmsg/mqcreate1.c

```c
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, flags;
 6     mqd_t   mqd;

 7     flags = O_RDWR | O_CREAT;
 8     while ( (c = Getopt(argc, argv, "e")) != -1) {
 9         switch (c) {
10         case 'e':
11             flags |= O_EXCL;
12             break;
13         }
14     }
15     if (optind != argc - 1)
16         err_quit("usage: mqcreate [ -e ] <name>");

17     mqd = Mq_open(argv[optind], flags, FILE_MODE, NULL);

18     Mq_close(mqd);
19     exit(0);
20 }
```

pxmsg/mqcreate1.c

Figure 5.2 Create a message queue with the exclusive-create flags specified.

8-16 We allow a `-e` option that specifies an exclusive create. (We say more about the `getopt` function and our `Getopt` wrapper with Figure 5.5.) Upon return, `getopt` stores in `optind` the index of the next argument to be processed.

17 We call `mq_open` with the IPC name from the command-line, without calling our `px_ipc_name` function (Section 2.2). This lets us see exactly how the implementation handles these Posix IPC names. (We do this with all our simple test programs throughout this book.)

Here is the output under Solaris 2.6:

```
solaris % mqcreate1 /temp.1234                        first create works
solaris % ls -l /tmp/.*1234
-rw-rw-rw-   1 rstevens other1  132632 Oct 23 17:08 /tmp/.MQDtemp.1234
-rw-rw-rw-   1 rstevens other1       0 Oct 23 17:08 /tmp/.MQLtemp.1234
-rw-r--r--   1 rstevens other1       0 Oct 23 17:08 /tmp/.MQPtemp.1234
solaris % mqcreate1 -e /temp.1234                     second create with -e fails
mq_open error for /temp.1234: File exists
```

(We call this version of our program `mqcreate1`, because we enhance it in Figure 5.5 after describing attributes.) The third file has the permissions that we specify with our `FILE_MODE` constant (read–write for the user, read-only for the group and other), but the other two files have different permissions. We guess that the filename containing `D` contains the data, the filename containing `L` is some type of lock, and the filename containing `P` specifies the permissions.

Under Digital Unix 4.0B, we can see the actual pathname that is created.

```
alpha % mqcreate1 /tmp/myq.1234
alpha % ls -l  /tmp/myq.1234
-rw-r--r--   1 rstevens system   11976 Oct 23 17:04 /tmp/myq.1234
alpha % mqcreate1 -e /tmp/myq.1234
mq_open error for /tmp/myq.1234: File exists
```

Example: `mqunlink` Program

Figure 5.3 is our `mqunlink` program, which removes a message queue from the system.

pxmsg/mqunlink.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    if (argc != 2)
        err_quit("usage: mqunlink <name>");

    Mq_unlink(argv[1]);

    exit(0);
}
```

pxmsg/mqunlink.c

Figure 5.3 `mq_unlink` a message queue.

We can remove the message queue that was created by our `mqcreate` program.

```
solaris % mqunlink /temp.1234
```

All three files in the `/tmp` directory that were shown earlier are removed.

5.3 `mq_getattr` and `mq_setattr` Functions

Each message queue has four attributes, all of which are returned by `mq_getattr` and one of which is set by `mq_setattr`.

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);

int mq_setattr(mqd_t mqdes, const struct mq_attr *attr, struct mq_attr *oattr);
```

Both return: 0 if OK, –1 on error

The `mq_attr` structure contains these attributes.

```
struct mq_attr {
  long  mq_flags;    /* message queue flag: 0, O_NONBLOCK */
  long  mq_maxmsg;   /* max number of messages allowed on queue */
  long  mq_msgsize;  /* max size of a message (in bytes) */
  long  mq_curmsgs;  /* number of messages currently on queue */
};
```

A pointer to one of these structures can be passed as the fourth argument to `mq_open`, allowing us to set both `mq_maxmsg` and `mq_msgsize` when the queue is created. The other two members of this structure are ignored by `mq_open`.

`mq_getattr` fills in the structure pointed to by *attr* with the current attributes for the queue.

`mq_setattr` sets the attributes for the queue, but only the `mq_flags` member of the `mq_attr` structure pointed to by *attr* is used, to set or clear the nonblocking flag. The other three members of the structure are ignored: the maximum number of messages per queue and the maximum number of bytes per message can be set only when the queue is created, and the number of messages currently on the queue can be fetched but not set.

Additionally, if the *oattr* pointer is nonnull, the previous attributes of the queue are returned (`mq_flags`, `mq_maxmsg`, and `mq_msgsize`), along with the current status of the queue (`mq_curmsgs`).

Example: `mqgetattr` Program

The program in Figure 5.4 opens a specified message queue and prints its attributes.

pxmsg/mqgetattr.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    mqd_t   mqd;
    struct mq_attr attr;

    if (argc != 2)
        err_quit("usage: mqgetattr <name>");

    mqd = Mq_open(argv[1], O_RDONLY);

    Mq_getattr(mqd, &attr);
    printf("max #msgs = %ld, max #bytes/msg = %ld, "
           "#currently on queue = %ld\n",
           attr.mq_maxmsg, attr.mq_msgsize, attr.mq_curmsgs);

    Mq_close(mqd);
    exit(0);
}
```

pxmsg/mqgetattr.c

Figure 5.4 Fetch and print the attributes of a message queue.

We can create a message queue and print its default attributes.

```
solaris % mqcreate1 /hello.world
solaris % mqgetattr /hello.world
max #msgs = 128, max #bytes/msg = 1024, #currently on queue = 0
```

We can now see that the file size listed by `ls` when we created a queue with the default attributes following Figure 5.2 was $128 \times 1024 + 1560 = 132,632$. The 1560 extra bytes are probably overhead information: 8 bytes per message plus an additional 536 bytes.

Example: `mqcreate` Program

We can modify our program from Figure 5.2, allowing us to specify the maximum number of messages for the queue and the maximum size of each message. We cannot specify one and not the other; both must be specified (but see Exercise 5.1). Figure 5.5 is the new program.

pxmsg/mqcreate.c

```
#include    "unpipc.h"

struct mq_attr attr;            /* mq_maxmsg and mq_msgsize both init to 0 */

int
main(int argc, char **argv)
{
    int     c, flags;
    mqd_t   mqd;

    flags = O_RDWR | O_CREAT;
    while ( (c = Getopt(argc, argv, "em:z:")) != -1) {
        switch (c) {
        case 'e':
            flags |= O_EXCL;
            break;

        case 'm':
            attr.mq_maxmsg = atol(optarg);
            break;

        case 'z':
            attr.mq_msgsize = atol(optarg);
            break;
        }
    }
    if (optind != argc - 1)
        err_quit("usage: mqcreate [ -e ] [ -m maxmsg -z msgsize ] <name>");

    if ((attr.mq_maxmsg != 0 && attr.mq_msgsize == 0) ||
        (attr.mq_maxmsg == 0 && attr.mq_msgsize != 0))
        err_quit("must specify both -m maxmsg and -z msgsize");

    mqd = Mq_open(argv[optind], flags, FILE_MODE,
                  (attr.mq_maxmsg != 0) ? &attr : NULL);

    Mq_close(mqd);
    exit(0);
}
```

pxmsg/mqcreate.c

Figure 5.5 Modification of Figure 5.2 allowing attributes to be specified.

To specify that a command-line option requires an argument, we specify a colon following the option character for the m and z options in the call to getopt. When processing the option character, optarg points to the argument.

Our Getopt wrapper function calls the standard library's getopt function and terminates the process if getopt detects an error: encountering an option letter not included in the third argument, or an option letter without a required argument (indicated by a colon following the option letter in the third argument). In either case, getopt writes an error message to standard error and returns an error, which causes our Getopt wrapper to terminate. For example, the following two errors are detected by getopt:

```
solaris % mqcreate -z
mqcreate: option requires an argument -- z
solaris % mqcreate -q
mqcreate: illegal option -- q
```

The following error (not specifying the required name argument) is detected by our program:

```
solaris % mqcreate
usage: mqcreate [ -e ] [ -m maxmsg -z msgsize ] <name>
```

If neither of the two new options are specified, we must pass a null pointer as the final argument to mq_open, else we pass a pointer to our attr structure.

We now run this new version of our program, specifying a maximum of 1024 messages, each message containing up to 8192 bytes.

```
solaris % mqcreate -e -m 1024 -z 8192 /foobar
solaris % ls -al /tmp/.*foobar
-rw-rw-rw-   1 rstevens other1 8397336 Oct 25 11:29 /tmp/.MQDfoobar
-rw-rw-rw-   1 rstevens other1       0 Oct 25 11:29 /tmp/.MQLfoobar
-rw-r--r--   1 rstevens other1       0 Oct 25 11:29 /tmp/.MQPfoobar
```

The size of the file containing the data for this queue accounts for the maximum number of maximum-sized messages ($1024 \times 8192 = 8,388,608$), and the remaining 8728 bytes of overhead allows room for 8 bytes per message (8×1024) plus an additional 536 bytes.

If we execute the same program under Digital Unix 4.0B, we have

```
alpha % mqcreate -m 256 -z 2048 /tmp/bigq
alpha % ls -l /tmp/bigq
-rw-r--r--   1 rstevens system  537288 Oct 25 15:38 /tmp/bigq
```

This implementation appears to allow room for the maximum number of maximum-sized messages ($256 \times 2048 = 524,288$) and the remaining 13000 bytes of overhead allows room for 48 bytes per message (48×256) plus an additional 712 bytes.

5.4 mq_send and mq_receive Functions

These two functions place a message onto a queue and take a message off a queue. Every message has a *priority*, which is an unsigned integer less than MQ_PRIO_MAX. Posix requires that this upper limit be at least 32.

Solaris 2.6 has an upper limit of 32, but this limit is 256 with Digital Unix 4.0B. We show how to obtain these values with Figure 5.8.

mq_receive always returns the oldest message of the highest priority from the specified queue, and the priority can be returned in addition to the actual contents of the message and its length.

This operation of mq_receive differs from that of the System V msgrcv (Section 6.4). System V messages have a type field, which is similar to the priority, but with msgrcv, we can specify three different scenarios as to which message is returned: the oldest message on the queue, the oldest message with a specific type, or the oldest message whose type is less than or equal to some value.

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *ptr, size_t len, unsigned int prio);

                                              Returns: 0 if OK, –1 on error

ssize_t mq_receive(mqd_t mqdes, char *ptr, size_t len, unsigned int *priop);

                          Returns: number of bytes in message if OK, –1 on error
```

The first three arguments to both functions are similar to the first three arguments for write and read, respectively.

Declaring the pointer argument to the buffer as a char* looks like a mistake. void* would be more consistent with other Posix.1 functions.

The value of the *len* argument for mq_receive must be at least as big as the maximum size of any message that can be added to this queue, the mq_msgsize member of the mq_attr structure for this queue. If *len* is smaller than this value, EMSGSIZE is returned immediately.

This means that most applications that use Posix message queues must call mq_getattr after opening the queue, to determine the maximum message size, and then allocate one or more read buffers of that size. By requiring that the buffer always be large enough for any message on the queue, mq_receive does not need to return a notification if the message is larger than the buffer. Compare, for example, the MSG_NOERROR flag and the E2BIG error possible with System V message queues (Section 6.4) and the MSG_TRUNC flag with the recvmsg function that is used with UDP datagrams (Section 13.5 of UNPv1).

prio is the priority of the message for mq_send, and its value must be less than MQ_PRIO_MAX. If *priop* is a nonnull pointer for mq_receive, the priority of the returned message is stored through this pointer. If the application does not need messages of differing priorities, then the priority can always be specified as 0 for mq_send, and the final argument for mq_receive can be a null pointer.

A 0-byte message *is* allowed. This instance is one in which what is important is not what is said in the standard (i.e., Posix.1), but what is *not* said: nowhere is a 0-byte message forbidden. The return value from mq_receive is the number of bytes in the message (if OK) or –1 (if an error), so a return value of 0 indicates a 0-length message.

One feature is missing from both Posix message queues and System V message queues: accurately identifying the sender of each message to the receiver. This information could be useful in many applications. Unfortunately, most IPC messaging mechanisms do not identify the sender. In Section 15.5, we describe how doors provide this identity. Section 14.8 of UNPv1 describes how BSD/OS provides this identity when a Unix domain socket is used. Section 15.3.1 of APUE describes how SVR4 passes the sender's identity across a pipe when a descriptor is passed across the pipe. The BSD/OS technique is not widely implemented, and although the SVR4 technique is part of Unix 98, it requires passing a descriptor across the pipe, which is normally more expensive than just passing data across a pipe. We cannot have the sender include its identity (e.g., its effective user ID) with the message, as we cannot trust the sender to tell the truth. Although the access permissions on a message queue determine whether the sender is allowed to place a message onto the queue, this still does not identify the sender. The possibility exists to create one queue per sender (which we talk about with regard to System V message queues in Section 6.8), but this does not scale well for large applications. Lastly, realize that if the message queue functions are implemented entirely as user functions (as we show in Section 5.8), and not within the kernel, then we could not trust any sender identity that accompanied the message, as it would be easy to forge.

Example: `mqsend` Program

Figure 5.6 shows our program that adds a message to a queue.

pxmsg/mqsend.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    mqd_t   mqd;
    void    *ptr;
    size_t  len;
    uint_t  prio;

    if (argc != 4)
        err_quit("usage: mqsend <name> <#bytes> <priority>");
    len = atoi(argv[2]);
    prio = atoi(argv[3]);

    mqd = Mq_open(argv[1], O_WRONLY);

    ptr = Calloc(len, sizeof(char));
    Mq_send(mqd, ptr, len, prio);

    exit(0);
}
```

pxmsg/mqsend.c

Figure 5.6 `mqsend` program.

Both the size of the message and its priority must be specified as command-line arguments. The buffer is allocated by `calloc`, which initializes it to 0.

Example: `mqreceive` Program

The program in Figure 5.7 reads the next message from a queue.

pxmsg/mqreceive.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, flags;
 6     mqd_t   mqd;
 7     ssize_t n;
 8     uint_t  prio;
 9     void   *buff;
10     struct mq_attr attr;

11     flags = O_RDONLY;
12     while ( (c = Getopt(argc, argv, "n")) != -1) {
13         switch (c) {
14         case 'n':
15             flags |= O_NONBLOCK;
16             break;
17         }
18     }
19     if (optind != argc - 1)
20         err_quit("usage: mqreceive [ -n ] <name>");

21     mqd = Mq_open(argv[optind], flags);
22     Mq_getattr(mqd, &attr);

23     buff = Malloc(attr.mq_msgsize);

24     n = Mq_receive(mqd, buff, attr.mq_msgsize, &prio);
25     printf("read %ld bytes, priority = %u\n", (long) n, prio);

26     exit(0);
27 }
```

pxmsg/mqreceive.c

Figure 5.7 `mqreceive` program.

Allow `-n` option to specify nonblocking

14-17 A command-line option of `-n` specifies nonblocking, which causes our program to return an error if no messages are in the queue.

Open queue and get attributes

21-25 We open the queue and then get its attributes by calling `mq_getattr`. We need to determine the maximum message size, because we must allocate a buffer of this size for the call to `mq_receive`. We print the size of the message that is read and its priority.

> Since `n` is a `size_t` datatype and we do not know whether this is an `int` or a `long`, we cast the value to be a long integer and use the `%ld` format string. On a 64-bit implementation, `int` will be a 32-bit integer, but `long` and `size_t` will both be 64-bit integers.

We can use these two programs to see how the priority field is used.

```
solaris % mqcreate /test1                          create and get attributes
solaris % mqgetattr /test1
max #msgs = 128, max #bytes/msg = 1024, #currently on queue = 0

solaris % mqsend /test1 100 99999                  send with invalid priority
mq_send error: Invalid argument

solaris % mqsend /test1 100 6                      100 bytes, priority of 6
solaris % mqsend /test1 50 18                      50 bytes, priority of 18
solaris % mqsend /test1 33 18                      33 bytes, priority of 18

solaris % mqreceive /test1
read 50 bytes, priority = 18                       oldest, highest priority message is returned
solaris % mqreceive /test1
read 33 bytes, priority = 18
solaris % mqreceive /test1
read 100 bytes, priority = 6
solaris % mqreceive -n /test1                      specify nonblocking; queue is empty
mq_receive error: Resource temporarily unavailable
```

We can see that `mq_receive` returns the oldest message of the highest priority.

5.5 Message Queue Limits

We have already encountered two limits for any given queue, both of which are established when the queue is created:

`mq_maxmsg`	the maximum number of messages on the queue, and
`mq_msgsize`	the maximum size of a given message.

No inherent limits exist on either value, although for the two implementations that we have looked at, room in the filesystem must exist for a file whose size is the product of these two numbers, plus some small amount of overhead. Virtual memory requirements may also exist based on the size of the queue (see Exercise 5.5).

Two other limits are defined by the implementation:

`MQ_OPEN_MAX`	the maximum number of message queues that a process can have open at once (Posix requires that this be at least 8), and
`MQ_PRIO_MAX`	the maximum value plus one for the priority of any message (Posix requires that this be at least 32).

These two constants are often defined in the `<unistd.h>` header and can also be obtained at run time by calling the `sysconf` function, as we show next.

Example: `mqsysconf` Program

The program in Figure 5.8 calls `sysconf` and prints the two implementation-defined limits for message queues.

pxmsg/mqsysconf.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    printf("MQ_OPEN_MAX = %ld, MQ_PRIO_MAX = %ld\n",
           Sysconf(_SC_MQ_OPEN_MAX), Sysconf(_SC_MQ_PRIO_MAX));
    exit(0);
}
```

pxmsg/mqsysconf.c

Figure 5.8 Call `sysconf` to obtain message queue limits.

If we execute this on our two systems, we obtain

```
solaris % mqsysconf
MQ_OPEN_MAX = 32, MQ_PRIO_MAX = 32

alpha % mqsysconf
MQ_OPEN_MAX = 64, MQ_PRIO_MAX = 256
```

5.6 `mq_notify` Function

One problem that we will see with System V message queues in Chapter 6 is their inability to notify a process when a message is placed onto a queue. We can block in a call to `msgrcv`, but that prevents us from doing anything else while we are waiting. If we specify the nonblocking flag for `msgrcv` (`IPC_NOWAIT`), we do not block, but we must continually call this function to determine when a message arrives. We said this is called *polling* and is a waste of CPU time. We want a way for the system to tell us when a message is placed onto a queue that was previously empty.

> This section and the remaining sections of this chapter contain advanced topics that you may want to skip on a first reading.

Posix message queues allow for an *asynchronous event notification* when a message is placed onto an empty message queue. This notification can be either

- the generation of a signal, or
- the creation of a thread to execute a specified function.

We establish this notification by calling `mq_notify`.

```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);

                                        Returns: 0 if OK, -1 on error
```

This function establishes or removes the asynchronous event notification for the specified queue. The `sigevent` structure is new with the Posix.1 realtime signals, which we say more about in the next section. This structure and all of the new signal-related constants introduced in this chapter are defined by `<signal.h>`.

```
union sigval {
  int    sival_int;              /* integer value */
  void  *sival_ptr;              /* pointer value */
};

struct sigevent {
  int             sigev_notify; /* SIGEV_{NONE,SIGNAL,THREAD} */
  int             sigev_signo;  /* signal number if SIGEV_SIGNAL */
  union sigval    sigev_value;  /* passed to signal handler or thread */
                                /* following two if SIGEV_THREAD */
  void          (*sigev_notify_function)(union sigval);
  pthread_attr_t  *sigev_notify_attributes;
};
```

We will show some examples shortly of the different ways to use this notification, but a few rules apply in general for this function.

1. If the *notification* argument is nonnull, then the process wants to be notified when a message arrives for the specified queue and the queue is empty. We say that "the process is registered for notification for the queue."

2. If the *notification* argument is a null pointer and if the process is currently registered for notification for the queue, the existing registration is removed.

3. Only one process at any given time can be registered for notification for a given queue.

4. When a message arrives for a queue that was previously empty and a process is registered for notification for the queue, the notification is sent only if no thread is blocked in a call to `mq_receive` for that queue. That is, blocking in a call to `mq_receive` takes precedence over any registration for notification.

5. When the notification is sent to the registered process, the registration is removed. The process must reregister (if desired) by calling `mq_notify` again.

One of the original problems with Unix signals was that a signal's action was reset to its default each time the signal was generated (Section 10.4 of APUE). Usually the first function called by a signal handler was `signal`, to reestablish the handler. This provided a small window of time, between the signal's generation and the process reestablishing its signal handler, during which another occurrence of that signal could terminate the process. At first glance, we seem to have a similar problem with `mq_notify`, since the process must reregister each time the notification occurs. But message queues are different from signals, because the notification cannot occur again until the queue is empty. Therefore, we must be careful to reregister *before* reading the message from the queue.

Example: Simple Signal Notification

Before getting into the details of Posix realtime signals or threads, we can write a simple program that causes `SIGUSR1` to be generated when a message is placed onto an empty queue. We show this program in Figure 5.9 and note that this program contains an error that we talk about in detail shortly.

pxmsg/mqnotifysig1.c

```
 1 #include    "unpipc.h"

 2 mqd_t   mqd;
 3 void   *buff;
 4 struct mq_attr attr;
 5 struct sigevent sigev;

 6 static void sig_usr1(int);

 7 int
 8 main(int argc, char **argv)
 9 {
10     if (argc != 2)
11         err_quit("usage: mqnotifysig1 <name>");

12         /* open queue, get attributes, allocate read buffer */
13     mqd = Mq_open(argv[1], O_RDONLY);
14     Mq_getattr(mqd, &attr);
15     buff = Malloc(attr.mq_msgsize);

16         /* establish signal handler, enable notification */
17     Signal(SIGUSR1, sig_usr1);
18     sigev.sigev_notify = SIGEV_SIGNAL;
19     sigev.sigev_signo = SIGUSR1;
20     Mq_notify(mqd, &sigev);

21     for ( ; ; )
22         pause();                /* signal handler does everything */
23     exit(0);
24 }

25 static void
26 sig_usr1(int signo)
27 {
28     ssize_t n;

29     Mq_notify(mqd, &sigev);     /* reregister first */
30     n = Mq_receive(mqd, buff, attr.mq_msgsize, NULL);
31     printf("SIGUSR1 received, read %ld bytes\n", (long) n);
32     return;
33 }
```

pxmsg/mqnotifysig1.c

Figure 5.9 Generate SIGUSR1 when message placed onto an empty queue (incorrect version).

Declare globals

2-6 We declare some globals that are used by both the `main` function and our signal handler (`sig_usr1`).

Open queue, get attributes, allocate read buffer

12-15 We open the message queue, obtain its attributes, and allocate a read buffer.

Establish signal handler, enable notification

16-20 We first establish our signal handler for `SIGUSR1`. We fill in the `sigev_notify` member of the `sigevent` structure with the `SIGEV_SIGNAL` constant, which says that we want a signal generated when the queue goes from empty to not-empty. We set the `sigev_signo` member to the signal that we want generated and call `mq_notify`.

Infinite loop

21-22 Our `main` function is then an infinite loop that goes to sleep in the `pause` function, which returns –1 each time a signal is caught.

Catch signal, read message

25-33 Our signal handler calls `mq_notify`, to reregister for the next event, reads the message, and prints its length. In this program, we ignore the received message's priority.

> The `return` statement at the end of `sig_usr1` is not needed, since there is no return value and falling off the end of the function is an implicit return to the caller. Nevertheless, the author always codes an explicit `return` at the end of a signal handler to reiterate that the return from this function is special. It might cause the premature return (with an error of `EINTR`) of a function call in the thread that handles the signal.

We now run this program from one window

```
solaris % mqcreate /test1               create queue
solaris % mqnotifysig1 /test1           start program from Figure 5.9
```

and then execute the following commands from another window:

```
solaris % mqsend /test1 50 16           send 50-byte message with priority of 16
```

As expected, our `mqnotifysig1` program outputs `SIGUSR1 received, read 50 bytes`.

We can verify that only one process at a time can be registered for the notification, by starting another copy of our program from another window:

```
solaris % mqnotifysig1 /test1
mq_notify error: Device busy
```

This error message corresponds to `EBUSY`.

Posix Signals: Async-Signal-Safe Functions

The problem with Figure 5.9 is that it calls `mq_notify`, `mq_receive`, and `printf` from the signal handler. None of these functions may be called from a signal handler.

Posix uses the term *async-signal-safe* to describe the functions that may be called from a signal handler. Figure 5.10 lists these Posix functions, along with a few that are added by Unix 98.

Functions not listed may not be called from a signal handler. Note that none of the standard I/O functions are listed and none of the `pthread_XXX` functions are listed. Of all the IPC functions covered in this text, only `sem_post`, `read`, and `write` are listed (we are assuming the latter two would be used with pipes and FIFOs).

> ANSI C lists four functions that may be called from a signal handler: `abort`, `exit`, `longjmp`, and `signal`. The first three are not listed as async-signal-safe by Unix 98.

access	fpathconf	rename	sysconf
aio_return	fstat	rmdir	tcdrain
aio_suspend	fsync	sem_post	tcflow
alarm	getegid	setgid	tcflush
cfgetispeed	geteuid	setpgid	tcgetattr
cfgetospeed	getgid	setsid	tcgetpgrp
cfsetispeed	getgroups	setuid	tcsendbreak
cfsetospeed	getpgrp	sigaction	tcsetattr
chdir	getpid	sigaddset	tcsetpgrp
chmod	getppid	sigdelset	time
chown	getuid	sigemptyset	timer_getoverrun
clock_gettime	kill	sigfillset	timer_gettime
close	link	sigismember	timer_settime
creat	lseek	signal	times
dup	mkdir	sigpause	umask
dup2	mkfifo	sigpending	uname
execle	open	sigprocmask	unlink
execve	pathconf	sigqueue	utime
_exit	pause	sigset	wait
fcntl	pipe	sigsuspend	waitpid
fdatasync	raise	sleep	write
fork	read	stat	

Figure 5.10 Functions that are async-signal-safe.

Example: Signal Notification

One way to avoid calling *any* function from a signal handler is to have the handler just set a global flag that some thread examines to determine when a message has been received. Figure 5.11 shows this technique, although it contains a different error, which we describe shortly.

Global variable

2 Since the only operation performed by our signal handler is to set `mqflag` nonzero, the global variables from Figure 5.9 need not be global. Reducing the number of global variables is always a good technique, especially when threads are being used.

Open message queue

15-18 We open the message queue, obtain its attributes, and allocate a receive buffer.

Initialize signal sets

19-22 We initialize three signal sets and turn on the bit for `SIGUSR1` in the set `newmask`.

Establish signal handler, enable notification

23-27 We establish a signal handler for `SIGUSR1`, fill in our `sigevent` structure, and call `mq_notify`.

pxmsg/mqnotifysig2.c

```c
#include    "unpipc.h"

volatile sig_atomic_t mqflag;   /* set nonzero by signal handler */
static void sig_usr1(int);

int
main(int argc, char **argv)
{
    mqd_t   mqd;
    void   *buff;
    ssize_t n;
    sigset_t zeromask, newmask, oldmask;
    struct mq_attr attr;
    struct sigevent sigev;

    if (argc != 2)
        err_quit("usage: mqnotifysig2 <name>");

        /* open queue, get attributes, allocate read buffer */
    mqd = Mq_open(argv[1], O_RDONLY);
    Mq_getattr(mqd, &attr);
    buff = Malloc(attr.mq_msgsize);

    Sigemptyset(&zeromask);     /* no signals blocked */
    Sigemptyset(&newmask);
    Sigemptyset(&oldmask);
    Sigaddset(&newmask, SIGUSR1);

        /* establish signal handler, enable notification */
    Signal(SIGUSR1, sig_usr1);
    sigev.sigev_notify = SIGEV_SIGNAL;
    sigev.sigev_signo = SIGUSR1;
    Mq_notify(mqd, &sigev);

    for ( ; ; ) {
        Sigprocmask(SIG_BLOCK, &newmask, &oldmask);     /* block SIGUSR1 */
        while (mqflag == 0)
            sigsuspend(&zeromask);
        mqflag = 0;             /* reset flag */

        Mq_notify(mqd, &sigev); /* reregister first */
        n = Mq_receive(mqd, buff, attr.mq_msgsize, NULL);
        printf("read %ld bytes\n", (long) n);
        Sigprocmask(SIG_UNBLOCK, &newmask, NULL);   /* unblock SIGUSR1 */
    }
    exit(0);
}

static void
sig_usr1(int signo)
{
    mqflag = 1;
    return;
}
```

pxmsg/mqnotifysig2.c

Figure 5.11 Signal handler just sets a flag for main thread (incorrect version).

Wait for signal handler to set flag

28-32 We call sigprocmask to block SIGUSR1, saving the current signal mask in oldmask. We then test the global mqflag in a loop, waiting for the signal handler to set it nonzero. As long as it is 0, we call sigsuspend, which atomically puts the calling thread to sleep and resets its signal mask to zeromask (no signals are blocked). Section 10.16 of APUE talks more about sigsuspend and why we must test the mqflag variable only when SIGUSR1 is blocked. Each time sigsuspend returns, SIGUSR1 is blocked.

Reregister and read message

33-36 When mqflag is nonzero, we reregister and then read the message from the queue. We then unblock SIGUSR1 and go back to the top of the for loop.

We mentioned that a problem still exists with this solution. Consider what happens if two messages arrive for the queue before the first message is read. We can simulate this by adding a sleep before the call to mq_notify. The fundamental problem is that the notification is sent *only* when a message is placed onto an empty queue. If two messages arrive for a queue before we can read the first, only one notification is sent: we read the first message and then call sigsuspend waiting for another message, which may never be sent. In the meantime, another message is already sitting on the queue waiting to be read that we are ignoring.

Example: Signal Notification with Nonblocking mq_receive

The correction to the problem just noted is to *always* read a message queue in a nonblocking mode when mq_notify is being used to generate a signal. Figure 5.12 shows a modification to Figure 5.11 that reads the message queue in a nonblocking mode.

Open message queue nonblocking

15-18 The first change is to specify O_NONBLOCK when the message queue is opened.

Read all messages from queue

34-38 The other change is to call mq_receive in a loop, processing each message on the queue. An error return of EAGAIN is OK and just means that no more messages exist.

Example: Signal Notification Using sigwait instead of a Signal Handler

Although the previous example is correct, it could be more efficient. Our program blocks, waiting for a message to arrive, by calling sigsuspend. When a message is placed onto an empty queue, the signal is generated, the main thread is stopped, the signal handler executes and sets the mqflag variable, the main thread executes again, finds mq_flag nonzero, and reads the message. An easier approach (and probably more efficient) would be to block in a function just waiting for the signal to be delivered, without having the kernel execute a signal handler just to set a flag. This capability is provided by sigwait.

pxmsg/mqnotifysig3.c

```
#include    "unpipc.h"

volatile sig_atomic_t mqflag;   /* set nonzero by signal handler */
static void sig_usr1(int);

int
main(int argc, char **argv)
{
    mqd_t   mqd;
    void   *buff;
    ssize_t n;
    sigset_t zeromask, newmask, oldmask;
    struct mq_attr attr;
    struct sigevent sigev;

    if (argc != 2)
        err_quit("usage: mqnotifysig3 <name>");

        /* open queue, get attributes, allocate read buffer */
    mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    Mq_getattr(mqd, &attr);
    buff = Malloc(attr.mq_msgsize);

    Sigemptyset(&zeromask);     /* no signals blocked */
    Sigemptyset(&newmask);
    Sigemptyset(&oldmask);
    Sigaddset(&newmask, SIGUSR1);

        /* establish signal handler, enable notification */
    Signal(SIGUSR1, sig_usr1);
    sigev.sigev_notify = SIGEV_SIGNAL;
    sigev.sigev_signo = SIGUSR1;
    Mq_notify(mqd, &sigev);

    for ( ; ; ) {
        Sigprocmask(SIG_BLOCK, &newmask, &oldmask);     /* block SIGUSR1 */
        while (mqflag == 0)
            sigsuspend(&zeromask);
        mqflag = 0;             /* reset flag */

        Mq_notify(mqd, &sigev); /* reregister first */
        while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0) {
            printf("read %ld bytes\n", (long) n);
        }
        if (errno != EAGAIN)
            err_sys("mq_receive error");
        Sigprocmask(SIG_UNBLOCK, &newmask, NULL);   /* unblock SIGUSR1 */
    }
    exit(0);
}

static void
sig_usr1(int signo)
{
    mqflag = 1;
    return;
}
```

pxmsg/mqnotifysig3.c

Figure 5.12 Using a signal notification to read a Posix message queue.

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);

                                Returns: 0 if OK, positive Exxx value on error
```

Before calling `sigwait`, we block some set of signals. We specify this set of signals as the *set* argument. `sigwait` then blocks until one or more of these signals is pending, at which time it returns one of the signals. That signal value is stored through the pointer *sig*, and the return value of the function is 0. This is called "synchronously waiting for an asynchronous event": we are using a signal but without an asynchronous signal handler.

Figure 5.13 shows the use of `mq_notify` with `sigwait`.

Initialize signal set and block `SIGUSR1`

18-20 One signal set is initialized to contain just `SIGUSR1`, and this signal is then blocked by `sigprocmask`.

Wait for signal

26-34 We now block, waiting for the signal, in a call to `sigwait`. When `SIGUSR1` is delivered, we reregister the notification and read all available messages.

> `sigwait` is often used with a multithreaded process. Indeed, looking at its function prototype, we see that its return value is 0 or one of the *Exxx* errors, which is the same as most of the Pthread functions. But `sigprocmask` cannot be used with a multithreaded process; instead, `pthread_sigmask` must be called, and it changes the signal mask of just the calling thread. The arguments for `pthread_sigmask` are identical to those for `sigprocmask`.
>
> Two variants of `sigwait` exist: `sigwaitinfo` also returns a `siginfo_t` structure (which we define in the next section) and is intended for use with reliable signals. `sigtimedwait` also returns a `siginfo_t` structure and allows the caller to specify a time limit.
>
> Most threads books, such as [Butenhof 1997], recommend using `sigwait` to handle all signals in a multithreaded process and never using asynchronous signal handlers.

Example: Posix Message Queues with `select`

A message queue descriptor (an `mqd_t` variable) is not a "normal" descriptor and cannot be used with either `select` or `poll` (Chapter 6 of UNPv1). Nevertheless, we can use them along with a pipe and the `mq_notify` function. (We show a similar technique in Section 6.9 with System V message queues, which involves a child process and a pipe.) First, notice from Figure 5.10 that the `write` function is async-signal-safe, so we can call it from a signal handler. Figure 5.14 shows our program.

pxmsg/mqnotifysig4.c

```c
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     signo;
    mqd_t   mqd;
    void   *buff;
    ssize_t n;
    sigset_t newmask;
    struct mq_attr attr;
    struct sigevent sigev;

    if (argc != 2)
        err_quit("usage: mqnotifysig4 <name>");

        /* open queue, get attributes, allocate read buffer */
    mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    Mq_getattr(mqd, &attr);
    buff = Malloc(attr.mq_msgsize);

    Sigemptyset(&newmask);
    Sigaddset(&newmask, SIGUSR1);
    Sigprocmask(SIG_BLOCK, &newmask, NULL);     /* block SIGUSR1 */

        /* establish signal handler, enable notification */
    sigev.sigev_notify = SIGEV_SIGNAL;
    sigev.sigev_signo = SIGUSR1;
    Mq_notify(mqd, &sigev);

    for ( ; ; ) {
        Sigwait(&newmask, &signo);
        if (signo == SIGUSR1) {
            Mq_notify(mqd, &sigev);     /* reregister first */
            while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0) {
                printf("read %ld bytes\n", (long) n);
            }
            if (errno != EAGAIN)
                err_sys("mq_receive error");
        }
    }
    exit(0);
}
```

pxmsg/mqnotifysig4.c

Figure 5.13 Using `mq_notify` with `sigwait`.

pxmsg/mqnotifysig5.c

```c
#include    "unpipc.h"

int     pipefd[2];
static void sig_usr1(int);
```

```
int
main(int argc, char **argv)
{
    int     nfds;
    char    c;
    fd_set  rset;
    mqd_t   mqd;
    void   *buff;
    ssize_t n;
    struct mq_attr attr;
    struct sigevent sigev;

    if (argc != 2)
        err_quit("usage: mqnotifysig5 <name>");

        /* open queue, get attributes, allocate read buffer */
    mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    Mq_getattr(mqd, &attr);
    buff = Malloc(attr.mq_msgsize);

    Pipe(pipefd);

        /* establish signal handler, enable notification */
    Signal(SIGUSR1, sig_usr1);
    sigev.sigev_notify = SIGEV_SIGNAL;
    sigev.sigev_signo = SIGUSR1;
    Mq_notify(mqd, &sigev);

    FD_ZERO(&rset);
    for ( ; ; ) {
        FD_SET(pipefd[0], &rset);
        nfds = Select(pipefd[0] + 1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(pipefd[0], &rset)) {
            Read(pipefd[0], &c, 1);
            Mq_notify(mqd, &sigev);     /* reregister first */
            while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0) {
                printf("read %ld bytes\n", (long) n);
            }
            if (errno != EAGAIN)
                err_sys("mq_receive error");
        }
    }
    exit(0);
}

static void
sig_usr1(int signo)
{
    Write(pipefd[1], "", 1);    /* one byte of 0 */
    return;
}
```

pxmsg/mqnotifysig5.c

Figure 5.14 Using a signal notification with a pipe.

Create a pipe

21 We create a pipe that the signal handler will write to when a notification is received for the message queue. This is an example of a pipe being used within a single process.

Call `select`

27-40 We initialize the descriptor set `rset` and each time around the loop turn on the bit corresponding to `pipefd[0]` (the read end of the pipe). We then call `select` waiting for only this descriptor, although in a typical application, this is where input or output on multiple descriptors would be multiplexed. When the read end of the pipe is readable, we reregister the message queue notification and read all available messages.

Signal handler

43-48 Our signal handler just `writes` 1 byte to the pipe. As we mentioned, this is an async-signal-safe operation.

Example: Initiate Thread

Another alternative is to set `sigev_notify` to `SIGEV_THREAD`, which causes a new thread to be created. The function specified by the `sigev_notify_function` is called with the parameter of `sigev_value`. The thread attributes for the new thread are specified by `sigev_notify_attributes`, which can be a null pointer if the default attributes are OK. Figure 5.15 shows an example of this technique.

We specify a null pointer for the new thread's argument (`sigev_value`), so nothing is passed to the thread start function. We could pass a pointer to the message queue descriptor as the argument, instead of declaring it as a global, but the new thread still needs the message queue attributes and the `sigev` structure (to reregister). We specify a null pointer for the new thread's attributes, so system defaults are used. These new threads are created as detached threads.

Unfortunately, neither of the systems being used for these examples, Solaris 2.6 and Digital Unix 4.0B, support `SIGEV_THREAD`. Both require that `sigev_notify` be either `SIGEV_NONE` or `SIGEV_SIGNAL`.

5.7 Posix Realtime Signals

Unix signals have gone through numerous evolutionary changes over the past years.

1. The signal model provided by Version 7 Unix (1978) was unreliable. Signals could get lost, and it was hard for a process to turn off selected signals while executing critical sections of code.
2. 4.3BSD (1986) added reliable signals.
3. System V Release 3.0 (1986) also added reliable signals, albeit differently from the BSD model.
4. Posix.1 (1990) standardized the BSD reliable signal model, and Chapter 10 of APUE describes this model in detail.

pxmsg/mqnotifythread1.c

```
#include    "unpipc.h"

mqd_t   mqd;
struct mq_attr attr;
struct sigevent sigev;

static void notify_thread(union sigval);    /* our thread function */

int
main(int argc, char **argv)
{
    if (argc != 2)
        err_quit("usage: mqnotifythread1 <name>");

    mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    Mq_getattr(mqd, &attr);

    sigev.sigev_notify = SIGEV_THREAD;
    sigev.sigev_value.sival_ptr = NULL;
    sigev.sigev_notify_function = notify_thread;
    sigev.sigev_notify_attributes = NULL;
    Mq_notify(mqd, &sigev);

    for ( ; ; )
        pause();                /* each new thread does everything */

    exit(0);
}

static void
notify_thread(union sigval arg)
{
    ssize_t n;
    void   *buff;

    printf("notify_thread started\n");
    buff = Malloc(attr.mq_msgsize);
    Mq_notify(mqd, &sigev);      /* reregister */

    while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0) {
        printf("read %ld bytes\n", (long) n);
    }
    if (errno != EAGAIN)
        err_sys("mq_receive error");

    free(buff);
    pthread_exit(NULL);
}
```

pxmsg/mqnotifythread1.c

Figure 5.15 `mq_notify` that initiates a new thread.

5. Posix.1 (1996) added realtime signals to the Posix model. This work originated from the Posix.1b realtime extensions (which was called Posix.4).

Almost every Unix system today provides Posix reliable signals, and newer systems are providing the Posix realtime signals. (Be careful to differentiate between reliable and realtime when describing signals.) We need to say more about the realtime signals, as we have already encountered some of the structures defined by this extension in the previous section (the `sigval` and `sigevent` structures).

Signals can be divided into two groups:

1. The realtime signals whose values are between `SIGRTMIN` and `SIGRTMAX`, inclusive. Posix requires that at least `RTSIG_MAX` of these realtime signals be provided, and the minimum value for this constant is 8.
2. All other signals: `SIGALRM`, `SIGINT`, `SIGKILL`, and so on.

> On Solaris 2.6, the normal Unix signals are numbered 1 through 37, and 8 realtime signals are defined with values from 38 through 45. On Digital Unix 4.0B, the normal Unix signals are numbered 1 through 32, and 16 realtime signals are defined with values from 33 through 48. Both implementations define `SIGRTMIN` and `SIGRTMAX` as macros that call `sysconf`, to allow their values to change in the future.

Next we note whether or not the new `SA_SIGINFO` flag is specified in the call to `sigaction` by the process that receives the signal. These differences lead to the four possible scenarios shown in Figure 5.16.

Signal	Call to `sigaction`	
	`SA_SIGINFO` specified	`SA_SIGINFO` not specified
`SIGRTMIN` through `SIGRTMAX`	realtime behavior guaranteed	realtime behavior unspecified
all other signals	realtime behavior unspecified	realtime behavior unspecified

Figure 5.16 Realtime behavior of Posix signals, depending on `SA_SIGINFO`.

What we mean in the three boxes labeled "realtime behavior unspecified" is that some implementations may provide realtime behavior and some may not. If we want real-time behavior, we *must* use the new realtime signals between `SIGRTMIN` and `SIGRTMAX`, and we *must* specify the `SA_SIGINFO` flag to `sigaction` when the signal handler is installed.

The term *realtime behavior* implies the following characteristics:

- Signals are queued. That is, if the signal is generated three times, it is delivered three times. Furthermore, multiple occurrences of a given signal are queued in a first-in, first-out (FIFO) order. We show an example of signal queueing shortly. For signals that are not queued, a signal that is generated three times can be delivered only once.
- When multiple, unblocked signals in the range `SIGRTMIN` through `SIGRTMAX` are queued, lower-numbered signals are delivered before higher-numbered signals. That is, `SIGRTMIN` is a "higher priority" than the signal numbered `SIGRTMIN+1`, which is a "higher priority" than the signal numbered `SIGRTMIN+2`, and so on.

- When a nonrealtime signal is delivered, the only argument to the signal handler is the signal number. Realtime signals carry more information than other signals. The signal handler for a realtime signal that is installed with the `SA_SIGINFO` flag set is declared as

  ```
  void func(int signo, siginfo_t *info, void *context);
  ```

  signo is the signal number, and the `siginfo_t` structure is defined as

  ```
  typedef struct {
    int           si_signo; /* same value as signo argument */
    int           si_code;  /* SI_{USER,QUEUE,TIMER,ASYNCIO,MESGQ} */
    union sigval  si_value; /* integer or pointer value from sender */
  } siginfo_t;
  ```

  What the *context* argument points to is implementation dependent.

  Technically a nonrealtime Posix signal handler is called with just one argument. Many Unix systems have an older, three-argument convention for signal handlers that predates the Posix realtime standard.

  `siginfo_t` is the only Posix structure defined as a `typedef` of a name ending in `_t`. In Figure 5.17 we declare pointers to these structures as `siginfo_t *` without the word `struct`.

- Some new functions are defined to work with the realtime signals. For example, the `sigqueue` function is used instead of the `kill` function, to send a signal to some process, and the new function allows the sender to pass a `sigval` union with the signal.

The realtime signals are generated by the following Posix.1 features, identified by the `si_code` value contained in the `siginfo_t` structure that is passed to the signal handler.

`SI_ASYNCIO`	The signal was generated by the completion of an asynchronous I/O request: the Posix `aio_XXX` functions, which we do not describe.
`SI_MESGQ`	The signal was generated when a message was placed onto an empty message queue, as we described in Section 5.6.
`SI_QUEUE`	The signal was sent by the `sigqueue` function. We show an example of this shortly.
`SI_TIMER`	The signal was generated by the expiration of a timer that was set by the `timer_settime` function, which we do not describe.
`SI_USER`	The signal was sent by the `kill` function.

If the signal was generated by some other event, `si_code` will be set to some value other than the ones just shown. The contents of the `si_value` member of the `siginfo_t` structure are valid only when `si_code` is `SI_ASYNCIO`, `SI_MESGQ`, `SI_QUEUE`, or `SI_TIMER`.

Example

Figure 5.17 is a simple program that demonstrates realtime signals. The program calls `fork`, the child blocks three realtime signals, the parent then sends nine signals (three occurrences each of three realtime signals), and the child then unblocks the signals and we see how many occurrences of each signal are delivered and the order in which the signals are delivered.

Print realtime signal numbers

10 We print the minimum and maximum realtime signal numbers, to see how many realtime signals the implementation supports. We cast the two constants to an integer, because some implementations define these two constants to be macros that call `sysconf`, as in

```
#define  SIGRTMAX  (sysconf(_SC_RTSIG_MAX))
```

and `sysconf` returns a long integer (see Exercise 5.4).

`fork`: child blocks three realtime signals

11-17 A child is spawned, and the child calls `sigprocmask` to block the three realtime signals that we are using: `SIGRTMAX`, `SIGRTMAX-1`, and `SIGRTMAX-2`.

Establish signal handler

18-21 We call our `signal_rt` function (which we show in Figure 5.18) to establish our function `sig_rt` as the handler for the three realtime signals. This function sets the `SA_SIGINFO` flag, and since these three signals are realtime signals, we expect realtime behavior. This function also sets the mask of signals to block while the signal handler is executing.

Wait for parent to generate the signals, then unblock the signals

22-25 We wait 6 seconds to allow the parent to generate the nine signals. We then call `sigprocmask` to unblock the three realtime signals. This should allow all the queued signals to be delivered. We pause for another 3 seconds, to let the signal handler call `printf` nine times, and then the child terminates.

