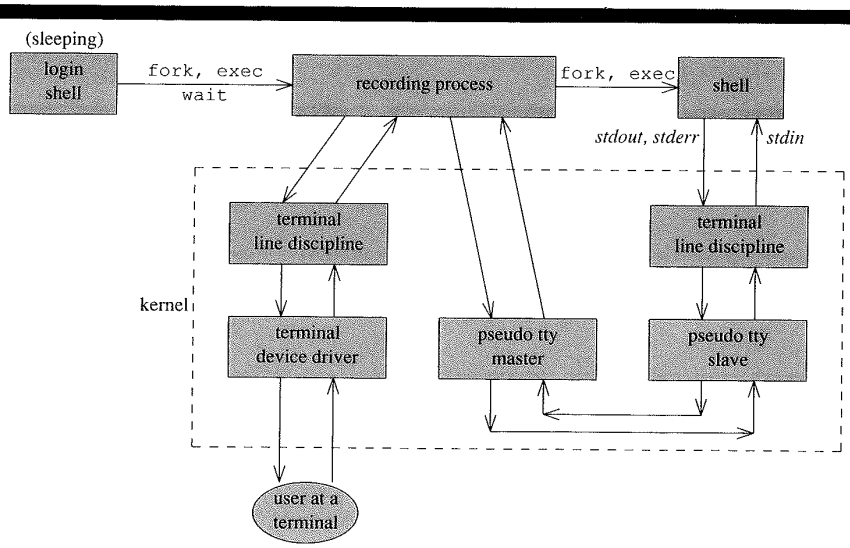
UNIX® NETWORK PROGRAMMING



W. RICHARD STEVENS

PRENTICE HALL SOFTWARE SERIES

UNIX® Network Programming

W. Richard Stevens
Health Systems International

PRENTICE HALL
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Stevens, W. Richard.
UNIX network programming / W. Richard Stevens.
p. cm.
Includes bibliographical references.
ISBN 0-13-949876-1
1. UNIX (Computer operating system) 2. Computer networks.
I. Title.
QA76.76.063S755 1990
005.7'1--dc20 89-25576
CIP

To Sally, Bill, and Ellen.

Editorial/production supervision: **Brendan M. Stewart**
Cover design: **Lundgren Graphics Ltd.**
Manufacturing buyer: **Ray Sintel**

Prentice Hall Software Series
Brian W. Kernighan, Advisor

A Division of Simon & Schuster
Englewood Cliffs, NJ 07632

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

ISBN 0-13-949876-1

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

3

Interprocess Communication

3.1 Introduction

Since network programming involves the interaction of two or more processes, we must look carefully at the different methods that are available for different processes to communicate with each other. In traditional single process programming, different modules within the single process can communicate with each other using global variables, function calls, and the arguments and results passed back and forth between functions and their callers. But, when dealing with separate processes, each executing within its own address space, there are more details to consider. When time-shared, multiprogramming operating systems were developed over 20 years ago, one design goal was to assure certain that separate processes wouldn't interfere with each other. For two processes to communicate with each other, they must both agree to it, and the operating system must provide some facilities for the *Interprocess Communication* (IPC).

While some form of IPC is required for network programming, the use of IPC is by no means restricted to processes executing on different systems, communicating through a network of some form. Indeed, there are many instances where IPC can and should be used between processes on a single computer system. We can diagram two processes using IPC on a single computer system as shown in Figure 3.1.

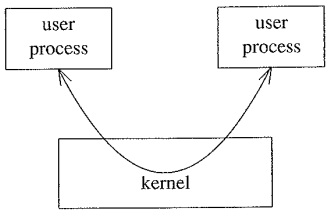


Figure 3.1 IPC between two processes on a single system.

This figure shows the information between the two processes going through the kernel; although this is typical, it is not a requirement for IPC.

To expand this to processes on different systems, using some form of networking between the two systems, we have the picture shown in Figure 3.2.

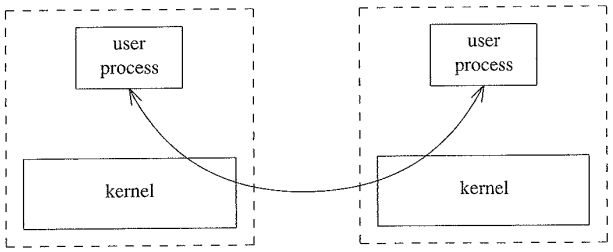


Figure 3.2 IPC between two processes on different systems.

In this chapter we consider several different methods of IPC.

- pipes
- FIFOs (named pipes)
- message queues
- semaphores
- shared memory

Before going into the IPC sections, we first describe file and record locking, an integral and necessary part of many multiple process systems.

3.2 File and Record Locking

There are occasions when multiple processes want to share some resource. It is essential that some form of mutual exclusion be provided so that only one process at a time can access the resource.

Consider the following scenario, which comes from the line printer daemon examples that we consider in more detail in Chapter 13. The process that places a job on the print queue (to be printed at a later time by another process) has to assign a unique sequence number to each print job. We cannot use the process ID as the sequence number, as it is possible for a print job to exist long enough for a given process ID to be reused. The technique used by the Unix 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:

- it reads the sequence number file,
- it uses the number,
- it increments the number and writes it back.

The problem is that in the time it takes a single process to execute these three steps, another process can perform the same three steps. Chaos can result. 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. Here is 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.

```
#define SEQFILE "seqno"         /* filename */
#define MAXBUFF    100

main()
{
        int     fd, i, n, pid, seqno;
        char    buff[MAXBUFF + 1];

        pid = getpid();
        if ( (fd = open(SEQFILE, 2)) < 0)
                err_sys("can't open %s", SEQFILE);

        for (i = 0; i < 20; i++) {
                my_lock(fd);                    /* lock the file */

                lseek(fd, 0L, 0);               /* rewind before read */
                if ( (n = read(fd, buff, MAXBUFF)) <= 0)
                        err_sys("read error");
                buff[n] = '\0';         /* null terminate for sscanf */

                if ( (n = sscanf(buff, "%d\n", &seqno)) != 1)
                        err_sys("sscanf error");
                printf("pid = %d, seq# = %d\n", pid, seqno);

                seqno++;                /* increment the sequence number */

                sprintf(buff, "%03d\n", seqno);
                n = strlen(buff);
                lseek(fd, 0L, 0);               /* rewind before write */
                if (write(fd, buff, n) != n)
```

```
                err_sys("write error");

            my_unlock(fd);              /* unlock the file */
        }
}
```

To show the results when locking is not used, the following functions provide no locking at all.

```
/*
 * Locking routines that do nothing.
 */

my_lock(fd)
int     fd;
{
        return;
}

my_unlock(fd)
int     fd;
{
        return;
}
```

If the sequence number is initialized to one, and a single copy of the program is run, we get the following output:

```
pid = 118, seq# = 1
pid = 118, seq# = 2
pid = 118, seq# = 3
pid = 118, seq# = 4
pid = 118, seq# = 5
pid = 118, seq# = 6
pid = 118, seq# = 7
pid = 118, seq# = 8
pid = 118, seq# = 9
pid = 118, seq# = 10
pid = 118, seq# = 11
pid = 118, seq# = 12
pid = 118, seq# = 13
pid = 118, seq# = 14
pid = 118, seq# = 15
pid = 118, seq# = 16
pid = 118, seq# = 17
pid = 118, seq# = 18
pid = 118, seq# = 19
pid = 118, seq# = 20
```

When the sequence number is again initialized to one, and the program (whose name is `a.out`) is run twice, using the shell command

```
a.out & a.out &
```

we get the following output:

```
pid = 186, seq# = 1

pid = 187, seq# = 1
pid = 187, seq# = 2
pid = 187, seq# = 3
pid = 187, seq# = 4
pid = 187, seq# = 5
pid = 187, seq# = 6
pid = 187, seq# = 7
pid = 187, seq# = 8
pid = 187, seq# = 9
pid = 187, seq# = 10

pid = 186, seq# = 2
pid = 186, seq# = 3
pid = 186, seq# = 4
pid = 186, seq# = 5

pid = 187, seq# = 5

pid = 186, seq# = 6
pid = 186, seq# = 7
pid = 186, seq# = 8
pid = 186, seq# = 9
pid = 186, seq# = 10
pid = 186, seq# = 11

pid = 187, seq# = 6
pid = 187, seq# = 7
pid = 187, seq# = 8
pid = 187, seq# = 9
pid = 187, seq# = 10

pid = 186, seq# = 12
pid = 186, seq# = 13
pid = 186, seq# = 14
pid = 186, seq# = 15
pid = 186, seq# = 16
pid = 186, seq# = 17
pid = 186, seq# = 18
pid = 186, seq# = 19
pid = 186, seq# = 20

pid = 187, seq# = 11
pid = 187, seq# = 12
pid = 187, seq# = 13
pid = 187, seq# = 14
```

(The output has been put into two columns and spaces inserted between the consecutive lines of output for a single process to make the output more readable.) This is not what we want. The first process (PID 186) reads the file and prints its value (1). Before this value gets incremented and written back to the file, the second process (PID 187) reads the file and uses the CPU while it updates the sequence number all the way to 10. Only then does the first process run again, and it resets the sequence number back to two. What has happened is that the operating system has switched between the two processes somewhere in the middle of the three steps outlined earlier. Note that each process reads, increments, and writes the sequence number file exactly 20 times (there are exactly 40 lines of output), but the sequence number keeps getting incremented then reset to some earlier value by the other process.

What we need is some way to allow a process to "lock out" other processes from accessing the sequence number file while the three steps are being performed.

Advisory Locking versus Mandatory Locking

Advisory locking means that the operating system maintains a correct knowledge of which files have been locked by which processes, but it does not prevent some process from writing to a file that is locked by another process. A process can ignore an advisory lock and write to a file that is locked, if the process has adequate permissions. Advisory locks are fine for what are called *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. In our example, 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. The other type of file locking, *mandatory locking*, is provided by some systems. Mandatory locks mean that the operating system checks every `read` and `write` request to verify that the operation does not interfere with a lock held by a process.

System V Release 2 provides advisory locking, as does 4.3BSD, while System V Release 3 provides both advisory and mandatory locking. By default, System V Release 3 provides advisory locking. To enable mandatory locking for a particular file, we must turn the group-execute bit *off* and turn the set-group-ID bit *on* for the file. Note that it makes no sense to have the set-user-ID bit on for a file without having the user-execute bit on also, and similarly for the set-group-ID bit and the group-execute bit. Therefore the mandatory locking option was added to System V Release 3 in this way, without affecting any existing user software. New system calls were not required. The `ls` command on System V Release 3 was enhanced to look for this special combination of bits and note that the file is marked for mandatory locking.

File Locking versus Record Locking

File locking locks an entire file, while *record locking* allows a process to lock a specified portion of a file. The definition of a record for Unix record locking is given by specifying a starting byte offset in the file and the number of bytes from that position. The term *record locking* is derived from other operating systems that enforce a record structure on disk files. A better term for the Unix feature is *range locking* since the Unix kernel does not support the concept of "records." It is a range of the file that is locked. Whether this range has any resemblance to a record is up to the application.

The System V `lockf` function has the calling sequence

```
#include <unistd.h>

int lockf(int fd, int function, long size);
```

where *function* has one of the following values:

F_ULOCK	Unlock a previously locked region
F_LOCK	Lock a region (blocking)
F_TLOCK	Test and lock a region (nonblocking)
F_TEST	Test a region to see if it is locked

The lockf function uses the current file offset (which the process can set using the lseek system call) and the *size* argument to define the "record." The record starts at the current offset and extends forward for a positive *size*, or extends backwards for a negative *size*. If the *size* is zero, the record affected extends from the current offset through the largest file offset (the end of file). Doing an lseek to the beginning of the file followed by a lockf with a *size* of zero locks the entire file.

The lockf function provides both the ability to set a lock and to test if a lock is set. When the *function* is F_TLOCK and the region is already locked by another process, the calling process is put to sleep until the region is available. This is termed *blocking*. The F_TLOCK operation, however, is termed a *nonblocking* call—if the region is not available, lockf returns immediately with a value of −1 and errno set to either EAGAIN or EACCES.† Also, the F_TEST operation allows a process to test if a lock is set, without setting a lock. Note that to do a *nonblocking* lock, we must use the F_TLOCK operation. If we write

```
...
if (lockf(fd, F_TEST, size) == 0) {
        rc = lockf(fd, F_LOCK, size);
        ...
}
```

there is a chance that some other process can issue a lockf call between the F_TEST and the F_LOCK, which causes the F_LOCK to block. The F_TLOCK provides the ability to do the test and set in a single operation, which is needed.

An example of a nonblocking lock is if a daemon is started and it wants to assure that only a single copy of it is running. When the daemon starts, it executes a nonblocking lock on a specified file. If the lock succeeds, then the daemon is the only copy running, but if it fails the daemon can exit, since it knows that a copy is already running. We will see an example of this exact scenario in the 4.3BSD line printer spooler in Section 13.2.

System V Release 2 Advisory Locking

The original System V Release 2 did not provide any type of file or record locking. But the enhanced version of System V Release 2 provided advisory file and record locking.

† System V Release 2 and System V Release 3 both return EACCES. The System V Release 3 manual warns that "portable application programs should expect and test for either value."

With System V Release 3, mandatory file and record locking are provided. Our locking functions are

```
/*
 * Locking routines for System V.
 */

#include        <unistd.h>

my_lock(fd)
int     fd;
{
        lseek(fd, 0L, 0);                       /* rewind before lockf */
        if (lockf(fd, F_LOCK, 0L) == -1)        /* 0L -> lock entire file */
                err_sys("can't F_LOCK");
}

my_unlock(fd)
int     fd;
{
        lseek(fd, 0L, 0);
        if (lockf(fd, F_ULOCK, 0L) == -1)
                err_sys("can't F_ULOCK");
}
```

System V record locking is really provided by the `fcntl` system call. The `lockf` function is provided by the standard C library as an interface to the `fcntl` system call.

4.3BSD Advisory Locking

We continue our sequence number example using the advisory file locking provided by 4.3BSD. The `flock` system call is provided to lock and unlock a file.