Parent sends the nine signals

27-36 The parent pauses for 3 seconds to let the child block all signals. The parent then generates three occurrences of each of the three realtime signals: `i` assumes three values, and `j` takes on the values 0, 1, and 2 for each value of `i`. We purposely generate the signals starting with the highest signal number, because we expect them to be delivered starting with the lowest signal number. We also send a different integer value (`sival_int`) with each signal, to verify that the three occurrences of a given signal are generated in FIFO order.

Signal handler

38-43 Our signal handler just prints the information about the signal that is delivered.

We noted with Figure 5.10 that `printf` is not async-signal-safe and should not be called from a signal handler. We call it here as a simple diagnostic tool in this little test program.

rtsignals/test1.c

```
#include    "unpipc.h"

static void sig_rt(int, siginfo_t *, void *);

int
main(int argc, char **argv)
{
    int     i, j;
    pid_t   pid;
    sigset_t newset;
    union sigval val;

    printf("SIGRTMIN = %d, SIGRTMAX = %d\n", (int) SIGRTMIN, (int) SIGRTMAX);

    if ( (pid = Fork()) == 0) {
            /* child: block three realtime signals */
        Sigemptyset(&newset);
        Sigaddset(&newset, SIGRTMAX);
        Sigaddset(&newset, SIGRTMAX - 1);
        Sigaddset(&newset, SIGRTMAX - 2);
        Sigprocmask(SIG_BLOCK, &newset, NULL);

            /* establish signal handler with SA_SIGINFO set */
        Signal_rt(SIGRTMAX, sig_rt, &newset);
        Signal_rt(SIGRTMAX - 1, sig_rt, &newset);
        Signal_rt(SIGRTMAX - 2, sig_rt, &newset);

        sleep(6);                /* let parent send all the signals */

        Sigprocmask(SIG_UNBLOCK, &newset, NULL);    /* unblock */
        sleep(3);                /* let all queued signals be delivered */
        exit(0);
    }
        /* parent sends nine signals to child */
    sleep(3);                    /* let child block all signals */
    for (i = SIGRTMAX; i >= SIGRTMAX - 2; i--) {
        for (j = 0; j <= 2; j++) {
            val.sival_int = j;
            Sigqueue(pid, i, val);
            printf("sent signal %d, val = %d\n", i, j);
        }
    }
    exit(0);
}

static void
sig_rt(int signo, siginfo_t *info, void *context)
{
    printf("received signal #%d, code = %d, ival = %d\n",
           signo, info->si_code, info->si_value.sival_int);
}
```

rtsignals/test1.c

Figure 5.17 Simple test program to demonstrate realtime signals.

We first run the program under Solaris 2.6, but the output is not what is expected.

```
solaris % test1
SIGRTMIN = 38, SIGRTMAX = 45                      8 realtime signals provided
                                                  3-second pause in here
sent signal 45, val = 0                           parent now sends the nine signals
sent signal 45, val = 1
sent signal 45, val = 2
sent signal 44, val = 0
sent signal 44, val = 1
sent signal 44, val = 2
sent signal 43, val = 0
sent signal 43, val = 1
sent signal 43, val = 2
solaris %                                         parent terminates, shell prompt printed
                                                  3-second pause before child unblocks the signals
received signal #45, code = -2, ival = 2          child catches the signals
received signal #45, code = -2, ival = 1
received signal #45, code = -2, ival = 0
received signal #44, code = -2, ival = 2
received signal #44, code = -2, ival = 1
received signal #44, code = -2, ival = 0
received signal #43, code = -2, ival = 2
received signal #43, code = -2, ival = 1
received signal #43, code = -2, ival = 0
```

The nine signals are queued, but the three signals are generated starting with the highest signal number (we expect the lowest signal number to be generated first). Then for a given signal, the queued signals appear to be delivered in LIFO, not FIFO, order. The `si_code` of −2 corresponds to `SI_QUEUE`.

We now run the program under Digital Unix 4.0B and see the expected results.

```
alpha % test1
SIGRTMIN = 33, SIGRTMAX = 48                      16 realtime signals provided
                                                  3-second pause in here
sent signal 48, val = 0                           parent now sends the nine signals
sent signal 48, val = 1
sent signal 48, val = 2
sent signal 47, val = 0
sent signal 47, val = 1
sent signal 47, val = 2
sent signal 46, val = 0
sent signal 46, val = 1
sent signal 46, val = 2
alpha %                                           parent terminates, shell prompt printed
                                                  3-second pause before child unblocks the signals
received signal #46, code = -1, ival = 0          child catches the signals
received signal #46, code = -1, ival = 1
received signal #46, code = -1, ival = 2
received signal #47, code = -1, ival = 0
received signal #47, code = -1, ival = 1
received signal #47, code = -1, ival = 2
received signal #48, code = -1, ival = 0
received signal #48, code = -1, ival = 1
received signal #48, code = -1, ival = 2
```

The nine signals are queued and then delivered in the order that we expect: the lowest-numbered-signal first, and for a given signal, the three occurrences are delivered in FIFO order.

The Solaris 2.6 implementation appears to have a bug.

`signal_rt` Function

On p. 120 of UNPv1, we show our `signal` function, which calls the Posix `sigaction` function to establish a signal handler that provides reliable Posix semantics. We now modify that function to provide realtime behavior. We call this new function `signal_rt` and show it in Figure 5.18.

lib/signal_rt.c

```
 1 #include    "unpipc.h"

 2 Sigfunc_rt *
 3 signal_rt(int signo, Sigfunc_rt *func, sigset_t *mask)
 4 {
 5     struct sigaction act, oact;

 6     act.sa_sigaction = func;    /* must store function addr here */
 7     act.sa_mask = *mask;        /* signals to block */
 8     act.sa_flags = SA_SIGINFO;  /* must specify this for realtime */
 9     if (signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
12 #endif
13     } else {
14 #ifdef  SA_RESTART
15         act.sa_flags |= SA_RESTART;     /* SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction(signo, &act, &oact) < 0)
19         return ((Sigfunc_rt *) SIG_ERR);
20     return (oact.sa_sigaction);
21 }
```

lib/signal_rt.c

Figure 5.18 `signal_rt` function to provide realtime behavior.

Simplify function prototype using `typedef`

1-3 In our `unpipc.h` header (Figure C.1), we define `Sigfunc_rt` as

```
typedef  void  Sigfunc_rt(int, siginfo_t *, void *);
```

We said earlier in this section that this is the function prototype for a signal handler installed with the `SA_SIGINFO` flag set.

Specify handler function

5-7 The `sigaction` structure changed when realtime signal support was added, with the addition of the new `sa_sigaction` member.

```
struct sigaction {
  void      (*sa_handler)(); /* SIG_DFL, SIG_IGN, or addr of signal handler */
  sigset_t  sa_mask;         /* additional signals to block */
  int       sa_flags;        /* signal options: SA_xxx */
  void      (*sa_sigaction)(int, siginfo_t, void *);
                             /* addr of signal handler if SA_SIGINFO set */
};
```

The rules are:

- If the `SA_SIGINFO` flag is set in the `sa_flags` member, then the `sa_sigaction` member specifies the address of the signal-handling function.
- If the `SA_SIGINFO` flag is not set in the `sa_flags` member, then the `sa_handler` member specifies the address of the signal-handling function.
- To specify the default action for a signal or to ignore a signal, set `sa_handler` to either `SIG_DFL` or `SIG_IGN`, and do not set `SA_SIGINFO`.

Set `SA_SIGINFO`

8-17 We always set the `SA_SIGINFO` flag, and also specify the `SA_RESTART` flag if the signal is not `SIGALRM`.

5.8 Implementation Using Memory-Mapped I/O

We now provide an implementation of Posix message queues using memory-mapped I/O, along with Posix mutexes and condition variables.

We cover mutexes and condition variables in Chapter 7 and memory-mapped I/O in Chapters 12 and 13. You may wish to skip this section until you have read those chapters.

Figure 5.19 shows a layout of the data structures that we use to implement Posix message queues. In this figure, we assume that the message queue was created to hold up to four messages of 7 bytes each.

Figure 5.20 shows our `mqueue.h` header, which defines the fundamental structures for this implementation.

`mqd_t` datatype

1 Our message queue descriptor is just a pointer to an `mq_info` structure. Each call to `mq_open` allocates one of these structures, and the pointer to this structure is what gets returned to the caller. This reiterates that a message queue descriptor need not be a small integer, like a file descriptor—the only Posix requirement is that this datatype cannot be an array type.

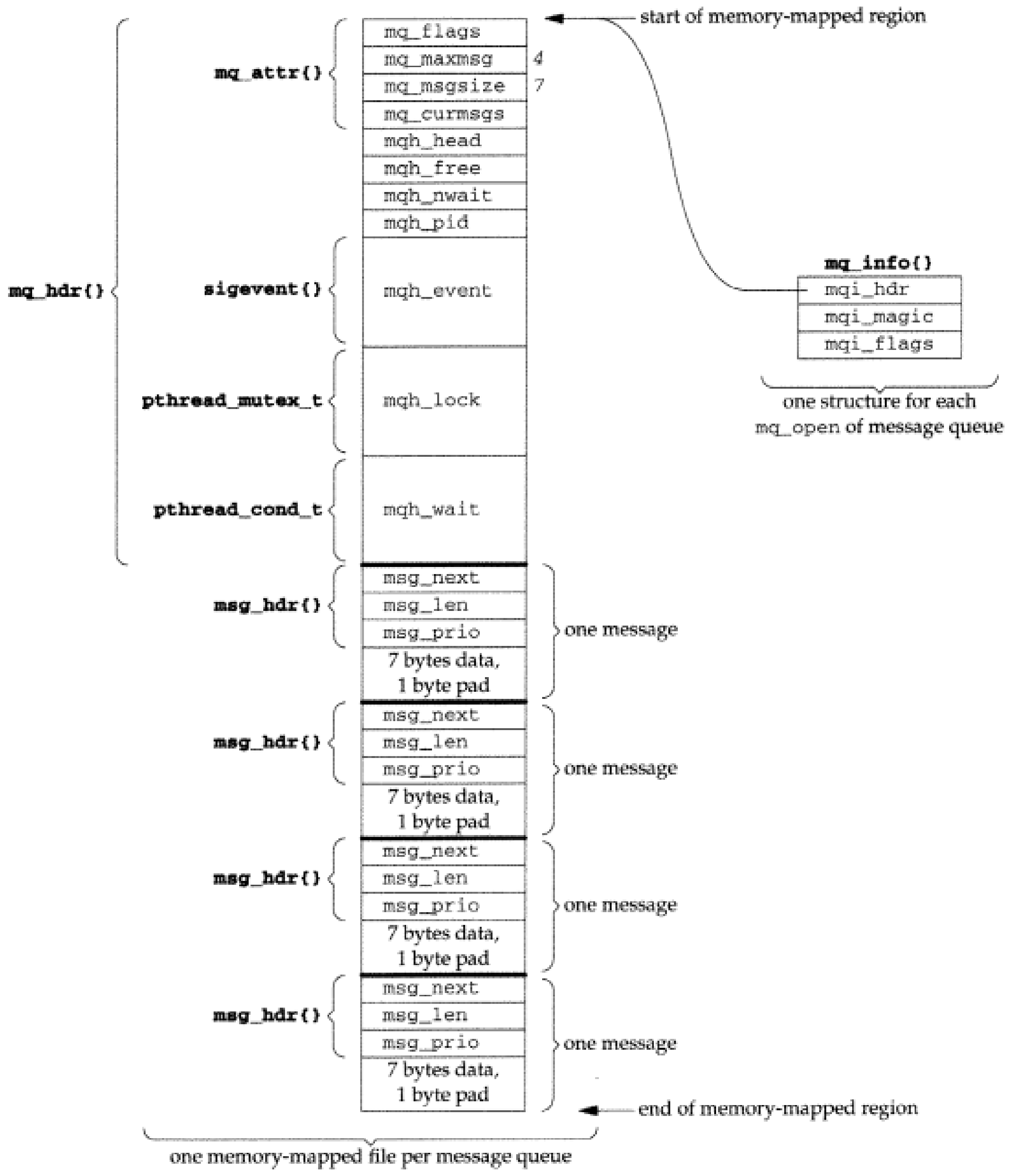

Figure 5.19 Layout of data structures to implement Posix message queues using a memory-mapped file.

my_pxmsg_mmap/mqueue.h

```
 1 typedef struct mq_info *mqd_t;  /* opaque datatype */

 2 struct mq_attr {
 3     long    mq_flags;           /* message queue flag: O_NONBLOCK */
 4     long    mq_maxmsg;          /* max number of messages allowed on queue */
 5     long    mq_msgsize;         /* max size of a message (in bytes) */
 6     long    mq_curmsgs;         /* number of messages currently on queue */
 7 };

 8         /* one mq_hdr{} per queue, at beginning of mapped file */
 9 struct mq_hdr {
10     struct mq_attr mqh_attr;  /* the queue's attributes */
11     long    mqh_head;           /* index of first message */
12     long    mqh_free;           /* index of first free message */
13     long    mqh_nwait;          /* #threads blocked in mq_receive() */
14     pid_t   mqh_pid;            /* nonzero PID if mqh_event set */
15     struct sigevent mqh_event;  /* for mq_notify() */
16     pthread_mutex_t mqh_lock;   /* mutex lock */
17     pthread_cond_t mqh_wait;    /* and condition variable */
18 };

19         /* one msg_hdr{} at the front of each message in the mapped file */
20 struct msg_hdr {
21     long    msg_next;           /* index of next on linked list */
22         /* msg_next must be first member in struct */
23     ssize_t msg_len;            /* actual length */
24     unsigned int msg_prio;      /* priority */
25 };

26         /* one mq_info{} malloc'ed per process per mq_open() */
27 struct mq_info {
28     struct mq_hdr *mqi_hdr;   /* start of mmap'ed region */
29     long    mqi_magic;          /* magic number if open */
30     int     mqi_flags;          /* flags for this process */
31 };
32 #define MQI_MAGIC   0x98765432

33         /* size of message in file is rounded up for alignment */
34 #define MSGSIZE(i)  ((((i) + sizeof(long)-1) / sizeof(long)) * sizeof(long))
```

my_pxmsg_mmap/mqueue.h

Figure 5.20 mqueue.h header.

mq_hdr structure

8-18 This structure appears at the beginning of the mapped file and contains all the per-queue information. The mq_flags member of the mqh_attr structure is not used, because the flags (the nonblocking flag is the only one defined) must be maintained on a per-open basis, not on a per-queue basis. The flags are maintained in the mq_info structure. We describe the remaining members of this structure as we use them in the various functions.

Note now that everything that we refer to as an *index* (the mqh_head and mqh_free members of this structure, and the msg_next member of the next structure) contains byte indexes from the beginning of the mapped file. For example, the size of the `mq_hdr` structure under Solaris 2.6 is 96 bytes, so the index of the first message following this header is 96. Each message in Figure 5.19 occupies 20 bytes (12 bytes for the `msg_hdr` structure and 8 bytes for the message data), so the indexes of the remaining three messages are 116, 136, and 156, and the size of this mapped file is 176 bytes. These indexes are used to maintain two linked lists in the mapped file: one list (`mqh_head`) contains all the messages currently on the queue, and the other (`mqh_free`) contains all the free messages on the queue. We cannot use actual memory pointers (addresses) for these list pointers, because the mapped file can start at different memory addresses in each process that maps the file (as we show in Figure 13.6).

`msg_hdr` structure

19-25 This structure appears at the beginning of each message in the mapped file. All messages are either on the message list or on the free list, and the `msg_next` member contains the index of the next message on the list (or 0 if this message is the end of the list). `msg_len` is the actual length of the message data, which for our example in Figure 5.19 can be between 0 and 7 bytes, inclusive. `msg_prio` is the priority assigned to the message by the caller of `mq_send`.

`mq_info` structure

26-32 One of these structures is dynamically allocated by `mq_open` when a queue is opened, and freed by `mq_close`. `mqi_hdr` points to the mapped file (the starting address returned by `mmap`). A pointer to this structure is the fundamental `mqd_t` datatype of our implementation, and this pointer is the return value from `mq_open`.

The `mqi_magic` member contains `MQI_MAGIC`, once this structure has been initialized, and is checked by each function that is passed an `mqd_t` pointer, to make certain that the pointer really points to an `mq_info` structure. `mqi_flags` contains the nonblocking flag for this open instance of the queue.

`MSGSIZE` macro

33-34 For alignment purposes, we want each message in the mapped file to start on a long integer boundary. Therefore, if the maximum size of each message is not so aligned, we add between 1 and 3 bytes of padding to the data portion of each message, as shown in Figure 5.19. This assumes that the size of a long integer is 4 bytes (which is true for Solaris 2.6), but if the size of a long integer is 8 bytes (as on Digital Unix 4.0), then the amount of padding will be between 1 and 7 bytes.

`mq_open` Function

Figure 5.21 shows the first part of our `mq_open` function, which creates a new message queue or opens an existing message queue.

my_pxmsg_mmap/mq_open.c

```
#include   "unpipc.h"
#include   "mqueue.h"

#include   <stdarg.h>
#define    MAX_TRIES   10       /* for waiting for initialization */

struct mq_attr defattr =
{0, 128, 1024, 0};
```

```
 7 mqd_t
 8 mq_open(const char *pathname, int oflag,...)
 9 {
10     int     i, fd, nonblock, created, save_errno;
11     long    msgsize, filesize, index;
12     va_list ap;
13     mode_t  mode;
14     int8_t *mptr;
15     struct stat statbuff;
16     struct mq_hdr *mqhdr;
17     struct msg_hdr *msghdr;
18     struct mq_attr *attr;
19     struct mq_info *mqinfo;
20     pthread_mutexattr_t mattr;
21     pthread_condattr_t cattr;

22     created = 0;
23     nonblock = oflag & O_NONBLOCK;
24     oflag &= ~O_NONBLOCK;
25     mptr = (int8_t *) MAP_FAILED;
26     mqinfo = NULL;
27   again:
28     if (oflag & O_CREAT) {
29         va_start(ap, oflag);     /* init ap to final named argument */
30         mode = va_arg(ap, va_mode_t) & ~S_IXUSR;
31         attr = va_arg(ap, struct mq_attr *);
32         va_end(ap);

33             /* open and specify O_EXCL and user-execute */
34         fd = open(pathname, oflag | O_EXCL | O_RDWR, mode | S_IXUSR);
35         if (fd < 0) {
36             if (errno == EEXIST && (oflag & O_EXCL) == 0)
37                 goto exists;     /* already exists, OK */
38             else
39                 return ((mqd_t) -1);
40         }
41         created = 1;
42             /* first one to create the file initializes it */
43         if (attr == NULL)
44             attr = &defattr;
45         else {
46             if (attr->mq_maxmsg <= 0 || attr->mq_msgsize <= 0) {
47                 errno = EINVAL;
48                 goto err;
49             }
50         }
```
my_pxmsg_mmap/mq_open.c

Figure 5.21 mq_open function: first part.

Handle variable argument list

29-32 This function can be called with either two or four arguments, depending on whether or not the O_CREAT flag is specified. When this flag is specified, the third argument is of type `mode_t`, but this is a primitive system datatype that can be any type of integer. The problem we encounter is on BSD/OS, which defines this datatype as an `unsigned short` integer (occupying 16 bits). Since an integer on this implementation occupies 32 bits, the C compiler expands an argument of this type from 16 to 32 bits, since all short integers are expanded to integers in the argument list. But if we specify `mode_t` in the call to `va_arg`, it will step past 16 bits of argument on the stack, when the argument has been expanded to occupy 32 bits. Therefore, we must define our own datatype, `va_mode_t`, that is an integer under BSD/OS, or of type `mode_t` under other systems. The following lines in our `unpipc.h` header (Figure C.1) handle this portability problem:

```
#ifdef   __bsdi__
#define  va_mode_t   int
#else
#define  va_mode_t   mode_t
#endif
```

30 We turn off the user-execute bit in the `mode` variable (`S_IXUSR`) for reasons that we describe shortly.

Create a new message queue

33-34 A regular file is created with the name specified by the caller, and the user-execute bit is turned on.

Handle potential race condition

35-40 If we were to just open the file, memory map its contents, and initialize the mapped file (as described shortly) when the `O_CREAT` flag is specified by the caller, we would have a race condition. A message queue is initialized by `mq_open` only if `O_CREAT` is specified by the caller *and* the message queue does not already exist. That means we need some method of detecting whether the message queue already exists. To do so, we always specify `O_EXCL` when we `open` the file that will be memory-mapped. But an error return of `EEXIST` from `open` becomes an error from `mq_open`, only if the caller specified `O_EXCL`. Otherwise, if `open` returns an error of `EEXIST`, the file already exists and we just skip ahead to Figure 5.23 as if the `O_CREAT` flag was not specified.

The possible race condition is because our use of a memory-mapped file to represent a message queue requires two steps to initialize a new message queue: first, the file must be created by `open`, and second, the contents of the file (described shortly) must be initialized. The problem occurs if two threads (in the same or different processes) call `mq_open` at about the same time. One thread can create the file, and then the system switches to the second thread before the first thread completes the initialization. This second thread detects that the file already exists (using the `O_EXCL` flag to `open`) and immediately tries to use the message queue. But the message queue cannot be used until the first thread initializes the message queue. We use the user-execute bit of the file to indicate that the message queue has been initialized. This bit is enabled only by the thread that actually creates the file (using the `O_EXCL` flag to detect which thread creates the file), and that thread initializes the message queue and then turns off the user-execute bit. We encounter similar race conditions in Figures 10.43 and 10.52.

Check attributes

42-50 If the caller specifies a null pointer for the final argument, we use the default attributes shown at the beginning of this figure: 128 messages and 1024 bytes per message. If the caller specifies the attributes, we verify that `mq_maxmsg` and `mq_msgsize` are positive.

The second part of our `mq_open` function is shown in Figure 5.22; it completes the initialization of a new queue.

my_pxmsg_mmap/mq_open.c

```
        /* calculate and set the file size */
    msgsize = MSGSIZE(attr->mq_msgsize);
    filesize = sizeof(struct mq_hdr) + (attr->mq_maxmsg *
                             (sizeof(struct msg_hdr) + msgsize));
    if (lseek(fd, filesize - 1, SEEK_SET) == -1)
        goto err;
    if (write(fd, "", 1) == -1)
        goto err;

        /* memory map the file */
    mptr = mmap(NULL, filesize, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, 0);
    if (mptr == MAP_FAILED)
        goto err;

        /* allocate one mq_info{} for the queue */
    if ( (mqinfo = malloc(sizeof(struct mq_info))) == NULL)
        goto err;

    mqinfo->mqi_hdr = mqhdr = (struct mq_hdr *) mptr;
    mqinfo->mqi_magic = MQI_MAGIC;
    mqinfo->mqi_flags = nonblock;

        /* initialize header at beginning of file */
        /* create free list with all messages on it */
    mqhdr->mqh_attr.mq_flags = 0;
    mqhdr->mqh_attr.mq_maxmsg = attr->mq_maxmsg;
    mqhdr->mqh_attr.mq_msgsize = attr->mq_msgsize;
    mqhdr->mqh_attr.mq_curmsgs = 0;
    mqhdr->mqh_nwait = 0;
    mqhdr->mqh_pid = 0;
    mqhdr->mqh_head = 0;
    index = sizeof(struct mq_hdr);
    mqhdr->mqh_free = index;
    for (i = 0; i < attr->mq_maxmsg - 1; i++) {
        msghdr = (struct msg_hdr *) &mptr[index];
        index += sizeof(struct msg_hdr) + msgsize;
        msghdr->msg_next = index;
    }
    msghdr = (struct msg_hdr *) &mptr[index];
    msghdr->msg_next = 0;    /* end of free list */

        /* initialize mutex & condition variable */
    if ( (i = pthread_mutexattr_init(&mattr)) != 0)
        goto pthreaderr;
```

```
 91         pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
 92         i = pthread_mutex_init(&mqhdr->mqh_lock, &mattr);
 93         pthread_mutexattr_destroy(&mattr);  /* be sure to destroy */
 94         if (i != 0)
 95             goto pthreaderr;

 96         if ( (i = pthread_condattr_init(&cattr)) != 0)
 97             goto pthreaderr;
 98         pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);
 99         i = pthread_cond_init(&mqhdr->mqh_wait, &cattr);
100         pthread_condattr_destroy(&cattr);   /* be sure to destroy */
101         if (i != 0)
102             goto pthreaderr;

103             /* initialization complete, turn off user-execute bit */
104         if (fchmod(fd, mode) == -1)
105             goto err;
106         close(fd);
107         return ((mqd_t) mqinfo);
108     }
```
my_pxmsg_mmap/mq_open.c

Figure 5.22 Second part of mq_open function: complete initialization of new queue.

Set the file size

51-58 We calculate the size of each message, rounding up to the next multiple of the size of a long integer. To calculate the file size, we also allocate room for the mq_hdr structure at the beginning of the file and the msg_hdr structure at the beginning of each message (Figure 5.19). We set the size of the newly created file using lseek and then writing one byte of 0. Just calling ftruncate (Section 13.3) would be easier, but we are not guaranteed that this works to increase the size of a file.

Memory map the file

59-63 The file is memory mapped by mmap.

Allocate mq_info structure

64-66 We allocate one mq_info structure for each call to mq_open. This structure is initialized.

Initialize mq_hdr structure

67-87 We initialize the mq_hdr structure. The head of the linked list of messages (mqh_head) is set to 0, and all the messages in the queue are added to the free list (mqh_free).

Initialize mutex and condition variable

88-102 Since Posix message queues can be shared by any process that knows the message queue's name and has adequate permission, we must initialize the mutex and condition variable with the PTHREAD_PROCESS_SHARED attribute. To do so for the message queue, we first initialize the attributes by calling pthread_mutexattr_init, then call pthread_mutexattr_setpshared to set the process-shared attribute in this structure, and then initialize the mutex by calling pthread_mutex_init. Nearly identical steps are done for the condition variable. We are careful to destroy the mutex or condition variable attributes that are initialized, even if an error occurs, because the calls to `pthread_mutexattr_init` or `pthread_condattr_init` might allocate memory (Exercise 7.3).

Turn off user-execute bit

103-107 Once the message queue is initialized, we turn off the user-execute bit. This indicates that the message queue has been initialized. We also `close` the file, since it has been memory mapped and there is no need to keep it open (taking up a descriptor).

Figure 5.23 shows the final part of our `mq_open` function, which opens an existing queue.

my_pxmsg_mmap/mq_open.c

```
exists:
      /* open the file then memory map */
  if ( (fd = open(pathname, O_RDWR)) < 0) {
      if (errno == ENOENT && (oflag & O_CREAT))
          goto again;
      goto err;
  }
      /* make certain initialization is complete */
  for (i = 0; i < MAX_TRIES; i++) {
      if (stat(pathname, &statbuff) == -1) {
          if (errno == ENOENT && (oflag & O_CREAT)) {
              close(fd);
              goto again;
          }
          goto err;
      }
      if ((statbuff.st_mode & S_IXUSR) == 0)
          break;
      sleep(1);
  }
  if (i == MAX_TRIES) {
      errno = ETIMEDOUT;
      goto err;
  }
  filesize = statbuff.st_size;
  mptr = mmap(NULL, filesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if (mptr == MAP_FAILED)
      goto err;
  close(fd);

      /* allocate one mq_info{} for each open */
  if ( (mqinfo = malloc(sizeof(struct mq_info))) == NULL)
      goto err;

  mqinfo->mqi_hdr = (struct mq_hdr *) mptr;
  mqinfo->mqi_magic = MQI_MAGIC;
  mqinfo->mqi_flags = nonblock;
  return ((mqd_t) mqinfo);
```

```
pthreaderr:
  errno = i;
err:
      /* don't let following function calls change errno */
  save_errno = errno;
  if (created)
      unlink(pathname);
  if (mptr != MAP_FAILED)
      munmap(mptr, filesize);
  if (mqinfo != NULL)
      free(mqinfo);
  close(fd);
  errno = save_errno;
  return ((mqd_t) -1);
}
```
my_pxmsg_mmap/mq_open.c

Figure 5.23 Third part of mq_open function: open an existing queue.

Open existing message queue

109-115 We end up here if either the O_CREAT flag is not specified or if O_CREAT is specified but the message queue already exists. In either case, we are opening an existing message queue. We open the file containing the message queue for reading and writing and memory map the file into the address space of the process (mmap).

> Our implementation is simplistic with regard to the open mode. Even if the caller specifies O_RDONLY, we must specify read–write access to both open and mmap, because we cannot read a message from a queue without changing the file. Similarly, we cannot write a message to a queue without reading the file. One way around this problem is to save the open mode (O_RDONLY, O_WRONLY, or O_RDWR) in the mq_info structure and then check this mode in the individual functions. For example, mq_receive should fail if the open mode was O_WRONLY.

Make certain that message queue is initialized

116-132 We must wait for the message queue to be initialized (in case multiple threads try to create the same message queue at about the same time). To do so, we call stat and look at the file's permissions (the st_mode member of the stat structure). If the user-execute bit is off, the message queue has been initialized.

This piece of code handles another possible race condition. Assume that two threads in different processes open the same message queue at about the same time. The first thread creates the file and then blocks in its call to lseek in Figure 5.22. The second thread finds that the file already exists and branches to exists where it opens the file again, and then blocks. The first thread runs again, but its call to mmap in Figure 5.22 fails (perhaps it has exceeded its virtual memory limit), so it branches to err and unlinks the file that it created. The second thread continues, but if we called fstat instead of stat, the second thread could time out in the for loop waiting for the file to be initialized. Instead, we call stat, and if it returns an error that the file does not exist and if the O_CREAT flag was specified, we branch to again (Figure 5.21) to create the file again. This possible race condition is why we also check for an error of ENOENT in the call to open.

Memory map file; allocate and initialize `mq_info` structure

133-144 The file is memory mapped, and the descriptor can then be closed. We allocate an `mq_info` structure and initialize it. The return value is a pointer to the `mq_info` structure that was allocated.

Handle errors

145-158 When an error is detected earlier in the function, the label `err` is branched to, with `errno` set to the value to be returned by `mq_open`. We are careful that the functions called to clean up after the error is detected do not affect the `errno` returned by this function.

`mq_close` Function

Figure 5.24 shows our `mq_close` function.

my_pxmsg_mmap/mq_close.c

```
#include    "unpipc.h"
#include    "mqueue.h"

int
mq_close(mqd_t mqd)
{
    long    msgsize, filesize;
    struct mq_hdr *mqhdr;
    struct mq_attr *attr;
    struct mq_info *mqinfo;

    mqinfo = mqd;
    if (mqinfo->mqi_magic != MQI_MAGIC) {
        errno = EBADF;
        return (-1);
    }
    mqhdr = mqinfo->mqi_hdr;
    attr = &mqhdr->mqh_attr;

    if (mq_notify(mqd, NULL) != 0)     /* unregister calling process */
        return (-1);

    msgsize = MSGSIZE(attr->mq_msgsize);
    filesize = sizeof(struct mq_hdr) + (attr->mq_maxmsg *
                                    (sizeof(struct msg_hdr) + msgsize));
    if (munmap(mqinfo->mqi_hdr, filesize) == -1)
        return (-1);

    mqinfo->mqi_magic = 0;      /* just in case */
    free(mqinfo);
    return (0);
}
```

my_pxmsg_mmap/mq_close.c

Figure 5.24 `mq_close` function.

Get pointers to structures

10-16 The argument is validated, and pointers are then obtained to the memory-mapped region (`mqhdr`) and the attributes (in the `mq_hdr` structure).

Unregister calling process

17-18 We call `mq_notify` to unregister the calling process for this queue. If the process is registered, it will be unregistered, but if it is not registered, no error is returned.

Unmap region and free memory

19-25 We calculate the size of the file for `munmap` and then free the memory used by the `mq_info` structure. Just in case the caller continues to use the message queue descriptor before that region of memory is reused by `malloc`, we set the magic number to 0, so that our message queue functions will detect the error.

Note that if the process terminates without calling `mq_close`, the same operations take place on process termination: the memory-mapped file is unmapped and the memory is freed.

`mq_unlink` Function

Our `mq_unlink` function shown in Figure 5.25 removes the name associated with our message queue. It just calls the Unix `unlink` function.

my_pxmsg_mmap/mq_unlink.c

```
#include    "unpipc.h"
#include    "mqueue.h"

int
mq_unlink(const char *pathname)
{
    if (unlink(pathname) == -1)
        return (-1);
    return (0);
}
```

my_pxmsg_mmap/mq_unlink.c

Figure 5.25 `mq_unlink` function.

`mq_getattr` Function

Figure 5.26 shows our `mq_getattr` function, which returns the current attributes of the specified queue.

Acquire queue's mutex lock

17-20 We must acquire the message queue's mutex lock before fetching the attributes, in case some other thread is in the middle of changing them.

my_pxmsg_mmap/mq_getattr.c

```
#include    "unpipc.h"
#include    "mqueue.h"

int
mq_getattr(mqd_t mqd, struct mq_attr *mqstat)
{
    int     n;
    struct mq_hdr *mqhdr;
    struct mq_attr *attr;
    struct mq_info *mqinfo;

    mqinfo = mqd;
    if (mqinfo->mqi_magic != MQI_MAGIC) {
        errno = EBADF;
        return (-1);
    }
    mqhdr = mqinfo->mqi_hdr;
    attr = &mqhdr->mqh_attr;
    if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
        errno = n;
        return (-1);
    }
    mqstat->mq_flags = mqinfo->mqi_flags;   /* per-open */
    mqstat->mq_maxmsg = attr->mq_maxmsg;    /* remaining three per-queue */
    mqstat->mq_msgsize = attr->mq_msgsize;
    mqstat->mq_curmsgs = attr->mq_curmsgs;

    pthread_mutex_unlock(&mqhdr->mqh_lock);
    return (0);
}
```

my_pxmsg_mmap/mq_getattr.c

Figure 5.26 `mq_getattr` function.

`mq_setattr` Function

Figure 5.27 shows our `mq_setattr` function, which sets the current attributes of the specified queue.

Return current attributes

22-27 If the third argument is a nonnull pointer, we return the previous attributes and current status before changing anything.

Change `mq_flags`

28-31 The only attribute that can be changed with this function is `mq_flags`, which we store in the `mq_info` structure.

my_pxmsg_mmap/mq_setattr.c

```
#include    "unpipc.h"
#include    "mqueue.h"

int
mq_setattr(mqd_t mqd, const struct mq_attr *mqstat,
            struct mq_attr *omqstat)
{
    int     n;
    struct mq_hdr *mqhdr;
    struct mq_attr *attr;
    struct mq_info *mqinfo;

    mqinfo = mqd;
    if (mqinfo->mqi_magic != MQI_MAGIC) {
        errno = EBADF;
        return (-1);
    }
    mqhdr = mqinfo->mqi_hdr;
    attr = &mqhdr->mqh_attr;
    if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
        errno = n;
        return (-1);
    }
    if (omqstat != NULL) {
        omqstat->mq_flags = mqinfo->mqi_flags;  /* previous attributes */
        omqstat->mq_maxmsg = attr->mq_maxmsg;
        omqstat->mq_msgsize = attr->mq_msgsize;
        omqstat->mq_curmsgs = attr->mq_curmsgs;     /* and current status */
    }
    if (mqstat->mq_flags & O_NONBLOCK)
        mqinfo->mqi_flags |= O_NONBLOCK;
    else
        mqinfo->mqi_flags &= ~O_NONBLOCK;

    pthread_mutex_unlock(&mqhdr->mqh_lock);
    return (0);
}
```

my_pxmsg_mmap/mq_setattr.c

Figure 5.27 `mq_setattr` function.

`mq_notify` Function

The `mq_notify` function shown in Figure 5.28 registers or unregisters the calling process for the queue. We keep track of the process currently registered for a queue by storing its process ID in the `mqh_pid` member of the `mq_hdr` structure. Only one process at a time can be registered for a given queue. When a process registers itself, we also save its specified `sigevent` structure in the `mqh_event` structure.

my_pxmsg_mmap/mq_notify.c

```
 1 #include    "unpipc.h"
 2 #include    "mqueue.h"

 3 int
 4 mq_notify(mqd_t mqd, const struct sigevent *notification)
 5 {
 6     int     n;
 7     pid_t   pid;
 8     struct mq_hdr *mqhdr;
 9     struct mq_info *mqinfo;

10     mqinfo = mqd;
11     if (mqinfo->mqi_magic != MQI_MAGIC) {
12         errno = EBADF;
13         return (-1);
14     }
15     mqhdr = mqinfo->mqi_hdr;
16     if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
17         errno = n;
18         return (-1);
19     }
20     pid = getpid();
21     if (notification == NULL) {
22         if (mqhdr->mqh_pid == pid) {
23             mqhdr->mqh_pid = 0; /* unregister calling process */
24         }                       /* no error if caller not registered */
25     } else {
26         if (mqhdr->mqh_pid != 0) {
27             if (kill(mqhdr->mqh_pid, 0) != -1 || errno != ESRCH) {
28                 errno = EBUSY;
29                 goto err;
30             }
31         }
32         mqhdr->mqh_pid = pid;
33         mqhdr->mqh_event = *notification;
34     }
35     pthread_mutex_unlock(&mqhdr->mqh_lock);
36     return (0);

37   err:
38     pthread_mutex_unlock(&mqhdr->mqh_lock);
39     return (-1);
40 }
```

my_pxmsg_mmap/mq_notify.c

Figure 5.28 `mq_notify` function.

Unregister calling process

20-24 If the second argument is a null pointer, the calling process is unregistered for this queue. Strangely, no error is specified if the calling process is not registered for this queue.

Register calling process

25-34 If some process is already registered, we check whether it still exists by sending it signal 0 (called the *null signal*). This performs the normal error checking, but does not send a signal and returns an error of ESRCH if the process does not exist. An error of EBUSY is returned if the previously registered process still exists. Otherwise, the process ID is saved, along with the caller's sigevent structure.

Our test for whether the previously registered process exists is not perfect. This process can terminate and then have its process ID reused at some later time.

mq_send Function

Figure 5.29 shows the first half of our mq_send function.

Initialize

14-29 Pointers are obtained to the structures that we will use, and the mutex lock for the queue is obtained. A check is made that the size of the message does not exceed the maximum message size for this queue.

Check for empty queue and send notification if applicable

30-38 If we are placing a message onto an empty queue, we check whether any process is registered for this queue *and* whether any thread is blocked in a call to mq_receive. For the latter check, we will see that our mq_receive function keeps a count (mqh_nwait) of the number of threads blocked on the empty queue. If this counter is nonzero, we do not send any notification to the registered process. We handle a notification of SIGEV_SIGNAL and call sigqueue to send the signal. The registered process is then unregistered.

Calling sigqueue to send the signal results in an si_code of SI_QUEUE being passed to the signal handler in the siginfo_t structure (Section 5.7), which is incorrect. Generating the correct si_code of SI_MESGQ from a user process is implementation dependent. Page 433 of [IEEE 1996] mentions that a hidden interface into the signal generation mechanism is required to generate this signal from a user library.

Check for full queue

39-48 If the queue is full but the O_NONBLOCK flag has been set, we return an error of EAGAIN. Otherwise, we wait on the condition variable mqh_wait, which we will see is signaled by our mq_receive function when a message is read from a full queue.

Our implementation is simplistic with regard to returning an error of EINTR if this call to mq_send is interrupted by a signal that is caught by the calling process. The problem is that pthread_cond_wait does not return an error when the signal handler returns: it can either return a value of 0 (which appears as a spurious wakeup) or it need not return at all. Ways around this exist, all nontrivial.

Figure 5.30 shows the second half of our mq_send function. At this point, we know the queue has room for the new message.

my_pxmsg_mmap/mq_send.c

```
#include    "unpipc.h"
#include    "mqueue.h"

int
mq_send(mqd_t mqd, const char *ptr, size_t len, unsigned int prio)
{
    int     n;
    long    index, freeindex;
    int8_t *mptr;
    struct sigevent *sigev;
    struct mq_hdr *mqhdr;
    struct mq_attr *attr;
    struct msg_hdr *msghdr, *nmsghdr, *pmsghdr;
    struct mq_info *mqinfo;

    mqinfo = mqd;
    if (mqinfo->mqi_magic != MQI_MAGIC) {
        errno = EBADF;
        return (-1);
    }
    mqhdr = mqinfo->mqi_hdr;    /* struct pointer */
    mptr = (int8_t *) mqhdr;    /* byte pointer */
    attr = &mqhdr->mqh_attr;
    if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
        errno = n;
        return (-1);
    }
    if (len > attr->mq_msgsize) {
        errno = EMSGSIZE;
        goto err;
    }
    if (attr->mq_curmsgs == 0) {
        if (mqhdr->mqh_pid != 0 && mqhdr->mqh_nwait == 0) {
            sigev = &mqhdr->mqh_event;
            if (sigev->sigev_notify == SIGEV_SIGNAL) {
                sigqueue(mqhdr->mqh_pid, sigev->sigev_signo,
                         sigev->sigev_value);
            }
            mqhdr->mqh_pid = 0; /* unregister */
        }
    } else if (attr->mq_curmsgs >= attr->mq_maxmsg) {
            /* queue is full */
        if (mqinfo->mqi_flags & O_NONBLOCK) {
            errno = EAGAIN;
            goto err;
        }
            /* wait for room for one message on the queue */
        while (attr->mq_curmsgs >= attr->mq_maxmsg)
            pthread_cond_wait(&mqhdr->mqh_wait, &mqhdr->mqh_lock);
    }
```

my_pxmsg_mmap/mq_send.c

Figure 5.29 mq_send function: first half.

my_pxmsg_mmap/mq_send.c

```
49         /* nmsghdr will point to new message */
50     if ( (freeindex = mqhdr->mqh_free) == 0)
51         err_dump("mq_send: curmsgs = %ld; free = 0", attr->mq_curmsgs);
52     nmsghdr = (struct msg_hdr *) &mptr[freeindex];
53     nmsghdr->msg_prio = prio;
54     nmsghdr->msg_len = len;
55     memcpy(nmsghdr + 1, ptr, len);  /* copy message from caller */
56     mqhdr->mqh_free = nmsghdr->msg_next;    /* new freelist head */

57         /* find right place for message in linked list */
58     index = mqhdr->mqh_head;
59     pmsghdr = (struct msg_hdr *) &(mqhdr->mqh_head);
60     while (index != 0) {
61         msghdr = (struct msg_hdr *) &mptr[index];
62         if (prio > msghdr->msg_prio) {
63             nmsghdr->msg_next = index;
64             pmsghdr->msg_next = freeindex;
65             break;
66         }
67         index = msghdr->msg_next;
68         pmsghdr = msghdr;
69     }
70     if (index == 0) {
71             /* queue was empty or new goes at end of list */
72         pmsghdr->msg_next = freeindex;
73         nmsghdr->msg_next = 0;
74     }
75         /* wake up anyone blocked in mq_receive waiting for a message */
76     if (attr->mq_curmsgs == 0)
77         pthread_cond_signal(&mqhdr->mqh_wait);
78     attr->mq_curmsgs++;

79     pthread_mutex_unlock(&mqhdr->mqh_lock);
80     return (0);

81   err:
82     pthread_mutex_unlock(&mqhdr->mqh_lock);
83     return (-1);
84 }
```

my_pxmsg_mmap/mq_send.c

Figure 5.30 `mq_send` function: second half.

Get index of free block to use

50-52 Since the number of free messages created when the queue was initialized equals `mq_maxmsg`, we should never have a situation where `mq_curmsgs` is less than `mq_maxmsg` with an empty free list.

Copy message

53-56 `nmsghdr` contains the address in the mapped memory of where the message is stored. The priority and length are stored in its `msg_hdr` structure, and then the contents of the message are copied from the caller.

Place new message onto linked list in correct location

57-74 The order of messages on our linked list is from highest priority at the front (`mqh_head`) to lowest priority at the end. When a new message is added to the queue and one or more messages of the same priority are already on the queue, the new message is added after the last message with its priority. Using this ordering, `mq_receive` always returns the first message on the linked list (which is the oldest message of the highest priority on the queue). As we step through the linked list, `pmsghdr` contains the address of the previous message in the list, because its `msg_next` value will contain the index of the new message.

> Our design can be slow when lots of messages are on the queue, forcing a traversal of a large number of list entries each time a message is written to the queue. A separate index could be maintained that remembers the location of the last message for each possible priority.

Wake up anyone blocked in `mq_receive`

75-77 If the queue was empty before we placed the message onto the queue, we call `pthread_cond_signal` to wake up any thread that might be blocked in `mq_receive`.

78 The number of messages currently on the queue, `mq_curmsgs`, is incremented.

`mq_receive` Function

Figure 5.31 shows the first half of our `mq_receive` function, which sets up the pointers that it needs, obtains the mutex lock, and verifies that the caller's buffer is large enough for the largest possible message.

Check for empty queue

30-40 If the queue is empty and the `O_NONBLOCK` flag is set, an error of `EAGAIN` is returned. Otherwise, we increment the queue's `mqh_nwait` counter, which was examined by our `mq_send` function in Figure 5.29, if the queue was empty and someone was registered for notification. We then wait on the condition variable, which is signaled by `mq_send` in Figure 5.29.

> As with our implementation of `mq_send`, our implementation of `mq_receive` is simplistic with regard to returning an error of `EINTR` if this call is interrupted by a signal that is caught by the calling process.

Figure 5.32 shows the second half of our `mq_receive` function. At this point, we know that a message is on the queue to return to the caller.

my_pxmsg_mmap/mq_receive.c

```c
#include    "unpipc.h"
#include    "mqueue.h"

ssize_t
mq_receive(mqd_t mqd, char *ptr, size_t maxlen, unsigned int *priop)
{
    int     n;
    long    index;
    int8_t *mptr;
    ssize_t len;
    struct mq_hdr *mqhdr;
    struct mq_attr *attr;
    struct msg_hdr *msghdr;
    struct mq_info *mqinfo;

    mqinfo = mqd;
    if (mqinfo->mqi_magic != MQI_MAGIC) {
        errno = EBADF;
        return (-1);
    }
    mqhdr = mqinfo->mqi_hdr;    /* struct pointer */
    mptr = (int8_t *) mqhdr;    /* byte pointer */
    attr = &mqhdr->mqh_attr;
    if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
        errno = n;
        return (-1);
    }
    if (maxlen < attr->mq_msgsize) {
        errno = EMSGSIZE;
        goto err;
    }
    if (attr->mq_curmsgs == 0) {    /* queue is empty */
        if (mqinfo->mqi_flags & O_NONBLOCK) {
            errno = EAGAIN;
            goto err;
        }
            /* wait for a message to be placed onto queue */
        mqhdr->mqh_nwait++;
        while (attr->mq_curmsgs == 0)
            pthread_cond_wait(&mqhdr->mqh_wait, &mqhdr->mqh_lock);
        mqhdr->mqh_nwait--;
    }
```

my_pxmsg_mmap/mq_receive.c

Figure 5.31 mq_receive function: first half.

my_pxmsg_mmap/mq_receive.c

```
41      if ( (index = mqhdr->mqh_head) == 0)
42          err_dump("mq_receive: curmsgs = %ld; head = 0", attr->mq_curmsgs);

43      msghdr = (struct msg_hdr *) &mptr[index];
44      mqhdr->mqh_head = msghdr->msg_next;     /* new head of list */
45      len = msghdr->msg_len;
46      memcpy(ptr, msghdr + 1, len);   /* copy the message itself */
47      if (priop != NULL)
48          *priop = msghdr->msg_prio;

49          /* just-read message goes to front of free list */
50      msghdr->msg_next = mqhdr->mqh_free;
51      mqhdr->mqh_free = index;

52          /* wake up anyone blocked in mq_send waiting for room */
53      if (attr->mq_curmsgs == attr->mq_maxmsg)
54          pthread_cond_signal(&mqhdr->mqh_wait);
55      attr->mq_curmsgs--;

56      pthread_mutex_unlock(&mqhdr->mqh_lock);
57      return (len);

58    err:
59      pthread_mutex_unlock(&mqhdr->mqh_lock);
60      return (-1);
61  }
```

my_pxmsg_mmap/mq_receive.c

Figure 5.32 `mq_receive` function: second half.