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

int flock(int fd, int operation);
```

fd is a file descriptor of an open file, and *operation* is built from the following constants:

`LOCK_SH`	Shared lock
`LOCK_EX`	Exclusive lock
`LOCK_UN`	Unlock
`LOCK_NB`	Don't block when locking

More than one shared lock can be applied to a file at any time, but a file cannot have both a shared lock and an exclusive lock, or multiple exclusive locks at any time. The permissible locking operations are

LOCK_SH	Shared lock (blocking)
LOCK_EX	Exclusive lock (blocking)
LOCK_SH | LOCK_NB	Shared lock (nonblocking)
LOCK_EX | LOCK_NB	Exclusive lock (nonblocking)
LOCK_UN	Unlock

If the lock was successful, zero is returned by `flock`, otherwise −1 is returned. If the lock fails because the file is already locked and a nonblocking lock was requested (LOCK_NB), the global `errno` is set to EWOULDBLOCK. This lets us code our locking functions as follows:

```
/*
 * Locking routines for 4.3BSD.
 */

#include        <sys/file.h>

my_lock(fd)
int     fd;
{
        if (flock(fd, LOCK_EX) == -1)
                err_sys("can't LOCK_EX");
}

my_unlock(fd)
int     fd;
{
        if (flock(fd, LOCK_UN) == -1)
                err_sys("can't LOCK_UN");
}
```

We can now use these locking functions and run our program under System V or 4.3BSD. Executing the shell command

```
a.out & a.out &
```

yields the output

```
pid = 308, seq# = 1

pid = 307, seq# = 2
pid = 307, seq# = 3
pid = 307, seq# = 4
pid = 307, seq# = 5
pid = 307, seq# = 6
pid = 307, seq# = 7
pid = 307, seq# = 8
pid = 307, seq# = 9
pid = 307, seq# = 10
pid = 307, seq# = 11
pid = 307, seq# = 12
pid = 307, seq# = 13
pid = 307, seq# = 14
pid = 307, seq# = 15

pid = 308, seq# = 16
pid = 308, seq# = 17
pid = 308, seq# = 18
pid = 308, seq# = 19
pid = 308, seq# = 20
pid = 308, seq# = 21
pid = 308, seq# = 22
pid = 308, seq# = 23
pid = 308, seq# = 24
pid = 308, seq# = 25
pid = 308, seq# = 26
pid = 308, seq# = 27

pid = 307, seq# = 28
pid = 307, seq# = 29
pid = 307, seq# = 30

pid = 308, seq# = 31
pid = 308, seq# = 32
pid = 308, seq# = 33
pid = 308, seq# = 34

pid = 307, seq# = 35
pid = 307, seq# = 36
pid = 307, seq# = 37

pid = 308, seq# = 38
pid = 308, seq# = 39
pid = 308, seq# = 40
```

(Again we have inserted blank lines at the points when the process ID changes and made the output multicolumn.) This is the way this example should work—the sequence number increments up to 40 in order.

Other Unix Locking Techniques

There are three features of the Unix filesystem implementation that can be exploited to provide a type of locking. These techniques create and use an ancillary file as an indicator that the process has the shared resource locked. If the ancillary file exists, the resource is locked by some other process, otherwise the calling process must create the ancillary file to lock the resource. The three techniques that we show in this section use the following features:

1. The `link` system call fails if the name of the new link to the file already exists.
2. The `creat` system call fails if the file already exists *and* if the caller does not have write permission for the file.
3. Newer versions of Unix, including both 4.3BSD and System V, support options to the `open` system call that cause it to fail if the file already exists.

The first two techniques are applicable to all versions of Unix, and have been used since Version 7 (or earlier).

A first idea is to create the ancillary file and use it as an indicator that a process has a lock on some other file. This won't work, however, since the `creat` system call does not fail if the file already exists. If the file does exist, `creat` truncates the file to zero length.

Another idea is to test if the ancillary file exists, and if not, create the file.

```
if ( (fd = open(file, 0)) < 0)
        fd = creat(file, 0644);
...
```

This won't work either, as it takes two separate system calls to test if a file exists (`open`), and then create the file if it does not exist (`creat`). In the time between the two system calls some other process can do the same test and then `creat` the same file. When the process that was interrupted runs again, it issues the `creat` system call, but no error occurs even though the file already exists.

Our first correct implementation uses the fact that the `link` system call fails if the name of the new link to the file already exists.

```
int link(char *existingpath, char *newpath);
```

This system call takes an existing file and creates another directory link to that file. (Recall that a Unix file can have any number of links (i.e., names) that refer to the same file.) Unlike the `creat` system call, if the new pathname specified by the `link` already exists, an error is returned and the global `errno` is set to `EEXIST`. Our technique is to create a unique temporary file, whose name is based on the process ID of the process. Once this file is created, we try to form another `link` to it, under the "well-known" name of the ancillary lock file. If the `link` succeeds, then the process has the lock, and the lock file is pointed to by two directory entries—the temporary filename based on the process ID and the well-known name of the lock file. The temporary filename can now be removed, leaving only a single link to the file. Using this technique for our lock functions, we have

```
/*
 * Locking routines using the link() system call.
 */

#define LOCKFILE        "seqno.lock"

#include        <sys/errno.h>
extern int      errno;

my_lock(fd)
int     fd;
{
        int     tempfd;
        char    tempfile[30];

        sprintf(tempfile, "LCK%d", getpid());

        /*
```

```
         * Create a temporary file, then close it.
         * If the temporary file already exists, the creat() will
         * just truncate it to 0-length.
         */

        if ( (tempfd = creat(tempfile, 0444)) < 0)
                err_sys("can't creat temp file");
        close(tempfd);

        /*
         * Now try to rename the temporary file to the lock file.
         * This will fail if the lock file already exists (i.e., if
         * some other process already has a lock).
         */

        while (link(tempfile, LOCKFILE) < 0) {
                if (errno != EEXIST)
                        err_sys("link error");
                sleep(1);
        }
        if (unlink(tempfile) < 0)
                err_sys("unlink error for tempfile");
}

my_unlock(fd)
int     fd;
{
        if (unlink(LOCKFILE) < 0)
                err_sys("unlink error for LOCKFILE");
}
```

The second technique uses the fact that the `creat` system call returns an error if the file exists *and* write permission is denied. In describing this system call in Section 2.3, we stated that it truncates an existing file to zero-length if it already exists. This is normally true, but if the file access permissions don't allow writing to the file for the process calling `creat`, it fails and `errno` is set to `EACCES`. There is one caveat to this technique—this error is not returned if the process has superuser permissions, since the superuser can write to any file. Our technique is to create a temporary lock file with all write permissions disabled, and if the `creat` succeeds, the calling process knows it has the lock, and the sequence number file can be safely modified. The unlock operation removes the temporary lock file, since the `unlink` system call succeeds if the caller has write permission in the directory. The caller does not need either read or write permission for the file being unlinked. Our code for this technique is

```
/*
 * Locking routines using a creat() system call with all
 * permissions turned off.
 */

#include        <sys/errno.h>
extern int      errno;
```

```
#define LOCKFILE        "seqno.lock"
#define TEMPLOCK        "temp.lock"

my_lock(fd)
int     fd;
{
        int     tempfd;

        /*
         * Try to create a temporary file, with all write
         * permissions turned off.  If the temporary file already
         * exists, the creat() will fail.
         */

        while ( (tempfd = creat(TEMPLOCK, 0)) < 0) {
                if (errno != EACCES)
                        err_sys("creat error");
                sleep(1);
        }
        close(tempfd);
}

my_unlock(fd)
int     fd;
{
        if (unlink(TEMPLOCK) < 0)
                err_sys("unlink error for tempfile");
}
```

The third technique uses the option flags for the `open` system call that are provided by most recent versions of Unix—both 4.3BSD and System V, for example, provide it. The `<fcntl.h>` header file defines the option flags for the `open` system call. If both `O_CREAT` and `O_EXCL` are specified, the `open` fails if the file already exists, otherwise the file is created. The POSIX standard [IEEE 1988] specifically states that the test for the existence of the file and the creation of the file if it doesn't exist, must be atomic with regard to other processes trying to do the same thing.

Our code for this technique is

```
/*
 * Locking routines using a open() system call with both
 * O_CREAT and O_EXCL specified.
 */

#include        <fcntl.h>
#include        <sys/errno.h>
extern int      errno;

#define LOCKFILE        "seqno.lock"
#define PERMS           0666

my_lock(fd)
int     fd;
{
```

```
        int     tempfd;

        /*
         * Try to create the lock file, using open() with both
         * O_CREAT (create file if it doesn't exist) and O_EXCL
         * (error if create and file already exists).
         * If this fails, some other process has the lock.
         */

        while ( (tempfd = open(LOCKFILE, O_RDWR|O_CREAT|O_EXCL, PERMS)) < 0) {
                if (errno != EEXIST)
                        err_sys("open error for lock file");
                sleep(1);
        }
        close(tempfd);
}

my_unlock(fd)
int     fd;
{
        if (unlink(LOCKFILE) < 0)
                err_sys("unlink error for lock file");
}
```

There are a few points to consider about these three techniques.

1. The first two techniques work under any version of Unix.
2. These techniques take longer to execute than actual file locking system calls, as multiple system calls (such as `creat`, `close`, `link`, and `unlink`) are required, and several filesystem operations are required.
3. An ancillary lock file is required, in addition to the file containing the resource that is shared. In our example, we need both the sequence number file and the ancillary lock file. Since the lock file must be created each time it is needed, a single file cannot be used.
4. Ancillary files such as a lock file might be left around after a system crash and some technique must be devised to handle this (i.e., remove old lock files).
5. The `link` system call in the first technique cannot create a link between files on different logical filesystems, so the temporary file cannot normally be placed in the `/tmp` filesystem. (Many larger Unix systems place the `/tmp` directory in its own filesystem.)
6. The second technique does not work if the processes competing for the resource run with superuser privileges.
7. When the ancillary lock is owned by another process, the process wanting the lock doesn't know when to check again. In our examples we wait for one second. The ideal condition is for the process wanting the lock to be notified when the lock is available.

8. The process that obtains the lock can terminate, intentionally or unintentionally, without releasing the lock.

Cases 4 and 8 require some technique for another process to determine when an existing lock is not in use. A common solution is to assume a timeout period based on the application (a few minutes, perhaps) and remove locks that are older than this. The last-access time or last-modify time in the `stat` structure for the lock file can be used to determine how long a lock has been around. Since this technique fails if the resource is really in use for longer than the timeout period, an enhancement to this is to store the process ID of the process that has the lock in the lock file. This way, the `kill` system call, with a signal argument of zero can be used to determine if the process is still in existence. (We'll see a similar example in Section 13.2—the 4.3BSDline printer spooler stores the process ID of the current server in a file so that it can `kill` it at a later time, if necessary.)

Given the potential problems with these older techniques, true record locking or file locking should be used when available.

We'll return to this file locking example in Section 3.10 when we present another solution using semaphores.

3.3 A Simple Client–Server Example

The client–server example shown in Figure 3.3 is used throughout this chapter for illustration of the various methods of IPC.

The client reads a filename from the standard input and writes it to the IPC channel. The server reads this filename from the IPC channel and tries to open the file for reading. If the server can open the file, it responds by reading the file and writing it to the IPC channel, otherwise the server responds with an ASCII error message stating that it can't open the file. The client then reads from the IPC channel, writing what it receives to the standard output. If the file can't be read by the server, the client reads an error message from the IPC channel. Otherwise the client reads the contents of the file.

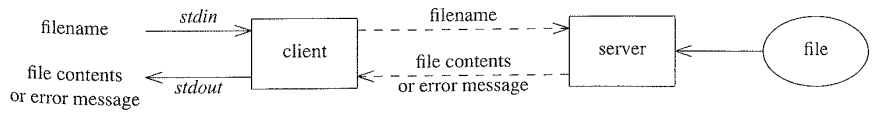


Figure 3.3 Client–server example.

The two dashed lines in Figure 3.3, between the client and server, correspond to some form of IPC.

3.4 Pipes

Pipes are provided with all flavors of Unix. A pipe provides a one-way flow of data. A pipe is created by the `pipe` system call.

```
int pipe(int *filedes);
```

Two file descriptors are returned—*filedes[0]* which is open for reading, and *filedes[1]* which is open for writing. Pipes are of little use within a single process, but here is a simple example that shows how they are created and used.

```
main()
{
        int     pipefd[2], n;
        char    buff[100];

        if (pipe(pipefd) < 0)
                err_sys("pipe error");

        printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
        if (write(pipefd[1], "hello world\n", 12) != 12)
                err_sys("write error");

        if ( (n = read(pipefd[0], buff, sizeof(buff))) <= 0)
                err_sys("read error");

        write(1, buff, n);      /* fd 1 = stdout */

        exit(0);
}
```

The output of this program is

```
hello world
read fd = 3, write fd = 4
```

Note that the "hello world" string is printed on the standard output *before* the output of the `printf`. The reason for this is because the `printf` output is buffered by the standard I/O library, and since its output does not fill a standard I/O buffer (typically 512 or 1024 bytes), the buffer is not output until the process terminates. The `write` function, however, is not buffered at all.

A diagram of what a pipe looks like in a single process is shown in Figure 3.4. A pipe has a finite size, always at least 4096 bytes. The rules for reading and writing a pipe—when there is either no data in the pipe, or when the pipe is full—are provided in the next section on FIFOs.

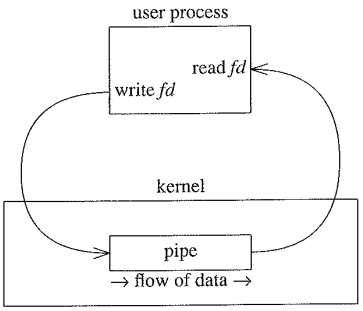


Figure 3.4 Pipe in a single process.

Pipes are typically used to communicate between two different processes in the following way. First, a process creates a pipe and then `forks` to create a copy of itself, as shown in Figure 3.5.

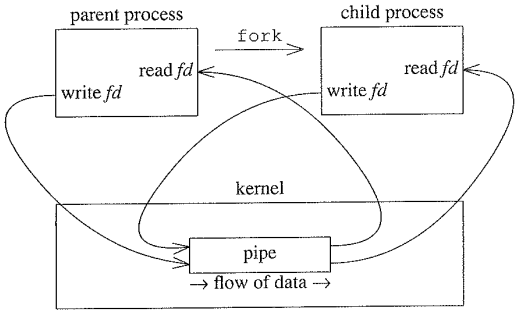


Figure 3.5 Pipe in a single process, immediately after `fork`.

Next the parent process closes the read end of the pipe and the child process closes the write end of the pipe. This provides a one-way flow of data between the two processes as shown in Figure 3.6.

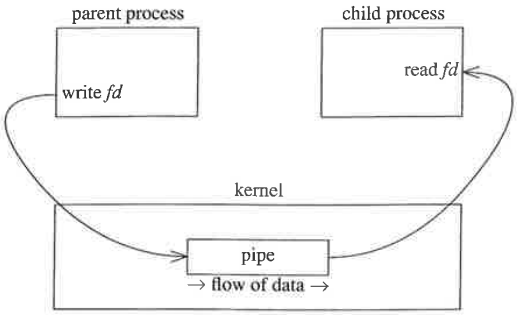


Figure 3.6 Pipe between two processes.

When a user enters a command such as

```
who | sort | lpr
```

to a Unix shell, the shell does the steps shown above to create three processes with two pipes between them. (`who` is a program that outputs the login names, terminal names, and login times of all users on the system. The `sort` program orders this list by login names, and `lpr` is a 4.3BSD program that sends the result to the line printer.) We show this pipeline in Figure 3.7.

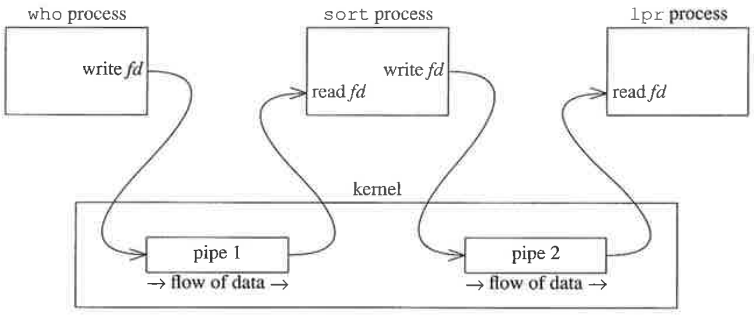


Figure 3.7 Pipes between three processes in a shell pipeline.

Note that all the pipes shown so far have all been 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

- create pipe 1, create pipe 2,
- fork,
- parent closes read end of pipe 1,

- parent closes write end of pipe 2,
- child closes write end of pipe 1,
- child closes read end of pipe 2.

This generates the picture shown in Figure 3.8.

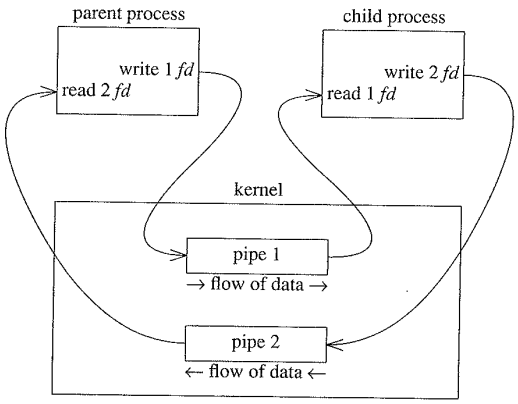


Figure 3.8 Two pipes to provide a bidirectional flow of data.

Let's now implement the client–server example described in the previous section using pipes. The `main` function creates the pipe and `fork`s. The client then runs in the parent process and the server runs in the child process.

```
main()
{
        int     childpid, pipe1[2], pipe2[2];

        if (pipe(pipe1) < 0 || pipe(pipe2) < 0)
                err_sys("can't create pipes");

        if ( (childpid = fork()) < 0) {
                err_sys("can't fork");

        } else if (childpid > 0) {                      /* parent */
                close(pipe1[0]);
                close(pipe2[1]);

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

                while (wait((int *) 0) != childpid)     /* wait for child */
                        ;

                close(pipe1[1]);
                close(pipe2[0]);
                exit(0);
```

```
                                                        /* child */
        } else {
                close(pipe1[1]);
                close(pipe2[0]);

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

                close(pipe1[0]);
                close(pipe2[1]);
                exit(0);
        }
}
```

The client function is

```
#include        <stdio.h>

#define MAXBUFF         1024

client(readfd, writefd)
int     readfd;
int     writefd;
{
        char    buff[MAXBUFF];
        int     n;

        /*
         * Read the filename from standard input,
         * write it to the IPC descriptor.
         */

        if (fgets(buff, MAXBUFF, stdin) == NULL)
                err_sys("client: filename read error");

        n = strlen(buff);
        if (buff[n-1] == '\n')
                n--;                    /* ignore newline from fgets() */
        if (write(writefd, buff, n) != n)
                err_sys("client: filename write error");;

        /*
         * Read the data from the IPC descriptor and write
         * to standard output.
         */

        while ( (n = read(readfd, buff, MAXBUFF)) > 0)
                if (write(1, buff, n) != n)     /* fd 1 = stdout */
                        err_sys("client: data write error");
        if (n < 0)
                err_sys("client: data read error");
}
```

The server function is

```
#include        <stdio.h>

#define MAXBUFF         1024

server(readfd, writefd)
int     readfd;
int     writefd;
{
        char            buff[MAXBUFF];
        char            errmesg[256], *sys_err_str();
        int             n, fd;
        extern int      errno;

        /*
         * Read the filename from the IPC descriptor.
         */

        if ( (n = read(readfd, buff, MAXBUFF)) <= 0)
                err_sys("server: filename read error");
        buff[n] = '\0';         /* null terminate filename */

        if ( (fd = open(buff, 0)) < 0) {
                /*
                 * Error.  Format an error message and send it back
                 * to the client.
                 */

                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(buff, errmesg);
                n = strlen(buff);
                if (write(writefd, buff, n) != n)
                        err_sys("server: errmesg write error");
        } else {
                /*
                 * Read the data from the file and write to
                 * the IPC descriptor.
                 */

                while ( (n = read(fd, buff, MAXBUFF)) > 0)
                        if (write(writefd, buff, n) != n)
                                err_sys("server: data write error");
                if (n < 0)
                        err_sys("server: read error");
        }
}
```

The function `sys_err_str` returns a pointer to an error message string that it builds, based on the system's `errno` value. This function, and other error handling functions that are used throughout the book, are listed in the "Standard Error Routines" section of Appendix A.

The standard I/O library provides a 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(char *command, char *type);
```

command is a shell command line. It is invoked by the 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 the value of *type* is r the calling process reads the standard output of the *command*. If the *type* is w the calling process writes to the standard input of the command. If the popen call fails, a value of NULL is returned. The function

```
#include <stdio.h>

int pclose(FILE *stream);
```

closes an I/O *stream* that was created by popen, returning the exit status of the *command*, or −1 if the *stream* was not created by popen.

We can provide another solution to our client–server example using the popen function and the Unix cat program.

```
#include        <stdio.h>

#define MAXLINE         1024

main()
{
        int     n;
        char    line[MAXLINE], command[MAXLINE + 10];
        FILE    *fp;

        /*
         * Read the filename from standard input.
         */

        if (fgets(line, MAXLINE, stdin) == NULL)
                err_sys("filename read error");

        /*
         * Use popen to create a pipe and execute the command.
         */

        sprintf(command, "cat %s", line);
        if ( (fp = popen(command, "r")) == NULL)
                err_sys("popen error");

        /*
         * Read the data from the FILE pointer and write
         * to standard output.
```

```
         */

        while ((fgets(line, MAXLINE, fp)) != NULL) {
                n = strlen(line);
                if (write(1, line, n) != n)
                        err_sys("data write error");
        }

        if (ferror(fp))
                err_sys("fgets error");

        pclose(fp);
        exit(0);
}
```

Another use of `popen` is to determine the current working directory of a process. Recall that the `chdir` system call changes the current working directory for a process, but there is not an equivalent system call to obtain its current value. System V provides a `getcwd` function to do this. 4.3BSD provides a similar, but not identical function, `getwd`.

The following program obtains the current working directory and prints it, using the Unix `pwd` command.

```
#include        <stdio.h>

#define MAXLINE         255

main()
{
        FILE    *fp;
        char    line[MAXLINE];

        if ( (fp = popen("/bin/pwd", "r")) == NULL)
                err_sys("popen error");

        if (fgets(line, MAXLINE, fp) == NULL)
                err_sys("fgets error");

        printf("%s", line);     /* pwd inserts the newline */

        pclose(fp);
        exit(0);
}
```

The biggest disadvantage with pipes is that they can only be used between processes that have a parent process in common. This is because a pipe is passed from one process to another by the `fork` system call, and the fact that all open files are shared between the parent and child after a `fork`. There is no way for two totally unrelated processes to create a pipe between them and use it for IPC.

3.5 FIFOs

FIFO stands for *First In, First Out.* A Unix FIFO is similar to a pipe. It is a one-way flow of data, with the first byte written to it being the first byte read from it. Unlike pipes, however, a FIFO has a name associated with it, allowing unrelated processes to access a single FIFO. Indeed, FIFOs are also called *named pipes.* FIFOs are in System V but not 4.3BSD. FIFOs came with System III Unix, but have never been well documented, and are therefore ignored by most programmers. FIFOs are used by the System V line printer system, as we see in Section 13.4.

A FIFO is created by the `mknod` system call. This call is normally reserved to the superuser to create new device entries, but any user can create a FIFO.

```
int mknod(char *pathname, int mode, int dev);
```

The *pathname* is a normal Unix pathname, and this is the name of the FIFO. The *mode* argument specifies the file mode access mode (read and write permissions for owner, group, and world) and is logically or'ed with the `S_IFIFO` flag from `<sys/stat.h>` to specify that a FIFO is being created. The *dev* value is ignored for a FIFO. A FIFO can also be created by the `mknod` command, as in

```
/etc/mknod name p
```

Once a FIFO is created, it must be opened for reading or writing, using either the `open` system call, or one of the standard I/O open functions—`fopen`, or `freopen`. Note that it takes three system calls to create a FIFO and open it for reading and writing (`mknod`, `open`, and `open`) while the single `pipe` system call does the same thing.

The `O_NDELAY` flag for a file, which is set either on the call to `open` or by calling `fcntl` after the file has been opened, sets the "no delay" flag for either a pipe or a FIFO. (Since a pipe is opened as part of the `pipe` system call, the only way to set this flag for a pipe is with the `fcntl` system call.) The effect of this flag is shown in Figure 3.9.

The reason the first two rows in Figure 3.9 have the restriction for read-only or write-only is that if a FIFO is opened by a process for *both* reading and writing, there can't be any waiting for a reader or writer.

A pipe or FIFO follows these rules for reading and writing:

- A `read` requesting less data than is in the pipe or FIFO returns only the requested amount of data. The remainder is left for subsequent `read`s.
- If a process asks to read more data than is currently available in the pipe FIFO, only the data available is returned. The process must be prepared to handle a return value from `read` that is less than the requested amount.
- If there is no data in the pipe or FIFO, and if no processes have it open for writing, a `read` returns zero, signifying the end of file. If the reader has specified `O_NDELAY`, it cannot tell if a return value of zero means there is no data currently available or if there are no writers left.

Condition	Normal	O_NDELAY set
open FIFO, read-only with no process having the FIFO open for writing	wait until a process opens the FIFO for writing	return immediately, no error
open FIFO, write-only with no process having the FIFO open for reading	wait until a process opens the FIFO for reading	return an error immediately, errno set to ENXIO
read pipe or FIFO, no data	wait until there is data in the pipe or FIFO, or until no processes have it open for writing; return a value of zero if no processes have it open for writing, otherwise return the count of data	return immediately, return value of zero
write, pipe or FIFO is full	wait until space is available, then write data	return immediately, return value of zero

Figure 3.9 Effect of O_NDELAY flag on pipes and FIFOs.

- If a process writes less than the capacity of a pipe (which is at least 4096 bytes) the write is guaranteed to be *atomic*. This means that if two processes each write to a 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 mix the data from the two processes—i.e., part of the data from one process, followed by part of the data from the other process. If, however, the write specifies more data than the pipe can hold, there is no guarantee that the write operation is atomic.
- If a process writes to a pipe or FIFO, but there are no processes in existence that have it open for reading, the SIGPIPE signal is generated, and the write returns zero with errno set to EPIPE. If the process has not called signal to handle the SIGPIPE notification, the default action is to terminate the process. The only way the write returns is if the process either ignores the SIGPIPE signal, or if it handles the signal and returns from its signal handler.†

Consider a daemon that uses a FIFO to receive client requests (such as the System V print spooler). The daemon opens the FIFO for read-only and its typical state is waiting

† Neither the System V Release 2 nor the System V Release 3 manuals note that this signal is generated for *both* pipes and FIFOs—they mention only pipes. As stated at the beginning of this section, FIFOs are the forgotten (and undocumented) child of IPC.

in a `read` system call for a client request. Client processes are started and they open the FIFO for writing, write their request, and exit. What happens is the `read` returns zero (end of file) to the daemon every time a client process terminates, if no other clients have the FIFO open for writing. The daemon has to then `open` the FIFO again (for read-only) and it waits here until a client process `opens` the FIFO for writing. To avoid this, a useful technique is for the daemon to open the FIFO two times—once for reading and once for writing. The file descriptor returned for reading is used to `read` the client requests, and the file descriptor for writing is never used. By having the FIFO always open for writing (as long as the daemon process exists) the reads do not return an EOF, but wait for the next client request.