Return message to caller

43-51 `msghdr` points to the `msg_hdr` of the first message on the queue, which is what we return. The space occupied by this message becomes the new head of the free list.

Wake up anyone blocked in `mq_send`

52-54 If the queue was full before we took the message off the queue, we call `pthread_cond_signal`, in case anyone is blocked in `mq_send` waiting for room for a message.

5.9 Summary

Posix message queues are simple: a new queue is created or an existing queue is opened by `mq_open`; queues are closed by `mq_close`, and the queue names are removed by `mq_unlink`. Messages are placed onto a queue with `mq_send` and read with `mq_receive`. Attributes of the queue can be queried and set with `mq_getattr` and `mq_setattr`, and the function `mq_notify` lets us register a signal to be sent, or a thread to be invoked, when a message is placed onto an empty queue. Small integer priorities are assigned to each message on the queue, and `mq_receive` always returns the oldest message of the highest priority each time it is called.

Using mq_notify introduced us to the Posix realtime signals, named SIGRTMIN through SIGRTMAX. When the signal handler for these signals is installed with the SA_SIGINFO flag set, (1) these signals are queued, (2) the queued signals are delivered in a FIFO order, and (3) two additional arguments are passed to the signal handler.

Finally, we implemented most of the Posix message queue features in about 500 lines of C code, using memory-mapped I/O, along with a Posix mutex and a Posix condition variable. This implementation showed a race condition dealing with the creation of a new queue; we will encounter this same race condition in Chapter 10 when implementing Posix semaphores.

Exercises

5.1 With Figure 5.5, we said that if the *attr* argument to mq_open is nonnull when a new queue is created, both of the members mq_maxmsg and mq_msgsize must be specified. How could we allow either of these to be specified, instead of requiring both, with the one not specified assuming the system's default value?

5.2 Modify Figure 5.9 so that it does not call mq_notify when the signal is delivered. Then send two messages to the queue and verify that the signal is not generated for the second message. Why?

5.3 Modify Figure 5.9 so that it does not read the message from the queue when the signal is delivered. Instead, just call mq_notify and print that the signal was received. Then send two messages to the queue and verify that the signal is not generated for the second message. Why?

5.4 What happens if we remove the cast to an integer for the two constants in the first printf in Figure 5.17?

5.5 Modify Figure 5.5 as follows: before calling mq_open, print a message and sleep for 30 seconds. After mq_open returns, print another message, sleep for 30 seconds, and then call mq_close. Compile and run the program, specifying a large number of messages (a few hundred thousand) and a maximum message size of (say) 10 bytes. The goal is to create a large message queue (megabytes) and then see whether the implementation uses memory-mapped files. During the first 30-second pause, run a program such as ps and look at the memory size of the program. Do this again, after mq_open has returned. Can you explain what happens?

5.6 What happens in the call to memcpy in Figure 5.30 when the caller of mq_send specifies a length of 0?

5.7 Compare a message queue to the full-duplex pipes that we described in Section 4.4. How many message queues are needed for two-way communication between a parent and child?

5.8 In Figure 5.24, why don't we destroy the mutex and condition variable?

5.9 Posix says that a message queue descriptor cannot be an array type. Why?

5.10 Where does the main function in Figure 5.14 spend most of its time? What happens every time a signal is delivered? How do we handle this scenario?

5.11 Not all implementations support the `PTHREAD_PROCESS_SHARED` attributes for mutexes and condition variables. Redo the implementation of Posix message queues in Section 5.8 to use Posix semaphores (Chapter 10) instead of mutexes and condition variables.

5.12 Extend the implementation of Posix message queues in Section 5.8 to support `SIGEV_THREAD`.

6

System V Message Queues

6.1 Introduction

System V message queues are identified by a *message queue identifier*. Any process with adequate privileges (Section 3.5) can place a message onto a given queue, and any process with adequate privileges can read a message from a given queue. As with Posix message queues, there is no requirement that some process be waiting for a message to arrive on a queue before some process writes a message to that queue.

For every message queue in the system, the kernel maintains the following structure of information, defined by including `<sys/msg.h>`:

```
struct msqid_ds {
  struct ipc_perm  msg_perm;   /* read-write perms: Section 3.3 */
  struct msg      *msg_first;  /* ptr to first message on queue */
  struct msg      *msg_last;   /* ptr to last message on queue */
  msglen_t         msg_cbytes; /* current # bytes on queue */
  msgqnum_t        msg_qnum;   /* current # of messages on queue */
  msglen_t         msg_qbytes; /* max # of bytes allowed on queue */
  pid_t            msg_lspid;  /* pid of last msgsnd() */
  pid_t            msg_lrpid;  /* pid of last msgrcv() */
  time_t           msg_stime;  /* time of last msgsnd() */
  time_t           msg_rtime;  /* time of last msgrcv() */
  time_t           msg_ctime;  /* time of last msgctl()
                                  (that changed the above) */
};
```

Unix 98 does not require the `msg_first`, `msg_last`, or `msg_cbytes` members. Nevertheless, these three members are found in the common System V derived implementations. Naturally, no requirement exists that the messages on a queue be maintained as a linked list, as implied by the `msg_first` and `msg_last` members. If these two pointers are present, they point to kernel memory and are largely useless to an application.

We can picture a particular message queue in the kernel as a linked list of messages, as shown in Figure 6.1. Assume that three messages are on a queue, with lengths of 1 byte, 2 bytes, and 3 bytes, and that the messages were written in that order. Also assume that these three messages were written with *types* of 100, 200, and 300, respectively.

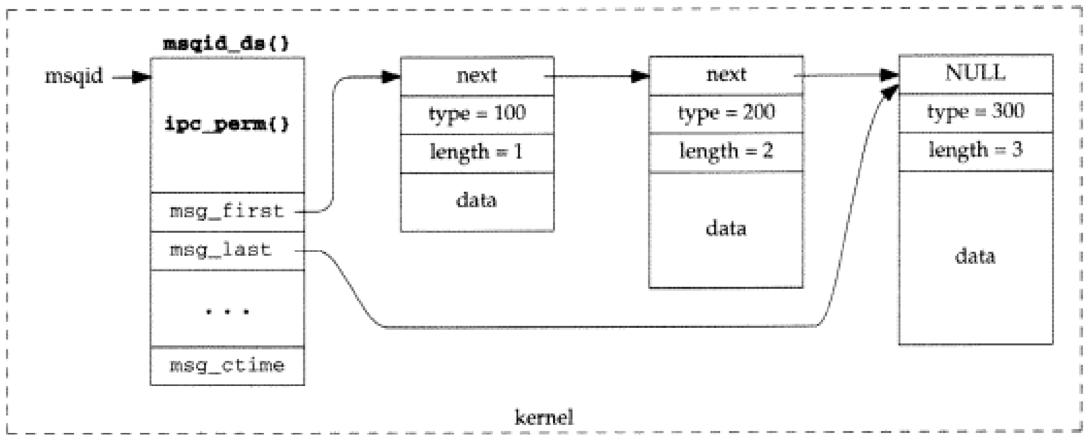

Figure 6.1 System V message queue structures in kernel.

In this chapter, we look at the functions for manipulating System V message queues and implement our file server example from Section 4.2 using message queues.

6.2 `msgget` Function

A new message queue is created, or an existing message queue is accessed with the `msgget` function.

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

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

Returns: nonnegative identifier if OK, −1 on error

The return value is an integer identifier that is used in the other three `msg` functions to refer to this queue, based on the specified *key*, which can be a value returned by `ftok` or the constant `IPC_PRIVATE`, as shown in Figure 3.3.

oflag is a combination of the read–write permission values shown in Figure 3.6. This can be bitwise-ORed with either `IPC_CREAT` or `IPC_CREAT | IPC_EXCL`, as discussed with Figure 3.4.

When a new message queue is created, the following members of the `msqid_ds` structure are initialized:

- The uid and cuid members of the msg_perm structure are set to the effective user ID of the process, and the gid and cgid members are set to the effective group ID of the process.
- The read–write permission bits in *oflag* are stored in msg_perm.mode.
- msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are set to 0.
- msg_ctime is set to the current time.
- msg_qbytes is set to the system limit.

6.3 msgsnd Function

Once a message queue is opened by msgget, we put a message onto the queue using msgsnd.

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

int msgsnd(int msqid, const void *ptr, size_t length, int flag);
```

Returns: 0 if OK, –1 on error

msqid is an identifier returned by msgget. *ptr* is a pointer to a structure with the following template, which is defined in <sys/msg.h>.

```
struct msgbuf {
  long  mtype;      /* message type, must be > 0 */
  char  mtext[1];   /* message data */
};
```

The message type must be greater than 0, since nonpositive message types are used as a special indicator to the msgrcv function, which we describe in the next section.

The name mtext in the msgbuf structure definition is a misnomer; the data portion of the message is not restricted to text. Any form of data is allowed, binary data or text. The kernel does not interpret the contents of the message data at all.

We use the term "template" to describe this structure, because what *ptr* points to is just a long integer containing the message type, immediately followed by the message itself (if the length of the message is greater than 0 bytes). But most applications do not use this definition of the msgbuf structure, since the amount of data (1 byte) is normally inadequate. No compile-time limit exists to the amount of data in a message (this limit can often be changed by the system administrator), so rather than declare a structure with a huge amount of data (more data than a given implementation may support), this template is defined instead. Most applications then define their own message structure, with the data portion defined by the needs of the application.

For example, if some application wanted to exchange messages consisting of a 16-bit integer followed by an 8-byte character array, it could define its own structure as:

```
#define MY_DATA   8

typedef struct my_msgbuf {
  long     mtype;      /* message type */
  int16_t  mshort;     /* start of message data */
  char     mchar[MY_DATA];
} Message;
```

The *length* argument to `msgsnd` specifies the length of the message in bytes. This is the length of the user-defined data that follows the long integer message type. The length can be 0. In the example just shown, the length could be passed as `sizeof(Message) - sizeof(long)`.

The *flag* argument can be either 0 or `IPC_NOWAIT`. This flag makes the call to `msgsnd` *nonblocking*: the function returns immediately if no room is available for the new message. This condition can occur if

- too many bytes are already on the specified queue (the `msg_qbytes` value in the `msqid_ds` structure), or
- too many messages exist systemwide.

If one of these two conditions exists and if `IPC_NOWAIT` is specified, `msgsnd` returns an error of `EAGAIN`. If one of these two conditions exists and if `IPC_NOWAIT` is not specified, then the thread is put to sleep until

- room exists for the message,
- the message queue identified by *msqid* is removed from the system (in which case, an error of `EIDRM` is returned), or
- the calling thread is interrupted by a caught signal (in which case, an error of `EINTR` is returned).

6.4 `msgrcv` Function

A message is read from a message queue using the `msgrcv` function.

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

ssize_t msgrcv(int msqid, void *ptr, size_t length, long type, int flag);
```

Returns: number of bytes of data read into buffer if OK, –1 on error

The *ptr* argument specifies where the received message is to be stored. As with `msgsnd`, this pointer points to the long integer type field (Figure 4.26) that is returned immediately before the actual message data.

length specifies the size of the data portion of the buffer pointed to by *ptr*. This is the maximum amount of data that is returned by the function. This length excludes the long integer type field.

type specifies which message on the queue is desired:

- If *type* is 0, the first message on the queue is returned. Since each message queue is maintained as a FIFO list (first-in, first-out), a *type* of 0 specifies that the oldest message on the queue is to be returned.
- If *type* is greater than 0, the first message whose type equals *type* is returned.
- If *type* is less than 0, the first message with the *lowest* type that is less than or equal to the absolute value of the *type* argument is returned.

Consider the message queue example shown in Figure 6.1, which contains three messages:

- the first message has a type of 100 and a length of 1,
- the second has a type of 200 and a length of 2, and
- the last message has a type of 300 and a length of 3.

Figure 6.2 shows the message returned for different values of *type*.

type	Type of message returned
0	100
100	100
200	200
300	300
-100	100
-200	100
-300	100

Figure 6.2 Messages returned by msgrcv for different values of *type*.

The *flag* argument specifies what to do if a message of the requested type is not on the queue. If the IPC_NOWAIT bit is set and no message is available, the msgrcv function returns immediately with an error of ENOMSG. Otherwise, the caller is blocked until one of the following occurs:

1. a message of the requested type is available,
2. the message queue identified by *msqid* is removed from the system (in which case, an error of EIDRM is returned), or
3. the calling thread is interrupted by a caught signal (in which case, an error of EINTR is returned).

An additional bit in the *flag* argument can be specified: MSG_NOERROR. When set, this specifies that if the actual data portion of the received message is greater than the *length* argument, just truncate the data portion and return without an error. Not specifying the MSG_NOERROR flag causes an error return of E2BIG if *length* is not large enough to receive the entire message.

On successful return, `msgrcv` returns the number of bytes of data in the received message. This does not include the bytes needed for the long integer message type that is also returned through the *ptr* argument.

6.5 `msgctl` Function

The `msgctl` function provides a variety of control operations on a message queue.

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

int msgctl(int msqid, int cmd, struct msqid_ds *buff);

                                        Returns: 0 if OK, -1 on error
```

Three commands are provided:

`IPC_RMID` Remove the message queue specified by *msqid* from the system. Any messages currently on the queue are discarded. We have already seen an example of this operation in Figure 3.7. The third argument to the function is ignored for this command.

`IPC_SET` Set the following four members of the `msqid_ds` structure for the message queue from the corresponding members in the structure pointed to by the *buff* argument: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`.

`IPC_STAT` Return to the caller (through the *buff* argument) the current `msqid_ds` structure for the specified message queue.

Example

The program in Figure 6.3 creates a message queue, puts a message containing 1 byte of data onto the queue, issues the `IPC_STAT` command to `msgctl`, executes the `ipcs` command using the `system` function, and then removes the queue using the `IPC_RMID` command to `msgctl`.

We write a 1-byte message to the queue, so we just use the standard `msgbuf` structure defined in `<sys/msg.h>`.

Executing this program gives us

```
solaris % ctl
read-write: 664, cbytes = 1, qnum = 1, qbytes = 4096
IPC status from <running system> as of Mon Oct 20 15:36:40 1997
T         ID      KEY        MODE          OWNER    GROUP
Message Queues:
q       1150   00000000 --rw-rw-r-- rstevens   other1
```

The values are as expected. The key value of 0 is the common value for `IPC_PRIVATE`, as we mentioned in Section 3.2. On this system there is a limit of 4096 bytes per message queue. Since we wrote a message with 1 byte of data, and since `msg_cbytes` is 1, *svmsg/ctl.c*

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     msqid;
    struct msqid_ds info;
    struct msgbuf buf;

    msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);

    buf.mtype = 1;
    buf.mtext[0] = 1;
    Msgsnd(msqid, &buf, 1, 0);

    Msgctl(msqid, IPC_STAT, &info);
    printf("read-write: %03o, cbytes = %lu, qnum = %lu, qbytes = %lu\n",
           info.msg_perm.mode & 0777, (ulong_t) info.msg_cbytes,
           (ulong_t) info.msg_qnum, (ulong_t) info.msg_qbytes);

    system("ipcs -q");

    Msgctl(msqid, IPC_RMID, NULL);
    exit(0);
}
```

svmsg/ctl.c

Figure 6.3 Example of `msgctl` function with `IPC_STAT` command.

this limit is apparently just for the data portion of the messages, and does not include the long integer message type associated with each message.

6.6 Simple Programs

Since System V message queues are kernel-persistent, we can write a small set of programs to manipulate these queues, and see what happens.

`msgcreate` Program

Figure 6.4 shows our `msgcreate` program, which creates a message queue.

9-12 We allow a command-line option of `-e` to specify the `IPC_EXCL` flag.

16 The pathname that is required as a command-line argument is passed as an argument to `ftok`. The resulting key is converted into an identifier by `msgget`. (See Exercise 6.1.)

`msgsnd` Program

Our `msgsnd` program is shown in Figure 6.5, and it places one message of a specified length and type onto a queue.

svmsg/msgcreate.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     c, oflag, mqid;

    oflag = SVMSG_MODE | IPC_CREAT;
    while ( (c = Getopt(argc, argv, "e")) != -1) {
        switch (c) {
        case 'e':
            oflag |= IPC_EXCL;
            break;
        }
    }
    if (optind != argc - 1)
        err_quit("usage: msgcreate [ -e ] <pathname>");

    mqid = Msgget(Ftok(argv[optind], 1), oflag);
    exit(0);
}
```

svmsg/msgcreate.c

Figure 6.4 Create a System V message queue.

svmsg/msgsnd.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     mqid;
    size_t  len;
    long    type;
    struct msgbuf *ptr;

    if (argc != 4)
        err_quit("usage: msgsnd <pathname> <#bytes> <type>");
    len = atoi(argv[2]);
    type = atoi(argv[3]);

    mqid = Msgget(Ftok(argv[1], 1), MSG_W);

    ptr = Calloc(sizeof(long) + len, sizeof(char));
    ptr->mtype = type;

    Msgsnd(mqid, ptr, len, 0);

    exit(0);
}
```

svmsg/msgsnd.c

Figure 6.5 Add a message to a System V message queue.

We allocate a pointer to a generic `msgbuf` structure but then allocate the actual structure (e.g., the output buffer) by calling `calloc`, based on the size of the message. This function initializes the buffer to 0.

msgrcv Program

Figure 6.6 shows our `msgrcv` function, which reads a message from a queue. An optional `-n` argument specifies nonblocking, and an optional `-t` argument specifies the *type* argument for `msgrcv`.

svmsg/msgrcv.c

```
 1 #include    "unpipc.h"

 2 #define MAXMSG  (8192 + sizeof(long))

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     c, flag, mqid;
 7     long    type;
 8     ssize_t n;
 9     struct msgbuf *buff;

10     type = flag = 0;
11     while ( (c = Getopt(argc, argv, "nt:")) != -1) {
12         switch (c) {
13         case 'n':
14             flag |= IPC_NOWAIT;
15             break;

16         case 't':
17             type = atol(optarg);
18             break;
19         }
20     }
21     if (optind != argc - 1)
22         err_quit("usage: msgrcv [ -n ] [ -t type ] <pathname>");

23     mqid = Msgget(Ftok(argv[optind], 1), MSG_R);

24     buff = Malloc(MAXMSG);

25     n = Msgrcv(mqid, buff, MAXMSG, type, flag);
26     printf("read %d bytes, type = %ld\n", n, buff->mtype);

27     exit(0);
28 }
```

svmsg/msgrcv.c

Figure 6.6 Read a message from a System V message queue.

2 No simple way exists to determine the maximum size of a message (we talk about this and other limits in Section 6.10), so we define our own constant for this limit.

msgrmid Program

To remove a message queue, we call `msgctl` with a command of `IPC_RMID`, as shown in Figure 6.7.

svmsg/msgrmid.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     mqid;

    if (argc != 2)
        err_quit("usage: msgrmid <pathname>");

    mqid = Msgget(Ftok(argv[1], 1), 0);
    Msgctl(mqid, IPC_RMID, NULL);

    exit(0);
}
```

svmsg/msgrmid.c

Figure 6.7 Remove a System V message queue.

Examples

We now use the four programs that we have just shown. We first create a message queue and write three messages to the queue.

```
solaris % msgcreate /tmp/no/such/file
ftok error for pathname "/tmp/no/such/file" and id 0: No such file or directory
solaris % touch /tmp/test1
solaris % msgcreate /tmp/test1
solaris % msgsnd /tmp/test1 1 100
solaris % msgsnd /tmp/test1 2 200
solaris % msgsnd /tmp/test1 3 300
solaris % ipcs -qo
IPC status from <running system> as of Sat Jan 10 11:25:45 1998
T         ID      KEY         MODE        OWNER    GROUP CBYTES  QNUM
Message Queues:
q        100   0x0000113e --rw-r--r-- rstevens   other1      6     3
```

We first try to create a message queue using a pathname that does not exist. This demonstrates that the pathname argument for `ftok` must exist. We then create the file `/tmp/test1` and create a message queue using this pathname. Three messages are placed onto the queue: the three lengths are 1, 2, and 3 bytes, and the three types are respectively 100, 200, and 300 (recall Figure 6.1). The `ipcs` program shows 3 messages comprising a total of 6 bytes on the queue.

We next demonstrate the use of the *type* argument to `msgrcv` in reading the messages in an order other than FIFO.

```
solaris % msgrcv -t 200 /tmp/test1
read 2 bytes, type = 200
solaris % msgrcv -t -300 /tmp/test1
read 1 bytes, type = 100
solaris % msgrcv /tmp/test1
read 3 bytes, type = 300
solaris % msgrcv -n /tmp/test1
msgrcv error: No message of desired type
```

The first example requests the message with a type field of 200, the second example requests the message with the lowest type field less than or equal to 300, and the third example requests the first message on the queue. The last execution of our `msgrcv` program shows the `IPC_NOWAIT` flag.

What happens if we specify a positive *type* argument to `msgrcv` but no message with that type exists on the queue?

```
solaris % ipcs -qo
IPC status from <running system> as of Sat Jan 10 11:37:01 1998
T         ID      KEY         MODE          OWNER     GROUP CBYTES  QNUM
Message Queues:
q        100   0x0000113e --rw-r--r-- rstevens   other1      0      0
solaris % msgsnd /tmp/test1 1 100
solaris % msgrcv -t 999 /tmp/test1
^?                                          type our interrupt key to terminate
solaris % msgrcv -n -t 999 /tmp/test1
msgrcv error: No message of desired type
solaris % grep desired /usr/include/sys/errno.h
#define ENOMSG  35      /* No message of desired type */
solaris % msgrmid /tmp/test1
```

We first execute `ipcs` to verify that the queue is empty, and then place a message of length 1 with a type of 100 onto the queue. When we ask for a message of type 999, the program blocks (in the call to `msgrcv`), waiting for a message of that type to be placed onto the queue. We interrupt this by terminating the program with our interrupt key. We then specify the `-n` flag to prevent blocking, and see that the error `ENOMSG` is returned in this scenario. We then remove the queue from the system with our `msgrmid` program. We could have removed the queue using the system-provided command

```
solaris % ipcrm -q 100
```

which specifies the message queue identifier, or using

```
solaris % ipcrm -Q 0x113e
```

which specifies the message queue key.

`msgrcvid` Program

We now demonstrate that to access a System V message queue, we need not call `msgget`: all we need to know is the message queue identifier (easily obtained with `ipcs`) and read permission for the queue. Figure 6.8 shows a simplification of our `msgrcv` program from Figure 6.6.

We do not call `msgget`. Instead, the caller specifies the message queue identifier on the command line.

svmsg/msgrcvid.c

```
#include    "unpipc.h"

#define MAXMSG  (8192 + sizeof(long))

int
main(int argc, char **argv)
{
    int     mqid;
    ssize_t n;
    struct msgbuf *buff;

    if (argc != 2)
        err_quit("usage: msgrcvid <mqid>");
    mqid = atoi(argv[1]);

    buff = Malloc(MAXMSG);

    n = Msgrcv(mqid, buff, MAXMSG, 0, 0);
    printf("read %d bytes, type = %ld\n", n, buff->mtype);

    exit(0);
}
```

svmsg/msgrcvid.c

Figure 6.8 Read from a System V message queue knowing only the identifier.

Here is an example of this technique:

```
solaris % touch /tmp/testid
solaris % msgcreate /tmp/testid
solaris % msgsnd /tmp/testid 4 400
solaris % ipcs -qo
IPC status from <running system> as of Wed Mar 25 09:48:28 1998
T         ID      KEY        MODE        OWNER    GROUP CBYTES  QNUM
Message Queues:
q        150   0x0000118a --rw-r--r-- rstevens   other1      4     1
solaris % msgrcvid 150
read 4 bytes, type = 400
```

We obtain the identifier of 150 from `ipcs`, and this is the command-line argument to our `msgrcvid` program.

This same feature applies to System V semaphores (Exercise 11.1) and System V shared memory (Exercise 14.1).

6.7 Client–Server Example

We now code our client–server example from Section 4.2 to use two message queues. One queue is for messages from the client to the server, and the other queue is for messages in the other direction.

Our header `svmsg.h` is shown in Figure 6.9. We include our standard header and define the keys for each message queue.

svmsgcliserv/svmsg.h

```
#include    "unpipc.h"

#define MQ_KEY1 1234L
#define MQ_KEY2 2345L
```

svmsgcliserv/svmsg.h

Figure 6.9 `svmsg.h` header for client–server using message queues.

The `main` function for the server is shown in Figure 6.10. Both message queues are created and if either already exists, it is OK, because we do not specify the `IPC_EXCL` flag. The `server` function is the one shown in Figure 4.30 that calls our `mesg_send` and `mesg_recv` functions, versions of which we show shortly.

svmsgcliserv/server_main.c

```
#include    "svmsg.h"

void    server(int, int);

int
main(int argc, char **argv)
{
    int     readid, writeid;

    readid = Msgget(MQ_KEY1, SVMSG_MODE | IPC_CREAT);
    writeid = Msgget(MQ_KEY2, SVMSG_MODE | IPC_CREAT);

    server(readid, writeid);

    exit(0);
}
```

svmsgcliserv/server_main.c

Figure 6.10 Server `main` function using message queues.

svmsgcliserv/client_main.c

```
#include    "svmsg.h"

void    client(int, int);

int
main(int argc, char **argv)
{
    int     readid, writeid;

        /* assumes server has created the queues */
    writeid = Msgget(MQ_KEY1, 0);
    readid = Msgget(MQ_KEY2, 0);

    client(readid, writeid);

        /* now we can delete the queues */
    Msgctl(readid, IPC_RMID, NULL);
    Msgctl(writeid, IPC_RMID, NULL);

    exit(0);
}
```

svmsgcliserv/client_main.c

Figure 6.11 Client `main` function using message queues.

Figure 6.11 shows the main function for the client. The two message queues are opened and our client function from Figure 4.29 is called. This function calls our mesg_send and mesg_recv functions, which we show next.

Both the client and server functions use the message format shown in Figure 4.25. These two functions also call our mesg_send and mesg_recv functions. The versions of these functions that we showed in Figures 4.27 and 4.28 called write and read, which worked with pipes and FIFOs, but we need to recode these two functions to work with message queues. Figures 6.12 and 6.13 show these new versions. Notice that the arguments to these two functions do not change from the versions that called write and read, because the first integer argument can contain either an integer descriptor (for a pipe or FIFO) or an integer message queue identifier.

svmsgcliserv/mesg_send.c

```
#include    "mesg.h"

ssize_t
mesg_send(int id, struct mymesg *mptr)
{
    return (msgsnd(id, &(mptr->mesg_type), mptr->mesg_len, 0));
}
```

svmsgcliserv/mesg_send.c

Figure 6.12 mesg_send function that works with message queues.

svmsgcliserv/mesg_recv.c

```
#include    "mesg.h"

ssize_t
mesg_recv(int id, struct mymesg *mptr)
{
    ssize_t n;

    n = msgrcv(id, &(mptr->mesg_type), MAXMESGDATA, mptr->mesg_type, 0);
    mptr->mesg_len = n;         /* return #bytes of data */

    return (n);                 /* -1 on error, 0 at EOF, else >0 */
}
```

svmsgcliserv/mesg_recv.c

Figure 6.13 mesg_recv function that works with message queues.

6.8 Multiplexing Messages

Two features are provided by the type field that is associated with each message on a queue:

1. The type field can be used to identify the messages, allowing multiple processes to *multiplex* messages onto a single queue. One value of the type field is used for messages from the clients to the server, and a different value that is unique for each client is used for messages from the server to the clients. Naturally, the process ID of the client can be used as the type field that is unique for each client.

2. The type field can be used as a priority field. This lets the receiver read the messages in an order other than first-in, first-out (FIFO). With pipes and FIFOs, the data must be read in the order in which it was written. With System V message queues, we can read the messages in any order that is consistent with the values we associate with the message types. Furthermore, we can call `msgrcv` with the `IPC_NOWAIT` flag to read any messages of a given type from the queue, but return immediately if no messages of the specified type exist.

Example: One Queue per Application

Recall our simple example of a server process and a single client process. With either pipes or FIFOs, two IPC channels are required to exchange data in both directions, since these types of IPC are unidirectional. With a message queue, a single queue can be used, having the type of each message signify whether the message is from the client to the server, or vice versa.

Consider the next complication, a server with multiple clients. Here we can use a type of 1, say, to indicate a message from any client to the server. If the client passes its process ID as part of the message, the server can send its messages to the client processes, using the client's process ID as the message type. Each client then specifies its process ID as the *type* argument to `msgrcv`. Figure 6.14 shows how a single message queue can be used to multiplex these messages between multiple clients and one server.

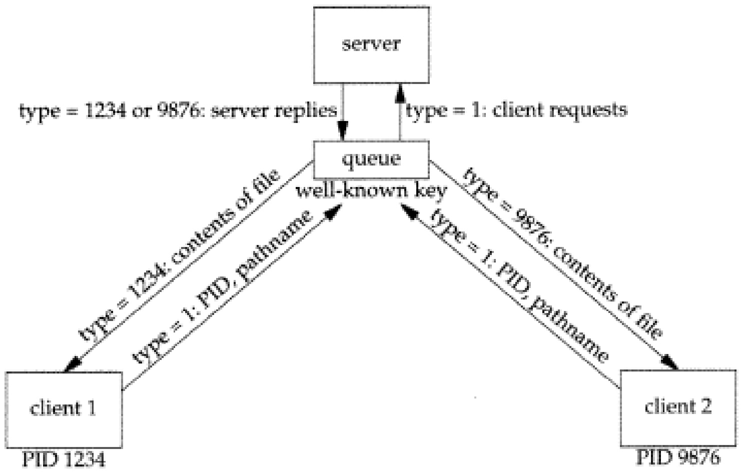
server
type = 1234 or 9876: server replies
type = 1: client requests
queue
well-known key
type = 1234: contents of file
type = 1: PID, pathname
type = 9876: contents of file
type = 1: PID, pathname
client 1
PID 1234
client 2
PID 9876

Figure 6.14 Multiplexing messages between multiple clients and one server.

A potential for deadlock always exists when one IPC channel is used by both the clients and the server. Clients can fill up the message queue (in this example), preventing the server from sending a reply. The clients are then blocked in `msgsnd`, as is the server. One convention that can detect this deadlock is for the server to always use a nonblocking write to the message queue.

We now redo our client–server example using a single message queue with different message types for messages in each direction. These programs use the convention that messages with a type of 1 are from the client to the server, and all other messages have a type equal to the process ID of the client. This client–server requires that the client request contain the client's process ID along with the pathname, similar to what we did in Section 4.8.

Figure 6.15 shows the server `main` function. The `svmsg.h` header was shown in Figure 6.9. Only one message queue is created, and if it already exists, it is OK. The same message queue identifier is used for both arguments to the `server` function.

svmsgmpx1q/server_main.c

```
#include    "svmsg.h"

void    server(int, int);

int
main(int argc, char **argv)
{
    int     msqid;

    msqid = Msgget(MQ_KEY1, SVMSG_MODE | IPC_CREAT);

    server(msqid, msqid);       /* same queue for both directions */

    exit(0);
}
```

svmsgmpx1q/server_main.c

Figure 6.15 Server `main` function.

The `server` function does all the server processing, and is shown in Figure 6.16. This function is a combination of Figure 4.23, our FIFO server that read commands consisting of a process ID and a pathname, and Figure 4.30, which used our `mesg_send` and `mesg_recv` functions. Notice that the process ID sent by the client is used as the message type for all messages sent by the server to the client. Also, this `server` is an infinite loop that is called once and never returns, reading each client request and sending back the replies. Our server is an iterative server, as we discussed in Section 4.9.

Figure 6.17 shows the client `main` function. The client opens the message queue, which the server must have already created.

The `client` function shown in Figure 6.18 does all of the processing for our client. This function is a combination of Figure 4.24, which sent a process ID followed by a pathname, and Figure 4.29, which used our `mesg_send` and `mesg_recv` functions. Note that the type of messages requested from `mesg_recv` equals the process ID of the client.

Our `client` and `server` functions both use the `mesg_send` and `mesg_recv` functions from Figures 6.12 and 6.13.

svmsgmpx1q/server.c

```
#include    "mesg.h"

void
server(int readfd, int writefd)
{
    FILE   *fp;
    char   *ptr;
    pid_t   pid;
    ssize_t n;
    struct mymesg mesg;

    for ( ; ; ) {
            /* read pathname from IPC channel */
        mesg.mesg_type = 1;
        if ( (n = Mesg_recv(readfd, &mesg)) == 0) {
            err_msg("pathname missing");
            continue;
        }
        mesg.mesg_data[n] = '\0';   /* null terminate pathname */

        if ( (ptr = strchr(mesg.mesg_data, ' ')) == NULL) {
            err_msg("bogus request: %s", mesg.mesg_data);
            continue;
        }
        *ptr++ = 0;             /* null terminate PID, ptr = pathname */
        pid = atol(mesg.mesg_data);
        mesg.mesg_type = pid;   /* for messages back to client */

        if ( (fp = fopen(ptr, "r")) == NULL) {
                /* error: must tell client */
            snprintf(mesg.mesg_data + n, sizeof(mesg.mesg_data) - n,
                     ": can't open, %s\n", strerror(errno));
            mesg.mesg_len = strlen(ptr);
            memmove(mesg.mesg_data, ptr, mesg.mesg_len);
            Mesg_send(writefd, &mesg);

        } else {
                /* fopen succeeded: copy file to IPC channel */
            while (Fgets(mesg.mesg_data, MAXMESGDATA, fp) != NULL) {
                mesg.mesg_len = strlen(mesg.mesg_data);
                Mesg_send(writefd, &mesg);
            }
            Fclose(fp);
        }

            /* send a 0-length message to signify the end */
        mesg.mesg_len = 0;
        Mesg_send(writefd, &mesg);
    }
}
```

svmsgmpx1q/server.c

Figure 6.16 server function.

svmsgmpx1q/client_main.c

```
#include    "svmsg.h"

void    client(int, int);

int
main(int argc, char **argv)
{
    int     msqid;

        /* server must create the queue */
    msqid = Msgget(MQ_KEY1, 0);

    client(msqid, msqid);       /* same queue for both directions */

    exit(0);
}
```

svmsgmpx1q/client_main.c

Figure 6.17 Client `main` function.

svmsgmpx1q/client.c

```
#include    "mesg.h"

void
client(int readfd, int writefd)
{
    size_t  len;
    ssize_t n;
    char   *ptr;
    struct mymesg mesg;

        /* start buffer with pid and a blank */
    snprintf(mesg.mesg_data, MAXMESGDATA, "%ld ", (long) getpid());
    len = strlen(mesg.mesg_data);
    ptr = mesg.mesg_data + len;

        /* read pathname */
    Fgets(ptr, MAXMESGDATA - len, stdin);
    len = strlen(mesg.mesg_data);
    if (mesg.mesg_data[len - 1] == '\n')
        len--;                  /* delete newline from fgets() */
    mesg.mesg_len = len;
    mesg.mesg_type = 1;

        /* write PID and pathname to IPC channel */
    Mesg_send(writefd, &mesg);

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

svmsgmpx1q/client.c

Figure 6.18 `client` function.

Example: One Queue per Client

We now modify the previous example to use one queue for all the client requests to the server and one queue per client for that client's responses. Figure 6.19 shows the design.

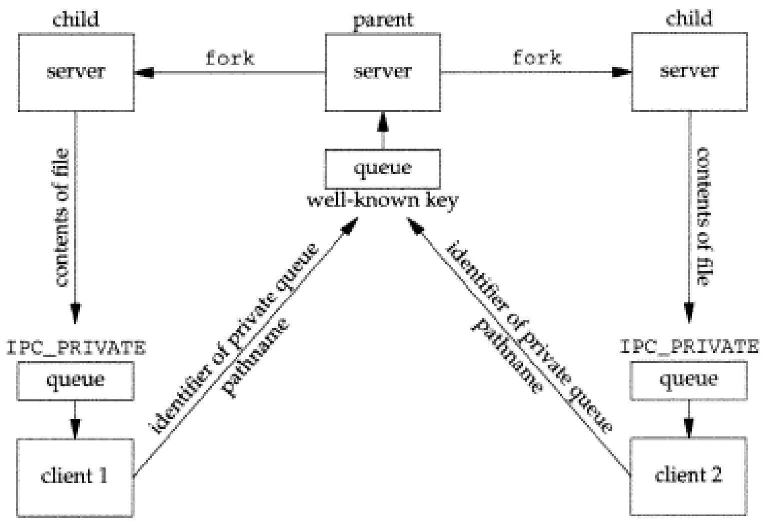

Figure 6.19 One queue per server and one queue per client.

The server's queue has a key that is well-known to the clients, but each client creates its own queue with a key of `IPC_PRIVATE`. Instead of passing its process ID with the request, each client passes the identifier of its private queue to the server, and the server sends its reply to the client's queue. We also write this server as a concurrent server, with one `fork` per client.

> One potential problem with this design occurs if a client dies, in which case, messages may be left in its private queue forever (or at least until the kernel reboots or someone explicitly deletes the queue).

The following headers and functions do not change from previous versions:

- `mesg.h` header (Figure 4.25),
- `svmsg.h` header (Figure 6.9),
- server `main` function (Figure 6.15), and
- `mesg_send` function (Figure 4.27).

Our client `main` function is shown in Figure 6.20; it has changed slightly from Figure 6.17. We open the server's well-known queue (`MQ_KEY1`) and then create our own queue with a key of `IPC_PRIVATE`. The two queue identifiers become the arguments to the `client` function (Figure 6.21). When the client is done, its private queue is removed.

svmsgmpxnq/client_main.c

```
#include    "svmsg.h"

void    client(int, int);

int
main(int argc, char **argv)
{
    int     readid, writeid;

        /* server must create its well-known queue */
    writeid = Msgget(MQ_KEY1, 0);
        /* we create our own private queue */
    readid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);

    client(readid, writeid);

        /* and delete our private queue */
    Msgctl(readid, IPC_RMID, NULL);

    exit(0);
}
```

svmsgmpxnq/client_main.c

Figure 6.20 Client `main` function.

svmsgmpxnq/client.c

```
#include    "mesg.h"

void
client(int readid, int writeid)
{
    size_t  len;
    ssize_t n;
    char   *ptr;
    struct mymesg mesg;

        /* start buffer with msqid and a blank */
    snprintf(mesg.mesg_data, MAXMESGDATA, "%d ", readid);
    len = strlen(mesg.mesg_data);
    ptr = mesg.mesg_data + len;

        /* read pathname */
    Fgets(ptr, MAXMESGDATA - len, stdin);
    len = strlen(mesg.mesg_data);
    if (mesg.mesg_data[len - 1] == '\n')
        len--;                  /* delete newline from fgets() */
    mesg.mesg_len = len;
    mesg.mesg_type = 1;

        /* write msqid and pathname to server's well-known queue */
    Mesg_send(writeid, &mesg);

        /* read from our queue, write to standard output */
    while ( (n = Mesg_recv(readid, &mesg)) > 0)
        Write(STDOUT_FILENO, mesg.mesg_data, n);
}
```

svmsgmpxnq/client.c

Figure 6.21 `client` function.

Figure 6.21 is the `client` function. This function is nearly identical to Figure 6.18, but instead of passing the client's process ID as part of the request, the identifier of the client's private queue is passed instead. The message type in the `mesg` structure is also left as 1, because that is the type used for messages in both directions.

Figure 6.23 is the `server` function. The main change from Figure 6.16 is writing this function as an infinite loop that calls `fork` for each client request.

Establish signal handler for `SIGCHLD`

10 Since we are spawning a child for each client, we must worry about zombie processes. Sections 5.9 and 5.10 of UNPv1 talk about this in detail. Here we establish a signal handler for the `SIGCHLD` signal, and our function `sig_chld` (Figure 6.22) is called when a child terminates.

12-18 The server parent blocks in the call to `mesg_recv` waiting for the next client message to arrive.

25-45 A child is created with `fork`, and the child tries to open the requested file, sending back either an error message or the contents of the file. We purposely put the call to `fopen` in the child, instead of the parent, just in case the file is on a remote filesystem, in which case, the opening of the file could take some time if any network problems occur.

Our handler for the `SIGCHLD` function is shown in Figure 6.22. This is copied from Figure 5.11 of UNPv1.

svmsgmpxnq/sigchldwaitpid.c

```
#include    "unpipc.h"

void
sig_chld(int signo)
{
    pid_t   pid;
    int     stat;

    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0) ;
    return;
}
```

svmsgmpxnq/sigchldwaitpid.c

Figure 6.22 `SIGCHLD` signal handler that calls `waitpid`.

Each time our signal handler is called, it calls `waitpid` in a loop, fetching the termination status of any children that have terminated. Our signal handler then returns. This can create a problem, because the parent process spends most of its time blocked in a call to `msgrcv` in the function `mesg_recv` (Figure 6.13). When our signal handler returns, this call to `msgrcv` is *interrupted*. That is, the function returns an error of `EINTR`, as described in Section 5.9 of UNPv1.

We must handle this interrupted system call, and Figure 6.24 shows the new version of our `Mesg_recv` wrapper function. We allow an error of `EINTR` from `mesg_recv` (which just calls `msgrcv`), and when this happens, we just call `mesg_recv` again.

svmsgmpxnq/server.c

```
#include    "mesg.h"

void
server(int readid, int writeid)
{
    FILE   *fp;
    char   *ptr;
    ssize_t n;
    struct mymesg mesg;
    void    sig_chld(int);

    Signal(SIGCHLD, sig_chld);

    for ( ; ; ) {
            /* read pathname from our well-known queue */
        mesg.mesg_type = 1;
        if ( (n = Mesg_recv(readid, &mesg)) == 0) {
            err_msg("pathname missing");
            continue;
        }
        mesg.mesg_data[n] = '\0';    /* null terminate pathname */

        if ( (ptr = strchr(mesg.mesg_data, ' ')) == NULL) {
            err_msg("bogus request: %s", mesg.mesg_data);
            continue;
        }
        *ptr++ = 0;                /* null terminate msgid, ptr = pathname */
        writeid = atoi(mesg.mesg_data);

        if (Fork() == 0) {        /* child */
            if ( (fp = fopen(ptr, "r")) == NULL) {
                    /* error: must tell client */
                snprintf(mesg.mesg_data + n, sizeof(mesg.mesg_data) - n,
                         ": can't open, %s\n", strerror(errno));
                mesg.mesg_len = strlen(ptr);
                memmove(mesg.mesg_data, ptr, mesg.mesg_len);
                Mesg_send(writeid, &mesg);

            } else {
                    /* fopen succeeded: copy file to client's queue */
                while (Fgets(mesg.mesg_data, MAXMESGDATA, fp) != NULL) {
                    mesg.mesg_len = strlen(mesg.mesg_data);
                    Mesg_send(writeid, &mesg);
                }
                Fclose(fp);
            }

                /* send a 0-length message to signify the end */
            mesg.mesg_len = 0;
            Mesg_send(writeid, &mesg);
            exit(0);                /* child terminates */
        }
        /* parent just loops around */
    }
}
```

svmsgmpxnq/server.c

Figure 6.23 `server` function.

svmsgmpxnq/mesg_recv.c

```
ssize_t
Mesg_recv(int id, struct mymesg *mptr)
{
    ssize_t n;

    do {
        n = mesg_recv(id, mptr);
    } while (n == -1 && errno == EINTR);

    if (n == -1)
        err_sys("mesg_recv error");

    return (n);
}
```

svmsgmpxnq/mesg_recv.c

Figure 6.24 Mesg_recv wrapper function that handles an interrupted system call.

6.9 Message Queues with select and poll

One problem with System V message queues is that they are known by their own identifiers, and not by descriptors. This means that we cannot use either select or poll (Chapter 6 of UNPv1) with these message queues.

> Actually, one version of Unix, IBM's AIX, extends select to handle System V message queues in addition to descriptors. But this is nonportable and works only with AIX.

This missing feature is often uncovered when someone wants to write a server that handles both network connections and IPC connections. Network communications using either the sockets API or the XTI API (UNPv1) use descriptors, allowing either select or poll to be used. Pipes and FIFOs also work with these two functions, because they too are identified by descriptors.

One solution to this problem is for the server to create a pipe and then spawn a child, with the child blocking in a call to msgrcv. When a message is ready to be processed, msgrcv returns, and the child reads the message from the queue and writes the message to the pipe. The server parent can then select on the pipe, in addition to some network connections. The downside is that these messages are then processed three times: once when read by the child using msgrcv, again when written to the pipe by the child, and again when read from the pipe by the parent. To avoid this extra processing, the parent could create a shared memory segment that is shared between itself and the child, and then use the pipe as a flag between the parent and child (Exercise 12.5).

> In Figure 5.14 we showed a solution using Posix message queues that did not require a fork. We can use a single process with Posix message queues, because they provide a notification capability that generates a signal when a message arrives for an empty queue. System V message queues do not provide this capability, so we must fork a child and have the child block in a call to msgrcv.

Another missing feature from System V message queues, when compared to network programming, is the inability to peek at a message, something provided with the `MSG_PEEK` flag to the `recv`, `recvfrom`, and `recvmsg` functions (p. 356 of UNPv1). If such a facility were provided, then the parent–child scenario just described (to get around the `select` problem) could be made more efficient by having the child specify the peek flag to `msgrcv` and just write 1 byte to the pipe when a message was ready, and let the parent read the message.

6.10 Message Queue Limits

As we noted in Section 3.8, certain system limits often exist on message queues. Figure 6.25 shows the values for some different implementations. The first column is the traditional System V name for the kernel variable that contains this limit.

Name	Description	DUnix 4.0B	Solaris 2.6
`msgmax`	max #bytes per message	8192	2048
`msgmnb`	max #bytes on any one message queue	16384	4096
`msgmni`	max #message queues, systemwide	64	50
`msgtql`	max #messages, systemwide	40	40

Figure 6.25 Typical system limits for System V message queues.