We can write our simple client–server example using FIFOs. The client function and the server function don't change—their use of `read` and `write` doesn't change between pipes and FIFOs. The `main` function does change.

```
#include        <sys/types.h>
#include        <sys/stat.h>
#include        <sys/errno.h>
extern int      errno;

#define FIFO1   "/tmp/fifo.1"
#define FIFO2   "/tmp/fifo.2"
#define PERMS   0666

main()
{
        int     childpid, readfd, writefd;

        if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST))
                err_sys("can't create fifo 1: %s", FIFO1);
        if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
                unlink(FIFO1);
                err_sys("can't create fifo 2: %s", FIFO2);
        }

        if ( (childpid = fork()) < 0) {
                err_sys("can't fork");

        } else if (childpid > 0) {                      /* parent */
                if ( (writefd = open(FIFO1, 1)) < 0)
                        err_sys("parent: can't open write fifo");
                if ( (readfd = open(FIFO2, 0)) < 0)
                        err_sys("parent: can't open read fifo");

                client(readfd, writefd);

                while (wait((int *) 0) != childpid)     /* wait for child */
                        ;

                close(readfd);
                close(writefd);

                if (unlink(FIFO1) < 0)
```

```
                err_sys("parent: can't unlink %s", FIFO1);
        if (unlink(FIFO2) < 0)
                err_sys("parent: can't unlink %s", FIFO2);
        exit(0);
                                        /* child */
    } else {
        if ( (readfd = open(FIFO1, 0)) < 0)
                err_sys("child: can't open read fifo");
        if ( (writefd = open(FIFO2, 1)) < 0)
                err_sys("child: can't open write fifo");

        server(readfd, writefd);

        close(readfd);
        close(writefd);
        exit(0);
    }
}
```

We first call `mknod` to create the FIFOs, realizing that they might already exist. After the `fork` both processes must open each of the two FIFOs as desired. The parent process removes the FIFOs with the `unlink` system call, after `wait`ing for the child to terminate.

The order of the `open` calls is important, and avoids a deadlock condition. When the parent opens `FIFO1` for writing, it waits until the child opens it for reading. If the first call to `open` in the child were for `FIFO2` instead of `FIFO1`, then the child would wait for the parent to open `FIFO2` for writing. Each process would be waiting for the other, and neither would proceed. This is called a *deadlock*. The solution is either to code the calls to `open` as shown, or to specify the `O_NDELAY` option to `open`.

When we used pipes to implement this example, the client and server had to originate from the same process, since pipes cannot be shared between unrelated processes. With FIFOs, however, we do not have this restriction. Nevertheless, we used the same program structure for the FIFO example above as we used for the pipe example. We now rewrite the FIFO example, splitting it into two separate programs—one for the client and one for the server.

The `main` function for the client is

```
#include        "fifo.h"

main()
{
        int     readfd, writefd;

        /*
         * Open the FIFOs.  We assume the server has already created them.
         */

        if ( (writefd = open(FIFO1, 1)) < 0)
                err_sys("client: can't open write fifo: %s", FIFO1);
        if ( (readfd = open(FIFO2, 0)) < 0)
                err_sys("client: can't open read fifo: %s", FIFO2);
```

```
        client(readfd, writefd);

        close(readfd);
        close(writefd);

        /*
         * Delete the FIFOs, now that we're finished.
         */

        if (unlink(FIFO1) < 0)
                err_sys("client: can't unlink %s", FIFO1);
        if (unlink(FIFO2) < 0)
                err_sys("client: can't unlink %s", FIFO2);

        exit(0);
}
```

The server `main` function is

```
#include        "fifo.h"

main()
{
        int     readfd, writefd;

        /*
         * Create the FIFOs, then open them - one for reading and one
         * for writing.
         */

        if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST))
                err_sys("can't create fifo: %s", FIFO1);
        if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
                unlink(FIFO1);
                err_sys("can't create fifo: %s", FIFO2);
        }

        if ( (readfd = open(FIFO1, 0)) < 0)
                err_sys("server: can't open read fifo: %s", FIFO1);
        if ( (writefd = open(FIFO2, 1)) < 0)
                err_sys("server: can't open write fifo: %s", FIFO2);

        server(readfd, writefd);

        close(readfd);
        close(writefd);

        exit(0);
}
```

To run this program we first start the server process in the background

```
server &
```

and then run the client in the foreground (since the client reads from the standard input and writes to the standard output):

```
client
```

Alternatively, we could just start the client program by hand, and have it invoke the server program. If the two programs are on the same computer system (as they must be if a FIFO is being used for IPC), the client can `fork` and `exec` the server process. We will see examples later when the client process invokes the required server process automatically. Indeed, when we invoke a client program, we should not need to worry about starting the server process that the client needs—this should be done automatically by the client.

3.6 Streams and Messages

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

There are times, however, when a process wants to impose some structure on the data being transferred. This can happen when the data consists of variable-length messages and it is required that the reader know where the message boundaries are so that it knows when a single message has been read. Many Unix processes that need to impose a message structure on top of a stream based IPC facility do it using the newline character to separate each message. The writing process appends a newline to each message and the reading process reads from the IPC channel, one line at a time. In Chapter 13 we will see that both the 4.3BSD line printer spooler and the System V line printer spooler use this method of delineating messages on an IPC stream.

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` system call, which returns a file descriptor, the standard I/O function `fdopen` must be used to associate an open file descriptor with a standard I/O *stream*. 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 is what the Unix message queue form of IPC does. We can also add more structure to either a pipe or FIFO. We define a message in our `mesg.h` header file as

```
/*
 * Definition of "our" message.
 *
 * You may have to change the 4096 to a smaller value, if message queues
 * on your system were configured with "msgmax" less than 4096.
 */

#define MAXMESGDATA     (4096-16)
                                /* we don't want sizeof(Mesg) > 4096 */

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

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

Each message has a *type* which we define as a long integer whose value must be greater than zero. We ignore the type field for now, but return to it in Section 3.9 when we describe message queues. We also specify a *length* for each message, and allow the length to be zero. If we use the stream format for messages, with newlines separating each message, the length is implied by the newline character at the end of each message. What we are doing with the `Mesg` definition is to precede each message with its length, instead of using newlines to separate the messages. The only advantage this has over the newline separation is if binary messages are being exchanged—with the length preceding the message the actual content of the message does not have to be scanned to find the end. We define two functions to send and receive messages.

```
#include        "mesg.h"

/*
 * Send a message by writing on a file descriptor.
 * The mesg_len, mesg_type and mesg_data fields must be filled
 * in by the caller.
 */

mesg_send(fd, mesgptr)
int     fd;
Mesg    *mesgptr;
{
        int     n;

        /*
         * Write the message header and the optional data.
         * First calculate the length of the length field, plus the
         * type field, plus the optional data.
```

```
         */

        n = MESGHDRSIZE + mesgptr->mesg_len;

        if (write(fd, (char *) mesgptr, n) != n)
                err_sys("message write error");
}

/*
 * Receive a message by reading on a file descriptor.
 * Fill in the mesg_len, mesg_type and mesg_data fields, and return
 * mesg_len as the return value also.
 */

int
mesg_recv(fd, mesgptr)
int     fd;
Mesg    *mesgptr;
{
        int     n;

        /*
         * Read the message header first.  This tells us how much
         * data follows the message for the next read.
         * An end-of-file on the file descriptor causes us to
         * return 0.  Hence, we force the assumption on the caller
         * that a 0-length message means EOF.
         */

        if ( (n = read(fd, (char *) mesgptr, MESGHDRSIZE)) == 0)
                return(0);              /* end of file */
        else if (n != MESGHDRSIZE)
                err_sys("message header read error");

        /*
         * Read the actual data, if there is any.
         */

        if ( (n = mesgptr->mesg_len) > 0)
                if (read(fd, mesgptr->mesg_data, n) != n)
                        err_sys("data read error");
        return(n);
}
```

We now change the client and server functions to use the `mesg_send` and `mesg_recv` functions.

```
#include        <stdio.h>
#include        "mesg.h"

Mesg    mesg;

client(ipcreadfd, ipcwritefd)
int     ipcreadfd;
```

```
int     ipcwritefd;
{
        int     n;

        /*
         * Read the filename from standard input, write it as
         * a message to the IPC descriptor.
         */

        if (fgets(mesg.mesg_data, MAXMESGDATA, stdin) == NULL)
                err_sys("filename read error");

        n = strlen(mesg.mesg_data);
        if (mesg.mesg_data[n-1] == '\n')
                n--;                    /* ignore newline from fgets() */
        mesg.mesg_len = n;
        mesg.mesg_type = 1L;
        mesg_send(ipcwritefd, &mesg);

        /*
         * Receive the message from the IPC descriptor and write
         * the data to the standard output.
         */

        while ( (n = mesg_recv(ipcreadfd, &mesg)) > 0)
                if (write(1, mesg.mesg_data, n) != n)
                        err_sys("data write error");

        if (n < 0)
                err_sys("data read error");
}

#include        <stdio.h>
#include        "mesg.h"

extern int      errno;

Mesg    mesg;

server(ipcreadfd, ipcwritefd)
int     ipcreadfd;
int     ipcwritefd;
{
        int     n, filefd;
        char    errmesg[256], *sys_err_str();

        /*
         * Read the filename message from the IPC descriptor.
         */

        mesg.mesg_type = 1L;
        if ( (n = mesg_recv(ipcreadfd, &mesg)) <= 0)
                err_sys("server: filename read error");
        mesg.mesg_data[n] = '\0';       /* null terminate filename */
```

```
        if ( (filefd = open(mesg.mesg_data, 0)) < 0) {
                /*
                 * Error.  Format an error message and send it back
                 * to the client.
                 */

                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(mesg.mesg_data, errmesg);
                mesg.mesg_len = strlen(mesg.mesg_data);
                mesg_send(ipcwritefd, &mesg);

        } else {
                /*
                 * Read the data from the file and send a message to
                 * the IPC descriptor.
                 */

                while ( (n = read(filefd, mesg.mesg_data, MAXMESGDATA)) > 0) {
                        mesg.mesg_len = n;
                        mesg_send(ipcwritefd, &mesg);
                }
                close(filefd);

                if (n < 0)
                        err_sys("server: read error");
        }

        /*
         * Send a message with a length of 0 to signify the end.
         */

        mesg.mesg_len = 0;
        mesg_send(ipcwritefd, &mesg);
}
```

The `main` functions that call the `client` and `server` functions don't change at all. We can use either the pipe version or the FIFO version.

3.7 Name Spaces

Pipes do not have names but FIFOs have a Unix pathname to identify them. As we move to other forms of IPC in the following sections, additional naming conventions are used. The set of possible names for a given type of IPC is called its *name space*. The name space is important because for all forms of IPC other than plain pipes, the name is how the client and server "connect" to exchange messages. All forms of IPC that we cover in this chapter are restricted to use on a single computer system. When we describe methods for IPC between processes on *different* systems, the choice of names from the available name space becomes important.

Figure 3.10 summarizes the naming conventions used by the different forms of IPC. The last column specifies how a process accesses a particular form of IPC.

IPC type	Name space	Identification
pipe	(no name)	file descriptor
FIFO	pathname	file descriptor
message queue	`key_t` key	identifier
shared memory	`key_t` key	identifier
semaphore	`key_t` key	identifier
socket—Unix domain	pathname	file descriptor
socket—other domains	(domain dependent)	file descriptor

Figure 3.10 Name spaces for IPC.

`key_t` Keys

The function `ftok` is provided by the System V standard C library to convert a pathname and project identifier into a System V IPC key.† These System V IPC keys are used to identify message queues, shared memory, and semaphores, all of which are described in the following sections:

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

key_t ftok(char *pathname, char proj);
```

The file `<sys/types.h>` defines the `key_t` datatype, which is typically a 32-bit integer.

An IPC application should have 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 only need a single IPC channel between them, a *proj* 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 a *proj* of one, and the other a *proj* of two, for example. Once the *pathname* and *proj* are agreed on by the client and server, then both can call the `ftok` function to convert these into the same IPC key.

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 which can change the key returned by `ftok` to the next caller.

† For some reason this function is found in the System V manual under *stdipc*(3C), instead of its real name.

Note that the *proj* argument to `ftok` is an 8-bit character, not a pointer to a character string.

The normal implementation of the `ftok` function is as follows. It combines the 8-bit *proj* value, along with the numeric i-node number of the disk file corresponding to the specified *pathname*, along with the minor device number of the filesystem on which the disk file resides. The combination of these three values produces a 32-bit key. By using the i-node number of the file within the filesystem that it resides on, along with the minor device number of that filesystem (which corresponds to the disk partition of a single disk controller), this function attempts to generate a unique key value. To *guarantee* a unique key for every possible pathname on a system, we need to combine the 32-bit i-node number, the 8-bit major device number, and the 8-bit minor device number. The algorithm chosen by `ftok` is a compromise, since there is a nonzero (but small) chance that two different pathnames can generate the same key value. But few real i-node numbers are larger than 16 bits and to have two different pathnames generate the same key, both files must have the same i-node number and the same minor device number, but have different major device numbers.

3.8 System V IPC

The three types of IPC

- message queues
- semaphores
- shared memory

are collectively referred to as "System V IPC." They share many similarities in the system calls that access them, and in the information that the kernel maintains on them. A summary of their system calls is shown in Figure 3.11.

	Message queue	Semaphore	Shared memory
include file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
system call to create or open	`msgget`	`semget`	`shmget`
system call for control operations	`msgctl`	`semctl`	`shmctl`
system calls for IPC operations	`msgsnd`	`semop`	`shmat`
	`msgrcv`		`shmdt`

Figure 3.11 Summary of System V IPC system calls.

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

```
struct ipc_perm {
   ushort  uid;      /* owner's user id */
   ushort  gid;      /* owner's group id */
   ushort  cuid;     /* creator's user id */
```

```
    ushort  cgid;    /* creator's group id */
    ushort  mode;    /* access modes */
    ushort  seq;     /* slot usage sequence number */
    key_t   key;     /* key */
};
```

This structure, and other manifest constants for the System V IPC system calls, are defined in <sys/ipc.h>. The three ctl system calls are used to access this structure and modify it.

The three get system calls that create or open an IPC channel, all take a *key* value, whose type is key_t, and return an integer identifier. The sequence of steps is shown in Figure 3.12.