Many SVR4-derived implementations have additional limits, inherited from their original implementation: `msgssz` is often 8, and this is the "segment" size (in bytes) in which the message data is stored. A message with 21 bytes of data would be stored in 3 of these segments, with the final 3 bytes of the last segment unused. `msgseg` is the number of these segments that are allocated, often 1024. Historically, this has been stored in a short integer and must therefore be less than 32768. The total number of bytes available for all message data is the product of these two variables, often 8 × 1024 bytes.

The intent of this section is to show some typical values, to aid in planning for portability. When a system runs applications that make heavy use of message queues, kernel tuning of these (or similar) parameters is normally required (which we described in Section 3.8).

Example

Figure 6.26 is a program that determines the four limits shown in Figure 6.25.

svmsg/limits.c

```
#include    "unpipc.h"

#define MAX_DATA    64*1024
#define MAX_NMESG   4096
#define MAX_NIDS    4096
int     max_mesg;

struct mymesg {
    long    type;
    char    data[MAX_DATA];
} mesg;

int
main(int argc, char **argv)
{
    int     i, j, msqid, qid[MAX_NIDS];

        /* first try and determine maximum amount of data we can send */
    msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
    mesg.type = 1;
    for (i = MAX_DATA; i > 0; i -= 128) {
        if (msgsnd(msqid, &mesg, i, 0) == 0) {
            printf("maximum amount of data per message = %d\n", i);
            max_mesg = i;
            break;
        }
        if (errno != EINVAL)
            err_sys("msgsnd error for length %d", i);
    }
    if (i == 0)
        err_quit("i == 0");
    Msgctl(msqid, IPC_RMID, NULL);

        /* see how many messages of varying size can be put onto a queue */
    mesg.type = 1;
    for (i = 8; i <= max_mesg; i *= 2) {
        msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
        for (j = 0; j < MAX_NMESG; j++) {
            if (msgsnd(msqid, &mesg, i, IPC_NOWAIT) != 0) {
                if (errno == EAGAIN)
                    break;
                err_sys("msgsnd error, i = %d, j = %d", i, j);
                break;
            }
        }
        printf("%d %d-byte messages were placed onto queue,", j, i);
        printf(" %d bytes total\n", i * j);
        Msgctl(msqid, IPC_RMID, NULL);
    }

        /* see how many identifiers we can "open" */
    mesg.type = 1;
    for (i = 0; i <= MAX_NIDS; i++) {
        if ( (qid[i] = msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT)) == -1) {
            printf("%d identifiers open at once\n", i);
            break;
        }
    }
    for (j = 0; j < i; j++)
        Msgctl(qid[j], IPC_RMID, NULL);

    exit(0);
}
```
svmsg/limits.c

Figure 6.26 Determine the system limits on System V message queues.

Determine maximum message size

14-28 To determine the maximum message size, we try to send a message containing 65536 bytes of data, and if this fails, we try a message containing 65408 bytes of data, and so on, until the call to msgsnd succeeds.

How many messages of varying size can be put onto a queue?

29-44 Next we start with 8-byte messages and see how many can be placed onto a given queue. Once we determine this limit, we delete the queue (discarding all these messages) and try again with 16-byte messages. We keep doing so until we pass the maximum message size that was determined in the first step. We expect smaller messages to encounter a limit on the total number of messages per queue and larger messages to encounter a limit on the total number of bytes per queue.

How many identifiers can be open at once?

45-54 Normally a system limit exists on the maximum number of message queue identifiers that can be open at any time. We determine this by just creating queues until msgget fails.

We first run this program under Solaris 2.6 and then Digital Unix 4.0B, and the results confirm the values shown in Figure 6.25.

```
solaris % limits
maximum amount of data per message = 2048
40 8-byte messages were placed onto queue, 320 bytes total
40 16-byte messages were placed onto queue, 640 bytes total
40 32-byte messages were placed onto queue, 1280 bytes total
40 64-byte messages were placed onto queue, 2560 bytes total
32 128-byte messages were placed onto queue, 4096 bytes total
16 256-byte messages were placed onto queue, 4096 bytes total
8 512-byte messages were placed onto queue, 4096 bytes total
4 1024-byte messages were placed onto queue, 4096 bytes total
2 2048-byte messages were placed onto queue, 4096 bytes total
50 identifiers open at once

alpha % limits
maximum amount of data per message = 8192
40 8-byte messages were placed onto queue, 320 bytes total
40 16-byte messages were placed onto queue, 640 bytes total
40 32-byte messages were placed onto queue, 1280 bytes total
40 64-byte messages were placed onto queue, 2560 bytes total
40 128-byte messages were placed onto queue, 5120 bytes total
40 256-byte messages were placed onto queue, 10240 bytes total
32 512-byte messages were placed onto queue, 16384 bytes total
16 1024-byte messages were placed onto queue, 16384 bytes total
8 2048-byte messages were placed onto queue, 16384 bytes total
4 4096-byte messages were placed onto queue, 16384 bytes total
2 8192-byte messages were placed onto queue, 16384 bytes total
63 identifiers open at once
```

The reason for the limit of 63 identifiers under Digital Unix, and not the 64 shown in Figure 6.25, is that one identifier is already being used by a system daemon.

6.11 Summary

System V message queues are similar to Posix message queues. New applications should consider using Posix message queues, but lots of existing code uses System V message queues. Nevertheless, recoding an application to use Posix message queues, instead of System V message queues, should not be hard. The main feature missing from Posix message queues is the ability to read messages of a specified priority from the queue. Neither form of message queue uses real descriptors, making it hard to use either `select` or `poll` with a message queue.

Exercises

6.1 Modify Figure 6.4 to accept a pathname argument of `IPC_PRIVATE` and create a message queue with a private key if this is specified. What changes must then be made to the remaining programs in Section 6.6?

6.2 Why did we use a type of 1 in Figure 6.14 for messages to the server?

6.3 What happens in Figure 6.14 if a malicious client sends many messages to the server but never reads any of the server's replies? What changes with Figure 6.19 for this type of client?

6.4 Redo the implementation of Posix message queues from Section 5.8 to use System V message queues instead of memory-mapped I/O.

Part 3

Synchronization

7

Mutexes and Condition Variables

7.1 Introduction

This chapter begins our discussion of synchronization: how to synchronize the actions of multiple threads or multiple processes. Synchronization is normally needed to allow the sharing of data between threads or processes. Mutexes and condition variables are the building blocks of synchronization.

Mutexes and condition variables are from the Posix.1 threads standard, and can always be used to synchronize the various threads within a process. Posix also allows a mutex or condition variable to be used for synchronization between multiple processes, if the mutex or condition variable is stored in memory that is shared between the processes.

> This is an option for Posix but required by Unix 98 (e.g., the "process shared mutex/CV" line in Figure 1.5).

In this chapter, we introduce the classic producer–consumer problem and use mutexes and condition variables in our solution of this problem. We use multiple threads for this example, instead of multiple processes, because having multiple threads share the common data buffer that is assumed in this problem is trivial, whereas sharing a common data buffer between multiple processes requires some form of shared memory (which we do not describe until Part 4). We provide additional solutions to this problem in Chapter 10 using semaphores.

7.2 Mutexes: Locking and Unlocking

A mutex, which stands for *mutual exclusion*, is the most basic form of synchronization. A mutex is used to protect a *critical region*, to make certain that only one thread at a time executes the code within the region (assuming a mutex that is being shared by the threads) or that only one process at a time executes the code within the region (assuming a mutex is being shared by the processes). The normal outline of code to protect a critical region looks like

```
lock_the_mutex(...);
critical region
unlock_the_mutex(...);
```

Since only one thread at a time can lock a given mutex, this guarantees that only one thread at a time can be executing the instructions within the critical region.

Posix mutexes are declared as variables with a datatype of `pthread_mutex_t`. If the mutex variable is statically allocated, we can initialize it to the constant `PTHREAD_MUTEX_INITIALIZER`, as in

```
static pthread_mutex_t  lock = PTHREAD_MUTEX_INITIALIZER;
```

If we dynamically allocate a mutex (e.g., by calling `malloc`) or if we allocate a mutex in shared memory, we must initialize it at run time by calling the `pthread_mutex_init` function, as we show in Section 7.7.

> You may encounter code that omits the initializer because that implementation defines the initializer to be 0 (and statically allocated variables are automatically initialized to 0). But this is incorrect code.

The following three functions lock and unlock a mutex:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mptr);

int pthread_mutex_trylock(pthread_mutex_t *mptr);

int pthread_mutex_unlock(pthread_mutex_t *mptr);

                    All three return: 0 if OK, positive Exxx value on error
```

If we try to lock a mutex that is already locked by some other thread, `pthread_mutex_lock` blocks until the mutex is unlocked. `pthread_mutex_trylock` is a nonblocking function that returns `EBUSY` if the mutex is already locked.

> If multiple threads are blocked waiting for a mutex, which thread runs when the mutex is unlocked? One of the features added by the 1003.1b–1993 standard is an option for priority scheduling. We do not cover this area, but suffice it to say that different threads can be assigned different priorities, and the synchronization functions (mutexes, read–write locks, and semaphores) will wake up the highest priority thread that is blocked. Section 5.5 of [Butenhof 1997] provides more details on the Posix.1 realtime scheduling feature.

Although we talk of a critical region being protected by a mutex, what is really protected is the *data* being manipulated within the critical region. That is, a mutex is normally used to protect *shared data* that is being shared between multiple threads or between multiple processes.

Mutex locks are *cooperative* locks. That is, if the shared data is a linked list (for example), then all the threads that manipulate the linked list must obtain the mutex lock before manipulating the list. Nothing can prevent one thread from manipulating the linked list without first obtaining the mutex.

7.3 Producer–Consumer Problem

One of the classic problems in synchronization is called the *producer–consumer* problem, also known as the *bounded buffer* problem. One or more producers (threads or processes) are creating data items that are then processed by one or more consumers (threads or processes). The data items are passed between the producers and consumers using some type of IPC.

We deal with this problem all the time with Unix pipes. That is, the shell pipeline

```
grep pattern chapters.* | wc -l
```

is such a problem. `grep` is the single producer and `wc` is the single consumer. A Unix pipe is used as the form of IPC. The required synchronization between the producer and consumer is handled by the kernel in the way in which it handles the `writes` by the producer and the `reads` by the consumer. If the producer gets ahead of the consumer (i.e., the pipe fills up), the kernel puts the producer to sleep when it calls `write`, until more room is in the pipe. If the consumer gets ahead of the producer (i.e., the pipe is empty), the kernel puts the consumer to sleep when it calls `read`, until some data is in the pipe.

This type of synchronization is *implicit*; that is, the producer and consumer are not even aware that it is being performed by the kernel. If we were to use a Posix or System V message queue as the form of IPC between the producer and consumer, the kernel would again handle the synchronization.

When shared memory is being used as the form of IPC between the producer and the consumer, however, some type of *explicit* synchronization must be performed by the producers and consumers. We will demonstrate this using a mutex. The example that we use is shown in Figure 7.1.

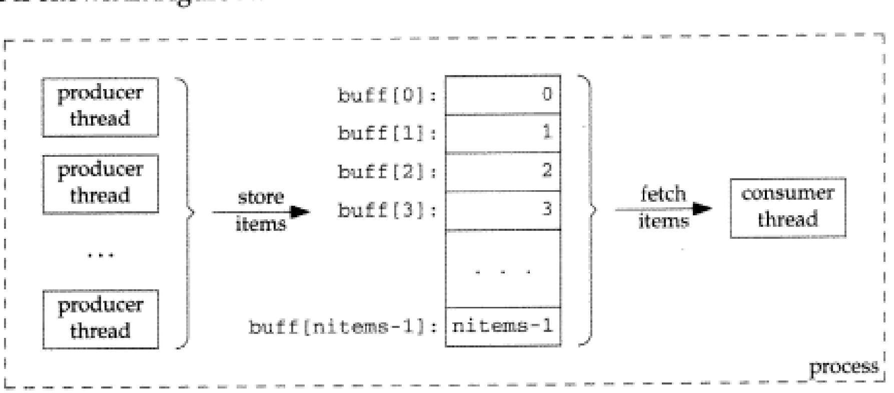

Figure 7.1 Producer–consumer example: multiple producer threads, one consumer thread.

We have multiple producer threads and a single consumer thread, in a single process. The integer array `buff` contains the items being produced and consumed (i.e., the shared data). For simplicity, the producers just set `buff[0]` to 0, `buff[1]` to 1, and so on. The consumer just goes through this array and verifies that each entry is correct.

In this first example, we concern ourselves only with synchronization between the multiple producer threads. We do not start the consumer thread until all the producers are done. Figure 7.2 is the `main` function for our example.

mutex/prodcons2.c

```c
#include    "unpipc.h"

#define MAXNITEMS       1000000
#define MAXNTHREADS         100

int     nitems;             /* read-only by producer and consumer */
struct {
    pthread_mutex_t mutex;
    int     buff[MAXNITEMS];
    int     nput;
    int     nval;
} shared = {
    PTHREAD_MUTEX_INITIALIZER
};

void    *produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
    int     i, nthreads, count[MAXNTHREADS];
    pthread_t tid_produce[MAXNTHREADS], tid_consume;

    if (argc != 3)
        err_quit("usage: prodcons2 <#items> <#threads>");
    nitems = min(atoi(argv[1]), MAXNITEMS);
    nthreads = min(atoi(argv[2]), MAXNTHREADS);

    Set_concurrency(nthreads);
        /* start all the producer threads */
    for (i = 0; i < nthreads; i++) {
        count[i] = 0;
        Pthread_create(&tid_produce[i], NULL, produce, &count[i]);
    }

        /* wait for all the producer threads */
    for (i = 0; i < nthreads; i++) {
        Pthread_join(tid_produce[i], NULL);
        printf("count[%d] = %d\n", i, count[i]);
    }

        /* start, then wait for the consumer thread */
    Pthread_create(&tid_consume, NULL, consume, NULL);
    Pthread_join(tid_consume, NULL);

    exit(0);
}
```

mutex/prodcons2.c

Figure 7.2 `main` function.

Globals shared between the threads

4–12 These variables are shared between the threads. We collect them into a structure named `shared`, along with the mutex, to reinforce that these variables should be accessed only when the mutex is held. `nput` is the next index to store in the `buff` array, and `nval` is the next value to store (0, 1, 2, and so on). We allocate this structure and initialize the mutex that is used for synchronization between the producer threads.

> We will always try to collect shared data with their synchronization variables (mutex, condition variable, or semaphore) into a structure as we have done here, as a good programming technique. In many cases, however, the shared data is dynamically allocated, say as a linked list. We might be able to store the head of the linked list in a structure with the synchronization variables (as we did with our `mq_hdr` structure in Figure 5.20), but other shared data (the rest of the list) is not in the structure. Therefore, this solution is often not perfect.

Command-line arguments

19–22 The first command-line argument specifies the number of items for the producers to store, and the next argument is the number of producer threads to create.

Set concurrency level

23 `set_concurrency` is a function of ours that tells the threads system how many threads we would like to run concurrently. Under Solaris 2.6, this is just a call to `thr_setconcurrency` and is required if we want the multiple producer threads to each have a chance to execute. If we omit this call under Solaris, only the first producer thread runs. Under Digital Unix 4.0B, our `set_concurrency` function does nothing (because all the threads within a process compete for the processor by default).

> Unix 98 requires a function named `pthread_setconcurrency` that performs the same function. This function is needed with threads implementations that multiplex user threads (what we create with `pthread_create`) onto a smaller set of kernel execution entities (e.g., kernel threads). These are commonly referred to as many-to-few, two-level, or M-to-N implementations. Section 5.6 of [Butenhof 1997] discusses the relationship between user threads and kernel entities in more detail.

Create producer threads

24–28 The producer threads are created, and each executes the function `produce`. We save the thread ID of each in the `tid_produce` array. The argument to each producer thread is a pointer to an element of the `count` array. We first initialize the counter to 0, and each thread then increments this counter each time it stores an item in the buffer. We print this array of counters when we are done, to see how many items were stored by each producer thread.

Wait for producer threads, then start consumer thread

29–36 We wait for all the producer threads to terminate, also printing each thread's counter, and only then start a single consumer thread. This is how (for the time being) we avoid any synchronization issues between the producers and consumer. We wait for the consumer to finish and then terminate the process.

Figure 7.3 shows the `produce` and `consume` functions for our example.

mutex/prodcons2.c

```
39 void *
40 produce(void *arg)
41 {
42     for ( ; ; ) {
43         Pthread_mutex_lock(&shared.mutex);
44         if (shared.nput >= nitems) {
45             Pthread_mutex_unlock(&shared.mutex);
46             return (NULL);      /* array is full, we're done */
47         }
48         shared.buff[shared.nput] = shared.nval;
49         shared.nput++;
50         shared.nval++;
51         Pthread_mutex_unlock(&shared.mutex);
52         *((int *) arg) += 1;
53     }
54 }

55 void *
56 consume(void *arg)
57 {
58     int     i;

59     for (i = 0; i < nitems; i++) {
60         if (shared.buff[i] != i)
61             printf("buff[%d] = %d\n", i, shared.buff[i]);
62     }
63     return (NULL);
64 }
```

mutex/prodcons2.c

Figure 7.3 produce and consume functions.

Generate the data items

42-53 The critical region for the producer consists of the test for whether we are done

```
if (shared.nput >= nitems)
```

followed by the three lines

```
shared.buff[shared.nput] = shared.nval;
shared.nput++;
shared.nval++;
```

We protect this region with a mutex lock, being certain to unlock the mutex when we are done. Notice that the increment of the `count` element (through the pointer `arg`) is not part of the critical region because each thread has its own counter (the `count` array in the `main` function). Therefore, we do not include this line of code within the region locked by the mutex, because as a general programming principle, we should always strive to minimize the amount of code that is locked by a mutex.

Consumer verifies contents of array

59-62 The consumer just verifies that each item in the array is correct and prints a message if an error is found. As we said earlier, only one instance of this function is run and only after all the producer threads have finished, so no need exists for any synchronization.

If we run the program just described, specifying one million items and five producer threads, we have

```
solaris % prodcons2 1000000 5
count[0] = 167165
count[1] = 249891
count[2] = 194221
count[3] = 191815
count[4] = 196908
```

As we mentioned, if we remove the call to `set_concurrency` under Solaris 2.6, `count[0]` then becomes 1000000 and the remaining counts are all 0.

If we remove the mutex locking from this example, it fails, as expected. That is, the consumer detects many instances of `buff[i]` not equal to `i`. We can also verify that the removal of the mutex locking has no effect if only one producer thread is run.

7.4 Locking versus Waiting

We now demonstrate that mutexes are for *locking* and cannot be used for *waiting*. We modify our producer–consumer example from the previous section to start the consumer thread right after all the producer threads have been started. This lets the consumer thread process the data as it is being generated by the producer threads, unlike Figure 7.2, in which we did not start the consumer until all the producer threads were finished. But we must now synchronize the consumer with the producers to make certain that the consumer processes only data items that have already been stored by the producers.

Figure 7.4 shows the `main` function. All the lines prior to the declaration of `main` have not changed from Figure 7.2.

mutex/prodcons3.c

```
int
main(int argc, char **argv)
{
    int      i, nthreads, count[MAXNTHREADS];
    pthread_t tid_produce[MAXNTHREADS], tid_consume;

    if (argc != 3)
        err_quit("usage: prodcons3 <#items> <#threads>");
    nitems = min(atoi(argv[1]), MAXNITEMS);
    nthreads = min(atoi(argv[2]), MAXNTHREADS);

        /* create all producers and one consumer */
    Set_concurrency(nthreads + 1);
    for (i = 0; i < nthreads; i++) {
        count[i] = 0;
        Pthread_create(&tid_produce[i], NULL, produce, &count[i]);
    }
    Pthread_create(&tid_consume, NULL, consume, NULL);
30         /* wait for all producers and the consumer */
31     for (i = 0; i < nthreads; i++) {
32         Pthread_join(tid_produce[i], NULL);
33         printf("count[%d] = %d\n", i, count[i]);
34     }
35     Pthread_join(tid_consume, NULL);

36     exit(0);
37 }
```
mutex/prodcons3.c

Figure 7.4 `main` function: start consumer immediately after starting producers.

24 We increase the concurrency level by one, to account for the additional consumer thread.

25–29 We create the consumer thread immediately after creating the producer threads.

The `produce` function does not change from Figure 7.3.

We show in Figure 7.5 the `consume` function, which calls our new `consume_wait` function.

mutex/prodcons3.c
```
54 void
55 consume_wait(int i)
56 {
57     for ( ; ; ) {
58         Pthread_mutex_lock(&shared.mutex);
59         if (i < shared.nput) {
60             Pthread_mutex_unlock(&shared.mutex);
61             return;             /* an item is ready */
62         }
63         Pthread_mutex_unlock(&shared.mutex);
64     }
65 }

66 void *
67 consume(void *arg)
68 {
69     int     i;

70     for (i = 0; i < nitems; i++) {
71         consume_wait(i);
72         if (shared.buff[i] != i)
73             printf("buff[%d] = %d\n", i, shared.buff[i]);
74     }
75     return (NULL);
76 }
```
mutex/prodcons3.c

Figure 7.5 `consume_wait` and `consume` functions.

Consumer must wait

71 The only change to the `consume` function is to call `consume_wait` before fetching the next item from the array.

Wait for producers

57-64 Our `consume_wait` function must wait until the producers have generated the `i`th item. To check this condition, the producer's mutex is locked and `i` is compared to the producer's `nput` index. We must acquire the mutex lock before looking at `nput`, since this variable may be in the process of being updated by one of the producer threads.

The fundamental problem is: what can we do when the desired item is *not* ready? All we do in Figure 7.5 is loop around again, unlocking and locking the mutex each time. This is calling *spinning* or *polling* and is a waste of CPU time.

We could also sleep for a short amount of time, but we do not know how long to sleep. What is needed is another type of synchronization that lets a thread (or process) sleep until some event occurs.

7.5 Condition Variables: Waiting and Signaling

A mutex is for *locking* and a condition variable is for *waiting*. These are two different types of synchronization and both are needed.

A condition variable is a variable of type `pthread_cond_t`, and the following two functions are used with these variables.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);

int pthread_cond_signal(pthread_cond_t *cptr);
```

Both return: 0 if OK, positive `Exxx` value on error

The term "signal" in the second function's name does not refer to a Unix `SIGxxx` signal.

We choose what defines the "condition" to wait for and be notified of: we test this in our code.

A mutex is always associated with a condition variable. When we call `pthread_cond_wait` to wait for some condition to be true, we specify the address of the condition variable and the address of the associated mutex.

We explain the use of condition variables by recoding the example from the previous section. Figure 7.6 shows the global declarations.

Collect producer variables and mutex into a structure

7-13 The two variables `nput` and `nval` are associated with the `mutex`, and we put all three variables into a structure named `put`. This structure is used by the producers.

Collect counter, condition variable, and mutex into a structure

14-20 The next structure, `nready`, contains a counter, a condition variable, and a mutex. We initialize the condition variable to `PTHREAD_COND_INITIALIZER`.

The `main` function does not change from Figure 7.4.

mutex/prodcons6.c
```
#include    "unpipc.h"

#define MAXNITEMS       1000000
#define MAXNTHREADS         100

        /* globals shared by threads */
int     nitems;                 /* read-only by producer and consumer */
int     buff[MAXNITEMS];
struct {
    pthread_mutex_t mutex;
    int     nput;               /* next index to store */
    int     nval;               /* next value to store */
} put = {
    PTHREAD_MUTEX_INITIALIZER
};

struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int     nready;             /* number ready for consumer */
} nready = {
    PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER
};
```
mutex/prodcons6.c

Figure 7.6 Globals for our producer–consumer, using a condition variable.

The `produce` and `consume` functions do change, and we show them in Figure 7.7.

Place next item into array

50–58 We now use the mutex `put.mutex` to lock the critical section when the producer places a new item into the array.

Notify consumer

59–64 We increment the counter `nready.nready`, which counts the number of items ready for the consumer to process. Before doing this increment, if the value of the counter was 0, we call `pthread_cond_signal` to wake up any threads (e.g., the consumer) that may be waiting for this value to become nonzero. We can now see the interaction of the mutex and condition variable associated with this counter. The counter is shared between the producers and the consumer, so access to it must be when the associated mutex (`nready.mutex`) is locked. The condition variable is used for waiting and signaling.

Consumer waits for `nready.nready` to be nonzero

72–76 The consumer just waits for the counter `nready.nready` to be nonzero. Since this counter is shared among all the producers and the consumer, we can test its value only while we have its associated mutex locked. If, while we have the mutex locked, the value is 0, we call `pthread_cond_wait` to go to sleep. This does two actions atomically:

mutex/prodcons6.c

```
void *
produce(void *arg)
{
    for ( ; ; ) {
        Pthread_mutex_lock(&put.mutex);
        if (put.nput >= nitems) {
            Pthread_mutex_unlock(&put.mutex);
            return (NULL);      /* array is full, we're done */
        }
        buff[put.nput] = put.nval;
        put.nput++;
        put.nval++;
        Pthread_mutex_unlock(&put.mutex);

        Pthread_mutex_lock(&nready.mutex);
        if (nready.nready == 0)
            Pthread_cond_signal(&nready.cond);
        nready.nready++;
        Pthread_mutex_unlock(&nready.mutex);

        *((int *) arg) += 1;
    }
}

void *
consume(void *arg)
{
    int     i;

    for (i = 0; i < nitems; i++) {
        Pthread_mutex_lock(&nready.mutex);
        while (nready.nready == 0)
            Pthread_cond_wait(&nready.cond, &nready.mutex);
        nready.nready--;
        Pthread_mutex_unlock(&nready.mutex);

        if (buff[i] != i)
            printf("buff[%d] = %d\n", i, buff[i]);
    }
    return (NULL);
}
```

mutex/prodcons6.c

Figure 7.7 produce and consume functions.

1. the mutex `nready.mutex` is unlocked, and
2. the thread is put to sleep until some other thread calls `pthread_cond_signal` for this condition variable.

Before returning, `pthread_cond_wait` locks the mutex `nready.mutex`. Therefore, when it returns, and we find the counter nonzero, we decrement the counter (knowing that we have the mutex locked) and then unlock the mutex. Notice that when `pthread_cond_wait` returns, we *always* test the condition again, because *spurious* wakeups can occur: a wakeup when the desired condition is still not true. Implementations try to minimize the number of these spurious wakeups, but they can still occur.

In general, the code that signals a condition variable looks like the following:

```
struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  whatever variables maintain the condition
} var = { PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, ... };

Pthread_mutex_lock(&var.mutex);
set condition true
Pthread_cond_signal(&var.cond);
Pthread_mutex_unlock(&var.mutex);
```

In our example, the variable that maintains the condition was an integer counter, and setting the condition was just incrementing the counter. We added the optimization that the signal occurred only when the counter went from 0 to 1.

The code that tests the condition and goes to sleep waiting for the condition to be true normally looks like the following:

```
Pthread_mutex_lock(&var.mutex);
while (condition is false)
    Pthread_cond_wait(&var.cond, &var.mutex);
modify condition
Pthread_mutex_unlock(&var.mutex);
```

Avoiding Lock Conflicts

In the code fragment just shown, as well as in Figure 7.7, `pthread_cond_signal` is called by the thread that currently holds the mutex lock that is associated with the condition variable being signaled. In a worst-case scenario, we could imagine the system immediately scheduling the thread that is signaled; that thread runs and then immediately stops, because it cannot acquire the mutex. An alternative to our code in Figure 7.7 would be

```
int  dosignal;

Pthread_mutex_lock(&nready.mutex);
dosignal = (nready.nready == 0);
nready.nready++;
Pthread_mutex_unlock(&nready.mutex);

if (dosignal)
    Pthread_cond_signal(&nready.cond);
```

Here we do not signal the condition variable until we release the mutex. This is explicitly allowed by Posix: the thread calling `pthread_cond_signal` need not be the current owner of the mutex associated with the condition variable. But Posix goes on to say that if predictable scheduling behavior is required, then the mutex must be locked by the thread calling `pthread_cond_signal`.

7.6 Condition Variables: Timed Waits and Broadcasts

Normally, `pthread_cond_signal` awakens one thread that is waiting on the condition variable. In some instances, a thread knows that multiple threads should be awakened, in which case, `pthread_cond_broadcast` will wake up *all* threads that are blocked on the condition variable.

An example of a scenario in which multiple threads should be awakened occurs with the readers and writers problem that we describe in Chapter 8. When a writer is finished with a lock, it wants to awaken *all* queued readers, because multiple readers are allowed at the same time.

An alternate (and safer) way of thinking about a signal versus a broadcast is that you can always use a broadcast. A signal is an optimization for the cases in which you know that all the waiters are properly coded, only one waiter needs to be awakened, and which waiter is awakened does not matter. In all other situations, you must use a broadcast.

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cptr);

int pthread_cond_timedwait(pthread_cond_t *cptr, pthread_mutex_t *mptr,
                           const struct timespec *abstime);

                              Both return: 0 if OK, positive Exxx value on error
```

`pthread_cond_timedwait` lets a thread place a limit on how long it will block. *abstime* is a `timespec` structure:

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

This structure specifies the system time when the function must return, even if the condition variable has not been signaled yet. If this timeout occurs, `ETIMEDOUT` is returned.

This time value is an *absolute time*; it is not a *time delta*. That is, *abstime* is the system time—the number of seconds and nanoseconds past January 1, 1970, UTC—when the function should return. This differs from `select`, `pselect`, and `poll` (Chapter 6 of UNPv1), which all specify some number of fractional seconds in the future when the function should return. (`select` specifies microseconds in the future, `pselect` specifies nanoseconds in the future, and `poll` specifies milliseconds in the future.) The advantage in using an absolute time, instead of a delta time, is if the function prematurely returns (perhaps because of a caught signal): the function can be called again, without having to change the contents of the `timespec` structure.

7.7 Mutexes and Condition Variable Attributes

Our examples in this chapter of mutexes and condition variables have stored them as globals in a process in which they are used for synchronization between the threads within that process. We have initialized them with the two constants `PTHREAD_MUTEX_INITIALIZER` and `PTHREAD_COND_INITIALIZER`. Mutexes and condition variables initialized in this fashion assume the default attributes, but we can initialize these with other than the default attributes.

First, a mutex or condition variable is initialized or destroyed with the following functions:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mptr, const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mptr);

int pthread_cond_init(pthread_cond_t *cptr, const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cptr);
```

All four return: 0 if OK, positive Exxx value on error

Considering a mutex, *mptr* must point to a `pthread_mutex_t` variable that has been allocated, and `pthread_mutex_init` initializes that mutex. The `pthread_mutexattr_t` value, pointed to by the second argument to `pthread_mutex_init` (*attr*), specifies the attributes. If this argument is a null pointer, the default attributes are used.

Mutex attributes, a `pthread_mutexattr_t` datatype, and condition variable attributes, a `pthread_condattr_t` datatype, are initialized or destroyed with the following functions:

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

All four return: 0 if OK, positive Exxx value on error

Once a mutex attribute or a condition variable attribute has been initialized, separate functions are called to enable or disable certain attributes. For example, one attribute that we will use in later chapters specifies that the mutex or condition variable is to be shared between different processes, not just between different threads within a single process. This attribute is fetched or stored with the following functions.

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *valptr);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int value);

int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *valptr);

int pthread_condattr_setpshared(pthread_condattr_t *attr, int value);

                                   All four return: 0 if OK, positive Exxx value on error
```

The two `get` functions return the current value of this attribute in the integer pointed to by *valptr* and the two `set` functions set the current value of this attribute, depending on *value*. The *value* is either `PTHREAD_PROCESS_PRIVATE` or `PTHREAD_PROCESS_SHARED`. The latter is also referred to as the process-shared attribute.

This feature is supported only if the constant `_POSIX_THREAD_PROCESS_SHARED` is defined by including `<unistd.h>`. It is an optional feature with Posix.1 but required by Unix 98 (Figure 1.5).

The following code fragment shows how to initialize a mutex so that it can be shared between processes:

```
pthread_mutex_t     *mptr;  /* pointer to the mutex in shared memory */
pthread_mutexattr_t  mattr; /* mutex attribute datatype */

    . . .

    mptr = /* some value that points to shared memory */ ;
    Pthread_mutexattr_init(&mattr);
#ifdef _POSIX_THREAD_PROCESS_SHARED
    Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
#else
# error this implementation does not support _POSIX_THREAD_PROCESS_SHARED
#endif
    Pthread_mutex_init(mptr, &mattr);
```

We declare a `pthread_mutexattr_t` datatype named `mattr`, initialize it to the default attributes for a mutex, and then set the `PTHREAD_PROCESS_SHARED` attribute, which says that the mutex is to be shared between processes. `pthread_mutex_init` then initializes the mutex accordingly. The amount of shared memory that must be allocated for the mutex is `sizeof(pthread_mutex_t)`.

A nearly identical set of statements (replacing the five characters `mutex` with `cond`) is used to set the `PTHREAD_PROCESS_SHARED` attribute for a condition variable that is stored in shared memory for use by multiple processes.

We showed examples of these process-shared mutexes and condition variables in Figure 5.22.

Process Termination While Holding a Lock

When a mutex is shared between processes, there is always a chance that the process can terminate (perhaps involuntarily) while holding the mutex lock. There is no way to have the system automatically release held locks upon process termination. We will see that read–write locks and Posix semaphores share this property. The only type of synchronization locks that the kernel always cleans up automatically upon process termination is `fcntl` record locks (Chapter 9). When using System V semaphores, the application chooses whether a semaphore lock is automatically cleaned up or not by the kernel upon process termination (the `SEM_UNDO` feature that we talk about in Section 11.3).

A thread can also terminate while holding a mutex lock, by being canceled by another thread, or by calling `pthread_exit`. The latter should be of no concern, because the thread should know that it holds a mutex lock if it voluntarily terminates by calling `pthread_exit`. In the case of cancellation, the thread can install cleanup handlers that are called upon cancellation, which we demonstrate in Section 8.5. Fatal conditions for a thread normally result in termination of the entire process. For example, if a thread makes an invalid pointer reference, generating `SIGSEGV`, this terminates the entire process if the signal is not caught, and we are back to the previous condition dealing with the termination of the process.

Even if the system were to release a lock automatically when a process terminates, this may not solve the problem. The lock was protecting a critical region probably while some data was being updated. If the process terminates while it is in the middle of this critical region, what is the state of the data? A good chance exists that the data has some inconsistencies: for example, a new item may have been only partially entered into a linked list. If the kernel were to just unlock the mutex when the process terminates, the next process to use the linked list could find it corrupted.

In some examples, however, having the kernel clean up a lock (or a counter in the case of a semaphore) when the process terminates is OK. For example, a server might use a System V semaphore (with the `SEM_UNDO` feature) to count the number of clients currently being serviced. Each time a child is `fork`ed, it increments this semaphore, and when the child terminates, it decrements this semaphore. If the child terminates abnormally, the kernel will still decrement the semaphore. An example of when it is OK for the kernel to release a lock (not a counter as we just described) is shown in Section 9.7. The daemon obtains a write lock on one of its data files and holds this lock as long as it is running. Should someone try to start another copy of the daemon, the new copy will terminate when it cannot get the write lock, guaranteeing that only one copy of the daemon is ever running. But should the daemon terminate abnormally, the kernel releases the write lock, allowing another copy to be started.

7.8 Summary

Mutexes are used to protect critical regions of code, so that only one thread at a time is executing within the critical region. Sometimes a thread obtains a mutex lock and then discovers that it needs to wait for some condition to be true. When this happens, the thread waits on a condition variable. A condition variable is always associated with a mutex. The `pthread_cond_wait` function that puts the thread to sleep unlocks the mutex before putting the thread to sleep and relocks the mutex before waking up the thread at some later time. The condition variable is signaled by some other thread, and that signaling thread has the option of waking up one thread (`pthread_cond_signal`) or all threads that are waiting for the condition to be true (`pthread_cond_broadcast`).

Mutexes and condition variables can be statically allocated and statically initialized. They can also be dynamically allocated, which requires that they be dynamically initialized. Dynamic initialization allows us to specify the process-shared attribute, allowing the mutex or condition variable to be shared between different processes, assuming that the mutex or condition variable is stored in memory that is shared between the different processes.

Exercises

7.1 Remove the mutex locking from Figure 7.3 and verify that the example fails if more than one producer thread is run.

7.2 What happens in Figure 7.2 if the call to `Pthread_join` for the consumer thread is removed?

7.3 Write a program that just calls `pthread_mutexattr_init` and `pthread_condattr_init` in an infinite loop. Watch the memory usage of the process, using a program such as `ps`. What happens? Now add the appropriate calls to `pthread_mutexattr_destroy` and `pthread_condattr_destroy` and verify that no memory leak occurs.

7.4 In Figure 7.7, the producer calls `pthread_cond_signal` only when the counter `nready.nready` goes from 0 to 1. To see what this optimization does, add a counter each time `pthread_cond_signal` is called, and print this counter in the main thread when the consumer is done.

8

Read-Write Locks

8.1 Introduction

A mutex lock blocks all other threads from entering what we call a *critical region*. This critical region usually involves accessing or updating one or more pieces of data that are shared between the threads. But sometimes, we can distinguish between *reading* a piece of data and *modifying* a piece of data.

We now describe a *read–write lock* and distinguish between obtaining the read–write lock for reading and obtaining the read–write lock for writing. The rules for allocating these read–write locks are:

1. Any number of threads can hold a given read–write lock for reading as long as no thread holds the read–write lock for writing.
2. A read–write lock can be allocated for writing only if no thread holds the read–write lock for reading or writing.

Stated another way, any number of threads can have read access to a given piece of data as long as no thread is modifying that piece of data. A piece of data can be modified only if no other thread is reading or modifying the data.

In some applications, the data is read more often than the data is modified, and these applications can benefit from using read–write locks instead of mutex locks. Allowing multiple readers at any given time can provide more concurrency, while still protecting the data while it is modified from any other readers or writers.

This sharing of access to a given resource is also known as *shared–exclusive* locking, because obtaining a read–write lock for reading is called a *shared lock*, and obtaining a read–write lock for writing is called an *exclusive lock*. Other terms for this type of problem (multiple readers and one writer) are the *readers and writers* problem and *readers–writer* locks. (In the last term, "readers" is intentionally plural, and "writer" is intentionally singular, emphasizing the multiple-readers but single-writer nature of the problem.)

A common analogy for a read–write lock is accessing bank accounts. Multiple threads can be reading the balance of an account at the same time, but as soon as one thread wants to update a given balance, that thread must wait for all readers to finish reading that balance, and then only the updating thread should be allowed to modify the balance. No readers should be allowed to read the balance until the update is complete.

The functions that we describe in this chapter are defined by Unix 98 because read–write locks were not part of the 1996 Posix.1 standard. These functions were developed by a collection of Unix vendors in 1995 known as the Aspen Group, along with other extensions that were not defined by Posix.1. A Posix working group (1003.1j) is currently developing a set of Pthreads extensions that includes read–write locks, which will hopefully be the same as described in this chapter.

8.2 Obtaining and Releasing Read–Write Locks

A read–write lock has a datatype of `pthread_rwlock_t`. If a variable of this type is statically allocated, it can be initialized by assigning to it the constant `PTHREAD_RWLOCK_INITIALIZER`.

`pthread_rwlock_rdlock` obtains a read-lock, blocking the calling thread if the read–write lock is currently held by a writer. `pthread_rwlock_wrlock` obtains a write-lock, blocking the calling thread if the read–write lock is currently held by either another writer or by one or more readers. `pthread_rwlock_unlock` releases either a read lock or a write lock.

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwptr);

int pthread_rwlock_wrlock(pthread_rwlock_t *rwptr);

int pthread_rwlock_unlock(pthread_rwlock_t *rwptr);
```

All return: 0 if OK, positive Exxx value on error

The following two functions try to obtain either a read lock or a write lock, but if the lock cannot be granted, an error of `EBUSY` is returned instead of putting the calling thread to sleep.

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwptr);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwptr);
```

Both return: 0 if OK, positive Exxx value on error

8.3 Read–Write Lock Attributes

We mentioned that a statically allocated read–write lock can be initialized by assigning it the value PTHREAD_RWLOCK_INITIALIZER. These variables can also be dynamically initialized by calling pthread_rwlock_init. When a thread no longer needs a read–write lock, it can call the function pthread_rwlock_destroy.

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwptr,
                        const pthread_rwlockattr_t *attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwptr);
```

Both return: 0 if OK, positive Exxx value on error

When initializing a read–write lock, if *attr* is a null pointer, the default attributes are used. To assign other than these defaults, the following two functions are provided:

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Both return: 0 if OK, positive Exxx value on error

Once an attribute object of datatype pthread_rwlockattr_t has been initialized, separate functions are called to enable or disable certain attributes. The only attribute currently defined is PTHREAD_PROCESS_SHARED, which specifies that the read–write lock is to be shared between different processes, not just between different threads within a single process. The following two functions fetch and set this attribute:

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *valptr);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int value);
```

Both return: 0 if OK, positive Exxx value on error

The first function returns the current value of this attribute in the integer pointed to by *valptr*. The second function sets the current value of this attribute to *value*, which is either PTHREAD_PROCESS_PRIVATE or PTHREAD_PROCESS_SHARED.

8.4 Implementation Using Mutexes and Condition Variables

Read–write locks can be implemented using just mutexes and condition variables. In this section, we examine one possible implementation. Our implementation gives preference to waiting writers. This is not required and there are other alternatives.

This section and the remaining sections of this chapter contain advanced topics that you may want to skip on a first reading.

Other implementations of read–write locks merit study. Section 7.1.2 of [Butenhof 1997] provides an implementation that gives priority to waiting readers and includes cancellation handling (which we say more about shortly). Section B.18.2.3.1 of [IEEE 1996] provides another implementation that gives priority to waiting writers and also includes cancellation handling. Chapter 14 of [Kleiman, Shah, and Smaalders 1996] provides an implementation that gives priority to waiting writers. The implementation shown in this section is from Doug Schmidt's ACE package, `http://www.cs.wustl.edu/~schmidt/ACE.html` (Adaptive Communications Environment). All four implementations use mutexes and condition variables.

`pthread_rwlock_t` Datatype

Figure 8.1 shows our `pthread_rwlock.h` header, which defines the basic `pthread_rwlock_t` datatype and the function prototypes for the functions that operate on read–write locks. Normally, these are found in the `<pthread.h>` header.

my_rwlock/pthread_rwlock.h

```c
#ifndef __pthread_rwlock_h
#define __pthread_rwlock_h

typedef struct {
    pthread_mutex_t rw_mutex;     /* basic lock on this struct */
    pthread_cond_t rw_condreaders;  /* for reader threads waiting */
    pthread_cond_t rw_condwriters;  /* for writer threads waiting */
    int     rw_magic;             /* for error checking */
    int     rw_nwaitreaders;      /* the number waiting */
    int     rw_nwaitwriters;      /* the number waiting */
    int     rw_refcount;
        /* -1 if writer has the lock, else # readers holding the lock */
} pthread_rwlock_t;

#define RW_MAGIC    0x19283746

        /* following must have same order as elements in struct above */
#define PTHREAD_RWLOCK_INITIALIZER  { PTHREAD_MUTEX_INITIALIZER, \
            PTHREAD_COND_INITIALIZER, PTHREAD_COND_INITIALIZER, \
            RW_MAGIC, 0, 0, 0 }

typedef int pthread_rwlockattr_t;   /* dummy; not supported */

        /* function prototypes */
int     pthread_rwlock_destroy(pthread_rwlock_t *);
int     pthread_rwlock_init(pthread_rwlock_t *, pthread_rwlockattr_t *);
int     pthread_rwlock_rdlock(pthread_rwlock_t *);
int     pthread_rwlock_tryrdlock(pthread_rwlock_t *);
int     pthread_rwlock_trywrlock(pthread_rwlock_t *);
int     pthread_rwlock_unlock(pthread_rwlock_t *);
int     pthread_rwlock_wrlock(pthread_rwlock_t *);
```

```
27          /* and our wrapper functions */
28 void     Pthread_rwlock_destroy(pthread_rwlock_t *);
29 void     Pthread_rwlock_init(pthread_rwlock_t *, pthread_rwlockattr_t *);
30 void     Pthread_rwlock_rdlock(pthread_rwlock_t *);
31 int      Pthread_rwlock_tryrdlock(pthread_rwlock_t *);
32 int      Pthread_rwlock_trywrlock(pthread_rwlock_t *);
33 void     Pthread_rwlock_unlock(pthread_rwlock_t *);
34 void     Pthread_rwlock_wrlock(pthread_rwlock_t *);

35 #endif   /* __pthread_rwlock_h */
```
my_rwlock/pthread_rwlock.h

Figure 8.1 Definition of `pthread_rwlock_t` datatype.

3-13 Our `pthread_rwlock_t` datatype contains one mutex, two condition variables, one flag, and three counters. We will see the use of all these in the functions that follow. Whenever we examine or manipulate this structure, we must hold the `rw_mutex`. When the structure is successfully initialized, the `rw_magic` member is set to `RW_MAGIC`. This member is then tested by all the functions to check that the caller is passing a pointer to an initialized lock, and then set to 0 when the lock is destroyed.

Note that `rw_refcount` always indicates the current status of the read–write lock: –1 indicates a write lock (and only one of these can exist at a time), 0 indicates the lock is available, and a value greater than 0 means that many read locks are currently held.

14-17 We also define the static initializer for this datatype.

`pthread_rwlock_init` Function

Our first function, `pthread_rwlock_init`, dynamically initializes a read–write lock and is shown in Figure 8.2.

7-8 We do not support assigning attributes with this function, so we check that the `attr` argument is a null pointer.

9-19 We initialize the mutex and two condition variables that are in our structure. All three counters are set to 0 and `rw_magic` is set to the value that indicates that the structure is initialized.

20-25 If the initialization of the mutex or condition variables fails, we are careful to destroy the initialized objects and return an error.

`pthread_rwlock_destroy` Function

Figure 8.3 shows our `pthread_rwlock_destroy` function, which destroys a read–write lock when the caller is finished with it.

8-13 We first check that the lock is not in use and then call the appropriate destroy functions for the mutex and two condition variables.

my_rwlock/pthread_rwlock_init.c

```
#include    "unpipc.h"
#include    "pthread_rwlock.h"

int
pthread_rwlock_init(pthread_rwlock_t *rw, pthread_rwlockattr_t *attr)
{
    int     result;

    if (attr != NULL)
        return (EINVAL);        /* not supported */

    if ( (result = pthread_mutex_init(&rw->rw_mutex, NULL)) != 0)
        goto err1;
    if ( (result = pthread_cond_init(&rw->rw_condreaders, NULL)) != 0)
        goto err2;
    if ( (result = pthread_cond_init(&rw->rw_condwriters, NULL)) != 0)
        goto err3;
    rw->rw_nwaitreaders = 0;
    rw->rw_nwaitwriters = 0;
    rw->rw_refcount = 0;
    rw->rw_magic = RW_MAGIC;

    return (0);

  err3:
    pthread_cond_destroy(&rw->rw_condreaders);
  err2:
    pthread_mutex_destroy(&rw->rw_mutex);
  err1:
    return (result);            /* an errno value */
}
```

my_rwlock/pthread_rwlock_init.c

Figure 8.2 `pthread_rwlock_init` function: initialize a read–write lock.

my_rwlock/pthread_rwlock_destroy.c

```
#include    "unpipc.h"
#include    "pthread_rwlock.h"

int
pthread_rwlock_destroy(pthread_rwlock_t *rw)
{
    if (rw->rw_magic != RW_MAGIC)
        return (EINVAL);
    if (rw->rw_refcount != 0 ||
        rw->rw_nwaitreaders != 0 || rw->rw_nwaitwriters != 0)
        return (EBUSY);

    pthread_mutex_destroy(&rw->rw_mutex);
    pthread_cond_destroy(&rw->rw_condreaders);
    pthread_cond_destroy(&rw->rw_condwriters);
    rw->rw_magic = 0;

    return (0);
}
```

my_rwlock/pthread_rwlock_destroy.c

Figure 8.3 `pthread_rwlock_destroy` function: destroy a read–write lock.

`pthread_rwlock_rdlock` Function

Our `pthread_rwlock_rdlock` function is shown in Figure 8.4.

my_rwlock/pthread_rwlock_rdlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_rdlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;

 7     if (rw->rw_magic != RW_MAGIC)
 8         return (EINVAL);

 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return (result);

11         /* give preference to waiting writers */
12     while (rw->rw_refcount < 0 || rw->rw_nwaitwriters > 0) {
13         rw->rw_nwaitreaders++;
14         result = pthread_cond_wait(&rw->rw_condreaders, &rw->rw_mutex);
15         rw->rw_nwaitreaders--;
16         if (result != 0)
17             break;
18     }
19     if (result == 0)
20         rw->rw_refcount++;      /* another reader has a read lock */

21     pthread_mutex_unlock(&rw->rw_mutex);
22     return (result);
23 }
```

my_rwlock/pthread_rwlock_rdlock.c

Figure 8.4 `pthread_rwlock_rdlock` function: obtain a read lock.

9-10 Whenever we manipulate the `pthread_rwlock_t` structure, we must lock the `rw_mutex` member.

11-18 We cannot obtain a read lock if (a) the `rw_refcount` is less than 0 (meaning a writer currently holds the lock), or (b) if threads are waiting to obtain a write lock (`rw_nwaitwriters` is greater than 0). If either of these conditions is true, we increment `rw_nwaitreaders` and call `pthread_cond_wait` on the `rw_condreaders` condition variable. We will see shortly that when a read–write lock is unlocked, a check is first made for any waiting writers, and if none exist, then a check is made for any waiting readers. If readers are waiting, the `rw_condreaders` condition variable is broadcast.

19-20 When we get the read lock, we increment `rw_refcount`. The mutex is released.

> A problem exists in this function: if the calling thread blocks in the call to `pthread_cond_wait` and the thread is then canceled, the thread terminates while it holds the mutex lock, and the counter `rw_nwaitreaders` is wrong. The same problem exists in our implementation of `pthread_rwlock_wrlock` in Figure 8.6. We correct these problems in Section 8.5.

pthread_rwlock_tryrdlock Function

Figure 8.5 shows our implementation of `pthread_rwlock_tryrdlock`, the nonblocking attempt to obtain a read lock.

my_rwlock/pthread_rwlock_tryrdlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_tryrdlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;

 7     if (rw->rw_magic != RW_MAGIC)
 8         return (EINVAL);

 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return (result);

11     if (rw->rw_refcount < 0 || rw->rw_nwaitwriters > 0)
12         result = EBUSY;         /* held by a writer or waiting writers */
13     else
14         rw->rw_refcount++;      /* increment count of reader locks */

15     pthread_mutex_unlock(&rw->rw_mutex);
16     return (result);
17 }
```

my_rwlock/pthread_rwlock_tryrdlock.c

Figure 8.5 `pthread_rwlock_tryrdlock` function: try to obtain a read lock.

11-14 If a writer currently holds the lock, or if threads are waiting for a write lock, `EBUSY` is returned. Otherwise, we obtain the lock by incrementing `rw_refcount`.

pthread_rwlock_wrlock Function

Our `pthread_rwlock_wrlock` function is shown in Figure 8.6.

11-17 As long as readers are holding read locks or a writer is holding a write lock (`rw_refcount` is not equal to 0), we must block. To do so, we increment `rw_nwaitwriters` and call `pthread_cond_wait` on the `rw_condwriters` condition variable. We will see that this condition variable is signaled when the read–write lock is unlocked and writers are waiting.

18-19 When we obtain the write lock, we set `rw_refcount` to –1.

pthread_rwlock_trywrlock Function

The nonblocking function `pthread_rwlock_trywrlock` is shown in Figure 8.7.

11-14 If `rw_refcount` is nonzero, the lock is currently held by either a writer or one or more readers (which one does not matter) and `EBUSY` is returned. Otherwise, we obtain the write lock and `rw_refcount` is set to –1.

my_rwlock/pthread_rwlock_wrlock.c

```c
#include    "unpipc.h"
#include    "pthread_rwlock.h"

int
pthread_rwlock_wrlock(pthread_rwlock_t *rw)
{
    int     result;

    if (rw->rw_magic != RW_MAGIC)
        return (EINVAL);

    if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
        return (result);

    while (rw->rw_refcount != 0) {
        rw->rw_nwaitwriters++;
        result = pthread_cond_wait(&rw->rw_condwriters, &rw->rw_mutex);
        rw->rw_nwaitwriters--;
        if (result != 0)
            break;
    }
    if (result == 0)
        rw->rw_refcount = -1;

    pthread_mutex_unlock(&rw->rw_mutex);
    return (result);
}
```

my_rwlock/pthread_rwlock_wrlock.c

Figure 8.6 `pthread_rwlock_wrlock` function: obtain a write lock.

my_rwlock/pthread_rwlock_trywrlock.c

```c
#include    "unpipc.h"
#include    "pthread_rwlock.h"

int
pthread_rwlock_trywrlock(pthread_rwlock_t *rw)
{
    int     result;

    if (rw->rw_magic != RW_MAGIC)
        return (EINVAL);

    if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
        return (result);

    if (rw->rw_refcount != 0)
        result = EBUSY;         /* held by either writer or reader(s) */
    else
        rw->rw_refcount = -1;   /* available, indicate a writer has it */

    pthread_mutex_unlock(&rw->rw_mutex);
    return (result);
}
```

my_rwlock/pthread_rwlock_trywrlock.c

Figure 8.7 `pthread_rwlock_trywrlock` function: try to obtain a write lock.

`pthread_rwlock_unlock` Function

Our final function, `pthread_rwlock_unlock`, is shown in Figure 8.8.

my_rwlock/pthread_rwlock_unlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_unlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;

 7     if (rw->rw_magic != RW_MAGIC)
 8         return (EINVAL);

 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return (result);

11     if (rw->rw_refcount > 0)
12         rw->rw_refcount--;      /* releasing a reader */
13     else if (rw->rw_refcount == -1)
14         rw->rw_refcount = 0;    /* releasing a writer */
15     else
16         err_dump("rw_refcount = %d", rw->rw_refcount);

17         /* give preference to waiting writers over waiting readers */
18     if (rw->rw_nwaitwriters > 0) {
19         if (rw->rw_refcount == 0)
20             result = pthread_cond_signal(&rw->rw_condwriters);
21     } else if (rw->rw_nwaitreaders > 0)
22         result = pthread_cond_broadcast(&rw->rw_condreaders);

23     pthread_mutex_unlock(&rw->rw_mutex);
24     return (result);
25 }
```

my_rwlock/pthread_rwlock_unlock.c

Figure 8.8 `pthread_rwlock_unlock` function: release a read lock or a write lock.

11-16 If `rw_refcount` is currently greater than 0, then a reader is releasing a read lock. If `rw_refcount` is currently –1, then a writer is releasing a write lock.

17-22 If a writer is waiting, the `rw_condwriters` condition variable is signaled if the lock is available (i.e., if the reference count is 0). We know that only one writer can obtain the lock, so `pthread_cond_signal` is called to wake up one thread. If no writers are waiting but one or more readers are waiting, we call `pthread_cond_broadcast` on the `rw_condreaders` condition variable, because all the waiting readers can obtain a read lock. Notice that we do not grant any additional read locks as soon as a writer is waiting; otherwise, a stream of continual read requests could block a waiting writer forever. For this reason, we need two separate `if` tests, and cannot write

```
        /* give preference to waiting writers over waiting readers */
    if (rw->rw_nwaitwriters > 0 && rw->rw_refcount == 0)
        result = pthread_cond_signal(&rw->rw_condwriters);
    } else if (rw->rw_nwaitreaders > 0)
        result = pthread_cond_broadcast(&rw->rw_condreaders);
```

We could also omit the test of `rw->rw_refcount`, but that can result in calls to `pthread_cond_signal` when read locks are still allocated, which is less efficient.

8.5 Thread Cancellation

We alluded to a problem with Figure 8.4 if the calling thread gets blocked in the call to `pthread_cond_wait` and the thread is then canceled. A thread may be *canceled* by any other thread in the same process when the other thread calls `pthread_cancel`, a function whose only argument is the thread ID to cancel.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Returns: 0 if OK, positive Exxx value on error

Cancellation can be used, for example, if multiple threads are started to work on a given task (say finding a record in a database) and the first thread that completes the task then cancels the other tasks. Another example is when multiple threads start on a task and one thread finds an error, necessitating that it and the other threads stop.

To handle the possibility of being canceled, any thread can install (push) and remove (pop) cleanup handlers.

```
#include <pthread.h>

void pthread_cleanup_push(void (*function)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

These handlers are just functions that are called

- when the thread is canceled (by some thread calling `pthread_cancel`), or
- when the thread voluntarily terminates (either by calling `pthread_exit` or returning from its thread start function).

The cleanup handlers can restore any state that needs to be restored, such as unlocking any mutexes or semaphores that the thread currently holds.

The *function* argument to `pthread_cleanup_push` is the address of the function that is called, and *arg* is its single argument. `pthread_cleanup_pop` always removes the function at the top of the cancellation cleanup stack of the calling threads and calls the function if *execute* is nonzero.

We encounter thread cancellation again with Figure 15.31 when we see that a doors server is canceled if the client terminates while a procedure call is in progress.

Example

An example is the easiest way to demonstrate the problem with our implementation in the previous section. Figure 8.9 shows a time line of our test program, and Figure 8.10 shows the program.

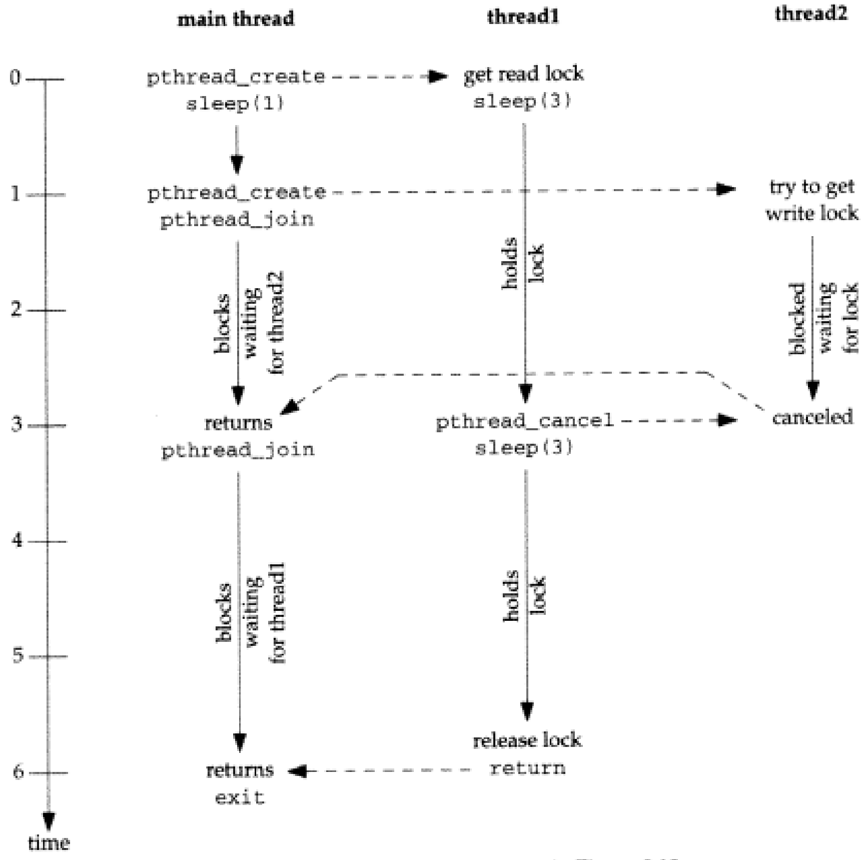

Figure 8.9 Time line of program in Figure 8.10.

Create two threads

10-13 Two threads are created, the first thread executing the function `thread1` and the second executing the function `thread2`. We sleep for a second after creating the first thread, to allow it to obtain a read lock.

Wait for threads to terminate

14-23 We wait for the second thread first, and verify that its status is `PTHREAD_CANCEL`. We then wait for the first thread to terminate and verify that its status is a null pointer. We then print the three counters in the `pthread_rwlock_t` structure and destroy the lock.

my_rwlock_cancel/testcancel.c

```
#include    "unpipc.h"
#include    "pthread_rwlock.h"

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
pthread_t tid1, tid2;
void   *thread1(void *), *thread2(void *);

int
main(int argc, char **argv)
{
    void   *status;

    Set_concurrency(2);
    Pthread_create(&tid1, NULL, thread1, NULL);
    sleep(1);                   /* let thread1() get the lock */
    Pthread_create(&tid2, NULL, thread2, NULL);

    Pthread_join(tid2, &status);
    if (status != PTHREAD_CANCELED)
        printf("thread2 status = %p\n", status);
    Pthread_join(tid1, &status);
    if (status != NULL)
        printf("thread1 status = %p\n", status);

    printf("rw_refcount = %d, rw_nwaitreaders = %d, rw_nwaitwriters = %d\n",
           rwlock.rw_refcount, rwlock.rw_nwaitreaders,
           rwlock.rw_nwaitwriters);
    Pthread_rwlock_destroy(&rwlock);

    exit(0);
}

void *
thread1(void *arg)
{
    Pthread_rwlock_rdlock(&rwlock);
    printf("thread1() got a read lock\n");
    sleep(3);                   /* let thread2 block in pthread_rwlock_wrlock() */
    pthread_cancel(tid2);
    sleep(3);
    Pthread_rwlock_unlock(&rwlock);
    return (NULL);
}

void *
thread2(void *arg)
{
    printf("thread2() trying to obtain a write lock\n");
    Pthread_rwlock_wrlock(&rwlock);
    printf("thread2() got a write lock\n");     /* should not get here */
    sleep(1);
    Pthread_rwlock_unlock(&rwlock);
    return (NULL);
}
```

my_rwlock_cancel/testcancel.c

Figure 8.10 Test program to show thread cancellation.

thread1 function

26-36 This thread obtains a read lock and then sleeps for 3 seconds. This pause allows the other thread to call `pthread_rwlock_wrlock` and block in its call to `pthread_cond_wait`, because a write lock cannot be granted while a read lock is active. The first thread then calls `pthread_cancel` to cancel the second thread, sleeps another 3 seconds, releases its read lock, and terminates.

thread2 function

37-46 The second thread tries to obtain a write lock (which it cannot get, since the first thread has already obtained a read lock). The remainder of this function should never be executed.

If we run this program using the functions from the previous section, we get

```
solaris % testcancel
thread1() got a read lock
thread2() trying to obtain a write lock
```

and we never get back a shell prompt. The program is hung. The following steps have occurred:

1. The second thread calls `pthread_rwlock_wrlock` (Figure 8.6), which blocks in its call to `pthread_cond_wait`.
2. The `sleep(3)` in the first thread returns, and `pthread_cancel` is called.
3. The second thread is canceled (it is terminated). When a thread is canceled while it is blocked in a condition variable wait, the mutex is reacquired before calling the first cancellation cleanup handler. (We have not installed any cancellation cleanup handlers yet, but the mutex is still reacquired before the thread is canceled.) Therefore, when the second thread is canceled, it holds the mutex lock for the read–write lock, and the value of `rw_nwaitwriters` in Figure 8.6 has been incremented.
4. The first thread calls `pthread_rwlock_unlock`, but it blocks forever in its call to `pthread_mutex_lock` (Figure 8.8), because the mutex is still locked by the thread that was canceled.

If we remove the call to `pthread_rwlock_unlock` in our `thread1` function, the main thread will print

```
rw_refcount = 1, rw_nwaitreaders = 0, rw_nwaitwriters = 1
pthread_rwlock_destroy error: Device busy
```

The first counter is 1 because we removed the call to `pthread_rwlock_unlock`, but the final counter is 1 because that is the counter that was incremented by the second thread before it was canceled.

The correction for this problem is simple. First we add two lines of code (preceded by a plus sign) to our `pthread_rwlock_rdlock` function in Figure 8.4 that bracket the call to `pthread_cond_wait`:

```
        rw->rw_nwaitreaders++;
+       pthread_cleanup_push(rwlock_cancelrdwait, (void *) rw);
        result = pthread_cond_wait(&rw->rw_condreaders, &rw->rw_mutex);
+       pthread_cleanup_pop(0);
        rw->rw_nwaitreaders--;
```

The first new line of code establishes a cleanup handler (our `rwlock_cancelrdwait` function), and its single argument will be the pointer `rw`. If `pthread_cond_wait` returns, our second new line of code removes the cleanup handler. The single argument of 0 to `pthread_cleanup_pop` specifies that the handler is not called. If this argument is nonzero, the cleanup handler is first called and then removed.

If the thread is canceled while it is blocked in its call to `pthread_cond_wait`, no return is made from this function. Instead, the cleanup handlers are called (after reacquiring the associated mutex, which we mentioned in step 3 earlier).

Figure 8.11 shows our `rwlock_cancelrdwait` function, which is our cleanup handler for `pthread_rwlock_rdlock`.

my_rwlock_cancel/pthread_rwlock_rdlock.c

```
 3 static void
 4 rwlock_cancelrdwait(void *arg)
 5 {
 6     pthread_rwlock_t *rw;

 7     rw = arg;
 8     rw->rw_nwaitreaders--;
 9     pthread_mutex_unlock(&rw->rw_mutex);
10 }
```

my_rwlock_cancel/pthread_rwlock_rdlock.c

Figure 8.11 `rwlock_cancelrdwait` function: cleanup handler for read lock.

8-9 The counter `rw_nwaitreaders` is decremented and the mutex is unlocked. This is the "state" that was established before the call to `pthread_cond_wait` that must be restored after the thread is canceled.

Our fix to our `pthread_rwlock_wrlock` function in Figure 8.6 is similar. First we add two new lines around the call to `pthread_cond_wait`:

```
        rw->rw_nwaitwriters++;
+       pthread_cleanup_push(rwlock_cancelwrwait, (void *) rw);
        result = pthread_cond_wait(&rw->rw_condwriters, &rw->rw_mutex);
+       pthread_cleanup_pop(0);
        rw->rw_nwaitwriters--;
```

Figure 8.12 shows our `rwlock_cancelwrwait` function, the cleanup handler for a write lock request.

8-9 The counter `rw_nwaitwriters` is decremented and the mutex is unlocked.

my_rwlock_cancel/pthread_rwlock_wrlock.c

```
static void
rwlock_cancelwrwait(void *arg)
{
    pthread_rwlock_t *rw;

    rw = arg;
    rw->rw_nwaitwriters--;
    pthread_mutex_unlock(&rw->rw_mutex);
}
```

my_rwlock_cancel/pthread_rwlock_wrlock.c

Figure 8.12 `rwlock_cancelwrwait` function: cleanup handler for write lock.

If we run our test program from Figure 8.10 with these new functions, the results are now correct.

```
solaris % testcancel
thread1() got a read lock
thread2() trying to obtain a write lock
rw_refcount = 0, rw_nwaitreaders = 0, rw_nwaitwriters = 0
```

The three counts are correct, `thread1` returns from its call to `pthread_rwlock_unlock`, and `pthread_rwlock_destroy` does not return `EBUSY`.

This section has been an overview of thread cancellation. There are more details; see, for example, Section 5.3 of [Butenhof 1997].

8.6 Summary

Read–write locks can provide more concurrency than a plain mutex lock when the data being protected is read more often than it is written. The read–write lock functions defined by Unix 98, which is what we have described in this chapter, or something similar, should appear in a future Posix standard. These functions are similar to the mutex functions from Chapter 7.

Read–write locks can be implemented easily using just mutexes and condition variables, and we have shown a sample implementation. Our implementation gives priority to waiting writers, but some implementations give priority to waiting readers.

Threads may be canceled while they are blocked in a call to `pthread_cond_wait`, and our implementation allowed us to see this occur. We provided a fix for this problem, using cancellation cleanup handlers.

Exercises

8.1 Modify our implementation in Section 8.4 to give preference to readers instead of writers.

8.2 Measure the performance of our implementation in Section 8.4 versus a vendor-provided implementation.

9

Record Locking

9.1 Introduction

The read–write locks described in the previous chapter are allocated in memory as variables of datatype `pthread_rwlock_t`. These variables can be within a single process when the read–write locks are shared among the threads within that process (the default), or within shared memory when the read–write locks are shared among the processes that share that memory (and assuming that the `PTHREAD_PROCESS_SHARED` attribute is specified when the read–write lock is initialized).

This chapter describes an extended type of read–write lock that can be used by related or unrelated processes to share the reading and writing of a file. The file that is being locked is referenced through its descriptor, and the function that performs the locking is `fcntl`. These types of locks are normally maintained within the kernel, and the owner of a lock is identified by its process ID. This means that these locks are for locking between different processes and not for locking between the different threads within one process.

In this chapter, we introduce our sequence-number-increment example. Consider the following scenario, which comes from the Unix print spoolers (the BSD `lpr` command and the System V `lp` command). The process that adds a job to the print queue (to be printed at a later time by another process) must assign a unique sequence number to each print job. The process ID, which is unique while the process is running, cannot be used as the sequence number, because a print job can exist long enough for a given process ID to be reused. A given process can also add multiple print jobs to a queue, and each job needs a unique number. The technique used by the print spoolers is to have a file for each printer that contains the next sequence number to be used. The file is just a single line containing the sequence number in ASCII. Each process that needs to assign a sequence number goes through three steps:

1. it reads the sequence number file,
2. it uses the number, and
3. it increments the number and writes it back.

The problem is that in the time a single process takes to execute these three steps, another process can perform the same three steps. Chaos can result, as we will see in some examples that follow.

> What we have just described is a mutual exclusion problem. It could be solved using mutexes from Chapter 7 or with the read–write locks from Chapter 8. What differs with this problem, however, is that we assume the processes are unrelated, which makes using these techniques harder. We could have the unrelated processes share memory (as we describe in Part 4) and then use some type of synchronization variable in that shared memory, but for unrelated processes, `fcntl` record locking is often easier to use. Another factor is that the problem we described with the line printer spoolers predates the availability of mutexes, condition variables, and read–write locks by many years. Record locking was added to Unix in the early 1980s, before shared memory and threads.

What is needed is for a process to be able to set a *lock* to say that no other process can access the file until the first process is done. Figure 9.2 shows a simple program that does these three steps. The functions `my_lock` and `my_unlock` are called to lock the file at the beginning and unlock the file when the process is done with the sequence number. We will show numerous implementations of these two functions.

20 We print the name by which the program is being run (`argv[0]`) each time around the loop when we print the sequence number, because we use this `main` function with various versions of our locking functions, and we want to see which version is printing the sequence number.

> Printing a process ID requires that we cast the variable of type `pid_t` to a `long` and then print it with the `%ld` format string. The problem is that the `pid_t` type is an integer type, but we do not know its size (`int` or `long`), so we must assume the largest. If we assumed an `int` and used a format string of `%d`, but the type was actually a `long`, the code would be wrong.

To show the results when locking is not used, the functions shown in Figure 9.1 provide no locking at all.

lock/locknone.c

```c
void
my_lock(int fd)
{
    return;
}

void
my_unlock(int fd)
{
    return;
}
```

lock/locknone.c

Figure 9.1 Functions that do no locking.

lock/lockmain.c

```
#include    "unpipc.h"

#define SEQFILE "seqno"         /* filename */

void    my_lock(int), my_unlock(int);

int
main(int argc, char **argv)
{
    int     fd;
    long    i, seqno;
    pid_t   pid;
    ssize_t n;
    char    line[MAXLINE + 1];

    pid = getpid();
    fd = Open(SEQFILE, O_RDWR, FILE_MODE);

    for (i = 0; i < 20; i++) {
        my_lock(fd);            /* lock the file */

        Lseek(fd, 0L, SEEK_SET);    /* rewind before read */
        n = Read(fd, line, MAXLINE);
        line[n] = '\0';         /* null terminate for sscanf */

        n = sscanf(line, "%ld\n", &seqno);
        printf("%s: pid = %ld, seq# = %ld\n", argv[0], (long) pid, seqno);

        seqno++;                /* increment sequence number */

        snprintf(line, sizeof(line), "%ld\n", seqno);
        Lseek(fd, 0L, SEEK_SET);    /* rewind before write */
        Write(fd, line, strlen(line));

        my_unlock(fd);          /* unlock the file */
    }
    exit(0);
}
```

lock/lockmain.c

Figure 9.2 `main` function for file locking example.

If the sequence number in the file is initialized to one, and a single copy of the program is run, we get the following output:

```
solaris % locknone
locknone: pid = 15491, seq# = 1
locknone: pid = 15491, seq# = 2
locknone: pid = 15491, seq# = 3
locknone: pid = 15491, seq# = 4
locknone: pid = 15491, seq# = 5
locknone: pid = 15491, seq# = 6
locknone: pid = 15491, seq# = 7
locknone: pid = 15491, seq# = 8
locknone: pid = 15491, seq# = 9
locknone: pid = 15491, seq# = 10
locknone: pid = 15491, seq# = 11
```

```
locknone: pid = 15491, seq# = 12
locknone: pid = 15491, seq# = 13
locknone: pid = 15491, seq# = 14
locknone: pid = 15491, seq# = 15
locknone: pid = 15491, seq# = 16
locknone: pid = 15491, seq# = 17
locknone: pid = 15491, seq# = 18
locknone: pid = 15491, seq# = 19
locknone: pid = 15491, seq# = 20
```

Notice that the `main` function (Figure 9.2) is in a file named `lockmain.c`, but when we compile and link edit this with the functions that perform no locking (Figure 9.1), we call the executable `locknone`. This is because we will provide other implementations of the two functions `my_lock` and `my_unlock` that use other locking techniques, so we name the executable based on the type of locking that we use.

When the sequence number is again initialized to one, and the program is run twice in the background, we have the following output:

```
solaris % locknone & locknone &
solaris % locknone: pid = 15498, seq# = 1
locknone: pid = 15498, seq# = 2
locknone: pid = 15498, seq# = 3
locknone: pid = 15498, seq# = 4
locknone: pid = 15498, seq# = 5
locknone: pid = 15498, seq# = 6
locknone: pid = 15498, seq# = 7
locknone: pid = 15498, seq# = 8
locknone: pid = 15498, seq# = 9
locknone: pid = 15498, seq# = 10
locknone: pid = 15498, seq# = 11
locknone: pid = 15498, seq# = 12
locknone: pid = 15498, seq# = 13
locknone: pid = 15498, seq# = 14
locknone: pid = 15498, seq# = 15
locknone: pid = 15498, seq# = 16
locknone: pid = 15498, seq# = 17
locknone: pid = 15498, seq# = 18
locknone: pid = 15498, seq# = 19
locknone: pid = 15498, seq# = 20          everything through this line is OK
locknone: pid = 15499, seq# = 1           this is wrong when kernel switches processes
locknone: pid = 15499, seq# = 2
locknone: pid = 15499, seq# = 3
locknone: pid = 15499, seq# = 4
locknone: pid = 15499, seq# = 5
locknone: pid = 15499, seq# = 6
locknone: pid = 15499, seq# = 7
locknone: pid = 15499, seq# = 8
locknone: pid = 15499, seq# = 9
locknone: pid = 15499, seq# = 10
locknone: pid = 15499, seq# = 11
locknone: pid = 15499, seq# = 12
locknone: pid = 15499, seq# = 13
locknone: pid = 15499, seq# = 14
```

```
locknone: pid = 15499, seq# = 15
locknone: pid = 15499, seq# = 16
locknone: pid = 15499, seq# = 17
locknone: pid = 15499, seq# = 18
locknone: pid = 15499, seq# = 19
locknone: pid = 15499, seq# = 20
```

The first thing we notice is that the shell's prompt is output before the first line of output from the program. This is OK and is common when running programs in the background.

The first 20 lines of output are OK and are generated by the first instance of the program (process ID 15498). But a problem occurs with the first line of output from the other instance of the program (process ID 15499): it prints a sequence number of 1, indicating that it probably was started first by the kernel, it read the sequence number file (with a value of 1), and the kernel then switched to the other process. This process only ran again when the other process terminated, and it continued executing with the value of 1 that it had read before the kernel switched processes. This is not what we want. Each process reads, increments, and writes the sequence number file 20 times (there are exactly 40 lines of output), so the ending value of the sequence number should be 40.

What we need is some way to allow a process to prevent other processes from accessing the sequence number file while the three steps are being performed. That is, we need these three steps to be performed as an *atomic operation* with regard to other processes. Another way to look at this problem is that the lines of code between the calls to `my_lock` and `my_unlock` in Figure 9.2 form a *critical region*, as we described in Chapter 7.

When we run two instances of the program in the background as just shown, the output is *nondeterministic*. There is no guarantee that each time we run the two programs we get the same output. This is OK if the three steps listed earlier are handled atomically with regard to other processes, generating an ending value of 40. But this is not OK if the three steps are not handled atomically, often generating an ending value less than 40, which is an error. For example, we do not care whether the first process increments the sequence number from 1 to 20, followed by the second process incrementing it from 21 to 40, or whether each process runs just long enough to increment the sequence number by two (the first process would print 1 and 2, then the next process would print 3 and 4, and so on).

Being nondeterministic does not make it incorrect. Whether the three steps are performed atomically is what makes the program correct or incorrect. Being nondeterministic, however, usually makes debugging these types of programs harder.

9.2 Record Locking versus File Locking

The Unix kernel has no notion whatsoever of records within a file. Any interpretation of records is up to the applications that read and write the file. Nevertheless, the term *record locking* is used to describe the locking features that are provided. But the application specifies a *byte range* within the file to lock or unlock. Whether this byte range has any relationship to one or more logical records within the file is left to the application.

Posix record locking defines one special byte range—a starting offset of 0 (the beginning of the file) and a length of 0—to specify the entire file. Our remaining discussion concerns record locking, with file locking just one special case.

The term *granularity* is used to denote the size of the object that can be locked. With Posix record locking, this granularity is a single byte. Normally the smaller the granularity, the greater the number of simultaneous users allowed. For example, assume five processes access a given file at about the same time, three readers and two writers. Also assume that all five are accessing different records in the file and that each of the five requests takes about the same amount of time, say 1 second. If the locking is done at the file level (the coarsest granularity possible), then all three readers can access their records at the same time, but both writers must wait until the readers are done. Then one writer can modify its record, followed by the other writer. The total time will be about 3 seconds. (We are ignoring lots of details in these timing assumptions, of course.) But if the locking granularity is the record (the finest granularity possible), then all five accesses can proceed simultaneously, since all five are working on different records. The total time would then be only 1 second.

> Berkeley-derived implementations of Unix support *file locking* to lock or unlock an entire file, with no capabilities to lock or unlock a range of bytes within the file. This is provided by the `flock` function.

History

Various techniques have been employed for file and record locking under Unix over the years. Early programs such as UUCP and line printer daemons used various tricks that exploited characteristics of the filesystem implementation. (We describe three of these filesystem techniques in Section 9.8.) These are slow, however, and better techniques were needed for the database systems that were being implemented in the early 1980s.

The first true file and record locking was added to Version 7 by John Bass in 1980, adding a new system call named `locking`. This provided mandatory record locking and was picked up by many versions of System III and Xenix. (We describe the differences between mandatory and advisory locking, and between record locking and file locking later in this chapter.)

4.2BSD provided file locking (not record locking) with its `flock` function in 1983. The 1984 `/usr/group` Standard (one of the predecessors to X/Open) defined the `lockf` function, which provided only exclusive locks (write locks), not shared locks (read locks).

In 1984, System V Release 2 (SVR2) provided advisory record locking through the `fcntl` function. The `lockf` function was also provided, but it was just a library function that called `fcntl`. (Many current systems still provide this implementation of `lockf` using `fcntl`.) In 1986, System V Release 3 (SVR3) added mandatory record locking to `fcntl` using the set-group-ID bit, as we describe in Section 9.5.

The 1988 Posix.1 standard standardized advisory file and record locking with the `fcntl` function, and that is what we describe in this chapter. The X/Open Portability Guide Issue 3 (XPG3, dated 1988) also specifies that record locking is to be provided through the `fcntl` function.

9.3 Posix `fcntl` Record Locking

The Posix interface for record locking is the `fcntl` function.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *arg */ );

                                  Returns: depends on cmd if OK, -1 on error
```

Three values of the *cmd* argument are used with record locking. These three commands require that the third argument, *arg*, be a pointer to an `flock` structure:

```
struct flock {
  short  l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
  short  l_whence;  /* SEEK_SET, SEEK_CUR, SEEK_END */
  off_t  l_start;   /* relative starting offset in bytes */
  off_t  l_len;     /* #bytes; 0 means until end-of-file */
  pid_t  l_pid;     /* PID returned by F_GETLK */
};
```

The three commands are:

`F_SETLK` Obtain (an `l_type` of either `F_RDLCK` or `F_WRLCK`) or release (an `l_type` of `F_UNLCK`) the lock described by the `flock` structure pointed to by *arg*.

If the lock cannot be granted to the process, the function returns immediately (it does not block) with an error of `EACCES` or `EAGAIN`.

`F_SETLKW` This command is similar to the previous command; however, if the lock cannot be granted to the process, the thread blocks until the lock can be granted. (The `W` at the end of this command name means "wait.")

`F_GETLK` Examine the lock pointed to by *arg* to see whether an existing lock would prevent this new lock from being granted. If no lock currently exists that would prevent the new lock from being granted, the `l_type` member of the `flock` structure pointed to by *arg* is set to `F_UNLCK`. Otherwise, information about the existing lock, including the process ID of the process holding the lock, is returned in the `flock` structure pointed to by *arg* (i.e., the contents of the structure are overwritten by this function).

Realize that issuing an `F_GETLK` followed by an `F_SETLK` is not an atomic operation. That is, if we call `F_GETLK` and it sets the `l_type` member to `F_UNLCK` on return, this does not guarantee that an immediate issue of the `F_SETLK` will return success. Another process could run between these two calls and obtain the lock that we want.

The reason that the `F_GETLK` command is provided is to return information about a lock when `F_SETLK` returns an error, allowing us to determine who has the region locked, and how (a read lock or a write lock). But even in this scenario, we must be prepared for the `F_GETLK` command to return that the region is unlocked, because the region can be unlocked between the `F_SETLK` and `F_GETLK` commands.

The `flock` structure describes the type of lock (a read lock or a write lock) and the byte range of the file to lock. As with `lseek`, the starting byte offset is specified as a relative offset (the `l_start` member) and how to interpret that relative offset (the `l_whence` member) as

- `SEEK_SET`: `l_start` relative to the beginning of the file,
- `SEEK_CUR`: `l_start` relative to the current byte offset of the file, and
- `SEEK_END`: `l_start` relative to the end of the file.

The `l_len` member specifies the number of consecutive bytes starting at that offset. A length of 0 means "from the starting offset to the largest possible value of the file offset." Therefore, two ways to lock the entire file are

1. specify an `l_whence` of `SEEK_SET`, an `l_start` of 0, and an `l_len` of 0; or
2. position the file to the beginning using `lseek` and then specify an `l_whence` of `SEEK_CUR`, an `l_start` of 0, and an `l_len` of 0.

The first of these two ways is most common, since it requires a single function call (`fcntl`) instead of two function calls. (See Exercise 9.10 also.)

A lock can be for reading or writing, and at most, one type of lock (read or write) can exist for any byte of a file. Furthermore, a given byte can have multiple read locks but only a single write lock. This corresponds to the read–write locks that we described in the previous chapter. Naturally, an error occurs if we request a read lock when the descriptor was not opened for reading, or request a write lock when the descriptor was not opened for writing.

All locks associated with a file for a given process are removed when a descriptor for that file is closed by that process, or when the process holding the descriptor terminates. Locks are not inherited by a child across a `fork`.

This cleanup of existing locks by the kernel when the process terminates is provided only by `fcntl` record locking and as an option with System V semaphores. The other synchronization techniques that we describe (mutexes, condition variables, read–write locks, and Posix semaphores) do not perform this cleanup on process termination. We talked about this at the end of Section 7.7.

Record locking should not be used with the standard I/O library, because of the internal buffering performed by the library. When a file is being locked, `read` and `write` should be used with the file to avoid problems.

Example

We now return to our example from Figure 9.2 and recode the two functions `my_lock` and `my_unlock` from Figure 9.1 to use Posix record locking. We show these functions in Figure 9.3.

lock/lockfcntl.c

```
#include    "unpipc.h"

void
my_lock(int fd)
{
    struct flock lock;

    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;             /* write lock entire file */

    Fcntl(fd, F_SETLKW, &lock);
}

void
my_unlock(int fd)
{
    struct flock lock;

    lock.l_type = F_UNLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;             /* unlock entire file */

    Fcntl(fd, F_SETLK, &lock);
}
```

lock/lockfcntl.c

Figure 9.3 Posix `fcntl` locking.

Notice that we must specify a write lock, to guarantee only one process at a time updates the sequence number. (See Exercise 9.4.) We also specify a command of `F_SETLKW` when obtaining the lock, because if the lock is not available, we want to block until it is available.

> Given the definition of the `flock` structure shown earlier, we might think we could initialize our structure in `my_lock` as
>
> ```
> static struct flock lock = { F_WRLCK, SEEK_SET, 0, 0, 0 };
> ```
>
> but this is wrong. Posix defines only the required members that must be in a structure, such as `flock`. Implementations can arrange these members in any order, and can also add implementation-specific members.

We do not show the output, but it appears correct. Realize that running our simple program from Figure 9.2 does not let us state that our program works. If the output is wrong, as we have seen, we can say that our program is *not* correct, but running two copies of the program, each looping 20 times is not an adequate test. The kernel could run one program that updates the sequence number 20 times, and then run the other program that updates the sequence number another 20 times. If no switch occurs between the two processes, we might never see the error. A better test is to run the functions from Figure 9.3 with a `main` function that increments the sequence number say, ten thousand times, without printing the value each time through the loop. If we initialize the sequence number to 1 and run 20 copies of this program at the same time, then we expect the ending value of the sequence number file to be 200,001.

Example: Simpler Macros

In Figure 9.3, to request or release a lock takes six lines of code. We must allocate a structure, fill in the structure, and then call `fcntl`. We can simplify our programs by defining the following seven macros, which are from Section 12.3 of APUE:

```
#define read_lock(fd, offset, whence, len) \
                   lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
#define readw_lock(fd, offset, whence, len) \
                    lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
#define write_lock(fd, offset, whence, len) \
                    lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
#define writew_lock(fd, offset, whence, len) \
                     lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
#define un_lock(fd, offset, whence, len) \
                 lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)

#define is_read_lockable(fd, offset, whence, len) \
                          lock_test(fd, F_RDLCK, offset, whence, len)
#define is_write_lockable(fd, offset, whence, len) \
                           lock_test(fd, F_WRLCK, offset, whence, len)
```

These macros use our `lock_reg` and `lock_test` functions, which are shown in Figures 9.4 and 9.5. When using these macros, we need not worry about the structure or the function that is actually called. The first three arguments to these macros are purposely the same as the first three arguments to the `lseek` function.

We also define two wrapper functions, `Lock_reg` and `Lock_test`, which terminate with an error upon an `fcntl` error, along with seven macros whose names also begin with a capital letter that call these two wrapper functions.

Using these macros, our `my_lock` and `my_unlock` functions from Figure 9.3 become

```
#define my_lock(fd)   (Writew_lock(fd, 0, SEEK_SET, 0))
#define my_unlock(fd) (Un_lock(fd, 0, SEEK_SET, 0))
```

lib/lock_reg.c

```
#include    "unpipc.h"

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type;          /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;       /* byte offset, relative to l_whence */
    lock.l_whence = whence;      /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;            /* #bytes (0 means to EOF) */

    return (fcntl(fd, cmd, &lock));      /* -1 upon error */
}
```

lib/lock_reg.c

Figure 9.4 Call `fcntl` to obtain or release a lock.

lib/lock_test.c

```
#include    "unpipc.h"

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type;          /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;       /* byte offset, relative to l_whence */
    lock.l_whence = whence;      /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;            /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) == -1)
        return (-1);             /* unexpected error */

    if (lock.l_type == F_UNLCK)
        return (0);              /* false, region not locked by another proc */
    return (lock.l_pid);         /* true, return positive PID of lock owner */
}
```

lib/lock_test.c

Figure 9.5 Call `fcntl` to test a lock.

9.4 Advisory Locking

Posix record locking is called *advisory locking*. This means the kernel maintains correct knowledge of all files that have been locked by each process, but it does not prevent a process from writing to a file that is read-locked by another process. Similarly, the kernel does not prevent a process from reading from a file that is write-locked by another process. A process can ignore an advisory lock and write to a file that is read-locked, or read from a file that is write-locked, assuming the process has adequate permissions to read or write the file.

Advisory locks are fine for *cooperating processes*. The programming of daemons used by network programming is an example of cooperative processes—the programs that access a shared resource, such as the sequence number file, are all under control of the system administrator. As long as the actual file containing the sequence number is not writable by any process, some random process cannot write to the file while it is locked.

Example: Noncooperating Processes

We can demonstrate that Posix record locking is advisory by running two instances of our sequence number program: one instance (`lockfcntl`) uses the functions from Figure 9.3 and locks the file before incrementing the sequence number, and the other (`locknone`) uses the functions from Figure 9.1 that perform no locking.

```
solaris % lockfcntl & locknone &
lockfcntl: pid = 18816, seq# = 1
lockfcntl: pid = 18816, seq# = 2
lockfcntl: pid = 18816, seq# = 3
```

```
lockfcntl: pid = 18816, seq# = 4
lockfcntl: pid = 18816, seq# = 5
lockfcntl: pid = 18816, seq# = 6
lockfcntl: pid = 18816, seq# = 7
lockfcntl: pid = 18816, seq# = 8
lockfcntl: pid = 18816, seq# = 9
lockfcntl: pid = 18816, seq# = 10
lockfcntl: pid = 18816, seq# = 11
locknone: pid = 18817, seq# = 11          switch processes; error
locknone: pid = 18817, seq# = 12
locknone: pid = 18817, seq# = 13
locknone: pid = 18817, seq# = 14
locknone: pid = 18817, seq# = 15
locknone: pid = 18817, seq# = 16
locknone: pid = 18817, seq# = 17
locknone: pid = 18817, seq# = 18
lockfcntl: pid = 18816, seq# = 12         switch processes; error
lockfcntl: pid = 18816, seq# = 13
lockfcntl: pid = 18816, seq# = 14
lockfcntl: pid = 18816, seq# = 15
lockfcntl: pid = 18816, seq# = 16
lockfcntl: pid = 18816, seq# = 17
lockfcntl: pid = 18816, seq# = 18
lockfcntl: pid = 18816, seq# = 19
lockfcntl: pid = 18816, seq# = 20
locknone: pid = 18817, seq# = 19          switch processes; error
locknone: pid = 18817, seq# = 20
locknone: pid = 18817, seq# = 21
locknone: pid = 18817, seq# = 22
locknone: pid = 18817, seq# = 23
locknone: pid = 18817, seq# = 24
locknone: pid = 18817, seq# = 25
locknone: pid = 18817, seq# = 26
locknone: pid = 18817, seq# = 27
locknone: pid = 18817, seq# = 28
locknone: pid = 18817, seq# = 29
locknone: pid = 18817, seq# = 30
```

Our `lockfcntl` program runs first, but while it is performing the three steps to increment the sequence number from 11 to 12 (and while it holds the lock on the entire file), the kernel switches processes and our `locknone` program runs. This new program reads the sequence number value of 11 before our `lockfcntl` program writes it back to the file. The advisory record lock held by the `lockfcntl` program has no effect on our `locknone` program.

9.5 Mandatory Locking

Some systems provide another type of record locking, called *mandatory locking*. With a mandatory lock, the kernel checks every `read` and `write` request to verify that the operation does not interfere with a lock held by a process. For a normal blocking descriptor, the `read` or `write` that conflicts with a mandatory lock puts the process to sleep until the lock is released. With a nonblocking descriptor, issuing a `read` or `write` that conflicts with a mandatory lock causes an error return of `EAGAIN`.

Posix.1 and Unix 98 define only advisory locking. Many implementations derived from System V, however, provide both advisory and mandatory locking. Mandatory record locking was introduced with System V Release 3.

To enable mandatory locking for a particular file,

- the group-execute bit must be *off*, and
- the set-group-ID bit must be *on*.

Note that having the set-user-ID bit on for a file without having the user-execute bit on also makes no sense, and similarly for the set-group-ID bit and the group-execute bit. Therefore, mandatory locking was added in this way, without affecting any existing user software. New system calls were not required.

On systems that support mandatory record locking, the `ls` command looks for this special combination of bits and prints an `l` or `L` to indicate that mandatory locking is enabled for that file. Similarly, the `chmod` command accepts a specification of `l` to enable mandatory locking for a file.

Example

On a first glance, using mandatory locking should solve the problem of an uncooperating process, since any `read`s or `write`s by the uncooperating process on the locked file will block that process until the lock is released. Unfortunately, the timing problems are more complex, as we can easily demonstrate.

To change our example using `fcntl` to use mandatory locking, all we do is change the permission bits of the `seqno` file. We also run a different version of the `main` function that takes the `for` loop limit from the first command-line argument (instead of using the constant 20) and does not call `printf` each time around the loop.

```
solaris % cat > seqno                          first initialize value to 1
1
^D                                             Control-D is our terminal end-of-file character
solaris % ls -l seqno
-rw-r--r--   1 rstevens other1          2 Oct  7 11:24 seqno
solaris % chmod +l seqno                        enable mandatory locking
solaris % ls -l seqno
-rw-r-lr--   1 rstevens other1          2 Oct  7 11:24 seqno
```

We now start two programs in the background: `loopfcntl` uses `fcntl` locking, and `loopnone` does no locking. We specify a command-line argument of 10,000, which is the number of times that each program reads, increments, and writes the sequence number.

```
solaris % loopfcntl 10000 & loopnone 10000 &    start both programs in the background
solaris % wait                                  wait for both background jobs to finish
solaris % cat seqno                             and look at the sequence number
14378                                           error: should be 20,001
```

Each time we run these two programs, the ending sequence number is normally between 14,000 and 16,000. If the locking worked as desired, the ending value would always be 20,001. To see where the error occurs, we need to draw a time line of the individual steps, which we show in Figure 9.6.