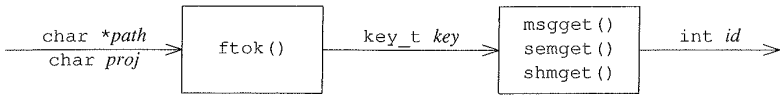


Figure 3.12 Generating IPC *id*s using ftok.

All three of the get system calls also take a *flag* argument that specifies the low-order 9 bits of the mode for the IPC channel, and whether a new IPC channel is being created, or if an existing one is being referenced. The rules for whether a new IPC channel is created or whether an existing one is referenced are

- Specifying a *key* of IPC_PRIVATE guarantees that a unique IPC channel is created. There are no combinations of *pathname* and *proj* that cause ftok to generate a *key* value of IPC_PRIVATE.
- Setting the IPC_CREAT bit of the *flag* word 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 *flag* word creates a new entry for the specified *key*, only if the entry does not already exist. If an existing entry is found, an error occurs, since the IPC channel already exists.

  Note that the IPC_EXCL bit does *not* guarantee the caller exclusive access to the IPC channel. Its only function is to guarantee that a new IPC channel is created, or else an error is returned. Even if we create a new IPC channel by specifying both IPC_CREAT and IPC_EXCL, other users can also access that channel, if they have adequate permission as specified by the low-order 9 bits of the channel's mode word.
- Setting the IPC_EXCL bit, without setting the IPC_CREAT bit, has no meaning.

The actual logic flow for opening an IPC channel is shown in Figure 3.13.

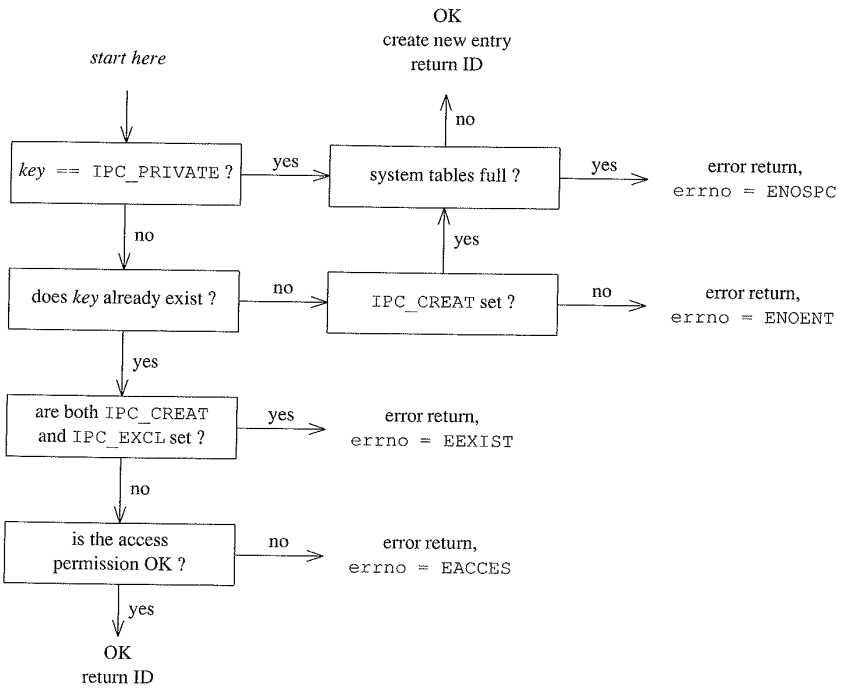


Figure 3.13 Logic for opening or creating an IPC channel.

Another way of looking at Figure 3.13 is shown in Figure 3.14.

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

Figure 3.14 Logic for creating or opening an IPC channel.

One thing to note is that for the middle line of Figure 3.14, the `IPC_CREAT` flag without `IPC_EXCL`, we do not get an indication whether a new entry has been created or if we are referencing an existing entry.

Whenever a new IPC channel is created, using one of the `get` system calls with the `IPC_CREAT` flag, the low-order 9 bits of the *flag* argument initialize the `mode` word in the `ipc_perm` structure. Additionally the `cuid` and `cgid` fields are set to the effective user ID and effective group ID of the calling process, respectively. The two fields `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 latter two IDs are called the "owner" IDs, while the `cuid` and `cgid` are called the "creator" IDs. The creator IDs never change, while a process can change the owner IDs, by calling the `ctl` system call for the IPC mechanism—`msgctl`, `semctl`, or `shmctl`. These three `ctl` system calls also allow a process to change the low-order 9 bits of the `mode` word for the IPC channel.

Two levels of checking are done whenever an IPC channel is accessed by any process.

1. Whenever a process accesses an existing IPC channel with one of the `get` system calls, an initial check is made that the caller's *flag* argument does not specify any access bits that are not in the `mode` word. For example, a server process can set the `mode` word for its input message queue so that the group-read and world-read permission bits are off. Any process that tries to specify a *flag* argument that includes these bits gets an error return from the `msgget` system call. But this test that's done by the `get` system calls is of little use. It implies that the caller knows which permission category it falls into—owner, group, or world. If the creator specifically turns off certain permission bits, and if the caller specifies these bits, the error is detected by the `get` system call. Any process, however, can totally bypass this check by just specifying a *flag* argument of zero if it knows that the IPC channel 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` system call, tests similar to those done for filesystem access are performed.
   - The superuser is always granted access.
   - If the effective user ID equals either the `uid` value or the `cuid` value for the IPC channel, and if the appropriate access bit is on in the `mode` word for the IPC channel, 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 channel (receiving a message from a message queue, for example), or the write-bit must be set for a write operation.
   - If the effective group ID equals either the `gid` value or the `cgid` value for the IPC channel, and if the appropriate access bit is on in the `mode` word for the IPC channel, permission is granted.
   - If none of the above tests are "true," the appropriate world access bit must be on in the `mode` word for the IPC channel, for permission to be allowed.

The ipc_perm structure 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 channel in the system. Every time an IPC channel is closed, the kernel increments the slot number, cycling it back to zero when it overflows.

This counter is needed for the following reasons. First consider the file descriptors maintained by the kernel for open files. They are small integers, but only have meaning within a single process—they are process specific numbers. If we try to read from file descriptor 4, say, in a process, this only works if the process has a file open on this descriptor. It has no effect whatsoever with a file that might be open on file descriptor 4 in some other unrelated process. IPC channels, however, are *systemwide* and not process specific. We obtain an IPC identifier (similar to a file descriptor) from one of the get system calls—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 system call is the same integer value in both processes. This means that a devious process could try to read a message using 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.

To avoid this potential problem, the designers of the System V IPC facilities decided to increase the possible range of identifier values to include *all* integers, not just small integers. This 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 an unsigned short integer (see the ipc_perm structure shown earlier in this section) it cycles after the table entry has been used 65,535 times. An an example of this pattern, the following program prints the first 10 identifier values returned by msgget.

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

#define KEY     ((key_t) 98765L)
#define PERMS   0666

main()
{
        int     i, msqid;

        for (i = 0; i < 10; i++) {
                if ( (msqid = msgget(KEY, PERMS | IPC_CREAT)) < 0)
                        err_sys("can't create message queue");

                printf("msqid = %d\n", msqid);
```

```
                if (msgctl(msqid, IPC_RMID, (struct msqid_ds *) 0) < 0)
                        err_sys("can't remove message queue");
        }
}
```

Its output is

```
msqid = 0
msqid = 50
msqid = 100
msqid = 150
msqid = 200
msqid = 250
msqid = 300
msqid = 350
msqid = 400
msqid = 450
```

3.9 Message Queues

Some form of message passing between processes is now part of many modern operating systems. Some operating systems restrict the passing of messages such that a process can only send a message to another specific process. System V has no such restriction. In the System V implementation of messages, all messages are stored in the kernel, and have an associated *message queue identifier*. It is this identifier, which we call an *msqid*, that identifies a particular queue of messages. Processes read and write messages to arbitrary queues. There is no requirement that any process be waiting for a message to arrive on a queue before some other process is allowed to write a message to that queue. This is in contrast to both pipes and FIFOs, where it made no sense to have a writer process unless a reader process also exists. It is possible for a process to write some messages to a queue, then exit, and have the messages read by another process at a later time.

Every message on a queue has the following attributes:

- long integer *type*;
- *length* of the data portion of the message (can be zero);
- *data* (if the *length* is greater than zero).

For every message queue in the system, the kernel maintains the following structure of information:

```
#include  <sys/types.h>
#include  <sys/ipc.h>       /* defines the ipc_perm structure */

struct msqid_ds {
  struct ipc_perm msg_perm;   /* operation permission struct */
  struct msg      *msg_first; /* ptr to first message on q */
  struct msg      *msg_last;  /* ptr to last message on q */
```

```
    ushort          msg_cbytes; /* current # bytes on q */
    ushort          msg_qnum;   /* current # of messages on q */
    ushort          msg_qbytes; /* max # of bytes allowed on q */
    ushort          msg_lspid;  /* pid of last msgsnd */
    ushort          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) */
};
```

The `ipc_perm` structure was described in Section 3.8 and contains the access permissions for this particular message queue. The `msg` structures are the internal data structure used by the kernel to maintain the linked list of messages on a particular queue.

We can picture a particular message queue in the kernel as a linked list of messages, as shown in Figure 3.15. 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.

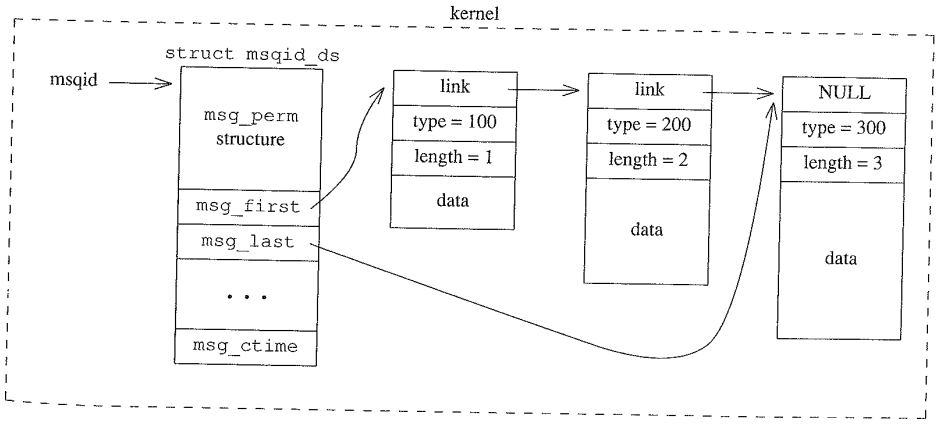


Figure 3.15 Message queue structures in kernel.

A new message queue is created, or an existing message queue is accessed with the `msgget` system call

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

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

The *msgflag* value is a combination of the constants shown in Figure 3.16.

Numeric	Symbolic	Description
0400	MSG_R	Read by owner
0200	MSG_W	Write by owner
0040	MSG_R >> 3	Read by group
0020	MSG_W >> 3	Write by group
0004	MSG_R >> 6	Read by world
0002	MSG_W >> 6	Write by world
	IPC_CREAT	(See Section 3.8)
	IPC_EXCL	(See Section 3.8)

Figure 3.16 *msgflag* values for `msgget` system call.

The value returned by `msgget` is the message queue identifier, *msqid*, or −1 if an error occurred.

Once a message queue is opened with `msgget`, we put a message on the queue using the `msgsnd` system call.

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

int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

The *ptr* argument is a pointer to a structure with the following template. (This template is defined in `<sys/msg.h>`.)

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

The name *mtext* 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 message type must be greater than zero, since a message type of zero is used as a special indicator to the `msgrcv` system call, which we describe later. By template we mean that the `ptr` argument must point to a long integer that contains the message type, followed by the message itself (if the length of the message is greater than zero bytes). The kernel does not interpret the contents of the message at all. If some cooperating processes wanted to exchange messages consisting of a short integer followed by an 8-byte character array, they can define their own structure:

```
typedef struct my_msgbuf {
  long   mtype;      /* message type */
  short  mshort;     /* start of message data */
  char   mchar[8];
} 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 zero.

The *flag* argument can be specified as either IPC_NOWAIT or as zero. The IPC_NOWAIT value allows the system call to return immediately if there is no room on the message queue for the new message. This condition can occur if either there are too many messages on the specified queue, or if there are too many messages systemwide. If there is no room for the message, and if IPC_NOWAIT is specified, msgsnd returns −1 and errno is set to EAGAIN. If the system call is successful, msgsnd returns zero.

A message is read from a message queue using the msgrcv system call.

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

int msgrcv(int msqid, struct msgbuf *ptr, int length, long msgtype,
           int flag);
```

The *ptr* argument is like the one for msgsnd, and specifies where the received message is stored. *len* specifies the size of the data portion of the structure pointed to by *ptr*. This is the maximum amount of data that is returned by the system call. If the MSG_NOERROR bit in the *flag* argument is set, this specifies that if the actual data portion of the received message is greater than *length*, just truncate the data portion and return without an error. Not specifying the MSG_NOERROR flag causes an error return if *length* is not large enough to receive the entire message.

The long integer *msgtype* argument specifies which message on the queue is desired.

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

Consider the message queue example shown in Figure 3.15, which has three messages: the first message has a type of 100 and a length of 1, the next 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 3.17 shows the message returned for different values of *msgtype*.

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, the msgrcv system call returns immediately if a message is not available. In this case the system call returns a −1 with errno set to ENOMSG. Otherwise, the caller is suspended until one of the following occurs:

msgtype	Type of message returned
0L	100
100L	100
200L	200
300L	300
-100L	100
-200L	100
-300L	100

Figure 3.17 Messages returned by `msgrcv` for different values of *msgtype*.

- a message of the requested type is available,
- the message queue is removed from the system,
- the process receives a signal that is caught.

Additionally, the `MSG_NOERROR` bit of the *flag* argument can be set, as mentioned above.

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

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

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

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

Our only use will be a *cmd* of `IPC_RMID` to remove a message queue from the system.

We now show the code for our client–server example, using 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 file `msgq.h` is

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

#include        <sys/errno.h>
extern int      errno;

#define MKEY1   1234L
#define MKEY2   2345L

#define PERMS   0666
```

The `main` function for the server is

```
#include        "msgq.h"

main()
{
        int     readid, writeid;

        /*
         * Create the message queues, if required.
         */

        if ( (readid = msgget(MKEY1, PERMS | IPC_CREAT)) < 0)
                err_sys("server: can't get message queue 1");
        if ( (writeid = msgget(MKEY2, PERMS | IPC_CREAT)) < 0)
                err_sys("server: can't get message queue 2");

        server(readid, writeid);

        exit(0);
}
```

The `main` function for the client is

```
#include        "msgq.h"

main()
{
        int     readid, writeid;

        /*
         * Open the message queues.  The server must have
         * already created them.
         */

        if ( (writeid = msgget(MKEY1, 0)) < 0)
                err_sys("client: can't msgget message queue 1");
        if ( (readid = msgget(MKEY2, 0)) < 0)
                err_sys("client: can't msgget message queue 2");

        client(readid, writeid);

        /*
         * Now we can delete the message queues.
         */

        if (msgctl(readid, IPC_RMID, (struct msqid_ds *) 0) < 0)
                err_sys("client: can't RMID message queue 1");
        if (msgctl(writeid, IPC_RMID, (struct msqid_ds *) 0) < 0)
                err_sys("client: can't RMID message queue 2");

        exit(0);
}
```

Both of these program use the `client` and `server` functions that were shown in Section 3.6. These two functions in turn call the following `mesg_send` and `mesg_recv` functions that use message queues.

```
#include        "mesg.h"

/*
 * Send a message using the System V message queues.
 * The mesg_len, mesg_type and mesg_data fields must be filled
 * in by the caller.
 */

mesg_send(id, mesgptr)
int     id;             /* really an msqid from msgget() */
Mesg    *mesgptr;
{
        /*
         * Send the message - the type followed by the optional data.
         */

        if (msgsnd(id, (char *) &(mesgptr->mesg_type),
                                        mesgptr->mesg_len, 0) != 0)
                err_sys("msgsnd error");
}

/*
 * Receive a message from a System V message queue.
 * The caller must fill in the mesg_type field with the desired type.
 * Return the number of bytes in the data portion of the message.
 * A 0-length data message implies end-of-file.
 */

int
mesg_recv(id, mesgptr)
int     id;             /* really an msqid from msgget() */
Mesg    *mesgptr;
{
        int     n;

        /*
         * Read the first message on the queue of the specified type.
         */

        n = msgrcv(id, (char *) &(mesgptr->mesg_type), MAXMESGDATA,
                                        mesgptr->mesg_type, 0);
        if ( (mesgptr->mesg_len = n) < 0)
                err_dump("msgrcv error");

        return(n);              /* n will be 0 at end of file */
}
```

Multiplexing Messages

The purpose of having a type associated with each message is to allow multiple processes to *multiplex* messages onto a single queue. Consider 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 if 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 process specifies the *msgtype* argument to `msgrcv` as its process ID. Figure 3.18 shows how a single message queue can be used to multiplex these messages between multiple processes.

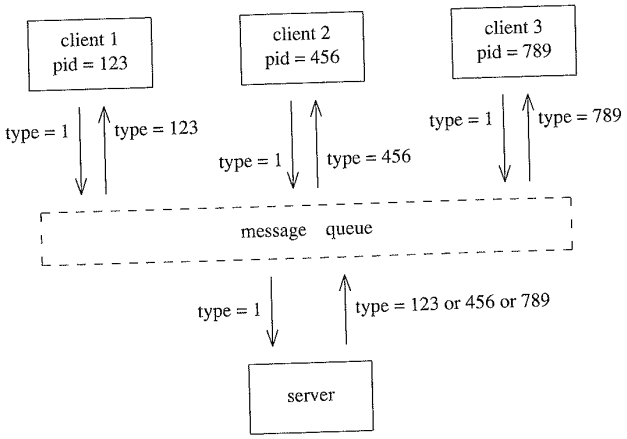


Figure 3.18 Multiplexing messages between three clients and one server.

Another feature provided by the type attribute of messages if the ability of the receiver to read the messages in an order other than first-in, first-out. With pipes and FIFOs, the data must be read in the order in which it was written. With message queues we can read the messages in any order that is consistent with the values we associate with the message types. We can, in essence, assign priorities to the messages by associating a priority to a type, or range of 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 there are no messages of the specified type.

Now we can redo the client–server example using a single message queue. These programs use the convention that messages with a type of 1 are from the client to the server, and messages with a type of 2 are from the server to the client. The server program is

```
#include        <stdio.h>
#include        "mesg.h"
#include        "msgq.h"

Mesg    mesg;

main()
{
        int     id;

        /*
         * Create the message queue, if required.
         */

        if ( (id = msgget(MKEY1, PERMS | IPC_CREAT)) < 0)
                err_sys("server: can't get message queue 1");

        server(id);

        exit(0);
}

server(id)
int     id;
{
        int     n, filefd;
        char    errmesg[256], *sys_err_str();

        /*
         * Read the filename message from the IPC descriptor.
         */

        mesg.mesg_type = 1L;            /* receive messages of this type */
        if ( (n = mesg_recv(id, &mesg)) <= 0)
                err_sys("server: filename read error");
        mesg.mesg_data[n] = '\0';       /* null terminate filename */

        mesg.mesg_type = 2L;            /* send messages of this type */
        if ( (filefd = open(mesg.mesg_data, 0)) < 0) {
                /*
                 * Error.  Format an error message and send it back
                 * to the client.
                 */

                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(mesg.mesg_data, errmesg);
                mesg.mesg_len = strlen(mesg.mesg_data);
                mesg_send(id, &mesg);
```

```
        } else {
                /*
                 * Read the data from the file and send a message to
                 * the IPC descriptor.
                 */

                while ( (n = read(filefd, mesg.mesg_data, MAXMESGDATA)) > 0) {
                        mesg.mesg_len = n;
                        mesg_send(id, &mesg);
                }
                close(filefd);

                if (n < 0)
                        err_sys("server: read error");
        }

        /*
         * Send a message with a length of 0 to signify the end.
         */

        mesg.mesg_len = 0;
        mesg_send(id, &mesg);
}
```

The client program is

```
#include        <stdio.h>
#include        "mesg.h"
#include        "msgq.h"

Mesg    mesg;

main()
{
        int     id;

        /*
         * Open the single message queue.  The server must have
         * already created it.
         */

        if ( (id = msgget(MKEY1, 0)) < 0)
                err_sys("client: can't msgget message queue 1");

        client(id);

        /*
         * Now we can delete the message queue.
         */

        if (msgctl(id, IPC_RMID, (struct msqid_ds *) 0) < 0)
                err_sys("client: can't RMID message queue 1");

        exit(0);
```

```
}

client(id)
int     id;
{
        int     n;

        /*
         * Read the filename from standard input, write it as
         * a message to the IPC descriptor.
         */

        if (fgets(mesg.mesg_data, MAXMESGDATA, stdin) == NULL)
                err_sys("filename read error");

        n = strlen(mesg.mesg_data);
        if (mesg.mesg_data[n-1] == '\n')
                n--;                    /* ignore the newline from fgets() */
        mesg.mesg_data[n] = '\0';       /* overwrite newline at end */
        mesg.mesg_len = n;
        mesg.mesg_type = 1L;            /* send messages of this type */
        mesg_send(id, &mesg);

        /*
         * Receive the message from the IPC descriptor and write
         * the data to the standard output.
         */

        mesg.mesg_type = 2L;    /* receive messages of this type */
        while( (n = mesg_recv(id, &mesg)) > 0)
                if (write(1, mesg.mesg_data, n) != n)
                        err_sys("data write error");

        if (n < 0)
                err_sys("data read error");
}
```

Both of these programs use the `mesg_send` and `mesg_recv` functions that were shown earlier in this section for the example that used two message queues.

Message Queue Limits

There are certain system limits on message queues. Some of these can be changed by a system administrator, by configuring a new kernel. Figure 3.19 shows the values for five different versions of Unix.†

† Note that three of these systems specify a per-queue maximum of 16,384 bytes when they only allow a systemwide maximum of 8,192 bytes. This doesn't make sense. Also, these three systems allow more total message queues (50) than messages (40), meaning we can't have one message per queue outstanding, at any point in time.

Parameter name	AT&T VAX Sys V Rel 2	DEC VAX Ultrix	AT&T Unix PC	Xenix 286	Xenix 386	Description
msgmax	8,192	8,192	8,192	8,192	8,192	max # of bytes per message
msgmnb	16,384	16,384	16,384	8,192	8,192	max # of bytes on any one message queue
msgmni	50	50	50	10	10	max # of message queues, systemwide
msgtql	40	40	40	60	60	max # of messages, systemwide
	8,192	8,192	8,192	8,192	8,192	max # of bytes of messages, systemwide

Figure 3.19 Typical limits for message queues.

The intent is to show some typical values, to aid in planning for portability. For example, if we write an application that uses messages that are 20,000 bytes long, it will not be portable. The magic number 8192 bytes for a maximum message size is similar to the magic number of 4096 bytes for a maximum write to a pipe of FIFO.

3.10 Semaphores

Semaphores are a synchronization primitive. As a form of IPC, they are not used for exchanging large amounts of data, as are pipes, FIFOs, and message queues, but are intended to let multiple processes synchronize their operations. Our main use of semaphores is to synchronize the access to shared memory segments, which we discuss in the next section. This section provides a general overview of the semaphore system calls.

Consider a semaphore as an integer valued variable that is a resource counter. The value of the variable at any point in time is the number of resource units available. If we have one resource, say a file that is shared, then the valid semaphore values are zero and one.

Since our use of semaphores is to provide resource synchronization between different processes, the actual semaphore value must be stored in the kernel. We show this in Figure 3.20.

To obtain a resource that is controlled by a semaphore, a process needs to test its current value, and if the current value is greater than zero, decrement the value by one. If the current value is zero, the process must wait until the value is greater than zero (i.e., wait for some other process to release the resource). To release a resource that is controlled by a semaphore, a process increments the semaphore value. If some other process has been waiting for the semaphore value to become greater than zero, that other process can now obtain the semaphore. We can show this in pseudocode as

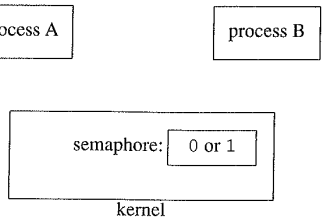


Figure 3.20 Semaphore value stored in kernel.

```
for ( ; ; ) {
        if (semaphore_value > 0) {
                semaphore_value--;
                break;
        }
}

/* we've obtained the semaphore */
```

The pitfall in this implementation of semaphores is that the test of the semaphore's current value, followed by the decrement of that value, is not an atomic operation. It must not be possible for one process to do the test and then be interrupted by another process that does the same test and then decrements the value. The System V implementation of semaphores is done in the kernel, where it is possible to guarantee that a group of operations on a semaphore is done atomically, with respect to other processes.

What we have described as a semaphore is a single *binary semaphore*—a semaphore with a single value that can be either zero or one. The System V implementation expands this in two directions.

1. A semaphore is not a single value but a *set* of nonnegative integer values. The number of nonnegative integer values in the set can be from one to some system defined maximum.
2. Each value in the set is not restricted to zero and one. Instead each value in the set can assume any nonnegative value, up to a system defined maximum value.

For every set of semaphores in the system, the kernel maintains the following structure of information.

```
#include  <sys/types.h>
#include  <sys/ipc.h>        /* defines the ipc_perm structure */

struct semid_ds {
  struct ipc_perm sem_perm;  /* operation permission struct */
  struct sem      *sem_base; /* ptr to first semaphore in set */
```

```
    ushort          sem_nsems; /* # of semaphores in set */
    time_t          sem_otime; /* time of last semop */
    time_t          sem_ctime; /* time of last change */
};
```

The `ipc_perm` structure was described in Section 3.8 and contains the access permissions for this particular semaphore. The `sem` structure is the internal data structure used by the kernel to maintain the set of values for a given semaphore. Every member of a semaphore set is described by the following structure:

```
struct sem {
  ushort  semval;   /* semaphore value, nonnegative */
  short   sempid;   /* pid of last operation */
  ushort  semncnt;  /* # awaiting semval > cval */
  ushort  semzcnt;  /* # awaiting semval = 0 */
};
```

In addition to maintaining the actual set of values for a semaphore, the kernel also maintains three other pieces of information for each value in the set—the process ID of the process that did the last operation on the value, a count of the number of processes waiting for the value to increase, and a count of the number of processes waiting for the value to become zero.

We can picture a particular semaphore in the kernel as being a `semid_ds` structure that points to an array of `sem` structures. If the semaphore has two members in its set, we would have the picture shown in Figure 3.21.

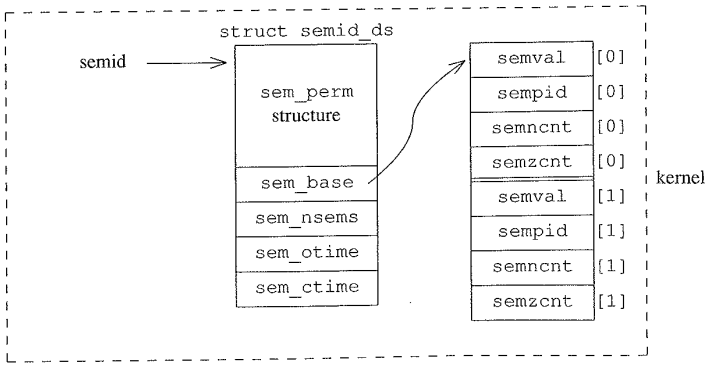


Figure 3.21 Kernel data structures for a semaphore set.

In Figure 3.21, the variable `sem_nsems` has a value of two and we have denoted each member of the set with the subscripts `[0]` and `[1]`.

A semaphore is created, or an existing semaphore is accessed with the `semget` system call.

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

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

The value returned by `semget` is the semaphore identifier, *semid*, or −1 if an error occurred.