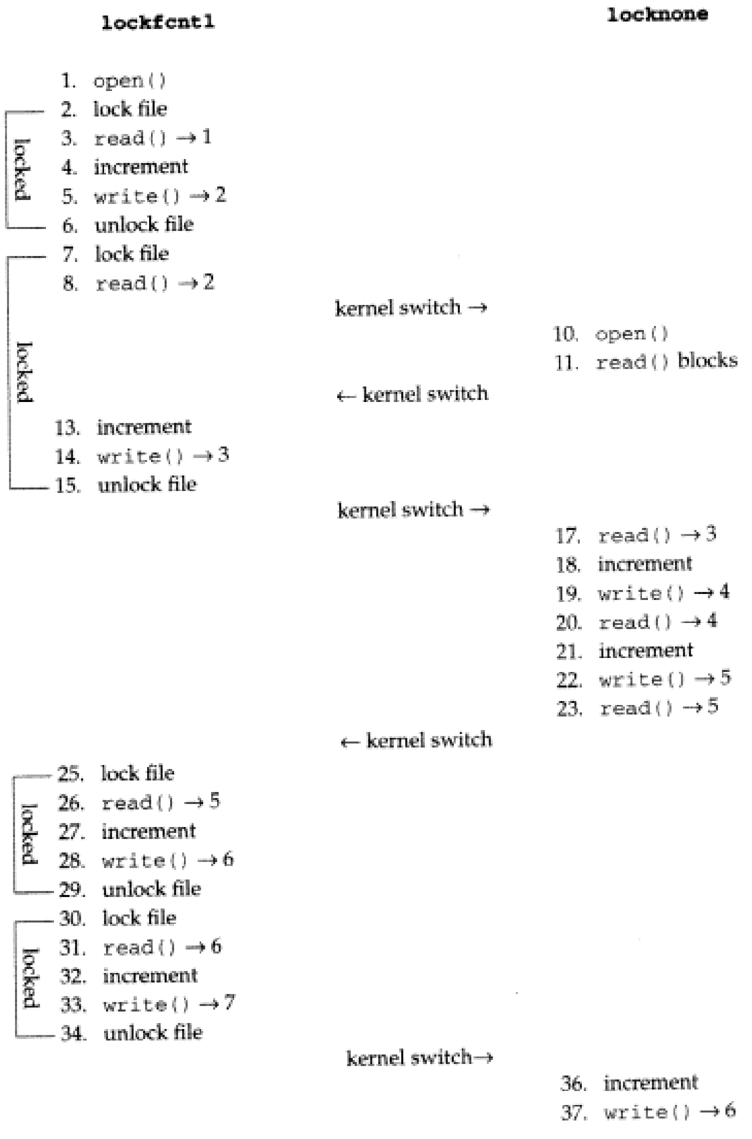

Figure 9.6 Time line of `loopfcntl` and `loopnone` programs.

We assume that the `loopfcntl` program starts first and executes the first eight steps shown in the figure. The kernel then switches processes while `loopfcntl` has a record lock on the sequence number file. `loopnone` is then started, but its first `read` blocks, because the file from which it is reading has an outstanding mandatory lock owned by another process. We assume that the kernel switches back to the first program and it executes steps 13, 14, and 15. This behavior is the type that we expect: the kernel blocks the `read` from the uncooperating process, because the file it is trying to read is locked by another process.

The kernel then switches to the `locknone` program and it executes steps 17 through 23. The `reads` and `writes` are allowed, because the first program unlocked the file in step 15. The problem, however, appears when the program `reads` the value of 5 in step 23 and the kernel then switches to the other process. It obtains the lock and also reads the value of 5. This process increments the value twice, storing a value of 7, before the next process runs in step 36. But the second process writes a value of 6 to the file, which is wrong.

What we see in this example is that mandatory locking prevents a process from reading a file that is locked (step 11), but this does not solve the problem. The problem is that the process on the left is allowed to update the file (steps 25 through 34) while the process on the right is in the middle of its three steps to update the sequence number (steps 23, 36, and 37). If multiple processes are updating a file, *all* the processes must cooperate using some form of locking. One rogue process can create havoc.

9.6 Priorities of Readers and Writers

In our implementation of read–write locks in Section 8.4, we gave priority to waiting writers over waiting readers. We now look at some details of the solution to the readers and writer problem provided by `fcntl` record locking. What we want to look at is how pending lock requests are handled when a region is already locked, something that is not specified by Posix.

Example: Additional Read Locks While a Write Lock Is Pending

The first question we ask is: if a resource is read-locked with a write lock queued, is another read lock allowed? Some solutions to the readers and writers problem do not allow another reader if a writer is already waiting, because if new read requests are continually allowed, a possibility exists that the already pending write request will never be allowed.

To test how `fcntl` record locking handles this scenario, we write a test program that obtains a read lock on an entire file and then `forks` two children. The first child tries to obtain a write lock (and will block, since the parent holds a read lock on the entire file), followed in time by the second child, which tries to obtain a read lock. Figure 9.7 shows a time line of these requests, and Figure 9.8 is our test program.

Parent opens file and obtains read lock

6-8 The parent opens the file and obtains a read lock on the entire file. Notice that we call `read_lock` (which does not block but returns an error if the lock cannot be granted) and not `readw_lock` (which can wait), because we expect this lock to be granted immediately. We also print a message with the current time (our `gf_time` function from p. 404 of UNPv1) when the lock is granted.

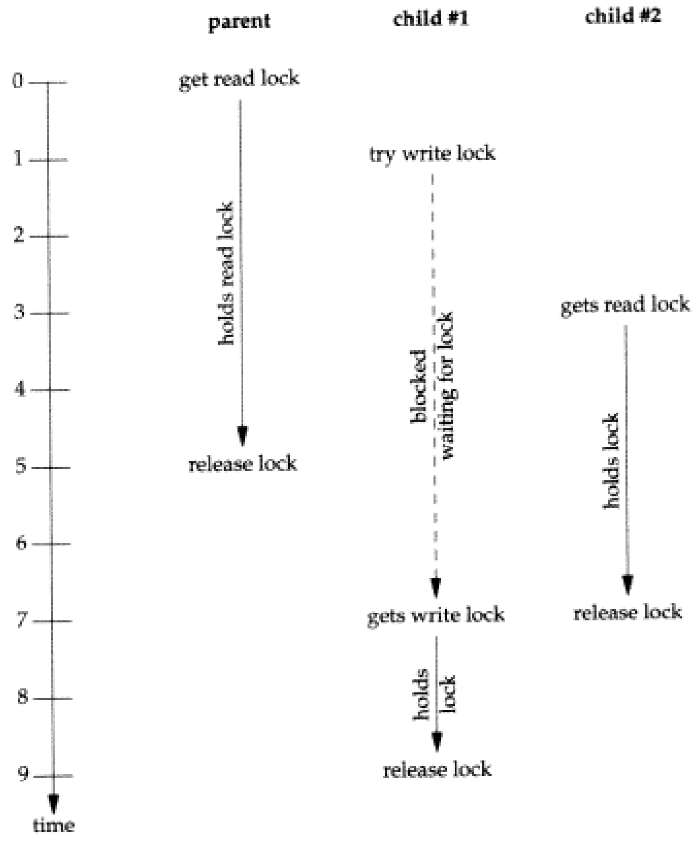

Figure 9.7 Determine whether another read lock is allowed while a write lock is pending.

`fork` first child

9-19 The first child is created and it sleeps for 1 second and then blocks while waiting for a write lock of the entire file. When the write lock is granted, this first child holds the lock for 2 seconds, releases the lock, and terminates.

`fork` second child

20-30 The second child is created, and it sleeps for 3 seconds to allow the first child's write lock to be pending, and then tries to obtain a read lock of the entire file. We can tell by the time on the message printed when `readw_lock` returns whether this read lock is queued or granted immediately. The lock is held for 4 seconds and released.

Parent holds read lock for 5 seconds

31-35 The parent holds the read lock for 5 seconds, releases the lock, and terminates.

lock/test2.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     fd;

    fd = Open("test1.data", O_RDWR | O_CREAT, FILE_MODE);

    Read_lock(fd, 0, SEEK_SET, 0);  /* parent read locks entire file */
    printf("%s: parent has read lock\n", Gf_time());

    if (Fork() == 0) {
            /* first child */
        sleep(1);
        printf("%s: first child tries to obtain write lock\n", Gf_time());
        Writew_lock(fd, 0, SEEK_SET, 0);    /* this should block */
        printf("%s: first child obtains write lock\n", Gf_time());
        sleep(2);
        Un_lock(fd, 0, SEEK_SET, 0);
        printf("%s: first child releases write lock\n", Gf_time());
        exit(0);
    }
    if (Fork() == 0) {
            /* second child */
        sleep(3);
        printf("%s: second child tries to obtain read lock\n", Gf_time());
        Readw_lock(fd, 0, SEEK_SET, 0);
        printf("%s: second child obtains read lock\n", Gf_time());
        sleep(4);
        Un_lock(fd, 0, SEEK_SET, 0);
        printf("%s: second child releases read lock\n", Gf_time());
        exit(0);
    }
        /* parent */
    sleep(5);
    Un_lock(fd, 0, SEEK_SET, 0);
    printf("%s: parent releases read lock\n", Gf_time());
    exit(0);
}
```

lock/test2.c

Figure 9.8 Determine whether another read lock is allowed while a write lock is pending.

The time line shown in Figure 9.7 is what we see under Solaris 2.6, Digital Unix 4.0B, and BSD/OS 3.1. That is, the read lock requested by the second child is granted even though a write lock is already pending from the first child. This allows for potential starvation of write locks as long as read locks are continually issued. Here is the output with some blank lines added between the major time events for readability:

```
alpha % test2
16:32:29.674453: parent has read lock

16:32:30.709197: first child tries to obtain write lock

16:32:32.725810: second child tries to obtain read lock
16:32:32.728739: second child obtains read lock

16:32:34.722282: parent releases read lock

16:32:36.729738: second child releases read lock
16:32:36.735597: first child obtains write lock

16:32:38.736938: first child releases write lock
```

Example: Do Pending Writers Have a Priority Over Pending Readers?

The next question we ask is: do pending writers have a priority over pending readers? Some solutions to the readers and writers problem build in this priority.

Figure 9.9 is our test program and Figure 9.10 is a time line of our test program.

Parent creates file and obtains write lock

6-8 The parent creates the file and obtains a write lock on the entire file.

`fork` and create first child

9-19 The first child is created, and it sleeps for 1 second and then requests a write lock on the entire file. We know this will block, since the parent has a write lock on the entire file and holds this lock for 5 seconds, but we want this request queued when the parent's lock is released.

`fork` and create second child

20-30 The second child is created, and it sleeps for 3 seconds and then requests a read lock on the entire file. This too will be queued when the parent releases its write lock.

Under both Solaris 2.6 and Digital Unix 4.0B, we see that the first child's write lock is granted before the second child's read lock, as we show in Figure 9.10. But this doesn't tell us that write locks have a priority over read locks, because the reason could be that the kernel grants the lock requests in FIFO order, regardless whether they are read locks or write locks. To verify this, we create another test program nearly identical to Figure 9.9, but with the read lock request occurring at time 1 and the write lock request occurring at time 3. These two programs show that Solaris and Digital Unix handle lock requests in a FIFO order, regardless of the type of lock request. These two programs also show that BSD/OS 3.1 gives priority to read requests.

lock/test3.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     fd;

    fd = Open("test1.data", O_RDWR | O_CREAT, FILE_MODE);

    Write_lock(fd, 0, SEEK_SET, 0);     /* parent write locks entire file */
    printf("%s: parent has write lock\n", Gf_time());

    if (Fork() == 0) {
            /* first child */
        sleep(1);
        printf("%s: first child tries to obtain write lock\n", Gf_time());
        Writew_lock(fd, 0, SEEK_SET, 0);    /* this should block */
        printf("%s: first child obtains write lock\n", Gf_time());
        sleep(2);
        Un_lock(fd, 0, SEEK_SET, 0);
        printf("%s: first child releases write lock\n", Gf_time());
        exit(0);
    }
    if (Fork() == 0) {
            /* second child */
        sleep(3);
        printf("%s: second child tries to obtain read lock\n", Gf_time());
        Readw_lock(fd, 0, SEEK_SET, 0);
        printf("%s: second child obtains read lock\n", Gf_time());
        sleep(4);
        Un_lock(fd, 0, SEEK_SET, 0);
        printf("%s: second child releases read lock\n", Gf_time());
        exit(0);
    }
    /* parent */
    sleep(5);
    Un_lock(fd, 0, SEEK_SET, 0);
    printf("%s: parent releases write lock\n", Gf_time());
    exit(0);
}
```

lock/test3.c

Figure 9.9 Test whether writers have a priority over readers.

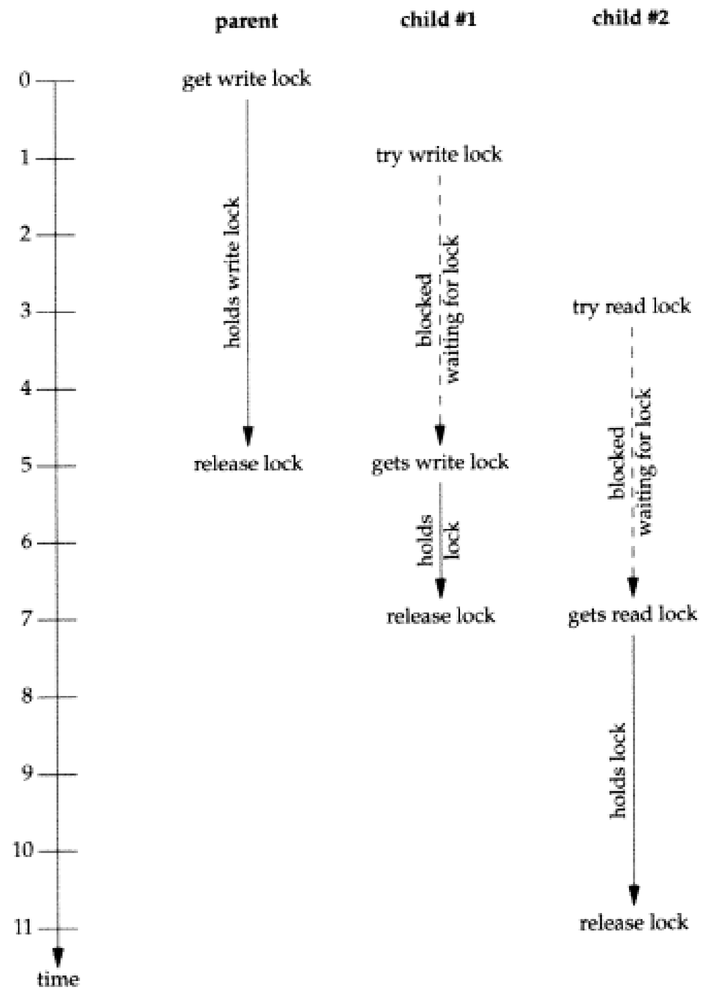

Figure 9.10 Test whether writers have a priority over readers.

Here is the output from Figure 9.9, from which we constructed the time line in Figure 9.10:

```
alpha % test3
16:34:02.810285: parent has write lock

16:34:03.848166: first child tries to obtain write lock

16:34:05.861082: second child tries to obtain read lock

16:34:07.858393: parent releases write lock
16:34:07.865222: first child obtains write lock

16:34:09.865987: first child releases write lock
16:34:09.872823: second child obtains read lock

16:34:13.873822: second child releases read lock
```

9.7 Starting Only One Copy of a Daemon

A common use for record locking is to make certain that only one copy of a program (such as a daemon) is running at a time. The code fragment shown in Figure 9.11 would be executed when a daemon starts.

lock/onedaemon.c

```
 1 #include    "unpipc.h"

 2 #define PATH_PIDFILE    "pidfile"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     pidfd;
 7     char    line[MAXLINE];

 8         /* open the PID file, create if nonexistent */
 9     pidfd = Open(PATH_PIDFILE, O_RDWR | O_CREAT, FILE_MODE);

10         /* try to write lock the entire file */
11     if (write_lock(pidfd, 0, SEEK_SET, 0) < 0) {
12         if (errno == EACCES || errno == EAGAIN)
13             err_quit("unable to lock %s, is %s already running?",
14                     PATH_PIDFILE, argv[0]);
15         else
16             err_sys("unable to lock %s", PATH_PIDFILE);
17     }
18         /* write my PID, leave file open to hold the write lock */
19     snprintf(line, sizeof(line), "%ld\n", (long) getpid());
20     Ftruncate(pidfd, 0);
21     Write(pidfd, line, strlen(line));

22     /* then do whatever the daemon does ... */

23     pause();
24 }
```

lock/onedaemon.c

Figure 9.11 Make certain only one copy of a program is running.

Open and lock a file

8-17 The daemon maintains a 1-line file that contains its process ID. This file is opened, being created if necessary, and then a write lock is requested on the entire file. If the lock is not granted, then we know that another copy of the program is running, and we print an error and terminate.

> Many Unix systems have their daemons write their process ID to a file. Solaris 2.6 stores some of these files in the `/etc` directory. Digital Unix and BSD/OS both store these files in the `/var/run` directory.

Write our PID to file

18-21 We truncate the file to 0 bytes and then write a line containing our PID. The reason for truncating the file is that the previous copy of the program (say before the system was rebooted) might have had a process ID of 23456, whereas this instance of the program has a process ID of 123. If we just wrote the line, without truncating the file, the contents would be `123\n6\n`. While the first line would still contain the process ID, it is cleaner and less confusing to avoid the possibility of a second line in the file.

Here is a test of the program in Figure 9.11:

```
solaris % onedaemon &           start first copy
[1]      22388
solaris % cat pidfile           check PID written to file
22388
solaris % onedaemon             and try to start a second copy
unable to lock pidfile, is onedaemon already running?
```

Other ways exist for a daemon to prevent another copy of itself from being started. A semaphore could also be used. The advantages in the method shown in this section are that many daemons already write their process ID to a file, and should the daemon prematurely crash, the record lock is automatically released by the kernel.

9.8 Lock Files

Posix.1 guarantees that if the `open` function is called with the `O_CREAT` (create the file if it does not already exist) and `O_EXCL` flags (exclusive open), the function returns an error if the file already exists. Furthermore, the check for the existence of the file and the creation of the file (if it does not already exist) must be *atomic* with regard to other processes. We can therefore use the file created with this technique as a lock. We are guaranteed that only one process at a time can create the file (i.e., obtain the lock), and to release the lock, we just `unlink` the file.

Figure 9.12 shows a version of our locking functions using this technique. If the `open` succeeds, we have the lock, and the `my_lock` function returns. We `close` the file because we do not need its descriptor: the lock is the existence of the file, regardless of whether the file is open or not. If `open` returns an error of `EEXIST`, then the file exists and we try the `open` again.

There are three problems with this technique.

1. If the process that currently holds the lock terminates without releasing the lock, the filename is not removed. There are ad hoc techniques to deal with this—check the last-access time of the file and assume it has been orphaned if it is older than some amount of time—but none are perfect. Another technique is to write the process ID of the process holding the lock into the lock file, so that other processes can read this process ID and check whether that process is still running. This is imperfect because process IDs are reused after some time.

   This scenario is not a problem with `fcntl` record locking, because when a process terminates, any record locks held by that process are automatically released.

2. If some other process currently has the file open, we just call `open` again, in an infinite loop. This is called *polling* and is a waste of CPU time. An alternate

lock/lockopen.c

```
#include    "unpipc.h"

#define LOCKFILE    "/tmp/seqno.lock"

void
my_lock(int fd)
{
    int     tempfd;

    while ( (tempfd = open(LOCKFILE, O_RDWR | O_CREAT | O_EXCL, FILE_MODE)) < 0) {
        if (errno != EEXIST)
            err_sys("open error for lock file");
        /* someone else has the lock, loop around and try again */
    }
    Close(tempfd);              /* opened the file, we have the lock */
}

void
my_unlock(int fd)
{
    Unlink(LOCKFILE);           /* release lock by removing file */
}
```

lock/lockopen.c

Figure 9.12 Lock functions using open with O_CREAT and O_EXCL flags.

technique would be to sleep for 1 second, and then try the open again. (We saw this same problem in Figure 7.5.)

This is not a problem with fcntl record locking, assuming that the process that wants the lock specifies the FSETLKW command. The kernel puts the process to sleep until the lock is available and then awakens the process.

3. Creating and deleting a second file by calling open and unlink involves the filesystem and normally takes much longer than calling fcntl twice (once to obtain the lock and once to release the lock). When the time was measured to execute 1000 loops within our program that increments the sequence number, fcntl record locking was faster than calling open and unlink by a factor of 75.

Two other quirks of the Unix filesystem have also been used to provide ad hoc locking. The first is that the link function fails if the name of the new link already exists. To obtain a lock, a unique temporary file is first created whose pathname contains the process ID (or some combination of the process ID and thread ID, if locking is needed between threads in different processes and between threads within the same process). The link function is then called to create a link to this file under the well-known pathname of the lock file. If this succeeds, then the temporary pathname can be unlinked. When the thread is finished with the lock, it just unlinks the well-known pathname. If the link fails with an error of EEXIST, the thread must try again (similar to what we did in Figure 9.12). One requirement of this technique is that the temporary file and the well-known pathname must both reside on the same filesystem, because most versions of Unix do not allow hard links (the result of the `link` function) across different filesystems.

The second quirk is based on `open` returning an error if the file exists, if `O_TRUNC` is specified, and if write permission is denied. To obtain a lock, we call `open`, specifying `O_CREAT | O_WRONLY | O_TRUNC` and a *mode* of 0 (i.e., the new file has no permission bits enabled). If this succeeds, we have the lock and we just `unlink` the pathname when we are done. If `open` fails with an error of `EACCES`, the thread must try again (similar to what we did in Figure 9.12). One caveat is that this trick does not work if the calling thread has superuser privileges.

The lesson from these examples is to use `fcntl` record locking. Nevertheless, you may encounter code that uses these older types of locking, often in programs written before the widespread implementation of `fcntl` locking.

9.9 NFS Locking

NFS is the Network File System and is discussed in Chapter 29 of TCPv1. `fcntl` record locking is an extension to NFS that is supported by most implementations of NFS. Unix systems normally support NFS record locking with two additional daemons: `lockd` and `statd`. When a process calls `fcntl` to obtain a lock, and the kernel detects that the descriptor refers to a file that is on an NFS-mounted filesystem, the local `lockd` sends the request to the server's `lockd`. The `statd` daemon keeps track of the clients holding locks and interacts with `lockd` to provide crash and recovery functions for NFS locking.

We should expect record locking for an NFS file to take longer than record locking for a local file, since network communication is required to obtain and release each lock. To test NFS record locking, all we need to change is the filename specified by `SEQFILE` in Figure 9.2. If we measure the time required for our program to execute 10,000 loops using `fcntl` record locking, it is about 80 times faster for a local file than for an NFS file. Also realize that when the sequence number file is on an NFS-mounted filesystem, network communication is involved for both the record locking and for the reading and writing of the sequence number.

> *Caveat emptor*: NFS record locking has been a problem for many years, and most of the problems have been caused by poor implementations. Despite the fact that the major Unix vendors have finally cleaned up their implementations, using `fcntl` record locking over NFS is still a religious issue for many. We will not take sides on this issue but will just note that `fcntl` record locking is supposed to work over NFS, but your success depends on the quality of the implementations, both client and server.

9.10 Summary

`fcntl` record locking provides advisory or mandatory locking of a file that is referenced through its open descriptor. These locks are for locking between different processes and not for locking between the different threads within one process. The term "record" is a misnomer because the Unix kernel has no concept of records within a file. A better term is "range locking," because we specify a range of bytes within the file to lock or unlock. Almost all uses of this type of record locking are advisory between cooperating processes, because even mandatory locking can lead to inconsistent data, as we showed.

With `fcntl` record locking, there is no guarantee as to the priority of pending readers versus pending writers, which is what we saw in Chapter 8 with read–write locks. If this is important to an application, tests similar to the ones we developed in Section 9.6 should be coded and run, or the application should provide its own read–write locks (as we did in Section 8.4), providing whatever priority is desired.

Exercises

9.1 Build the `locknone` program from Figures 9.2 and 9.1 and run it multiple times on your system. Verify that the program does not work without any locking, and that the results are nondeterministic.

9.2 Modify Figure 9.2 so that the standard output is unbuffered. What effect does this have?

9.3 Continue the previous exercise by also calling `putchar` for every character that is output to standard output, instead of calling `printf`. What effect does this have?

9.4 Change the lock in the `my_lock` function in Figure 9.3 to be a read lock instead of a write lock. What happens?

9.5 Change the call to `open` in the `loopmain.c` program to specify the `O_NONBLOCK` flag also. Build the `loopfcntlnonb` program and run two instances of it at the same time. Does anything change? Why?

9.6 Continue the previous exercise by using the nonblocking version of `loopmain.c` to build the `loopnonenonb` program (using the `locknone.c` file, which performs no locking). Enable the `seqno` file for mandatory locking. Run one instance of this program and another instance of the `loopfcntlnonb` program from the previous exercise at the same time. What happens?

9.7 Build the `loopfcntl` program and run it 10 times in the background from a shell script. Each of the 10 instances should specify a command-line argument of 10,000. First, time the shell script when advisory locking is used, and then change the permissions of the `seqno` file to enable mandatory locking. What effect does mandatory locking have on performance?

9.8 In Figures 9.8 and 9.9, why did we call `fork` to create child processes instead of calling `pthread_create` to create threads?

9.9 In Figure 9.11, we call `ftruncate` to set the size of the file to 0 bytes. Why don't we just specify the `O_TRUNC` flag for `open` instead?

9.10 If we are writing a threaded application that uses `fcntl` record locking, should we use `SEEK_SET`, `SEEK_CUR`, or `SEEK_END` when specifying the starting byte offset to lock, and why?

10

Posix Semaphores

10.1 Introduction

A *semaphore* is a primitive used to provide synchronization between various processes or between the various threads in a given process. We look at three types of semaphores in this text.

- Posix named semaphores are identified by Posix IPC names (Section 2.2) and can be used to synchronize processes or threads.
- Posix memory-based semaphores are stored in shared memory and can be used to synchronize processes or threads.
- System V semaphores (Chapter 11) are maintained in the kernel and can be used to synchronize processes or threads.

For now, we concern ourselves with synchronization between different processes. We first consider a *binary semaphore*: a semaphore that can assume only the values 0 or 1. We show this in Figure 10.1.

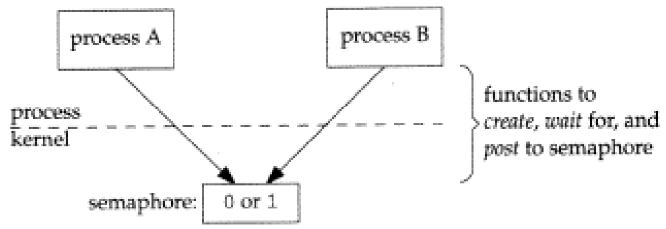

Figure 10.1 A binary semaphore being used by two processes.

We show that the semaphore is maintained by the kernel (which is true for System V semaphores) and that its value can be 0 or 1.

Posix semaphores need not be maintained in the kernel. Also, Posix semaphores are identified by names that might correspond to pathnames in the filesystem. Therefore, Figure 10.2 is a more realistic picture of what is termed a *Posix named semaphore*.

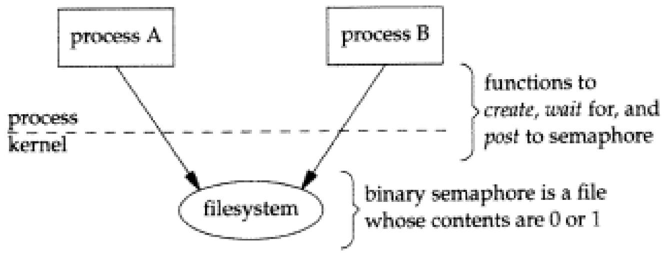
process A
process B
process
kernel
functions to *create*, *wait* for, and *post* to semaphore
filesystem
binary semaphore is a file whose contents are 0 or 1

Figure 10.2 A Posix named binary semaphore being used by two processes.

We must make one qualification with regard to Figure 10.2: although Posix named semaphores are identified by names that might correspond to pathnames in the filesystem, nothing requires that they actually be stored in a file in the filesystem. An embedded realtime system, for example, could use the name to identify the semaphore, but keep the actual semaphore value somewhere in the kernel. But if mapped files are used for the implementation (and we show such an implementation in Section 10.15), then the actual value does appear in a file and that file is mapped into the address space of all the processes that have the semaphore open.

In Figures 10.1 and 10.2, we note three operations that a process can perform on a semaphore:

1. *Create* a semaphore. This also requires the caller to specify the initial value, which for a binary semaphore is often 1, but can be 0.
2. *Wait* for a semaphore. This tests the value of the semaphore, waits (blocks) if the value is less than or equal to 0, and then decrements the semaphore value once it is greater than 0. This can be summarized by the pseudocode

```
while (semaphore_value <= 0)
    ;    /* wait; i.e., block the thread or process */
semaphore_value--;
/* we have the semaphore */
```

The fundamental requirement here is that the test of the value in the `while` statement, and its subsequent decrement (if its value was greater than 0), must be done as an *atomic* operation with respect to other threads or processes accessing this semaphore. (That is one reason System V semaphores were implemented in the mid-1980s within the kernel. Since the semaphore operations were system calls within the kernel, guaranteeing this atomicity with regard to other processes was easy.)

There are other common names for this operation: originally it was called *P* by Edsger Dijkstra, for the Dutch word *proberen* (meaning to try). It is also known as *down* (since the value of the semaphore is being decremented) and *lock*, but we will use the Posix term of *wait*.

3. *Post* to a semaphore. This increments the value of the semaphore and can be summarized by the pseudocode

   ```
   semaphore_value++;
   ```

   If any processes are blocked, waiting for this semaphore's value to be greater than 0, one of those processes can now be awoken. As with the wait code just shown, this post operation must also be atomic with regard to other processes accessing the semaphore.

   There are other common names for this operation: originally it was called *V* for the Dutch word *verhogen* (meaning to increment). It is also known as *up* (since the value of the semaphore is being incremented), *unlock*, and *signal*. We will use the Posix term of *post*.

Obviously, the actual semaphore code has more details than we show in the pseudocode for the wait and post operations: namely how to queue all the processes that are waiting for a given semaphore and then how to wake up one (of the possibly many processes) that is waiting for a given semaphore to be posted to. Fortunately, these details are handled by the implementation.

Notice that the pseudocode shown does not assume a binary semaphore with the values 0 and 1. The code works with semaphores that are initialized to any nonnegative value. These are called *counting semaphores*. These are normally initialized to some value *N*, which indicates the number of resources (say buffers) available. We show examples of both binary semaphores and counting semaphores throughout the chapter.

> We often differentiate between a binary semaphore and a counting semaphore, and we do so for our own edification. No difference exists between the two in the system code that implements a semaphore.

A binary semaphore can be used for mutual exclusion, just like a mutex. Figure 10.3 shows an example.

```
initialize mutex;                              initialize semaphore to 1;

pthread_mutex_lock(&mutex);                    sem_wait(&sem);
critical region                                critical region
pthread_mutex_unlock(&mutex);                  sem_post(&sem);
```

Figure 10.3 Comparison of mutex and semaphore to solve mutual exclusion problem.

We initialize the semaphore to 1. The call to `sem_wait` waits for the value to be greater than 0 and then decrements the value. The call to `sem_post` increments the value (from 0 to 1) and wakes up any threads blocked in a call to `sem_wait` for this semaphore.

Although semaphores can be used like a mutex, semaphores have a feature not provided by mutexes: a mutex must always be unlocked by the thread that locked the mutex, while a semaphore post need not be performed by the same thread that did the semaphore wait. We can show an example of this feature using two binary semaphores and a simplified version of the producer–consumer problem from Chapter 7. Figure 10.4 shows a producer that places an item into a shared buffer and a consumer that removes the item. For simplicity, assume that the buffer holds one item.

producer
shared buffer
consumer

Figure 10.4 Simple producer–consumer problem with a shared buffer.

Figure 10.5 shows the pseudocode for the producer and consumer.

```
              Producer                                      Consumer

initialize semaphore get to 0;
initialize semaphore put to 1;

for ( ; ; ) {                                for ( ; ; ) {
    sem_wait(&put);                              sem_wait(&get);
    put data into buffer                         process data in buffer
    sem_post(&get);                              sem_post(&put);
}                                            }
```

Figure 10.5 Pseudocode for simple producer–consumer.

The semaphore `put` controls whether the producer can place an item into the shared buffer, and the semaphore `get` controls whether the consumer can remove an item from the shared buffer. The steps that occur over time are as follows:

1. The producer initializes the buffer and the two semaphores.
2. Assume that the consumer then runs. It blocks in its call to `sem_wait` because the value of `get` is 0.
3. Sometime later, the producer starts. When it calls `sem_wait`, the value of `put` is decremented from 1 to 0, and the producer places an item into the buffer. It then calls `sem_post` to increment the value of `get` from 0 to 1. Since a thread is blocked on this semaphore (the consumer), waiting for its value to become positive, that thread is marked as ready-to-run. But assume that the producer continues to run. The producer then blocks in its call to `sem_wait` at the top of the `for` loop, because the value of `put` is 0. The producer must wait until the consumer empties the buffer.
4. The consumer returns from its call to `sem_wait`, which decrements the value of the `get` semaphore from 1 to 0. It processes the data in the buffer, and calls `sem_post`, which increments the value of `put` from 0 to 1. Since a thread is blocked on this semaphore (the producer), waiting for its value to become positive, that thread is marked as ready-to-run. But assume that the consumer continues to run. The consumer then blocks in its call to `sem_wait`, at the top of the `for` loop, because the value of `get` is 0.

5. The producer returns from its call to `sem_wait`, places data into the buffer, and this scenario just continues.

We assumed that each time `sem_post` was called, even though a process was waiting and was then marked as ready-to-run, the caller continued. Whether the caller continues or whether the thread that just became ready runs does not matter (you should assume the other scenario and convince yourself of this fact).

We can list three differences among semaphores and mutexes and condition variables.

1. A mutex must always be unlocked by the thread that locked the mutex, whereas a semaphore post need not be performed by the same thread that did the semaphore wait. This is what we just showed in our example.
2. A mutex is either locked or unlocked (a binary state, similar to a binary semaphore).
3. Since a semaphore has state associated with it (its count), a semaphore post is always remembered. When a condition variable is signaled, if no thread is waiting for this condition variable, the signal is lost. As an example of this feature, consider Figure 10.5 but assume that the first time through the producer loop, the consumer has not yet called `sem_wait`. The producer can still put the data item into the buffer, call `sem_post` on the `get` semaphore (incrementing its value from 0 to 1), and then block in its call to `sem_wait` on the `put` semaphore. Some time later, the consumer can enter its `for` loop and call `sem_wait` on the `get` variable, which will decrement the semaphore's value from 1 to 0, and the consumer then processes the buffer.

The Posix.1 Rationale states the following reason for providing semaphores along with mutexes and condition variables: "Semaphores are provided in this standard primarily to provide a means of synchronization for processes; these processes may or may not share memory. Mutexes and condition variables are specified as synchronization mechanisms between threads; these threads always share (some) memory. Both are synchronization paradigms that have been in widespread use for a number of years. Each set of primitives is particularly well matched to certain problems." We will see in Section 10.15 that it takes about 300 lines of C to implement counting semaphores with kernel persistence, using mutexes and condition variables—applications should not have to reinvent these 300 lines of C themselves. Even though semaphores are intended for interprocess synchronization and mutexes and condition variables are intended for interthread synchronization, semaphores can be used between threads and mutexes and condition variables can be used between processes. We should use whichever set of primitives fits the application.

We mentioned that Posix provides two types of semaphores: *named* semaphores and *memory-based* (also called *unnamed*) semaphores. Figure 10.6 compares the functions used for both types of semaphores.

Figure 10.2 illustrated a Posix named semaphore. Figure 10.7 shows a Posix memory-based semaphore within a process that is shared by two threads.

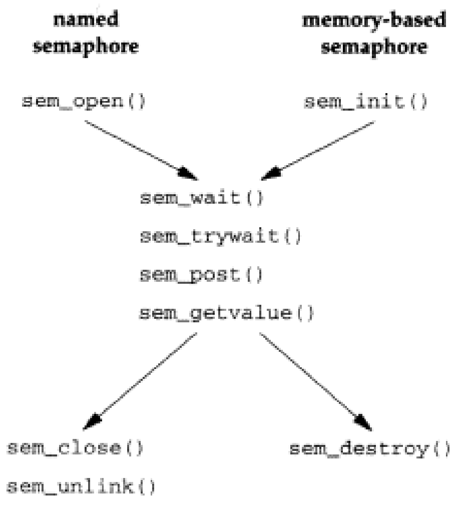

Figure 10.6 Function calls for Posix semaphores.

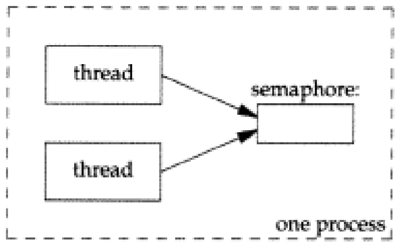

Figure 10.7 Memory-based semaphore shared between two threads within a process.

Figure 10.8 shows a Posix memory-based semaphore in shared memory (Part 4) that is shared by two processes. We show that the shared memory belongs to the address space of both processes.

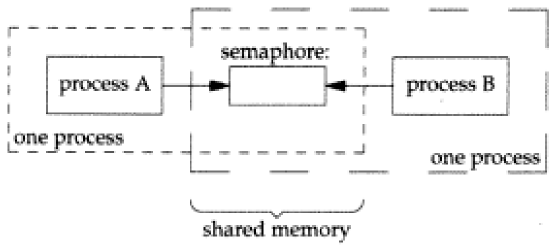

Figure 10.8 Memory-based semaphore in shared memory, shared by two processes.

In this chapter, we first describe Posix named semaphores and then Posix memory-based semaphores. We return to the producer–consumer problem from Section 7.3 and expand it to allow multiple producers with one consumer and finally multiple producers and multiple consumers. We then show that the common I/O technique of multiple buffers is just a special case of the producer–consumer problem.

We show three implementations of Posix named semaphores: the first using FIFOs, the next using memory-mapped I/O with mutexes and condition variables, and the last using System V semaphores.

10.2 sem_open, sem_close, and sem_unlink Functions

The function sem_open creates a new named semaphore or opens an existing named semaphore. A named semaphore can always be used to synchronize either threads or processes.

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );

                     Returns: pointer to semaphore if OK, SEM_FAILED on error
```

We described the rules about the *name* argument in Section 2.2.

The *oflag* argument is either 0, O_CREAT, or O_CREAT | O_EXCL, as described in Section 2.3. If O_CREAT is specified, then the third and fourth arguments are required: *mode* specifies the permission bits (Figure 2.4), and *value* specifies the initial value of the semaphore. This initial value cannot exceed SEM_VALUE_MAX, which must be at least 32767. Binary semaphores usually have an initial value of 1, whereas counting semaphores often have an initial value greater than 1.

If O_CREAT is specified (without specifying O_EXCL), the semaphore is initialized only if it does not already exist. Specifying O_CREAT if the semaphore already exists is not an error. This flag just means "create and initialize the semaphore if it does not already exist." But specifying O_CREAT | O_EXCL is an error if the semaphore already exists.

The return value is a pointer to a sem_t datatype. This pointer is then used as the argument to sem_close, sem_wait, sem_trywait, sem_post, and sem_getvalue.

The return value of SEM_FAILED to indicate an error is strange. A null pointer would make more sense. Earlier drafts that led to the Posix standard specified a return value of –1 to indicate an error, and many implementations define

```
#define  SEM_FAILED   ((sem_t *)(-1))
```

Posix.1 says little about the permission bits associated with a semaphore when it is created or opened by sem_open. Indeed, notice from Figure 2.3 and our discussion above that we do not even specify O_RDONLY, O_WRONLY, or O_RDWR in the *oflag* argument when opening a named semaphore. The two systems used for the examples in this book, Digital Unix 4.0B and Solaris 2.6, both require read access and write access to an existing semaphore for sem_open to succeed. The reason is probably that the two semaphore operations—post and wait—both read and change the value of the semaphore. Not having either read access or write access for an existing semaphore on these two implementations causes the sem_open function to return an error of EACCES ("Permission denied").

A named semaphore that was opened by `sem_open` is closed by `sem_close`.

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns: 0 if OK, –1 on error

This semaphore close operation also occurs *automatically* on process termination for any named semaphore that is still open. This happens whether the process terminates voluntarily (by calling `exit` or `_exit`), or involuntarily (by being killed by a signal).

Closing a semaphore does not remove the semaphore from the system. That is, Posix named semaphores are at least *kernel-persistent*: they retain their value even if no process currently has the semaphore open.

A named semaphore is removed from the system by `sem_unlink`.

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Returns: 0 if OK, –1 on error

Semaphores have a reference count of how many times they are currently open (just like files), and this function is similar to the `unlink` function for a file: the *name* can be removed from the filesystem while its reference count is greater than 0, but the destruction of the semaphore (versus removing its name from the filesystem) does not take place until the last `sem_close` occurs.

10.3 `sem_wait` and `sem_trywait` Functions

The `sem_wait` function tests the value of the specified semaphore, and if the value is greater than 0, the value is decremented and the function returns immediately. If the value is 0 when the function is called, the calling thread is put to sleep until the semaphore value is greater than 0, at which time it will be decremented, and the function then returns. We mentioned earlier that the "test and decrement" operation must be atomic with regard to other threads accessing this semaphore.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);
```

Both return: 0 if OK, –1 on error

The difference between sem_wait and sem_trywait is that the latter does not put the calling thread to sleep if the current value of the semaphore is already 0. Instead, an error of EAGAIN is returned.

sem_wait can return prematurely if it is interrupted by a signal, returning an error of EINTR.

10.4 sem_post and sem_getvalue Functions

When a thread is finished with a semaphore, it calls sem_post. As discussed in Section 10.1, this increments the value of the semaphore by 1 and wakes up any threads that are waiting for the semaphore value to become positive.

```
#include <semaphore.h>

int sem_post(sem_t *sem);

int sem_getvalue(sem_t *sem, int *valp);

                                        Both return: 0 if OK, -1 on error
```

sem_getvalue returns the current value of the semaphore in the integer pointed to by *valp*. If the semaphore is currently locked, then the value returned is either 0 or a negative number whose absolute value is the number of threads waiting for the semaphore to be unlocked.

We now see more differences among mutexes, condition variables, and semaphores. First, a mutex must always be unlocked by the thread that locked the mutex. Semaphores do not have this restriction: one thread can wait for a given semaphore (say, decrementing the semaphore's value from 1 to 0, which is the same as locking the semaphore), and another thread can post to the semaphore (say, incrementing the semaphore's value from 0 to 1, which is the same as unlocking the semaphore).

Second, since a semaphore has an associated value that is incremented by a post and decremented by a wait, a thread can post to a semaphore (say, incrementing its value from 0 to 1), even though no threads are waiting for the semaphore value to become positive. But if a thread calls pthread_cond_signal and no thread is currently blocked in a call to pthread_cond_wait, the signal is lost.

Lastly, of the various synchronization techniques—mutexes, condition variables, read–write locks, and semaphores—the only function that can be called from a signal handler is sem_post.

These three points should not be interpreted as a bias by the author towards semaphores. All the synchronization primitives that we have looked at—mutexes, condition variables, read–write locks, semaphores, and record locking—have their place. We have many choices for a given application and need to be aware of the differences between the various primitives. Also realize in the comparison just listed that mutexes are optimized for locking, condition variables are optimized for waiting, and a semaphore can do both, which may bring with it more overhead and complication.

10.5 Simple Programs

We now provide some simple programs that operate on Posix named semaphores, to learn more about their functionality and implementation. Since Posix named semaphores have at least kernel persistence, we can manipulate them across multiple programs.

semcreate Program

Figure 10.9 creates a named semaphore, allowing a -e option to specify an exclusive-create, and a -i option to specify an initial value (other than the default of 1).

pxsem/semcreate.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, flags;
 6     sem_t  *sem;
 7     unsigned int value;

 8     flags = O_RDWR | O_CREAT;
 9     value = 1;
10     while ( (c = Getopt(argc, argv, "ei:")) != -1) {
11         switch (c) {
12         case 'e':
13             flags |= O_EXCL;
14             break;

15         case 'i':
16             value = atoi(optarg);
17             break;
18         }
19     }
20     if (optind != argc - 1)
21         err_quit("usage: semcreate [ -e ] [ -i initalvalue ] <name>");

22     sem = Sem_open(argv[optind], flags, FILE_MODE, value);

23     Sem_close(sem);
24     exit(0);
25 }
```

pxsem/semcreate.c

Figure 10.9 Create a named semaphore.

Create semaphore

22 Since we always specify the O_CREAT flag, we must call sem_open with four arguments. The final two arguments, however, are used by sem_open only if the semaphore does not already exist.

Close semaphore

23 We call sem_close, although if this call were omitted, the semaphore is still closed (and the system resources released) when the process terminates.

`semunlink` Program

The program in Figure 10.10 unlinks a named semaphore.

pxsem/semunlink.c

```
 1 #include    "unpipc.h"

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

 7     Sem_unlink(argv[1]);

 8     exit(0);
 9 }
```

pxsem/semunlink.c

Figure 10.10 Unlink a named semaphore.

`semgetvalue` Program

Figure 10.11 is a simple program that opens a named semaphore, fetches its current value, and prints that value.

pxsem/semgetvalue.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     sem_t  *sem;
 6     int     val;

 7     if (argc != 2)
 8         err_quit("usage: semgetvalue <name>");

 9     sem = Sem_open(argv[1], 0);
10     Sem_getvalue(sem, &val);
11     printf("value = %d\n", val);

12     exit(0);
13 }
```

pxsem/semgetvalue.c

Figure 10.11 Get and print a semaphore's value.

Open semaphore

9 When we are opening a semaphore that must already exist, the second argument to `sem_open` is 0: we do not specify `O_CREAT` and there are no other `O_`*`xxx`* constants to specify.