The *nsems* argument specifies the number of semaphores in the set. If we are not creating a new semaphore set, but only accessing an existing set (i.e., we did not specify the `IPC_CREAT` flag for the *semflag* argument), we can specify this argument as zero. We cannot change the number of semaphores in a set once it is created.

The *semflag* value is a combination of the constants shown in Figure 3.22.

Numeric	Symbolic	Description
0400	SEM_R	Read by owner
0200	SEM_A	Alter by owner
0040	SEM_R >> 3	Read by group
0020	SEM_A >> 3	Alter by group
0004	SEM_R >> 6	Read by world
0002	SEM_A >> 6	Alter by world
	IPC_CREAT	(See Section 3.8)
	IPC_EXCL	(See Section 3.8)

Figure 3.22 *semflag* values for `semget` system call.

Once a semaphore set is opened with `semget`, operations are performed on one or more of the semaphore values in the set using the `semop` system call.

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

int semop(int semid, struct sembuf **opsptr, unsigned int nops);
```

The pointer *opsptr* points to an array of the following structures:

```
struct sembuf {
  ushort  sem_num;  /* semaphore # */
  short   sem_op;   /* semaphore operation */
  short   sem_flg;  /* operation flags */
};
```

The number of elements in the array of `sembuf` structures pointed to by *opsptr* is specified by the *nops* argument. Each element in this array specifies an operation for one particular semaphore value in the set. The particular semaphore value is specified by the `sem_num` value, which is zero for the first element, one for the second, and so on, up to *nsems−1*, where *nsems* is the number of semaphore values in the set (the second argument in the call to `semget` when the semaphore set was created).

The array of operations passed to the `semop` system call are guaranteed to be done atomically by the kernel. The kernel either does all the operations that are specified, or it doesn't do any of them. Each particular operation is specified by a `sem_op` value, which can be negative, zero, or positive.

1. If `sem_op` is positive, the value of `sem_val` is added to the semaphore's current value. This corresponds to the release of resources that a semaphore controls.
2. If `sem_op` is zero, the caller wants to wait until the semaphore's value becomes zero.
3. If `sem_op` is negative, the caller wants to wait until the semaphore's value becomes greater than or equal to the absolute value of `sem_op`. This corresponds to the allocation of resources.

The return value from `semop` is zero if all is OK, or −1 if an error occurred.†

There are a multitude of options available to the `semop` system call. We can specify, for example, the `IPC_NOWAIT` flag for the `sem_flg` value associated with an operation. This tells the system we don't want to wait if that operation can't be completed. There is also the capability of having the system remember adjustment values for a given semaphore so that if a process exits before releasing a semaphore, the system does the release for the process. We describe this option later in this section.

The `semctl` system call provides various control operations on a semaphore.

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

int semctl(int semid, int semnum, int cmd, union semun arg);

union semun {
  int               val;    /* used for SETVAL only */
  struct semid_ds   *buff;  /* used for IPC_STAT and IPC_SET */
  ushort            *array; /* used for IPC_GETALL & IPC_SETALL */
} arg;
```

We'll use a *cmd* of `IPC_RMID` to remove a semaphore from the system. We'll also use the `GETVAL` and `SETVAL` commands, to fetch and set a specific semaphore value. These two commands use the *semnum* argument to specify one member of the semaphore set. The `GETVAL` command returns the semaphore's value as the value of the system call. The `SETVAL` command sets the semaphore's value to *arg.val*.

† Both the System V Release 2 manual page for *semop*(2) and Bach [1986] say that the return value is the value of the last semaphore that was operated on, before it was changed. This is an error. The System V Release 3 manual page correctly states that the return value from a successful call is zero.

File Locking with Semaphores

Since semaphores can be thought of as a synchronization facility, we can return to the file locking examples from Section 3.2 and provide yet another implementation, using semaphores.

We create a binary semaphore—a single semaphore value that is either zero or one. To lock the semaphore we call `semop` to do two operations atomically. First, wait for the semaphore value to become zero (a `sem_op` value of zero), then increment the value to one. This is an example where multiple semaphore operations must be done atomically by the kernel. If it took two system calls to do this—one to test the value and wait for it to become zero, and another to increment the value—the operation would not work.

To unlock the resource, we call `semop` to decrement the semaphore value. Since we have the lock on the resource, we know that the semaphore value is one before the call, so the call cannot wait. We explicitly set `sem_flg` to `IPC_NOWAIT` so that if this "impossible condition" does happen, an error return occurs and the process aborts.

You might wonder why we don't initialize the semaphore value to one and use it as a resource counter, as we described earlier. Then, to allocate the resource we wait for its value to become greater than zero, and decrement it. The release of the resource would be done by incrementing the semaphore value. The reason we use the semaphore in our "backwards" fashion has to do with an initialization problem with System V semaphores. It is hard to initialize a semaphore to a value other than zero, as we soon see.

```
/*
 * Locking routines using semaphores.
 */

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

#define SEMKEY  123456L /* key value for semget() */
#define PERMS   0666

static struct sembuf    op_lock[2] = {
        0, 0, 0,        /* wait for sem#0 to become 0 */
        0, 1, 0         /* then increment sem#0 by 1 */
};

static struct sembuf    op_unlock[1] = {
        0, -1, IPC_NOWAIT       /* decrement sem#0 by 1 (sets it to 0) */
};

int     semid = -1;     /* semaphore id */

my_lock(fd)
int     fd;
{
        if (semid < 0) {
```

```
                if ( (semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
                        err_sys("semget error");
        }
        if (semop(semid, &op_lock[0], 2) < 0)
                err_sys("semop lock error");
}

my_unlock(fd)
int     fd;
{
        if (semop(semid, &op_unlock[0], 1) < 0)
                err_sys("semop unlock error");
}
```

There is a slight problem with this implementation—if the process aborts for some reason while it has the lock, the semaphore value is left at one. Any other process that tries to obtain the lock waits forever when it does the locking `semop` that first waits for the value to become zero. There are some ways around this.

- A process that has a lock can set up signal handlers to catch all possible signals, and remove the lock before terminating. The only problem with this is that there are some signals that a process can't catch, such as `SIGKILL`.
- The `my_lock` function can become more sophisticated. It can specify the `IPC_NOWAIT` flag on the first operation in the `op_lock` array. If the `semop` returns an error with `errno` equal to `EAGAIN`, the process can call the `semctl` system call and look at the `sem_ctime` value for the semaphore. If some long amount of time has passed since the last change to the semaphore (a few minutes, perhaps) the process can assume that the lock has been abandoned by some other process and can go ahead and remove the lock. The problem with this solution is that it takes an additional system call every time the resource is locked, and we have to guess what amount of time implies that the lock has been left around.
- The third solution is to tell the kernel when we obtain the lock that if this process terminates before releasing the lock, release it for the process.

This third solution is provided by the System V implementation of semaphores. Every semaphore value in the system can optionally have another value associated with it. This other value is maintained by the kernel and is called the *semaphore adjustment value*. The rules for this adjustment value are simple.

1. When a semaphore value is initialized, whether created by `semget`, or specifically set by `semctl`s `SETVAL` or `SETALL` commands, the adjustment value for that particular semaphore value is set to zero. Similarly, when a semaphore is deleted, any adjustment values associated with it are also deleted.

2. For every semop operation that specifies the SEM_UNDO flag, if the semaphore value goes up, the adjustment value goes down by the same amount. If the semaphore value goes down, the adjustment value goes up by the same amount.
3. When a process exits, the kernel automatically applies any adjustment values for that process.

Note that to take advantage of this feature, we must specify the SEM_UNDO flag for the corresponding operation. Also, we have to specify the SEM_UNDO flag consistently, that is, for both the allocation and deallocation, so that the corresponding adjustment value mirrors the semaphore's value.

We can change our file locking example to use this feature.

```
/*
 * Locking routines using semaphores.
 * Use the SEM_UNDO feature to have the kernel adjust the
 * semaphore value on premature exit.
 */

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

#define SEMKEY  123456L /* key value for semget() */
#define PERMS   0666

static struct sembuf    op_lock[2] = {
        0, 0, 0,        /* wait for sem#0 to become 0 */
        0, 1, SEM_UNDO  /* then increment sem#0 by 1 */
};

static struct sembuf    op_unlock[1] = {
        0, -1, (IPC_NOWAIT | SEM_UNDO)
                        /* decrement sem#0 by 1 (sets it to 0) */
};

int     semid = -1;     /* semaphore id */

my_lock(fd)
int     fd;
{
        if (semid < 0) {
                if ( (semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
                        err_sys("semget error");
        }
        if (semop(semid, &op_lock[0], 2) < 0)
                err_sys("semop lock error");
}

my_unlock(fd)
int     fd;
{
        if (semop(semid, &op_unlock[0], 1) < 0)
```

```
                err_sys("semop unlock error");
}
```

There is still a problem with this implementation, the semaphore is never removed from the system. The `main` function that calls our locking functions should call the `semctl` system call with a command argument of `IPC_RMID` before exiting, if it is the last process using the semaphore. Note that the undo feature that we described earlier only assures that the actual semaphore value gets adjusted as required if the process exits prematurely. It does not remove a semaphore that is not used by any active processes.

Simpler Semaphore Operations

The semaphore facility provided by System V is not simple to understand or use. There are some fundamental problems with its implementation.

- The creation of a semaphore with `semget` is independent of its initialization using `semctl`. This can easily lead to race conditions if we are not careful. As we saw in the file locking section of this chapter, when it takes two system calls to do what is logically a single operation, problems arise.
- Unless a semaphore is explicitly removed, it exists within the system, using system resources, until the system is rebooted. It is a remote condition where a semaphore that is not being used by any processes can be of any real use.

To get around these problems, the following functions provide an easier interface to the semaphore system calls. These functions handle the potential race conditions and also remove semaphores when they are no longer used (most of the time—see the comments in the code). We use these functions in the next section, to synchronize the access to shared memory. The technique used by these functions is to create a semaphore set made up of three members.

- The first member is the actual semaphore value that the user initializes, increments, and decrements. Functions are provided to increment and decrement by one, or by some other integer value.
- The second member of the set is used as a counter of the number of processes currently using the semaphore. This is so that the semaphore can be deleted when no more processes use it. Instead of being an actual counter of the number of processes, this member is initialized to a large value (10,000) and decremented every time a new process opens it, and incremented every time a process closes it. By always keeping the value of this member greater than zero, we don't have to worry about any operation on it blocking.
- The third member of the set is a lock variable for the semaphore. This is required to protect from race conditions when the semaphore is being initialized and closed. This lock variable is a binary semaphore that we handle in the same way