`semwait` Program

The program in Figure 10.12 opens a named semaphore, calls `sem_wait` (which will block if the semaphore's value is currently less than or equal to 0, and then decrements the semaphore value), fetches and prints the semaphore's value, and then blocks forever in a call to `pause`.

pxsem/semwait.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    sem_t  *sem;
    int     val;

    if (argc != 2)
        err_quit("usage: semwait <name>");

    sem = Sem_open(argv[1], 0);
    Sem_wait(sem);
    Sem_getvalue(sem, &val);
    printf("pid %ld has semaphore, value = %d\n", (long) getpid(), val);

    pause();                    /* blocks until killed */
    exit(0);
}
```

pxsem/semwait.c

Figure 10.12 Wait for a semaphore and print its value.

`sempost` Program

Figure 10.13 is a program that posts to a named semaphore (i.e., increments its value by one) and then fetches and prints the semaphore's value.

pxsem/sempost.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    sem_t  *sem;
    int     val;

    if (argc != 2)
        err_quit("usage: sempost <name>");

    sem = Sem_open(argv[1], 0);
    Sem_post(sem);
    Sem_getvalue(sem, &val);
    printf("value = %d\n", val);

    exit(0);
}
```

pxsem/sempost.c

Figure 10.13 Post to a semaphore.

Examples

We first create a named semaphore under Digital Unix 4.0B and print its (default) value.

```
alpha % semcreate /tmp/test1
alpha % ls -l /tmp/test1
-rw-r--r--   1 rstevens system       264 Nov 13 08:51 /tmp/test1
alpha % semgetvalue /tmp/test1
value = 1
```

As with Posix message queues, the system creates a file in the filesystem corresponding to the name that we specify for the named semaphore.

We now wait for the semaphore and then abort the program that holds the semaphore lock.

```
alpha % semwait /tmp/test1
pid 9702 has semaphore, value = 0      the value after sem_wait returns
^?                                     type our interrupt key to abort program
alpha % semgetvalue /tmp/test1
value = 0                              and value remains 0
```

This example shows two features that we mentioned earlier. First, the value of a semaphore is kernel-persistent. That is, the semaphore's value of 1 is maintained by the kernel from when the semaphore was created in our previous example, even though no program had the semaphore open during this time. Second, when we abort our `semwait` program that holds the semaphore lock, the value of the semaphore does not change. That is, the semaphore is not unlocked by the kernel when a process holding the lock terminates without releasing the lock. This differs from record locks, which we said in Chapter 9 are automatically released when the process holding the lock terminates without releasing the lock.

We now show that this implementation uses a negative semaphore value to indicate the number of processes waiting for the semaphore to be unlocked.

```
alpha % semgetvalue /tmp/test1
value = 0                              value is still 0 from previous example

alpha % semwait /tmp/test1 &           start in the background
[1]     9718                           it blocks, waiting for semaphore

alpha % semgetvalue /tmp/test1
value = -1                             one process waiting for semaphore

alpha % semwait /tmp/test1 &           start another in the background
[2]     9727                           it also blocks, waiting for semaphore

alpha % semgetvalue /tmp/test1
value = -2                             two processes waiting for semaphore

alpha % sempost /tmp/test1             now post to semaphore
value = -1                             value after sem_post returns
pid 9718 has semaphore, value = -1     output from semwait program

alpha % sempost /tmp/test1             post again to semaphore
value = 0
pid 9727 has semaphore, value = 0      output from other semwait program
```

When the value is –2 and we execute our `sempost` program, the value is incremented to –1 and one of the processes blocked in the call to `sem_wait` returns.

We now execute the same example under Solaris 2.6 to see the differences in the implementation.

```
solaris % semcreate /test2
solaris % ls -l /tmp/.*test2*
-rw-r--r--   1 rstevens other1    48 Nov 13 09:11 /tmp/.SEMDtest2
-rw-rw-rw-   1 rstevens other1     0 Nov 13 09:11 /tmp/.SEMLtest2
solaris % semgetvalue /test2
value = 1
```

As with Posix message queues, files are created in the `/tmp` directory containing the specified name as the filename suffixes. We see that the permissions on the first file correspond to the permissions specified in our call to `sem_open`, and we guess that the second file is used for locking.

We now verify that the kernel does not automatically post to a semaphore when the process holding the semaphore lock terminates without releasing the lock.

```
solaris % semwait /test2
pid 4133 has semaphore, value = 0
^?                                     type our interrupt key
solaris % semgetvalue /test2
value = 0                              value remains 0
```

Next we see how this implementation handles the semaphore value when processes are waiting for the semaphore.

```
solaris % semgetvalue /test2
value = 0                              value is still 0 from previous example

solaris % semwait /test2 &             start in the background
[1]     4257                           it blocks, waiting for semaphore

solaris % semgetvalue /test2
value = 0                              this implementation does not use negative values

solaris % semwait /test2 &             start another in the background
[2]     4263

solaris % semgetvalue /test2
value = 0                              value remains 0 with two processes waiting

solaris % sempost /test2               now post to semaphore
pid 4257 has semaphore, value = 0      output from semwait program
value = 0

solaris % sempost /test2
pid 4263 has semaphore, value = 0      output from other semwait program
value = 0
```

One difference in this output, compared to the previous output under Digital Unix, is when the semaphore is posted to: it appears that the waiting process runs before the process that posted to the semaphore.

10.6 Producer–Consumer Problem

In Section 7.3, we described the *producer–consumer* problem and showed some solutions in which multiple producer threads filled an array that was processed by one consumer thread.

1. In our first solution (Section 7.2), the consumer started only after the producers were finished, and we were able to solve this synchronization problem using a single mutex (to synchronize the producers).
2. In our next solution (Section 7.5), the consumer started before the producers were finished, and this required a mutex (to synchronize the producers) along with a condition variable and its mutex (to synchronize the consumer with the producers).

We now extend the producer–consumer problem by using the shared buffer as a circular buffer: after the producer fills the final entry (`buff[NBUFF-1]`), it goes back and fills the first entry (`buff[0]`), and the consumer does the same. This adds another synchronization problem in that the producer must not get ahead of the consumer. We still assume that the producer and consumer are threads, but they could also be processes, assuming that some way existed to share the buffer between the processes (e.g., shared memory, which we describe in Part 4).

Three conditions must be maintained by the code when the shared buffer is considered as a circular buffer:

1. The consumer cannot try to remove an item from the buffer when the buffer is empty.
2. The producer cannot try to place an item into the buffer when the buffer is full.
3. Shared variables may describe the current state of the buffer (indexes, counts, linked list pointers, etc.), so all buffer manipulations by the producer and consumer must be protected to avoid any race conditions.

Our solution using semaphores demonstrates three different types of semaphores:

1. A binary semaphore named `mutex` protects the critical regions: inserting a data item into the buffer (for the producer) and removing a data item from the buffer (for the consumer). A binary semaphore that is used as a mutex is initialized to 1. (Obviously, we could use a real mutex for this, instead of a binary semaphore. See Exercise 10.10.)
2. A counting semaphore named `nempty` counts the number of empty slots in the buffer. This semaphore is initialized to the number of slots in the buffer (`NBUFF`).
3. A counting semaphore named `nstored` counts the number of filled slots in the buffer. This semaphore is initialized to 0, since the buffer is initially empty.

Figure 10.14 shows the status of our buffer and the two counting semaphores when the program has finished its initialization. We have shaded the array elements that are unused.

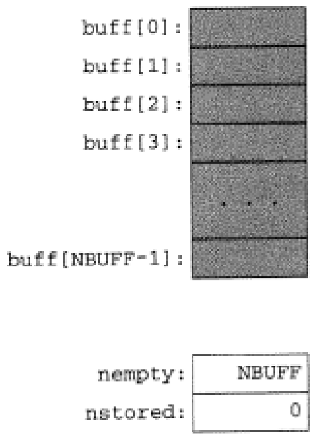

Figure 10.14 Buffer and the two counting semaphores after initialization.

In our example, the producer just stores the integers 0 through `NLOOP-1` into the buffer (`buff[0] = 0`, `buff[1] = 1`, and so on), using the buffer as a circular buffer. The consumer takes these integers from the buffer and verifies that they are correct, printing any errors to standard output.

Figure 10.15 shows the buffer and the counting semaphores after the producer has placed three items into the buffer, but before the consumer has taken any of these items from the buffer.

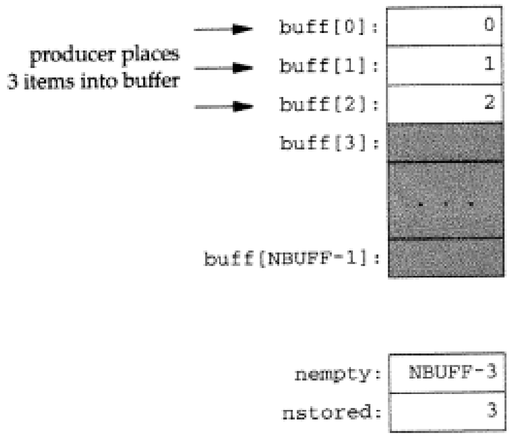

Figure 10.15 Buffer and semaphores after three items placed into buffer by producer.

We next assume that the consumer removes one item from the buffer, and we show this in Figure 10.16.

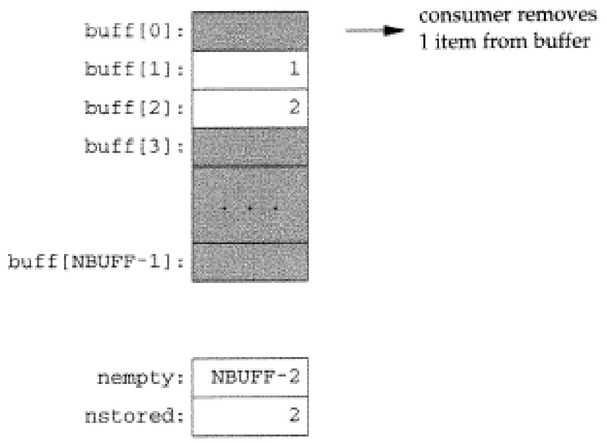

Figure 10.16 Buffer and semaphores after consumer removes first item from buffer.

Figure 10.17 is the `main` function that creates the three semaphores, creates two threads, waits for both threads to complete, and then removes the semaphores.

Globals

6–10 The buffer containing `NBUFF` items is shared between the two threads, as are the three semaphore pointers. As described in Chapter 7, we collect these into a structure to reiterate that the semaphores are used to synchronize access to the buffer.

Create semaphores

19–25 Three semaphores are created and their names are passed to our `px_ipc_name` function. We specify the `O_EXCL` flag because we need to initialize each semaphore to the correct value. If any of the three semaphores are still lying around from a previous run of this program that aborted, we could handle that by calling `sem_unlink` for each semaphore, ignoring any errors, before creating the semaphores. Alternately, we could check for an error of `EEXIST` from `sem_open` with the `O_EXCL` flag, and call `sem_unlink` followed by another call to `sem_open`, but this is more complicated. If we need to verify that only one copy of this program is running (which we could do before trying to create any of the semaphores), we would do so as described in Section 9.7.

Create two threads

26–29 The two threads are created, one as the producer and one as the consumer. No arguments are passed to the two threads.

30–36 The main thread then waits for both threads to terminate, and removes the three semaphores.

We could also call `sem_close` for each semaphore, but this happens automatically when the process terminates. Removing the name of a named semaphore, however, must be done explicitly.

Figure 10.18 shows the `produce` and `consume` functions.

pxsem/prodcons1.c

```
#include    "unpipc.h"

#define NBUFF    10
#define SEM_MUTEX   "mutex"     /* these are args to px_ipc_name() */
#define SEM_NEMPTY  "nempty"
#define SEM_NSTORED "nstored"

int     nitems;                  /* read-only by producer and consumer */
struct {                         /* data shared by producer and consumer */
    int     buff[NBUFF];
    sem_t  *mutex, *nempty, *nstored;
} shared;

void   *produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
    pthread_t tid_produce, tid_consume;

    if (argc != 2)
        err_quit("usage: prodcons1 <#items>");
    nitems = atoi(argv[1]);

        /* create three semaphores */
    shared.mutex = Sem_open(Px_ipc_name(SEM_MUTEX), O_CREAT | O_EXCL,
                            FILE_MODE, 1);
    shared.nempty = Sem_open(Px_ipc_name(SEM_NEMPTY), O_CREAT | O_EXCL,
                             FILE_MODE, NBUFF);
    shared.nstored = Sem_open(Px_ipc_name(SEM_NSTORED), O_CREAT | O_EXCL,
                              FILE_MODE, 0);

        /* create one producer thread and one consumer thread */
    Set_concurrency(2);
    Pthread_create(&tid_produce, NULL, produce, NULL);
    Pthread_create(&tid_consume, NULL, consume, NULL);

        /* wait for the two threads */
    Pthread_join(tid_produce, NULL);
    Pthread_join(tid_consume, NULL);

        /* remove the semaphores */
    Sem_unlink(Px_ipc_name(SEM_MUTEX));
    Sem_unlink(Px_ipc_name(SEM_NEMPTY));
    Sem_unlink(Px_ipc_name(SEM_NSTORED));
    exit(0);
}
```

pxsem/prodcons1.c

Figure 10.17 `main` function for semaphore solution to producer–consumer problem.

Producer waits until room for one item in buffer

44 The producer calls `sem_wait` on the `nempty` semaphore, to wait until room is available for another item in the buffer. The first time this statement is executed, the value of the semaphore will go from `NBUFF` to `NBUFF-1`.

pxsem/prodcons1.c

```
39 void *
40 produce(void *arg)
41 {
42     int     i;

43     for (i = 0; i < nitems; i++) {
44         Sem_wait(shared.nempty);    /* wait for at least 1 empty slot */
45         Sem_wait(shared.mutex);
46         shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
47         Sem_post(shared.mutex);
48         Sem_post(shared.nstored);   /* 1 more stored item */
49     }
50     return (NULL);
51 }

52 void *
53 consume(void *arg)
54 {
55     int     i;

56     for (i = 0; i < nitems; i++) {
57         Sem_wait(shared.nstored);   /* wait for at least 1 stored item */
58         Sem_wait(shared.mutex);
59         if (shared.buff[i % NBUFF] != i)
60             printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
61         Sem_post(shared.mutex);
62         Sem_post(shared.nempty);    /* 1 more empty slot */
63     }
64     return (NULL);
65 }
```

pxsem/prodcons1.c

Figure 10.18 `produce` and `consume` functions.

Producer stores item in buffer

45-48 Before storing the new item into the buffer, the producer must obtain the `mutex` semaphore. In our example, where the producer just stores a value into the array element indexed by `i % NBUFF`, no shared variables describe the status of the buffer (i.e., we do not use a linked list that we need to update each time we place an item into the buffer). Therefore, obtaining and releasing the `mutex` semaphore is not actually required. Nevertheless, we show it, because in general it is required for this type of problem (updating a buffer that is shared by multiple threads).

After the item is stored in the buffer, the `mutex` semaphore is released (its value goes from 0 to 1), and the `nstored` semaphore is posted to. The first time this statement is executed, the value of `nstored` will go from its initial value of 0 to 1.

Consumer waits for `nstored` semaphore

57-62 When the `nstored` semaphore's value is greater than 0, that many items are in the buffer to process. The consumer takes one item from the buffer and verifies that its value is correct, protecting this buffer access with the `mutex` semaphore. The consumer then posts to the `nempty` semaphore, telling the producer that another slot is empty.

Deadlock

What happens if we mistakenly swap the order of the two calls to `Sem_wait` in the consumer function (Figure 10.18)? If we assume the producer starts first (as in the solution shown for Exercise 10.1), it stores `NBUFF` items into the buffer, decrementing the value of the `nempty` semaphore from `NBUFF` to 0 and incrementing the value of the `nstored` semaphore from 0 to `NBUFF`. At that point, the producer blocks in the call `Sem_wait(shared.nempty)`, since the buffer is full and no empty slots are available for another item.

The consumer starts and verifies the first `NBUFF` items from the buffer. This decrements the value of the `nstored` semaphore from `NBUFF` to 0 and increments the value of the `nempty` semaphore from 0 to `NBUFF`. The consumer then blocks in the call `Sem_wait(shared.nstored)` after calling `Sem_wait(shared.mutex)`. The producer can resume, because the value of `nempty` is now greater than 0, but the producer then calls `Sem_wait(shared.mutex)` and blocks.

This is called a *deadlock*. The producer is waiting for the `mutex` semaphore, but the consumer is holding this semaphore and waiting for the `nstored` semaphore. But the producer cannot post to the `nstored` semaphore until it obtains the `mutex` semaphore. This is one of the problems with semaphores: if we make an error in our coding, our program does not work correctly.

Posix allows `sem_wait` to detect a deadlock and return an error of `EDEADLK`, but neither of the systems being used (Solaris 2.6 and Digital Unix 4.0B) detected this error with this example.

10.7 File Locking

We now return to our sequence number problem from Chapter 9 and provide versions of our `my_lock` and `my_unlock` functions that use Posix named semaphores. Figure 10.19 shows the two functions.

One semaphore is used for an advisory file lock, and the first time this function is called, the semaphore value is initialized to 1. To obtain the file lock, we call `sem_wait`, and to release the lock, we call `sem_post`.

10.8 `sem_init` and `sem_destroy` Functions

Everything so far in this chapter has dealt with the Posix *named* semaphores. These semaphores are identified by a *name* argument that normally references a file in the filesystem. But Posix also provides *memory-based* semaphores in which the application allocates the memory for the semaphore (that is, for a `sem_t` datatype, whatever that happens to be) and then has the system initialize this semaphore.

lock/lockpxsem.c

```
#include    "unpipc.h"

#define LOCK_PATH   "pxsemlock"

sem_t  *locksem;
int     initflag;

void
my_lock(int fd)
{
    if (initflag == 0) {
        locksem = Sem_open(Px_ipc_name(LOCK_PATH), O_CREAT, FILE_MODE, 1);
        initflag = 1;
    }
    Sem_wait(locksem);
}

void
my_unlock(int fd)
{
    Sem_post(locksem);
}
```

lock/lockpxsem.c

Figure 10.19 File locking using Posix named semaphores.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int shared, unsigned int value);

                                                    Returns: –1 on error

int sem_destroy(sem_t *sem);

                                           Returns: 0 if OK, –1 on error
```

A memory-based semaphore is initialized by `sem_init`. The *sem* argument points to the `sem_t` variable that the application must allocate. If *shared* is 0, then the semaphore is shared between the threads of a process, else the semaphore is shared between processes. When *shared* is nonzero, then the semaphore must be stored in some type of shared memory that is accessible to all the processes that will be using the semaphore. As with `sem_open`, the *value* argument is the initial value of the semaphore.

When we are done with a memory-based semaphore, `sem_destroy` destroys it.

`sem_open` does not need a parameter similar to *shared* or an attribute similar to `PTHREAD_PROCESS_SHARED` (which we saw with mutexes and condition variables in Chapter 7), because a named semaphore is *always* sharable between different processes.

Notice that there is nothing similar to `O_CREAT` for a memory-based semaphore: `sem_init` always initializes the semaphore value. Therefore, we must be careful to call `sem_init` only once for a given semaphore. (Exercise 10.2 shows the difference for a named semaphore.) The results are undefined if `sem_init` is called for a semaphore that has already been initialized.

Make certain you understand a fundamental difference between sem_open and sem_init. The former returns a pointer to a sem_t variable that the *function* has allocated and initialized. The first argument to sem_init, on the other hand, is a pointer to a sem_t variable that the *caller* must allocate and that the function then initializes.

Posix.1 warns that for a memory-based semaphore, only the location pointed to by the *sem* argument to sem_init can be used to refer to the semaphore, and using copies of this sem_t datatype is undefined.

sem_init returns −1 on an error, but does *not* return 0 on success. This is indeed strange, and a note in the Posix.1 Rationale says that a future update may specify a return of 0 on success.

A memory-based semaphore can be used when the name associated with a named semaphore is not needed. Named semaphores are normally used when different, unrelated processes are using the semaphore. The name is how each process identifies the semaphore.

In Figure 1.3, we say that memory-based semaphores have process persistence, but their persistence really depends on the type of memory in which the semaphore is stored. A memory-based semaphore remains in existence as long as the memory in which the semaphore is contained is valid.

- If a memory-based semaphore is being shared between the threads of a single process (the *shared* argument to sem_init is 0), then the semaphore has process persistence and disappears when the process terminates.
- If a memory-based semaphore is being shared between different processes (the *shared* argument to sem_init is 1), then the semaphore must be stored in shared memory and the semaphore remains in existence as long as the shared memory remains in existence. Recall from Figure 1.3 that Posix shared memory and System V shared memory both have kernel persistence. This means that a server can create a region of shared memory, initialize a Posix memory-based semaphore in that shared memory, and then terminate. Sometime later, one or more clients can open the region of shared memory and access the memory-based semaphore stored therein.

Be warned that the following code does *not* work as planned:

```
sem_t   mysem;

    Sem_init(&mysem, 1, 0);  /* 2nd arg of 1 -> shared between processes */

    if (Fork() == 0) {       /* child */
        . . .
        Sem_post(&mysem);
    }

    Sem_wait(&mysem);        /* parent; wait for child */
```

The problem here is that the semaphore mysem is not in shared memory—see Section 10.12. Memory is normally not shared between a parent and child across a fork. The child starts with a *copy* of the parent's memory, but this is not the same as shared memory. We talk more about shared memory in Part 4 of this book.

Example

As an example, we convert our producer–consumer example from Figures 10.17 and 10.18 to use memory-based semaphores. Figure 10.20 shows the program.

pxsem/prodcons2.c

```
#include    "unpipc.h"

#define NBUFF    10

int     nitems;                 /* read-only by producer and consumer */
struct {                        /* data shared by producer and consumer */
    int     buff[NBUFF];
    sem_t   mutex, nempty, nstored;     /* semaphores, not pointers */
} shared;

void   *produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
    pthread_t tid_produce, tid_consume;

    if (argc != 2)
        err_quit("usage: prodcons2 <#items>");
    nitems = atoi(argv[1]);

        /* initialize three semaphores */
    Sem_init(&shared.mutex, 0, 1);
    Sem_init(&shared.nempty, 0, NBUFF);
    Sem_init(&shared.nstored, 0, 0);

    Set_concurrency(2);
    Pthread_create(&tid_produce, NULL, produce, NULL);
    Pthread_create(&tid_consume, NULL, consume, NULL);

    Pthread_join(tid_produce, NULL);
    Pthread_join(tid_consume, NULL);

    Sem_destroy(&shared.mutex);
    Sem_destroy(&shared.nempty);
    Sem_destroy(&shared.nstored);
    exit(0);
}

void *
produce(void *arg)
{
    int     i;

    for (i = 0; i < nitems; i++) {
        Sem_wait(&shared.nempty);   /* wait for at least 1 empty slot */
        Sem_wait(&shared.mutex);
        shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
        Sem_post(&shared.mutex);
        Sem_post(&shared.nstored);  /* 1 more stored item */
    }
    return (NULL);
}
```

```
43 void *
44 consume(void *arg)
45 {
46     int     i;

47     for (i = 0; i < nitems; i++) {
48         Sem_wait(&shared.nstored);   /* wait for at least 1 stored item */
49         Sem_wait(&shared.mutex);
50         if (shared.buff[i % NBUFF] != i)
51             printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
52         Sem_post(&shared.mutex);
53         Sem_post(&shared.nempty);    /* 1 more empty slot */
54     }
55     return (NULL);
56 }
```

pxsem/prodcons2.c

Figure 10.20 Producer–consumer using memory-based semaphores.

Allocate semaphores

6 Our declarations for the three semaphores are now for three `sem_t` datatypes themselves, not for pointers to three of these datatypes.

Call `sem_init`

16-27 We call `sem_init` instead of `sem_open`, and then `sem_destroy` instead of `sem_unlink`. These calls to `sem_destroy` are really not needed, since the program is about to terminate.

The remaining changes are to pass pointers to the three semaphores in all the calls to `sem_wait` and `sem_post`.

10.9 Multiple Producers, One Consumer

The producer–consumer solution in Section 10.6 solves the classic one-producer, one-consumer problem. An interesting modification is to allow multiple producers with one consumer. We will start with the solution from Figure 10.20, which used memory-based semaphores. Figure 10.21 shows the global variables and `main` function.

Globals

4 The global `nitems` is the total number of items for all the producers to produce, and `nproducers` is the number of producer threads. Both are set from command-line arguments.

Shared structure

5-10 Two new variables are declared in the `shared` structure: `nput`, the index of the next buffer entry to store into (modulo `NBUFF`), and `nputval`, the next value to store in the buffer. These two variables are taken from our solution in Figures 7.2 and 7.3. These two variables are needed to synchronize the multiple producer threads.

pxsem/prodcons3.c

```
#include    "unpipc.h"

#define NBUFF        10
#define MAXNTHREADS 100

int     nitems, nproducers;     /* read-only by producer and consumer */

struct {                        /* data shared by producers and consumer */
    int     buff[NBUFF];
    int     nput;
    int     nputval;
    sem_t   mutex, nempty, nstored;     /* semaphores, not pointers */
} shared;

void   *produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
    int     i, count[MAXNTHREADS];
    pthread_t tid_produce[MAXNTHREADS], tid_consume;

    if (argc != 3)
        err_quit("usage: prodcons3 <#items> <#producers>");
    nitems = atoi(argv[1]);
    nproducers = min(atoi(argv[2]), MAXNTHREADS);

        /* initialize three semaphores */
    Sem_init(&shared.mutex, 0, 1);
    Sem_init(&shared.nempty, 0, NBUFF);
    Sem_init(&shared.nstored, 0, 0);

        /* create all producers and one consumer */
    Set_concurrency(nproducers + 1);
    for (i = 0; i < nproducers; i++) {
        count[i] = 0;
        Pthread_create(&tid_produce[i], NULL, produce, &count[i]);
    }
    Pthread_create(&tid_consume, NULL, consume, NULL);

        /* wait for all producers and the consumer */
    for (i = 0; i < nproducers; i++) {
        Pthread_join(tid_produce[i], NULL);
        printf("count[%d] = %d\n", i, count[i]);
    }
    Pthread_join(tid_consume, NULL);

    Sem_destroy(&shared.mutex);
    Sem_destroy(&shared.nempty);
    Sem_destroy(&shared.nstored);
    exit(0);
}
```

pxsem/prodcons3.c

Figure 10.21 `main` function that creates multiple producer threads.

New command-line arguments

17-20 Two new command-line arguments specify the total number of items to be stored into the buffer and the number of producer threads to create.

Create all the threads

21-41 The semaphores are initialized, and all the producer threads and one consumer thread are created. We then wait for all the threads to terminate. This code is nearly identical to Figure 7.2.

Figure 10.22 shows the `produce` function that is executed by each producer thread.

pxsem/prodcons3.c

```
43 void *
44 produce(void *arg)
45 {
46     for ( ; ; ) {
47         Sem_wait(&shared.nempty);    /* wait for at least 1 empty slot */
48         Sem_wait(&shared.mutex);

49         if (shared.nput >= nitems) {
50             Sem_post(&shared.nempty);
51             Sem_post(&shared.mutex);
52             return (NULL);       /* all done */
53         }
54         shared.buff[shared.nput % NBUFF] = shared.nputval;
55         shared.nput++;
56         shared.nputval++;

57         Sem_post(&shared.mutex);
58         Sem_post(&shared.nstored);   /* 1 more stored item */
59         *((int *) arg) += 1;
60     }
61 }
```

pxsem/prodcons3.c

Figure 10.22 Function executed by all the producer threads.

Mutual exclusion among producer threads

49-53 The change from Figure 10.18 is that the loop terminates when `nitems` of the values have been placed into the buffer by all the threads. Notice that multiple producer threads can acquire the `nempty` semaphore at the same time, but only one producer thread at a time can acquire the `mutex` semaphore. This protects the variables `nput` and `nputval` from being modified by more than one producer thread at a time.

Termination of producers

50-51 We must carefully handle the termination of the producer threads. After the last item is produced, each producer thread executes

```
Sem_wait(&shared.nempty);    /* wait for at least 1 empty slot */
```

at the top of the loop, which decrements the `nempty` semaphore. But before the thread terminates, it must increment this semaphore, because the thread does not store an item in the buffer during its last time around the loop. The terminating thread must also release the `mutex` semaphore, to allow the other producer threads to continue. If we did not increment `nempty` on thread termination and if we had more producer threads than buffer slots (say 14 threads and 10 buffer slots), the excess threads (4) would be blocked forever, waiting for the `nempty` semaphore, and would never terminate.

The `consume` function in Figure 10.23 just verifies that each entry in the buffer is correct, printing a message if an error is detected.

pxsem/prodcons3.c

```
void *
consume(void *arg)
{
    int     i;

    for (i = 0; i < nitems; i++) {
        Sem_wait(&shared.nstored);  /* wait for at least 1 stored item */
        Sem_wait(&shared.mutex);

        if (shared.buff[i % NBUFF] != i)
            printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);

        Sem_post(&shared.mutex);
        Sem_post(&shared.nempty);   /* 1 more empty slot */
    }
    return (NULL);
}
```

pxsem/prodcons3.c

Figure 10.23 Function executed by the one consumer thread.

Termination of the single consumer thread is simple—it just counts the number of items consumed.

10.10 Multiple Producers, Multiple Consumers

The next modification to our producer–consumer problem is to allow multiple producers and multiple consumers. Whether it makes sense to have multiple consumers depends on the application. The author has seen two applications that use this technique.

1. A program that converts IP addresses to their corresponding hostnames. Each consumer takes an IP address, calls `gethostbyaddr` (Section 9.6 of UNPv1), and appends the hostname to a file. Since each call to `gethostbyaddr` can take a variable amount of time, the order of the IP addresses in the buffer will normally not be the same as the order of the hostnames in the file appended by the consumer threads. The advantage in this scenario is that multiple calls to `gethostbyaddr` (each of which can take seconds) occur in parallel: one per consumer thread.

   This assumes a reentrant version of `gethostbyaddr`, and not all implementations have this property. If a reentrant version is not available, an alternative is to store the buffer in shared memory and use multiple processes instead of multiple threads.

2. A program that reads UDP datagrams, operates on the datagrams, and then writes the result to a database. One consumer thread processes each datagram, and multiple consumer threads are needed to overlap the potentially long processing of each datagram. Even though the datagrams are normally written to the database by the consumer threads in an order that differs from the original datagram order, the ordering of the records in the database handles this.

Figure 10.24 shows the global variables.

pxsem/prodcons4.c

```
 1 #include     "unpipc.h"

 2 #define NBUFF        10
 3 #define MAXNTHREADS 100

 4 int     nitems, nproducers, nconsumers;     /* read-only */

 5 struct {                          /* data shared by producers and consumers */
 6     int     buff[NBUFF];
 7     int     nput;                 /* item number: 0, 1, 2, ... */
 8     int     nputval;              /* value to store in buff[] */
 9     int     nget;                 /* item number: 0, 1, 2, ... */
10     int     ngetval;              /* value fetched from buff[] */
11     sem_t   mutex, nempty, nstored;     /* semaphores, not pointers */
12 } shared;

13 void   *produce(void *), *consume(void *);
```

pxsem/prodcons4.c

Figure 10.24 Global variables.

Globals and shared structure

4-12 The number of consumer threads is now a global variable and is set from a command-line argument. We have added two more variables to our `shared` structure: `nget`, the next item number for any one of the consumer threads to fetch, and `ngetval`, the corresponding value.

The `main` function, shown in Figure 10.25, is changed to create multiple consumer threads.

19-23 A new command-line argument specifies the number of consumer threads to create. We must also allocate an array (`tid_consume`) to hold all the consumer thread IDs, and an array (`conscount`) to hold our diagnostic count of how many items each consumer thread processes.

24-50 Multiple producer threads and multiple consumer threads are created and then waited for.

pxsem/prodcons4.c

```
int
main(int argc, char **argv)
{
    int     i, prodcount[MAXNTHREADS], conscount[MAXNTHREADS];
    pthread_t tid_produce[MAXNTHREADS], tid_consume[MAXNTHREADS];

    if (argc != 4)
        err_quit("usage: prodcons4 <#items> <#producers> <#consumers>");
    nitems = atoi(argv[1]);
    nproducers = min(atoi(argv[2]), MAXNTHREADS);
    nconsumers = min(atoi(argv[3]), MAXNTHREADS);

        /* initialize three semaphores */
    Sem_init(&shared.mutex, 0, 1);
    Sem_init(&shared.nempty, 0, NBUFF);
    Sem_init(&shared.nstored, 0, 0);

        /* create all producers and all consumers */
    Set_concurrency(nproducers + nconsumers);
    for (i = 0; i < nproducers; i++) {
        prodcount[i] = 0;
        Pthread_create(&tid_produce[i], NULL, produce, &prodcount[i]);
    }
    for (i = 0; i < nconsumers; i++) {
        conscount[i] = 0;
        Pthread_create(&tid_consume[i], NULL, consume, &conscount[i]);
    }

        /* wait for all producers and all consumers */
    for (i = 0; i < nproducers; i++) {
        Pthread_join(tid_produce[i], NULL);
        printf("producer count[%d] = %d\n", i, prodcount[i]);
    }
    for (i = 0; i < nconsumers; i++) {
        Pthread_join(tid_consume[i], NULL);
        printf("consumer count[%d] = %d\n", i, conscount[i]);
    }

    Sem_destroy(&shared.mutex);
    Sem_destroy(&shared.nempty);
    Sem_destroy(&shared.nstored);
    exit(0);
}
```

pxsem/prodcons4.c

Figure 10.25 `main` function that creates multiple producers and multiple consumers.

Our producer function contains one new line from Figure 10.22. When the producer threads terminate, the line preceded with the plus sign is new:

```
        if (shared.nput >= nitems) {
+           Sem_post(&shared.nstored);  /* let consumers terminate */
            Sem_post(&shared.nempty);
            Sem_post(&shared.mutex);
            return(NULL);             /* all done */
        }
```

We again must be careful when handling the termination of the producer threads and the consumer threads. After all the items in the buffer have been consumed, each consumer thread blocks in the call

```
Sem_wait(&shared.nstored);  /* wait for at least 1 stored item */
```

We have the producer threads increment the `nstored` semaphore to unblock the consumer threads, letting them see that they are done.

Our consumer function is shown in Figure 10.26.

pxsem/prodcons4.c

```
72 void *
73 consume(void *arg)
74 {
75     int     i;

76     for ( ; ; ) {
77         Sem_wait(&shared.nstored);  /* wait for at least 1 stored item */
78         Sem_wait(&shared.mutex);

79         if (shared.nget >= nitems) {
80             Sem_post(&shared.nstored);
81             Sem_post(&shared.mutex);
82             return (NULL);       /* all done */
83         }
84         i = shared.nget % NBUFF;
85         if (shared.buff[i] != shared.ngetval)
86             printf("error: buff[%d] = %d\n", i, shared.buff[i]);
87         shared.nget++;
88         shared.ngetval++;

89         Sem_post(&shared.mutex);
90         Sem_post(&shared.nempty);   /* 1 more empty slot */
91         *((int *) arg) += 1;
92     }
93 }
```

pxsem/prodcons4.c

Figure 10.26 Function executed by all consumer threads.

Termination of consumer threads

79-83 Our consumer function must now compare `nget` to `nitems`, to determine when it is done (similar to the producer function). After the last item has been consumed from the buffer, the consumer threads block, waiting for the `nstored` semaphore to be greater than 0. Therefore, as each consumer thread terminates, it increments `nstored` to let another consumer thread terminate.

10.11 Multiple Buffers

In a typical program that processes some data, we find a loop of the form

```
while ( (n = read(fdin, buff, BUFFSIZE)) > 0) {
    /* process the data */
    write(fdout, buff, n);
}
```

Many programs that process a text file, for example, read a line of input, process that line, and write a line of output. For text files, the calls to `read` and `write` are often replaced with calls to the standard I/O functions `fgets` and `fputs`.

Figure 10.27 shows one way to depict this operation, in which we identify a function named `reader` that reads the data from the input file and a function named `writer` that writes the data to the output file. One buffer is used.

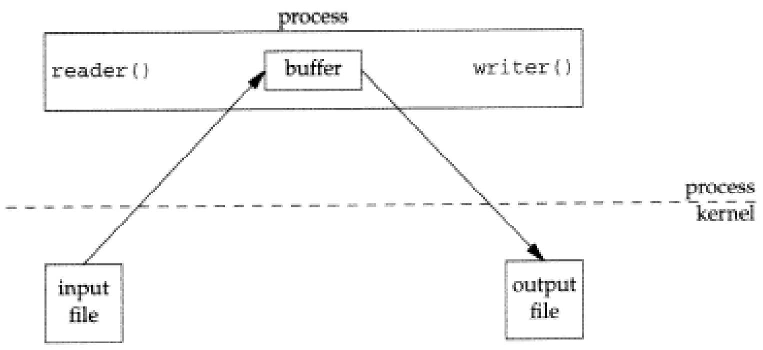

Figure 10.27 One process that reads data into a buffer and then writes the buffer out.

Figure 10.28 shows a time line of this operation. We have labeled the time line with numbers on the left, designating some arbitrary units of time. Time increases downward. We assume that a read operation takes 5 units of time, a write takes 7 units of time, and the processing time between the read and write consumes 2 units of time.

We can modify this application by dividing the processing into two threads, as shown in Figure 10.29. Here we use two threads, since a global buffer is automatically shared by the threads. We could also divide the copying into two processes, but that would require shared memory, which we have not yet covered.

Dividing the operation into two threads (or two processes) also requires some form of notification between the threads (or processes). The reader thread must notify the writer thread when the buffer is ready to be written, and the writer thread must notify the reader thread when the buffer is ready to be filled again. Figure 10.30 shows a time line for this operation.

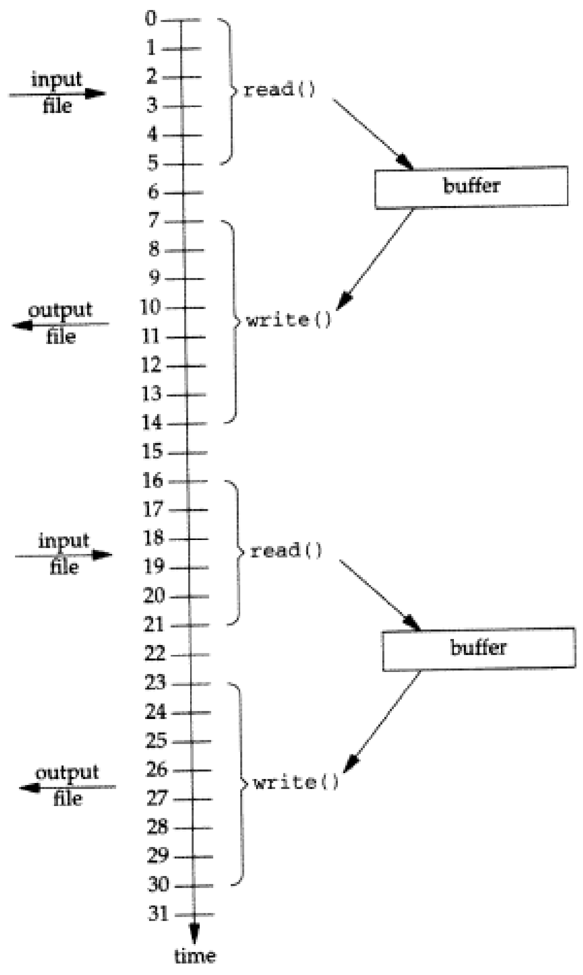

Figure 10.28 One process that reads data into a buffer and then writes the buffer out.

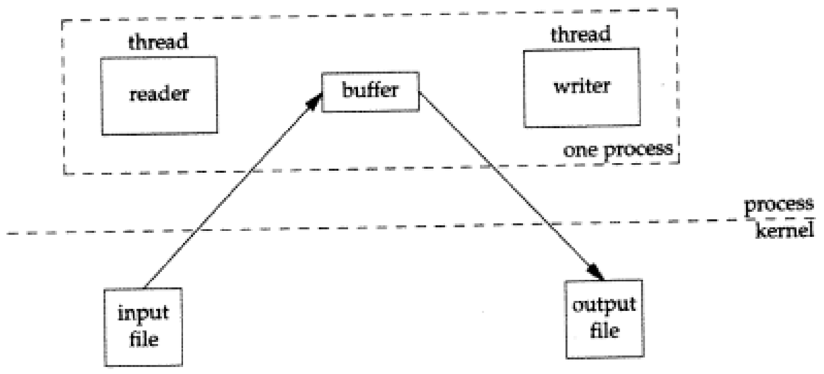

Figure 10.29 File copying divided into two threads.

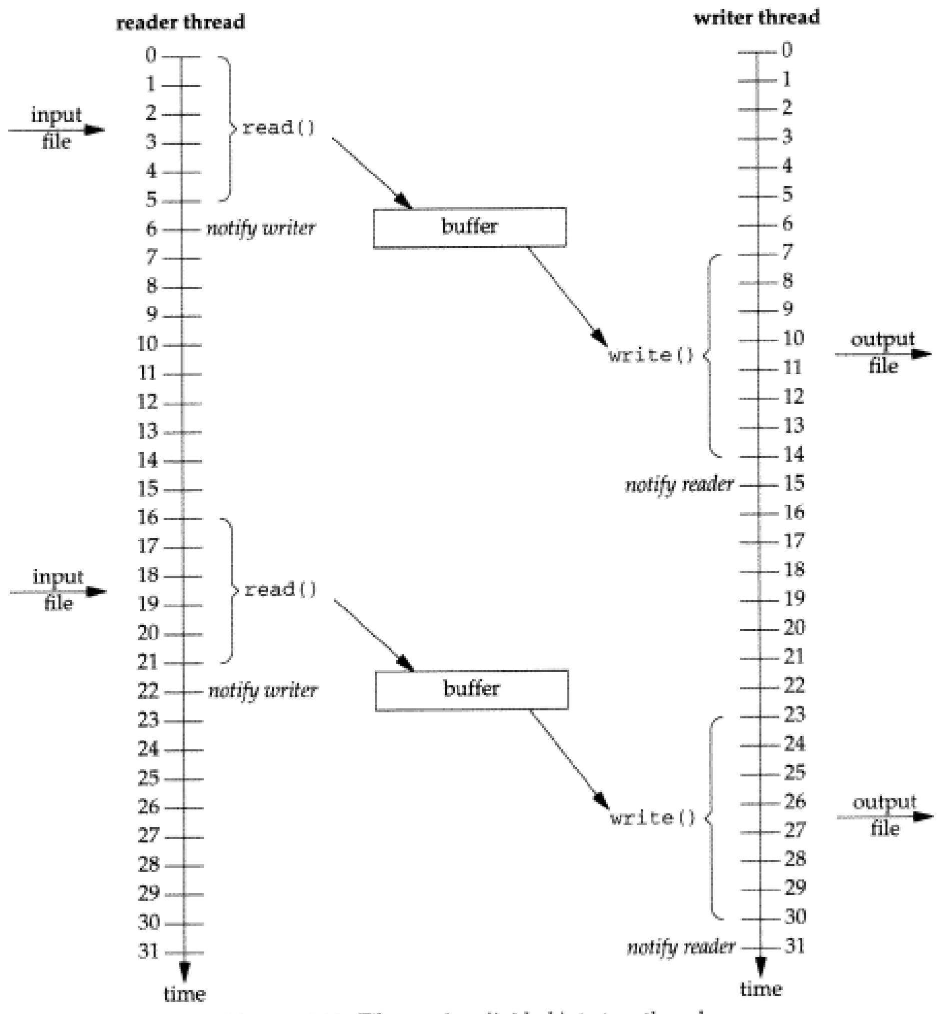

Figure 10.30 File copying divided into two threads.

We assume that the time to process the data in the buffer, along with the notification of the other thread, takes 2 units of time. The important thing to note is that dividing the reading and writing into two threads does not affect the total amount of time required to do the operation. We have not gained any speed advantage; we have only distributed the operation into two threads (or processes).

We are ignoring many fine points in these time lines. For example, most Unix kernels detect sequential reading of a file and do asynchronous *read ahead* of the next disk block for the reading process. This can improve the actual amount of time, called "clock time," that it takes to perform this type of operation. We are also ignoring the effect of other processes on our reading and writing threads, and the effects of the kernel's scheduling algorithms.

The next step in our example is to use two threads (or processes) and two buffers. This is the classic *double buffering* solution, and we show it in Figure 10.31.

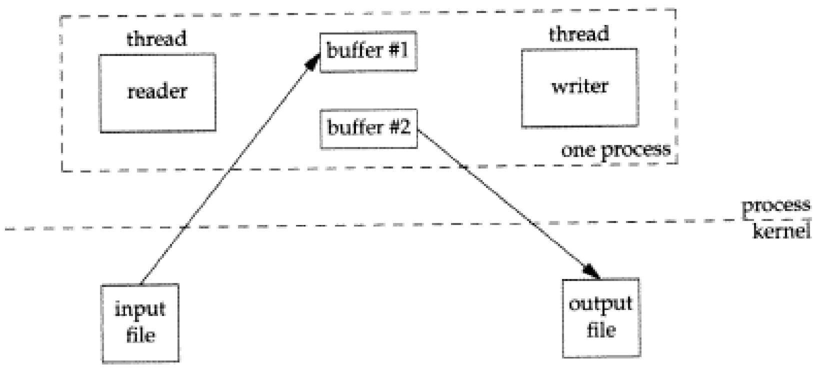

Figure 10.31 File copying divided into two threads using two buffers.

We show the reader thread reading into the first buffer while the writer thread is writing from the second buffer. The two buffers are then switched between the two threads.

Figure 10.32 shows a time line of double buffering. The reader first reads into buffer #1, and then notifies the writer that buffer #1 is ready for processing. The reader then starts reading into buffer #2, while the writer is writing buffer #1.

Note that we cannot go any faster than the slowest operation, which in our example is the write. Once the server has completed the first two reads, it has to wait the additional 2 units of time: the time difference between the write (7) and the read (5). The total clock time, however, will be almost halved for the double buffered case, compared to the single buffered case, for our hypothetical example.

Also note that the writes are now occurring as fast as they can: each write separated by only 2 units of time, compared to a separation of 9 units of time in Figures 10.28 and 10.30. This can help with some devices, such as tape drives, that operate faster if the data is written to the device as quickly as possible (this is called a *streaming* mode).

The interesting thing to note about the double buffering problem is that it is just a special case of the producer–consumer problem.

We now modify our producer–consumer to handle multiple buffers. We start with our solution from Figure 10.20 that used memory-based semaphores. Instead of just a double buffering solution, this solution handles any number of buffers (the `NBUFF` definition). Figure 10.33 shows the global variables and the `main` function.

Declare `NBUFF` buffers

2–9 Our `shared` structure now contains an array of another structure named `buff`, and this new structure contains a buffer and its count.

Open input file

18 The command-line argument is the pathname of a file that we will copy to standard output.

Figure 10.34 shows the `produce` and `consume` functions.

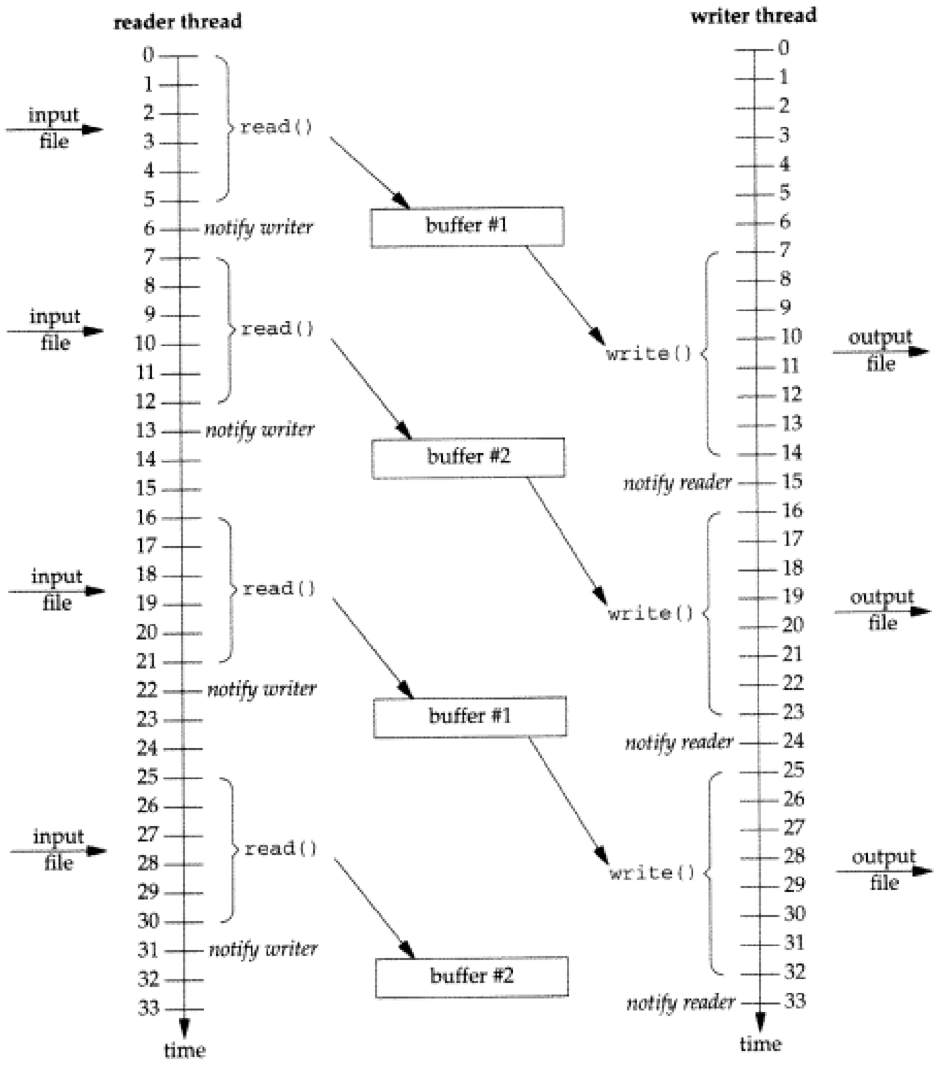

Figure 10.32 Time line for double buffering.

Empty critical region

40-42 The critical region that is locked by the `mutex` is empty for this example. If the data buffers were kept on a linked list, this would be where we could remove the buffer from the list, avoiding any conflict with the consumer's manipulation of this list. But in our example in which we just use the next buffer, with just one producer thread, nothing needs protection from the consumer. We still show the locking and unlocking of the `mutex`, to emphasize that this may be needed in other modifications to this code.

pxsem/mycat2.c

```
#include    "unpipc.h"

#define NBUFF    8

struct {                        /* data shared by producer and consumer */
    struct {
        char    data[BUFFSIZE]; /* a buffer */
        ssize_t n;              /* count of #bytes in the buffer */
    } buff[NBUFF];              /* NBUFF of these buffers/counts */
    sem_t   mutex, nempty, nstored;     /* semaphores, not pointers */
} shared;

int     fd;                     /* input file to copy to stdout */
void   *produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
    pthread_t tid_produce, tid_consume;

    if (argc != 2)
        err_quit("usage: mycat2 <pathname>");

    fd = Open(argv[1], O_RDONLY);

        /* initialize three semaphores */
    Sem_init(&shared.mutex, 0, 1);
    Sem_init(&shared.nempty, 0, NBUFF);
    Sem_init(&shared.nstored, 0, 0);

        /* one producer thread, one consumer thread */
    Set_concurrency(2);
    Pthread_create(&tid_produce, NULL, produce, NULL);  /* reader thread */
    Pthread_create(&tid_consume, NULL, consume, NULL);  /* writer thread */

    Pthread_join(tid_produce, NULL);
    Pthread_join(tid_consume, NULL);

    Sem_destroy(&shared.mutex);
    Sem_destroy(&shared.nempty);
    Sem_destroy(&shared.nstored);
    exit(0);
}
```

pxsem/mycat2.c

Figure 10.33 Global variable and `main` function.

Read data and increment `nstored` semaphore

43-49 Each time the producer obtains an empty buffer, it calls `read`. When `read` returns, the `nstored` semaphore is incremented, telling the consumer that the buffer is ready. When `read` returns 0 (end-of-file), the semaphore is incremented and the producer returns.

pxsem/mycat2.c

```
34 void *
35 produce(void *arg)
36 {
37     int     i;

38     for (i = 0;;) {
39         Sem_wait(&shared.nempty);   /* wait for at least 1 empty slot */

40         Sem_wait(&shared.mutex);
41             /* critical region */
42         Sem_post(&shared.mutex);

43         shared.buff[i].n = Read(fd, shared.buff[i].data, BUFFSIZE);
44         if (shared.buff[i].n == 0) {
45             Sem_post(&shared.nstored);  /* 1 more stored item */
46             return (NULL);
47         }
48         if (++i >= NBUFF)
49             i = 0;              /* circular buffer */

50         Sem_post(&shared.nstored);  /* 1 more stored item */
51     }
52 }

53 void *
54 consume(void *arg)
55 {
56     int     i;

57     for (i = 0;;) {
58         Sem_wait(&shared.nstored);  /* wait for at least 1 stored item */

59         Sem_wait(&shared.mutex);
60             /* critical region */
61         Sem_post(&shared.mutex);

62         if (shared.buff[i].n == 0)
63             return (NULL);
64         Write(STDOUT_FILENO, shared.buff[i].data, shared.buff[i].n);
65         if (++i >= NBUFF)
66             i = 0;              /* circular buffer */

67         Sem_post(&shared.nempty);   /* 1 more empty slot */
68     }
69 }
```

pxsem/mycat2.c

Figure 10.34 produce and consume functions.

Consumer thread

57-68 The consumer thread takes the buffers and writes them to standard output. A buffer containing a length of 0 indicates the end-of-file. As with the producer, the critical region protected by the `mutex` is empty.

In Section 22.3 of UNPv1, we developed an example using multiple buffers. In that example, the producer was the SIGIO signal handler, and the consumer was the main processing loop (the dg_echo function). The variable shared between the producer and consumer was the nqueue counter. The consumer blocked the SIGIO signal from being generated whenever it examined or modified this counter.

10.12 Sharing Semaphores between Processes

The rules for sharing memory-based semaphores between processes are simple: the semaphore itself (the sem_t datatype whose address is the first argument to sem_init) must reside in memory that is shared among all the processes that want to share the semaphore, and the second argument to sem_init must be 1.

These rules are similar to those for sharing a mutex, condition variable, or read–write lock between processes: the synchronization object itself (the pthread_mutex_t variable, or the pthread_cond_t variable, or the pthread_rwlock_t variable) must reside in memory that is shared among all the processes that want to share the object, and the object must be initialized with the PTHREAD_PROCESS_SHARED attribute.

With regard to named semaphores, different processes (related or unrelated) can always reference the same named semaphore by having each process call sem_open specifying the same name. Even though the pointers returned by sem_open might be different in each process that calls sem_open for a given name, the semaphore functions that use this pointer (e.g., sem_post and sem_wait) will all reference the same named semaphore.

But what if we call sem_open, which returns a pointer to a sem_t datatype, and then call fork? The description of the fork function in Posix.1 says "any semaphores that are open in the parent process shall also be open in the child process." This means that code of the following form is OK:

```
sem_t   *mutex;     /* global pointer that is copied across the fork() */

    ...
        /* parent creates named semaphore */
    mutex = Sem_open(Px_ipc_name(NAME), O_CREAT | O_EXCL, FILE_MODE, 0);

    if ( (childpid = Fork()) == 0) {
            /* child */
        ...
        Sem_wait(mutex);
        ...
    }
        /* parent */
    ...
    Sem_post(mutex);
    ...
```

The reason that we must be careful about knowing when we can and cannot share a semaphore between different processes is that the state of a semaphore might be contained in the sem_t datatype itself but it might also use other information (e.g., file descriptors). We will see in the next chapter that the only handle that a process has to describe a System V semaphore is its integer identifier that is returned by `semget`. Any process that knows that identifier can then access the semaphore. All the state information for a System V semaphore is contained in the kernel, and the integer identifier just tells the kernel which semaphore is being referenced.

10.13 Semaphore Limits

Two semaphore limits are defined by Posix:

`SEM_NSEMS_MAX`	the maximum number of semaphores that a process can have open at once (Posix requires that this be at least 256), and
`SEM_VALUE_MAX`	the maximum value of a semaphore (Posix requires that this be at least 32767).

These two constants are often defined in the `<unistd.h>` header and can also be obtained at run time by calling the `sysconf` function, as we show next.

Example: `semsysconf` Program

The program in Figure 10.35 calls `sysconf` and prints the two implementation-defined limits for semaphores.

pxsem/semsysconf.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    printf("SEM_NSEMS_MAX = %ld, SEM_VALUE_MAX = %ld\n",
           Sysconf(_SC_SEM_NSEMS_MAX), Sysconf(_SC_SEM_VALUE_MAX));
    exit(0);
}
```

pxsem/semsysconf.c

Figure 10.35 Call `sysconf` to obtain semaphore limits.

If we execute this on our two systems, we obtain

```
solaris % semsysconf
SEM_NSEMS_MAX = 2147483647, SEM_VALUE_MAX = 2147483647

alpha % semsysconf
SEM_NSEMS_MAX = 256, SEM_VALUE_MAX = 32767
```

10.14 Implementation Using FIFOs

We now provide an implementation of Posix named semaphores using FIFOs. Each named semaphore is implemented as a FIFO using the same name. The nonnegative number of bytes in the FIFO is the current value of the semaphore. The `sem_post` function writes 1 byte to the FIFO, and the `sem_wait` function reads 1 byte from the FIFO (blocking if the FIFO is empty, which is what we want). The `sem_open` function creates the FIFO if the `O_CREAT` flag is specified, opens it twice (once read-only, once write-only), and if a new FIFO has been created, writes the number of bytes specified by the initial value to the FIFO.

This section and the remaining sections of this chapter contain advanced topics that you may want to skip on a first reading.

We first show our `semaphore.h` header in Figure 10.36, which defines the fundamental `sem_t` datatype.

my_pxsem_fifo/semaphore.h

```
 1         /* the fundamental datatype */
 2 typedef struct {
 3     int     sem_fd[2];          /* two fds: [0] for reading, [1] for writing */
 4     int     sem_magic;          /* magic number if open */
 5 } sem_t;

 6 #define SEM_MAGIC   0x89674523

 7 #ifdef  SEM_FAILED
 8 #undef  SEM_FAILED
 9 #define SEM_FAILED  ((sem_t *)(-1))   /* avoid compiler warnings */
10 #endif
```

my_pxsem_fifo/semaphore.h

Figure 10.36 `semaphore.h` header.

`sem_t` datatype

1-5 Our semaphore data structure contains two descriptors, one for reading the FIFO and one for writing the FIFO. For similarity with pipes, we store both descriptors in a two-element array, with the first descriptor for reading and the second descriptor for writing.

The `sem_magic` member contains `SEM_MAGIC` once this structure has been initialized. This value is checked by each function that is passed a `sem_t` pointer, to make certain that the pointer really points to an initialized semaphore structure. This member is set to 0 when the semaphore is closed. This technique, although not perfect, can help detect some programming errors.

`sem_open` Function

Figure 10.37 shows our `sem_open` function, which creates a new semaphore or opens an existing semaphore.

my_pxsem_fifo/sem_open.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 #include    <stdarg.h>          /* for variable arg lists */

 4 sem_t *
 5 sem_open(const char *pathname, int oflag,...)
```

```
{
    int     i, flags, save_errno;
    char    c;
    mode_t  mode;
    va_list ap;
    sem_t *sem;
    unsigned int value;

    if (oflag & O_CREAT) {
        va_start(ap, oflag);    /* init ap to final named argument */
        mode = va_arg(ap, va_mode_t);
        value = va_arg(ap, unsigned int);
        va_end(ap);

        if (mkfifo(pathname, mode) < 0) {
            if (errno == EEXIST && (oflag & O_EXCL) == 0)
                oflag &= ~O_CREAT;  /* already exists, OK */
            else
                return (SEM_FAILED);
        }
    }
    if ( (sem = malloc(sizeof(sem_t))) == NULL)
        return(SEM_FAILED);
    sem->sem_fd[0] = sem->sem_fd[1] = -1;

    if ( (sem->sem_fd[0] = open(pathname, O_RDONLY | O_NONBLOCK)) < 0)
        goto error;
    if ( (sem->sem_fd[1] = open(pathname, O_WRONLY | O_NONBLOCK)) < 0)
        goto error;

        /* turn off nonblocking for sem_fd[0] */
    if ( (flags = fcntl(sem->sem_fd[0], F_GETFL, 0)) < 0)
        goto error;
    flags &= ~O_NONBLOCK;
    if (fcntl(sem->sem_fd[0], F_SETFL, flags) < 0)
        goto error;

    if (oflag & O_CREAT) {      /* initialize semaphore */
        for (i = 0; i < value; i++)
            if (write(sem->sem_fd[1], &c, 1) != 1)
                goto error;
    }
    sem->sem_magic = SEM_MAGIC;
    return (sem);

  error:
    save_errno = errno;
    if (oflag & O_CREAT)
        unlink(pathname);       /* if we created FIFO */
    close(sem->sem_fd[0]);      /* ignore error */
    close(sem->sem_fd[1]);      /* ignore error */
    free(sem);
    errno = save_errno;
    return (SEM_FAILED);
}
```
my_pxsem_fifo/sem_open.c

Figure 10.37 sem_open function.

Create a new semaphore

13-17 If the caller specifies the O_CREAT flag, then we know that four arguments are required, not two. We call va_start to initialize the variable ap to point to the last named argument (oflag). We then use ap and the implementation's va_arg function to obtain the values for the third and fourth arguments. We described the handling of the variable argument list and our va_mode_t datatype with Figure 5.21.

Create new FIFO

18-23 A new FIFO is created with the name specified by the caller. As we discussed in Section 4.6, this function returns an error of EEXIST if the FIFO already exists. If the caller of sem_open does not specify the O_EXCL flag, then this error is OK, but we do not want to initialize the FIFO later in the function, so we turn off the O_CREAT flag.

Allocate sem_t datatype and open FIFO for reading and writing

25-37 We allocate space for a sem_t datatype, which will contain two descriptors. We open the FIFO twice, once read-only and once write-only. We do not want to block in either call to open, so we specify the O_NONBLOCK flag when we open the FIFO read-only (recall Figure 4.21). We also specify the O_NONBLOCK flag when we open the FIFO write-only, but this is to detect overflow (e.g., if we try to write more than PIPE_BUF bytes to the FIFO). After the FIFO has been opened twice, we turn off the nonblocking flag on the read-only descriptor.

Initialize value of newly create semaphore

38-42 If a new semaphore has been created, we initialize its value by writing value number of bytes to the FIFO. If the initial value exceeds the implementation's PIPE_BUF limit, the call to write after the FIFO is full will return an error of EAGAIN.

sem_close Function

Figure 10.38 shows our sem_close function.

11-15 We close both descriptors and free the memory that was allocated for the sem_t datatype.

sem_unlink Function

Our sem_unlink function, shown in Figure 10.39, removes the name associated with our semaphore. It just calls the Unix unlink function.

sem_post Function

Figure 10.40 shows our sem_post function, which increments the value of a semaphore.

11-12 We write an arbitrary byte to the FIFO. If the FIFO was empty, this will wake up any processes that are blocked in a call to read on this FIFO, waiting for a byte of data.

my_pxsem_fifo/sem_close.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_close(sem_t *sem)
{
    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    sem->sem_magic = 0;         /* in case caller tries to use it later */
    if (close(sem->sem_fd[0]) == -1 || close(sem->sem_fd[1]) == -1) {
        free(sem);
        return (-1);
    }
    free(sem);
    return (0);
}
```

my_pxsem_fifo/sem_close.c

Figure 10.38 `sem_close` function.

my_pxsem_fifo/sem_unlink.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_unlink(const char *pathname)
{
    return (unlink(pathname));
}
```

my_pxsem_fifo/sem_unlink.c

Figure 10.39 `sem_unlink` function.

my_pxsem_fifo/sem_post.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_post(sem_t *sem)
{
    char    c;

    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    if (write(sem->sem_fd[1], &c, 1) == 1)
        return (0);
    return (-1);
}
```

my_pxsem_fifo/sem_post.c

Figure 10.40 `sem_post` function.

`sem_wait` Function

The final function is shown in Figure 10.41, `sem_wait`.

my_pxsem_fifo/sem_wait.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_wait(sem_t *sem)
 5 {
 6     char    c;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return (-1);
10     }
11     if (read(sem->sem_fd[0], &c, 1) == 1)
12         return (0);
13     return (-1);
14 }
```

my_pxsem_fifo/sem_wait.c

Figure 10.41 `sem_wait` function.

11-12 We `read` 1 byte from the FIFO, blocking if the FIFO is empty.

We have not implemented the `sem_trywait` function, but that could be done by enabling the nonblocking flag for the FIFO and calling `read`. We have also not implemented the `sem_getvalue` function. Some implementations return the number of bytes currently in a pipe or FIFO when the `stat` or `fstat` function is called, as the `st_size` member of the `stat` structure. But this is not guaranteed by Posix and is therefore nonportable. Implementations of these two Posix semaphore functions are shown in the next section.

10.15 Implementation Using Memory-Mapped I/O

We now provide an implementation of Posix named semaphores using memory-mapped I/O along with Posix mutexes and condition variables. An implementation similar to this is provided in Section B.11.3 (the Rationale) of [IEEE 1996].

We cover memory-mapped I/O in Chapters 12 and 13. You may wish to skip this section until you have read those chapters.

We first show our `semaphore.h` header in Figure 10.42, which defines the fundamental `sem_t` datatype.

`sem_t` datatype

1-7 Our semaphore data structure contains a mutex, a condition variable, and an unsigned integer containing the current value of the semaphore. As discussed with Figure 10.36, the `sem_magic` member contains `SEM_MAGIC` once this structure has been initialized.

my_pxsem_mmap/semaphore.h

```
        /* the fundamental datatype */
typedef struct {
    pthread_mutex_t sem_mutex;  /* lock to test and set semaphore value */
    pthread_cond_t sem_cond;    /* for transition from 0 to nonzero */
    unsigned int sem_count;     /* the actual semaphore value */
    int     sem_magic;          /* magic number if open */
} sem_t;

#define SEM_MAGIC   0x67458923

#ifdef  SEM_FAILED
#undef  SEM_FAILED
#define SEM_FAILED  ((sem_t *)(-1))   /* avoid compiler warnings */
#endif
```

my_pxsem_mmap/semaphore.h

Figure 10.42 `semaphore.h` header.

sem_open Function

Figure 10.43 shows the first half of our `sem_open` function, which creates a new semaphore or opens an existing semaphore.

Handle variable argument list

19-23 If the caller specifies the `O_CREAT` flag, then we know that four arguments are required, not two. We described the handling of the variable argument list and our `va_mode_t` datatype with Figure 5.21. We turn off the user-execute bit in the `mode` variable (`S_IXUSR`) for reasons that we describe shortly. A file is created with the name specified by the caller, and the user-execute bit is turned on.

Create a new semaphore and handle potential race condition

24-32 If, when the `O_CREAT` flag is specified by the caller, we were to just open the file, memory map its contents, and initialize the three members of the `sem_t` structure, we would have a race condition. We described this race condition with Figure 5.21, and the technique that we use is the same as shown there. We encounter a similar race condition in Figure 10.52.

Set the file size

33-37 We set the size of the newly created file by writing a zero-filled structure to the file. Since we know that the file has just been created with a size of 0, we call `write` to set the file size, and not `ftruncate`, because, as we note in Section 13.3, Posix does not guarantee that `ftruncate` works when the size of a regular file is being increased.

Memory map the file

38-42 The file is memory mapped by `mmap`. This file will contain the current value of the `sem_t` data structure, although since we have memory mapped the file, we just reference it through the pointer returned by `mmap`: we never call `read` or `write`.

my_pxsem_mmap/sem_open.c

```
#include    "unpipc.h"
#include    "semaphore.h"

#include    <stdarg.h>          /* for variable arg lists */
#define     MAX_TRIES   10      /* for waiting for initialization */

sem_t *
sem_open(const char *pathname, int oflag,...)
{
    int     fd, i, created, save_errno;
    mode_t  mode;
    va_list ap;
    sem_t *sem, seminit;
    struct stat statbuff;
    unsigned int value;
    pthread_mutexattr_t mattr;
    pthread_condattr_t cattr;

    created = 0;
    sem = MAP_FAILED;           /* [sic] */
  again:
    if (oflag & O_CREAT) {
        va_start(ap, oflag);    /* init ap to final named argument */
        mode = va_arg(ap, va_mode_t) & ~S_IXUSR;
        value = va_arg(ap, unsigned int);
        va_end(ap);

            /* open and specify O_EXCL and user-execute */
        fd = open(pathname, oflag | O_EXCL | O_RDWR, mode | S_IXUSR);
        if (fd < 0) {
            if (errno == EEXIST && (oflag & O_EXCL) == 0)
                goto exists;    /* already exists, OK */
            else
                return (SEM_FAILED);
        }
        created = 1;
            /* first one to create the file initializes it */
            /* set the file size */
        bzero(&seminit, sizeof(seminit));
        if (write(fd, &seminit, sizeof(seminit)) != sizeof(seminit))
            goto err;

            /* memory map the file */
        sem = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd, 0);
        if (sem == MAP_FAILED)
            goto err;

            /* initialize mutex, condition variable, and value */
        if ( (i = pthread_mutexattr_init(&mattr)) != 0)
            goto pthreaderr;
        pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
        i = pthread_mutex_init(&sem->sem_mutex, &mattr);
        pthread_mutexattr_destroy(&mattr);  /* be sure to destroy */
        if (i != 0)
            goto pthreaderr;
51          if ( (i = pthread_condattr_init(&cattr)) != 0)
52              goto pthreaderr;
53          pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);
54          i = pthread_cond_init(&sem->sem_cond, &cattr);
55          pthread_condattr_destroy(&cattr);    /* be sure to destroy */
56          if (i != 0)
57              goto pthreaderr;

58          if ( (sem->sem_count = value) > sysconf(_SC_SEM_VALUE_MAX)) {
59              errno = EINVAL;
60              goto err;
61          }
62              /* initialization complete, turn off user-execute bit */
63          if (fchmod(fd, mode) == -1)
64              goto err;
65          close(fd);
66          sem->sem_magic = SEM_MAGIC;
67          return (sem);
68      }
```
my_pxsem_mmap/sem_open.c

Figure 10.43 sem_open function: first half.

Initialize `sem_t` data structure

43-57 We initialize the three members of the `sem_t` data structure: the mutex, the condition variable, and the value of the semaphore. Since Posix named semaphores can be shared by any process that knows the semaphore's name and has adequate permission, we must specify the `PTHREAD_PROCESS_SHARED` attribute when initializing the mutex and condition variable. To do so for the semaphore, we first initialize the attributes by calling `pthread_mutexattr_init`, then set the process-shared attribute in this structure by calling `pthread_mutexattr_setpshared`, and then initialize the mutex by calling `pthread_mutex_init`. Three nearly identical steps are done for the condition variable. We are careful to destroy the attributes in the case of an error.

Initialize semaphore value

58-61 Finally, the initial value of the semaphore is stored. We compare this value to the maximum value allowed, which we obtain by calling `sysconf` (Section 10.13).

Turn off user-execute bit

62-67 Once the semaphore is initialized, we turn off the user-execute bit. This indicates that the semaphore has been initialized. We `close` the file, since it has been memory mapped and we do not need to keep it open.

Figure 10.44 shows the second half of our `sem_open` function. In Figure 5.23, we described a race condition that we handle here using the same technique.

Open existing semaphore

69-78 We end up here if either the `O_CREAT` flag is not specified or if `O_CREAT` is specified but the semaphore already exists. In either case, we are opening an existing semaphore. We `open` the file containing the `sem_t` datatype for reading and writing, and memory map the file into the address space of the process (`mmap`).

my_pxsem_mmap/sem_open.c

```
exists:
    if ( (fd = open(pathname, O_RDWR)) < 0) {
        if (errno == ENOENT && (oflag & O_CREAT))
            goto again;
        goto err;
    }
    sem = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
               MAP_SHARED, fd, 0);
    if (sem == MAP_FAILED)
        goto err;

        /* make certain initialization is complete */
    for (i = 0; i < MAX_TRIES; i++) {
        if (stat(pathname, &statbuff) == -1) {
            if (errno == ENOENT && (oflag & O_CREAT)) {
                close(fd);
                goto again;
            }
            goto err;
        }
        if ((statbuff.st_mode & S_IXUSR) == 0) {
            close(fd);
            sem->sem_magic = SEM_MAGIC;
            return (sem);
        }
        sleep(1);
    }
    errno = ETIMEDOUT;
    goto err;

pthreaderr:
    errno = i;
err:
        /* don't let munmap() or close() change errno */
    save_errno = errno;
    if (created)
        unlink(pathname);
    if (sem != MAP_FAILED)
        munmap(sem, sizeof(sem_t));
    close(fd);
    errno = save_errno;
    return (SEM_FAILED);
}
```

my_pxsem_mmap/sem_open.c

Figure 10.44 `sem_open` function: second half.

We can now see why Posix.1 states that "references to copies of the semaphore produce undefined results." When named semaphores are implemented using memory-mapped I/O, the semaphore (the `sem_t` datatype) is memory mapped into the address space of *all* processes that have the semaphore open. This is performed by `sem_open` in each process that opens the named semaphore. Changes made by one process (e.g., to the semaphore's count) are seen by all the other processes through the memory mapping. If we were to make our own copy of a `sem_t` data structure, this copy would no longer be shared by all the processes. Even though we might think it was working (the semaphore functions might not give any errors, at least until we call `sem_close`, which will unmap the memory, which would fail on the copy), no synchronization would occur with the other processes. Note from Figure 1.6, however, that memory-mapped regions in a parent are retained in the child across a `fork`, so a copy of a semaphore that is made by the kernel from a parent to a child across a `fork` is OK.

Make certain that semaphore is initialized

79-96 We must wait for the semaphore to be initialized (in case multiple threads try to create the same semaphore at about the same time). To do so, we call `stat` and look at the file's permissions (the `st_mode` member of the `stat` structure). If the user-execute bit is off, the semaphore has been initialized.

Error returns

97-108 When an error occurs, we are careful not to change `errno`.

`sem_close` Function

Figure 10.45 shows our `sem_close` function, which just calls `munmap` for the region that was memory mapped. Should the caller continue to use the pointer that was returned by `sem_open`, it should receive a `SIGSEGV` signal.

my_pxsem_mmap/sem_close.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_close(sem_t *sem)
{
    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    if (munmap(sem, sizeof(sem_t)) == -1)
        return(-1);

    return (0);
}
```

my_pxsem_mmap/sem_close.c

Figure 10.45 `sem_close` function.

`sem_unlink` Function

Our `sem_unlink` function shown in Figure 10.46 removes the name associated with our semaphore. It just calls the Unix `unlink` function.

`sem_post` Function

Figure 10.47 shows our `sem_post` function, which increments the value of a semaphore, awaking any threads waiting for the semaphore if the semaphore value has just become greater than 0.

my_pxsem_mmap/sem_unlink.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_unlink(const char *pathname)
{
    if (unlink(pathname) == -1)
        return (-1);
    return (0);
}
```

my_pxsem_mmap/sem_unlink.c

Figure 10.46 `sem_unlink` function.

my_pxsem_mmap/sem_post.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_post(sem_t *sem)
 5 {
 6     int     n;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return (-1);
10     }
11     if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
12         errno = n;
13         return (-1);
14     }
15     if (sem->sem_count == 0)
16         pthread_cond_signal(&sem->sem_cond);
17     sem->sem_count++;
18     pthread_mutex_unlock(&sem->sem_mutex);
19     return (0);
20 }
```

my_pxsem_mmap/sem_post.c

Figure 10.47 `sem_post` function.

11-18 We must acquire the semaphore's mutex lock before manipulating its value. If the semaphore's value will be going from 0 to 1, we call `pthread_cond_signal` to wake up anyone waiting for this semaphore.

`sem_wait` Function

The `sem_wait` function shown in Figure 10.48 waits for the value of the semaphore to exceed 0.

my_pxsem_mmap/sem_wait.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_wait(sem_t *sem)
 5 {
 6     int     n;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return (-1);
10     }
11     if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
12         errno = n;
13         return (-1);
14     }
15     while (sem->sem_count == 0)
16         pthread_cond_wait(&sem->sem_cond, &sem->sem_mutex);
17     sem->sem_count--;
18     pthread_mutex_unlock(&sem->sem_mutex);
19     return (0);
20 }
```

my_pxsem_mmap/sem_wait.c

Figure 10.48 `sem_wait` function.

11-18 We must acquire the semaphore's mutex lock before manipulating its value. If the value is 0, we go to sleep in a call to `pthread_cond_wait`, waiting for someone to call `pthread_cond_signal` for this semaphore, when its value goes from 0 to 1. Once the value is greater than 0, we decrement the value and release the mutex.

`sem_trywait` Function

Figure 10.49 shows the `sem_trywait` function, the nonblocking version of `sem_wait`.

11-22 We acquire the semaphore's mutex lock and then check its value. If the value is greater than 0, it is decremented and the return value is 0. Otherwise, the return value is –1 with `errno` set to `EAGAIN`.

`sem_getvalue` Function

Figure 10.50 shows our final function, `sem_getvalue`, which returns the current value of the semaphore.

11-16 We acquire the semaphore's mutex lock and return its value.

We can see from this implementation that semaphores are simpler to use than mutexes and condition variables.

my_pxsem_mmap/sem_trywait.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_trywait(sem_t *sem)
{
    int     n, rc;

    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
        errno = n;
        return (-1);
    }
    if (sem->sem_count > 0) {
        sem->sem_count--;
        rc = 0;
    } else {
        rc = -1;
        errno = EAGAIN;
    }
    pthread_mutex_unlock(&sem->sem_mutex);
    return (rc);
}
```

my_pxsem_mmap/sem_trywait.c

Figure 10.49 `sem_trywait` function.

my_pxsem_mmap/sem_getvalue.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_getvalue(sem_t *sem, int *pvalue)
{
    int     n;

    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
        errno = n;
        return (-1);
    }
    *pvalue = sem->sem_count;
    pthread_mutex_unlock(&sem->sem_mutex);
    return (0);
}
```

my_pxsem_mmap/sem_getvalue.c

Figure 10.50 `sem_getvalue` function.

10.16 Implementation Using System V Semaphores

We now provide one more implementation of Posix named semaphores using System V semaphores. Since implementations of the older System V semaphores are more common than the newer Posix semaphores, this implementation can allow applications to start using Posix semaphores, even if not supported by the operating system.

> We cover System V semaphores in Chapter 11. You may wish to skip this section until you have read that chapter.

We first show our `semaphore.h` header in Figure 10.51, which defines the fundamental `sem_t` datatype.

my_pxsem_svsem/semaphore.h

```
 1         /* the fundamental datatype */
 2 typedef struct {
 3     int     sem_semid;          /* the System V semaphore ID */
 4     int     sem_magic;          /* magic number if open */
 5 } sem_t;

 6 #define SEM_MAGIC   0x45678923

 7 #ifdef  SEM_FAILED
 8 #undef  SEM_FAILED
 9 #define SEM_FAILED  ((sem_t *)(-1))   /* avoid compiler warnings */
10 #endif

11 #ifndef SEMVMX
12 #define SEMVMX  32767          /* historical System V max value for sem */
13 #endif
```

my_pxsem_svsem/semaphore.h

Figure 10.51 `semaphore.h` header.

`sem_t` datatype

1–5 We implement a Posix named semaphore using a System V semaphore set consisting of one member. Our semaphore data structure contains the System V semaphore ID and a magic number (which we discussed with Figure 10.36).

`sem_open` Function

Figure 10.52 shows the first half of our `sem_open` function, which creates a new semaphore or opens an existing semaphore.

my_pxsem_svsem/sem_open.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 #include    <stdarg.h>          /* for variable arg lists */
 4 #define     MAX_TRIES   10      /* for waiting for initialization */

 5 sem_t *
 6 sem_open(const char *pathname, int oflag,...)
```

```
{
    int     i, fd, semflag, semid, save_errno;
    key_t   key;
    mode_t  mode;
    va_list ap;
    sem_t *sem;
    union semun arg;
    unsigned int value;
    struct semid_ds seminfo;
    struct sembuf initop;

        /* no mode for sem_open() w/out O_CREAT; guess */
    semflag = SVSEM_MODE;
    semid = -1;

    if (oflag & O_CREAT) {
        va_start(ap, oflag);    /* init ap to final named argument */
        mode = va_arg(ap, va_mode_t);
        value = va_arg(ap, unsigned int);
        va_end(ap);

            /* convert to key that will identify System V semaphore */
        if ( (fd = open(pathname, oflag, mode)) == -1)
            return (SEM_FAILED);
        close(fd);
        if ( (key = ftok(pathname, 1)) == (key_t) - 1)
            return (SEM_FAILED);

        semflag = IPC_CREAT | (mode & 0777);
        if (oflag & O_EXCL)
            semflag |= IPC_EXCL;

            /* create the System V semaphore with IPC_EXCL */
        if ( (semid = semget(key, 1, semflag | IPC_EXCL)) >= 0) {
                /* success, we're the first so initialize to 0 */
            arg.val = 0;
            if (semctl(semid, 0, SETVAL, arg) == -1)
                goto err;
                /* then increment by value to set sem_otime nonzero */
            if (value > SEMVMX) {
                errno = EINVAL;
                goto err;
            }
            initop.sem_num = 0;
            initop.sem_op = value;
            initop.sem_flg = 0;
            if (semop(semid, &initop, 1) == -1)
                goto err;
            goto finish;

        } else if (errno != EEXIST || (semflag & IPC_EXCL) != 0)
            goto err;
        /* else fall through */
    }
```
my_pxsem_svsem/sem_open.c

Figure 10.52 sem_open function: first half.

Create a new semaphore and handle variable argument list

20-24 If the caller specifies the `O_CREAT` flag, then we know that four arguments are required, not two. We described the handling of the variable argument list and our `va_mode_t` datatype with Figure 5.21.

Create ancillary file and map pathname into System V IPC key

25-30 A regular file is created with the pathname specified by the caller. We do so just to have a pathname for `ftok` to identify the semaphore. The caller's *oflag* argument for the semaphore, which can be either `O_CREAT` or `O_CREAT | O_EXCL`, is used in the call to `open`. This creates the file if it does not already exist and will cause an error return if the file already exists and `O_EXCL` is specified. The descriptor is closed, because the only use of this file is with `ftok`, which converts the pathname into a System V IPC key (Section 3.2).

Create System V semaphore set with one member

31-33 We convert the `O_CREAT` and `O_EXCL` constants into their corresponding System V `IPC_`*`xxx`* constants and call `semget` to create a System V semaphore set consisting of one member. We always specify `IPC_EXCL` to determine whether the semaphore exists or not.

Initialize semaphore

34-50 Section 11.2 describes a fundamental problem with initializing System V semaphores, and Section 11.6 shows the code that avoids the potential race condition. We use a similar technique here. The first thread to create the semaphore (recall that we always specify `IPC_EXCL`) initializes it to 0 with a command of `SETVAL` to `semctl`, and then sets its value to the caller's specified initial value with `semop`. We are guaranteed that the semaphore's `sem_otime` value is initialized to 0 by `semget` and will be set nonzero by the creator's call to `semop`. Therefore, any other thread that finds that the semaphore already exists knows that the semaphore has been initialized once the `sem_otime` value is nonzero.

Check initial value

40-44 We check the initial value specified by the caller because System V semaphores are normally stored as `unsigned shorts` (the `sem` structure in Section 11.1) with a maximum value of 32767 (Section 11.7), whereas Posix semaphores are normally stored as integers with possibly larger allowed values (Section 10.13). The constant `SEMVMX` is defined by some implementations to be the System V maximum value, or we define it to be 32767 in Figure 10.51.

51-53 If the semaphore already exists and the caller does not specify `O_EXCL`, this is not an error. In this situation, the code falls through to open (not create) the existing semaphore.

Figure 10.53 shows the second half of our `sem_open` function.

my_pxsem_svsem/sem_open.c

```
55     /*
56      * (O_CREAT not secified) or
57      * (O_CREAT without O_EXCL and semaphore already exists).
58      * Must open semaphore and make certain it has been initialized.
59      */
60     if ( (key = ftok(pathname, 1)) == (key_t) - 1)
61         goto err;
62     if ( (semid = semget(key, 0, semflag)) == -1)
63         goto err;

64     arg.buf = &seminfo;
65     for (i = 0; i < MAX_TRIES; i++) {
66         if (semctl(semid, 0, IPC_STAT, arg) == -1)
67             goto err;
68         if (arg.buf->sem_otime != 0)
69             goto finish;
70         sleep(1);
71     }
72     errno = ETIMEDOUT;
73   err:
74     save_errno = errno;         /* don't let semctl() change errno */
75     if (semid != -1)
76         semctl(semid, 0, IPC_RMID);
77     errno = save_errno;
78     return (SEM_FAILED);

79   finish:
80     if ( (sem = malloc(sizeof(sem_t))) == NULL)
81         goto err;

82     sem->sem_semid = semid;
83     sem->sem_magic = SEM_MAGIC;
84     return (sem);
85 }
```

my_pxsem_svsem/sem_open.c

Figure 10.53 `sem_open` function: second half.

Open existing semaphore

55-63 For an existing semaphore (the `O_CREAT` flag is not specified or `O_CREAT` is specified by itself and the semaphore already exists), we open the System V semaphore with `semget`. Notice that `sem_open` does not have a *mode* argument when `O_CREAT` is not specified, but `semget` requires the equivalent of a *mode* argument even if an existing semaphore is just being opened. Earlier in the function, we assigned a default value (the `SVSEM_MODE` constant from our `unpipc.h` header) that we pass to `semget` when `O_CREAT` is not specified.

Wait for semaphore to be initialized

64-72 We then verify that the semaphore has been initialized by calling `semctl` with a command of `IPC_STAT`, waiting for `sem_otime` to be nonzero.

Error returns

73-78 When an error occurs, we are careful not to change `errno`.

Allocate `sem_t` datatype

79-84 We allocate space for a `sem_t` datatype and store the System V semaphore ID in the structure. A pointer to the `sem_t` datatype is the return value from the function.

`sem_close` Function

Figure 10.54 shows our `sem_close` function, which just calls `free` to return the dynamically allocated memory that was used for the `sem_t` datatype.

my_pxsem_svsem/sem_close.c
```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_close(sem_t *sem)
{
    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    sem->sem_magic = 0;         /* just in case */

    free(sem);
    return (0);
}
```
my_pxsem_svsem/sem_close.c

Figure 10.54 `sem_close` function.

`sem_unlink` Function

Our `sem_unlink` function, shown in Figure 10.55, removes the ancillary file and the System V semaphore associated with our Posix semaphore.

Obtain System V key associated with pathname

8-16 `ftok` converts the pathname into a System V IPC key. The ancillary file is then removed by `unlink`. (We do so now, in case one of the remaining functions returns an error.) We open the System V semaphore with `semget` and then remove it with a command of `IPC_RMID` to `semctl`.

`sem_post` Function

Figure 10.56 shows our `sem_post` function, which increments the value of a semaphore.

11-16 We call `semop` with a single operation that increments the semaphore value by one.

`sem_wait` Function

The next function is shown in Figure 10.57; it is `sem_wait`, which waits for the value of the semaphore to exceed 0.

11-16 We call `semop` with a single operation that decrements the semaphore value by one.

my_pxsem_svsem/sem_unlink.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_unlink(const char *pathname)
{
    int     semid;
    key_t   key;

    if ( (key = ftok(pathname, 1)) == (key_t) - 1)
        return (-1);
    if (unlink(pathname) == -1)
        return (-1);
    if ( (semid = semget(key, 1, SVSEM_MODE)) == -1)
        return (-1);
    if (semctl(semid, 0, IPC_RMID) == -1)
        return (-1);
    return (0);
}
```

my_pxsem_svsem/sem_unlink.c

Figure 10.55 `sem_unlink` function.

my_pxsem_svsem/sem_post.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_post(sem_t *sem)
{
    struct sembuf op;

    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    op.sem_num = 0;
    op.sem_op = 1;
    op.sem_flg = 0;
    if (semop(sem->sem_semid, &op, 1) < 0)
        return (-1);
    return (0);
}
```

my_pxsem_svsem/sem_post.c

Figure 10.56 `sem_post` function.

`sem_trywait` Function

Our `sem_trywait` function, the nonblocking version of `sem_wait`, is shown in Figure 10.58.

13 The only change from our `sem_wait` function in Figure 10.57 is specifying `sem_flg` as `IPC_NOWAIT`. If the operation cannot be completed without blocking the calling thread, the return value from `semop` is `EAGAIN`, which is what `sem_trywait` must return if the operation cannot be completed without blocking.

my_pxsem_svsem/sem_wait.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_wait(sem_t *sem)
{
    struct sembuf op;

    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    op.sem_num = 0;
    op.sem_op = -1;
    op.sem_flg = 0;
    if (semop(sem->sem_semid, &op, 1) < 0)
        return (-1);
    return (0);
}
```

my_pxsem_svsem/sem_wait.c

Figure 10.57 `sem_wait` function.

my_pxsem_svsem/sem_trywait.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_trywait(sem_t *sem)
{
    struct sembuf op;

    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    op.sem_num = 0;
    op.sem_op = -1;
    op.sem_flg = IPC_NOWAIT;
    if (semop(sem->sem_semid, &op, 1) < 0)
        return (-1);
    return (0);
}
```

my_pxsem_svsem/sem_trywait.c

Figure 10.58 `sem_trywait` function.

`sem_getvalue` Function

The final function is shown in Figure 10.59; it is `sem_getvalue`, which returns the current value of the semaphore.

11-14 The current value of the semaphore is obtained with a command of `GETVAL` to `semctl`.

my_pxsem_svsem/sem_getvalue.c

```
#include    "unpipc.h"
#include    "semaphore.h"

int
sem_getvalue(sem_t *sem, int *pvalue)
{
    int     val;

    if (sem->sem_magic != SEM_MAGIC) {
        errno = EINVAL;
        return (-1);
    }
    if ( (val = semctl(sem->sem_semid, 0, GETVAL)) < 0)
        return (-1);
    *pvalue = val;
    return (0);
}
```

my_pxsem_svsem/sem_getvalue.c

Figure 10.59 `sem_getvalue` function.

10.17 Summary

Posix semaphores are counting semaphores, and three basic operations are provided:

1. create a semaphore,
2. wait for a semaphore's value to be greater than 0 and then decrement the value, and
3. post to a semaphore by incrementing its value and waking up any threads waiting for the semaphore.

Posix semaphores can be named or memory-based. Named semaphores can always be shared between different processes, whereas memory-based semaphores must be designated as process-shared when created. The persistence of these two types of semaphores also differs: named semaphores have at least kernel persistence, whereas memory-based semaphores have process persistence.

The producer–consumer problem is the classic example for demonstrating semaphores. In this chapter, our first solution had one producer thread and one consumer thread, our next solution allowed multiple producer threads and one consumer thread, and our final solution allowed multiple consumer threads. We then showed that the classic problem of double buffering is just a special case of the producer–consumer problem, with one producer and one consumer.

Three sample implementations of Posix semaphores were provided. The first, using FIFOs, is the simplest because much of the synchronization is handled by the kernel's `read` and `write` functions. The next implementation used memory-mapped I/O, similar to our implementation of Posix message queues in Section 5.8, and used a mutex and condition variable for synchronization. Our final implementation used System V semaphores, providing a simpler interface to these semaphores.

Exercises

10.1 Modify the `produce` and `consume` functions in Section 10.6 as follows. First, swap the order of the two calls to `Sem_wait` in the consumer, to generate a deadlock (as we discussed in Section 10.6). Next, add a call to `printf` before each call to `Sem_wait`, indicating which thread (the producer or the consumer) is waiting for which semaphore. Add another call to `printf` after the call to `Sem_wait`, indicating that the thread got the semaphore. Reduce the number of buffers to 2, and then build and run this program to verify that it leads to a deadlock.

10.2 Assume that we start four copies of our program that calls our `my_lock` function from Figure 10.19:

```
% lockpxsem & lockpxsem & lockpxsem & lockpxsem &
```

Each of the four processes starts with an `initflag` of 0, so each one calls `sem_open` specifying `O_CREAT`. Is this OK?

10.3 What happens in the previous exercise if one of the four programs terminates after calling `my_lock` but before calling `my_unlock`?

10.4 What could happen in Figure 10.37 if we did not initialize both descriptors to −1?

10.5 In Figure 10.37, why do we save the value of `errno` and then restore it, instead of coding the two calls to `close` as

```
if (sem->fd[0] >= 0)
    close(sem->fd[0]);
if (sem->fd[1] >= 0)
    close(sem->fd[1]);
```

10.6 What happens if two processes call our FIFO implementation of `sem_open` (Figure 10.37) at about the same time, both specifying `O_CREAT` with an initial value of 5? Can the FIFO ever be initialized (incorrectly) to 10?

10.7 With Figures 10.43 and 10.44, we described a possible race condition if two processes both try to create a semaphore at about the same time. Yet in the solution to the previous problem, we said that Figure 10.37 does not have a race condition. Explain.

10.8 Posix.1 makes it optional for `sem_wait` to detect that it has been interrupted by a caught signal and return `EINTR`. Write a test program to determine whether your implementation detects this or not.

Also run your test program using our implementations that use FIFOs (Section 10.14), memory-mapped I/O (Section 10.15), and System V semaphores (Section 10.16).

10.9 Which of our three implementations of `sem_post` are async-signal-safe (Figure 5.10)?

10.10 Modify the producer–consumer solution in Section 10.6 to use a `pthread_mutex_t` datatype for the `mutex` variable, instead of a semaphore. Does any measurable change in performance occur?

10.11 Compare the timing of named semaphores (Figures 10.17 and 10.18) with memory-based semaphores (Figure 10.20).