as the lock variable in the example shown previously in this section. Note that the lock variable is *not* needed when the value of the semaphore is being incremented or decremented. The kernel's `semop` system call handles this just fine. The problems arise in the creation, initialization, and closing of the semaphore.

```
/*
 * Provide an simpler and easier to understand interface to the System V
 * semaphore system calls.  There are 7 routines available to the user:
 *
 *      id = sem_create(key, initval);  # create with initial value or open
 *      id = sem_open(key);             # open (must already exist)
 *      sem_wait(id);                   # wait = P = down by 1
 *      sem_signal(id);                 # signal = V = up by 1
 *      sem_op(id, amount);             # wait   if (amount < 0)
 *                                      # signal if (amount > 0)
 *      sem_close(id);                  # close
 *      sem_rm(id);                     # remove (delete)
 *
 * We create and use a 3-member set for the requested semaphore.
 * The first member, [0], is the actual semaphore value, and the second
 * member, [1], is a counter used to know when all processes have finished
 * with the semaphore.  The counter is initialized to a large number,
 * decremented on every create or open and incremented on every close.
 * This way we can use the "adjust" feature provided by System V so that
 * any process that exit's without calling sem_close() is accounted
 * for.  It doesn't help us if the last process does this (as we have
 * no way of getting control to remove the semaphore) but it will
 * work if any process other than the last does an exit (intentional
 * or unintentional).
 * The third member, [2], of the semaphore set is used as a lock variable
 * to avoid any race conditions in the sem_create() and sem_close()
 * functions.
 */

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

#include        <errno.h>
extern int      errno;

#define BIGCOUNT        10000           /* initial value of process counter */

/*
 * Define the semaphore operation arrays for the semop() calls.
 */

static struct sembuf    op_lock[2] = {
        2, 0, 0,        /* wait for [2] (lock) to equal 0 */
        2, 1, SEM_UNDO  /* then increment [2] to 1 - this locks it */
                        /* UNDO to release the lock if processes exits
                           before explicitly unlocking */
};
```

```
static struct sembuf    op_endcreate[2] = {
        1, -1, SEM_UNDO,/* decrement [1] (proc counter) with undo on exit */
                        /* UNDO to adjust proc counter if process exits
                           before explicitly calling sem_close() */
        2, -1, SEM_UNDO /* then decrement [2] (lock) back to 0 */
};

static struct sembuf    op_open[1] = {
        1, -1, SEM_UNDO /* decrement [1] (proc counter) with undo on exit */
};

static struct sembuf    op_close[3] = {
        2, 0, 0,        /* wait for [2] (lock) to equal 0 */
        2, 1, SEM_UNDO, /* then increment [2] to 1 - this locks it */
        1, 1, SEM_UNDO  /* then increment [1] (proc counter) */
};

static struct sembuf    op_unlock[1] = {
        2, -1, SEM_UNDO /* decrement [2] (lock) back to 0 */
};

static struct sembuf    op_op[1] = {
        0, 99, SEM_UNDO /* decrement or increment [0] with undo on exit */
                        /* the 99 is set to the actual amount to add
                           or subtract (positive or negative) */
};

/*****************************************************************************
 * Create a semaphore with a specified initial value.
 * If the semaphore already exists, we don't initialize it (of course).
 * We return the semaphore ID if all OK, else -1.
 */

int
sem_create(key, initval)
key_t   key;
int     initval;        /* used if we create the semaphore */
{
        register int            id, semval;
        union semun {
                int             val;
                struct semid_ds *buf;
                ushort          *array;
        } semctl_arg;

        if (key == IPC_PRIVATE)
                return(-1);     /* not intended for private semaphores */

        else if (key == (key_t) -1)
                return(-1);     /* probably an ftok() error by caller */

again:
        if ( (id = semget(key, 3, 0666 | IPC_CREAT)) < 0)
                return(-1);     /* permission problem or tables full */
```

```
/*
 * When the semaphore is created, we know that the value of all
 * 3 members is 0.
 * Get a lock on the semaphore by waiting for [2] to equal 0,
 * then increment it.
 *
 * There is a race condition here.  There is a possibility that
 * between the semget() above and the semop() below, another
 * process can call our sem_close() function which can remove
 * the semaphore if that process is the last one using it.
 * Therefore, we handle the error condition of an invalid
 * semaphore ID specially below, and if it does happen, we just
 * go back and create it again.
 */

if (semop(id, &op_lock[0], 2) < 0) {
        if (errno == EINVAL)
                goto again;
        err_sys("can't lock");
}

/*
 * Get the value of the process counter.  If it equals 0,
 * then no one has initialized the semaphore yet.
 */

if ( (semval = semctl(id, 1, GETVAL, 0)) < 0)
        err_sys("can't GETVAL");

if (semval == 0) {
        /*
         * We could initialize by doing a SETALL, but that
         * would clear the adjust value that we set when we
         * locked the semaphore above.  Instead, we'll do 2
         * system calls to initialize [0] and [1].
         */

        semctl_arg.val = initval;
        if (semctl(id, 0, SETVAL, semctl_arg) < 0)
                err_sys("can SETVAL[0]");

        semctl_arg.val = BIGCOUNT;
        if (semctl(id, 1, SETVAL, semctl_arg) < 0)
                err_sys("can SETVAL[1]");
}

/*
 * Decrement the process counter and then release the lock.
 */

if (semop(id, &op_endcreate[0], 2) < 0)
        err_sys("can't end create");

return(id);
```

```
}

/*****************************************************************************
 * Open a semaphore that must already exist.
 * This function should be used, instead of sem_create(), if the caller
 * knows that the semaphore must already exist.  For example a client
 * from a client-server pair would use this, if its the server's
 * responsibility to create the semaphore.
 * We return the semaphore ID if all OK, else -1.
 */

int
sem_open(key)
key_t   key;
{
        register int    id;

        if (key == IPC_PRIVATE)
                return(-1);     /* not intended for private semaphores */

        else if (key == (key_t) -1)
                return(-1);     /* probably an ftok() error by caller */

        if ( (id = semget(key, 3, 0)) < 0)
                return(-1);     /* doesn't exist, or tables full */

        /*
         * Decrement the process counter.  We don't need a lock
         * to do this.
         */

        if (semop(id, &op_open[0], 1) < 0)
                err_sys("can't open");

        return(id);
}

/*****************************************************************************
 * Remove a semaphore.
 * This call is intended to be called by a server, for example,
 * when it is being shut down, as we do an IPC_RMID on the semaphore,
 * regardless whether other processes may be using it or not.
 * Most other processes should use sem_close() below.
 */

sem_rm(id)
int     id;
{
        if (semctl(id, 0, IPC_RMID, 0) < 0)
                err_sys("can't IPC_RMID");
}

/*****************************************************************************
 * Close a semaphore.
```

```
 * Unlike the remove function above, this function is for a process
 * to call before it exits, when it is done with the semaphore.
 * We "decrement" the counter of processes using the semaphore, and
 * if this was the last one, we can remove the semaphore.
 */

sem_close(id)
int     id;
{
        register int    semval;

        /*
         * The following semop() first gets a lock on the semaphore,
         * then increments [1] - the process counter.
         */

        if (semop(id, &op_close[0], 3) < 0)
                err_sys("can't semop");

        /*
         * Now that we have a lock, read the value of the process
         * counter to see if this is the last reference to the
         * semaphore.
         * There is a race condition here - see the comments in
         * sem_create().
         */

        if ( (semval = semctl(id, 1, GETVAL, 0)) < 0)
                err_sys("can't GETVAL");

        if (semval > BIGCOUNT)
                err_dump("sem[1] > BIGCOUNT");
        else if (semval == BIGCOUNT)
                sem_rm(id);
        else
                if (semop(id, &op_unlock[0], 1) < 0)
                        err_sys("can't unlock");        /* unlock */
}

/*****************************************************************************
 * Wait until a semaphore's value is greater than 0, then decrement
 * it by 1 and return.
 * Dijkstra's P operation.  Tanenbaum's DOWN operation.
 */

sem_wait(id)
int     id;
{
        sem_op(id, -1);
}

/*****************************************************************************
 * Increment a semaphore by 1.
 * Dijkstra's V operation.  Tanenbaum's UP operation.
```

```
 */

sem_signal(id)
int     id;
{
        sem_op(id, 1);
}

/*****************************************************************************
 * General semaphore operation.  Increment or decrement by a user-specified
 * amount (positive or negative; amount can't be zero).
 */

sem_op(id, value)
int     id;
int     value;
{
        if ( (op_op[0].sem_op = value) == 0)
                err_sys("can't have value == 0");

        if (semop(id, &op_op[0], 1) < 0)
                err_sys("sem_op error");
}
```

File Locking with Semaphores (Again)

We can use these simpler semaphore functions in our locking example. Here we have changed the `main` function from the version that was used in all the previous locking examples, so that here we call `sem_create` and `sem_close` outside the main loop.

```
/*
 * Locking example using the simpler semaphore operations.
 */

#include        <sys/types.h>

#define SEQFILE         "seqno"
#define SEMKEY          ((key_t) 23456L)
#define MAXBUFF         100

main()
{
        int     fd, i, n, pid, seqno, semid;
        char    buff[MAXBUFF];

        pid = getpid();
        if ( (fd = open(SEQFILE, 2)) < 0)
                err_sys("can't open %s", SEQFILE);
        if ( (semid = sem_create(SEMKEY, 1)) < 0)
                err_sys("can't open semaphore");

        for (i = 0; i < 20; i++) {
                sem_wait(semid);                /* get the lock */
```

```
                lseek(fd, 0L, 0);                /* rewind before read */
                if ( (n = read(fd, buff, MAXBUFF)) <= 0)
                        err_sys("read error");
                buff[n] = '\0';          /* null terminate for sscanf */

                if ( (n = sscanf(buff, "%d\n", &seqno)) != 1)
                        err_sys("sscanf error");
                printf("pid = %d, seq# = %d\n", pid, seqno);

                seqno++;

                sprintf(buff, "%03d\n", seqno);
                n = strlen(buff);
                lseek(fd, 0L, 0);                /* rewind before write */
                if (write(fd, buff, n) != n)
                        err_sys("write error");

                sem_signal(semid);               /* release the lock */
        }
        sem_close(semid);
}
```

Semaphore Limits

As with message queues, there are certain system limits with semaphores, some of which can be changed by reconfiguring the kernel. These are shown in Figure 3.23.

Parameter name	AT&T VAX Sys V Rel 2	DEC VAX Ultrix	AT&T Unix PC	Xenix 286	Xenix 386	Description
semmni	10	10	10	10	10	max # of unique semaphore sets, systemwide
semmns	60	60	60	40	40	max # of semaphores, systemwide
semmsl	25	25	25	10	10	max # of semaphores per semaphore set
semopn	10	10	10	5	5	max # of operations per semop call
semmnu	30	30	30	20	20	max # of undo structures, systemwide
semume	10	10	10	5	5	max # of undo entries per undo structure
semvmx	32,767	32,767	32,767	32,766	32,766	max value of any semaphore
semaem	16,384	16,384	16,384	16,384	16,384	max value of any semaphore's adjust-on-exit value

Figure 3.23 Typical limits for semaphores.

3.11 Shared Memory

Consider the normal steps involved in the client–server file copying program that we have been using for the example throughout this chapter.

- The server reads from the input file. Typically the data is read by the kernel into one of its internal block buffers and copied from there to the server's buffer (the second argument to the `read` system call). It should be noted that most Unix systems detect sequential reading, as is done by the server, and try to keep one block ahead of the `read` requests. This helps reduce the clock time required to copy a file, but the data for every `read` is still copied by the kernel from its block buffer to the caller's buffer.
- The server writes this data in a message, using one of the techniques described in this chapter—a pipe, FIFO or message queue. Any of these three forms of IPC require the data to be copied from the user's buffer into the kernel.
- The client reads the data from the IPC channel, again requiring the data be copied from the kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer, the second argument to the `write` system call, to the output file. This might involve just copying the data into a kernel buffer and returning, with the kernel doing the actual write operation to the device at some later time.

A total of four copies of the data are required. Additionally, these four copies are done between the kernel and a user process—an *intercontext* copy. While most Unix implementations try to speed up these copies as much as possible, they can still be expensive. Figure 3.24 depicts this data movement between the two processes.

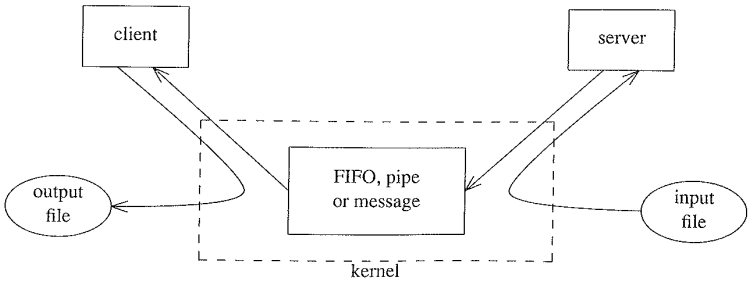


Figure 3.24 Typical movement of data between client and server.

The problem with these forms of IPC—pipes, FIFOs, and message queues—is that for two processes to exchange information, the information has to go through the kernel. Shared memory provides a way around this by letting two or more processes share a memory segment. There is, of course, a problem involved in multiple processes sharing

a piece of memory: the processes have to coordinate the use of the memory among themselves. (Sharing a common piece of memory is similar to sharing a disk file, such as the sequence number file used in all the file locking examples.) If one process is reading into some shared memory, for example, other processes must wait for the read to finish before processing the data. Fortunately this is an easy problem to solve, using semaphores for the synchronization.

The steps for the client–server example now become

- The server gets access to a shared memory segment using a semaphore.
- The server reads from the input file into the shared memory segment. The address to read into, the second argument to the `read` system call, points into shared memory.
- When the read is complete the server notifies the client, again using a semaphore.
- The client writes the data from the shared memory segment to the output file.

This is depicted in Figure 3.25.

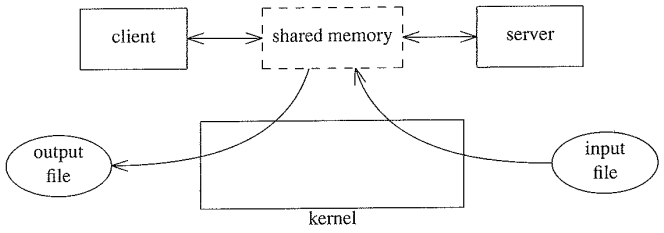


Figure 3.25 Movement of data between client and server using shared memory.

In this figure the data is only copied twice—from the input file into shared memory and from shared memory to the output file. Both of these copies probably involve the kernel's block buffers, as mentioned earlier.

For every shared memory segment, the kernel maintains the following structure of information:

```
#include  <sys/types.h>
#include  <sys/ipc.h>          /* defines the ipc_perm structure */

struct shmid_ds {
  struct ipc_perm shm_perm;    /* operation permission struct */
  int             shm_segsz;   /* segment size */
  struct XXX      shm_YYY;     /* implementation dependent info */
  ushort          shm_lpid;    /* pid of last operation */
  ushort          shm_cpid;    /* creator pid */
  ushort          shm_nattch;  /* current # attached */
  ushort          shm_cnattch; /* in-core # attached */
```

```
    time_t          shm_atime;    /* last attach time */
    time_t          shm_dtime;    /* last detach time */
    time_t          shm_ctime;    /* last change time */
};
```

The `ipc_perm` structure was described in Section 3.8 and contains the access permissions for the shared memory segment. Unlike message queues and semaphores, we can't describe the actual data structures used by the kernel to point to the shared memory segment, since it is hardware and implementation dependent. The notation `XXX` and `YYY` is used in the structure above to note this fact.

A shared memory segment is created, or an existing one is accessed with the `shmget` system call.

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

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

The value returned by `shmget` is the shared memory identifier, *shmid*, or −1 if an error occurs. The *size* argument specifies the size of the segment, in bytes. The *shmflag* argument is a combination of the constants shown in Figure 3.26.

Numeric	Symbolic	Description
0400	SHM_R	Read by owner
0200	SHM_W	Write by owner
0040	SHM_R >> 3	Read by group
0020	SHM_W >> 3	Write by group
0004	SHM_R >> 6	Read by world
0002	SHM_W >> 6	Write by world
	IPC_CREAT	(See Section 3.8)
	IPC_EXCL	(See Section 3.8)

Figure 3.26 *shmflag* values for `shmget` system call.

Note that the `shmget` call creates or opens a shared memory segment, but does not provide access to the segment for the calling process. We must attach the shared memory segment by calling the `shmat` system call.

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

char *shmat(int shmid, char *shmaddr, int shmflag);
```

This system call returns the starting address of the shared memory segment. The rules for determining this address are as follows:

- If the *shmaddr* argument is zero, the system selects the address for the caller.

- If the *shmaddr* argument is nonzero, the returned address depends whether the caller specifies the SHM_RND value for the *shmflag* argument:
  - If the SHM_RND value is not specified, the shared memory segment is attached at the address specified by the *shmaddr* argument.
  - If the SHM_RND value is specified, the shared memory segment is attached at the address specified by the *shmaddr* argument, rounded down by the constant SHMLBA (LBA stands for "lower boundary address").

For all practical purposes, the only portable calls to shmat specify the *shmaddr* of zero, and let the kernel select the address.

By default, the shared memory segment is attached for both reading and writing by the calling process. The SHM_RDONLY value can also be specified for the *shmflag* argument, specifying "read-only" access.

When a process is finished with a shared memory segment, it detaches the segment by calling the shmdt system call.

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

int shmdt(char *shmaddr);
```

This call does not delete the shared memory segment. To remove a shared memory segment, the shmctl system call is used.

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

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

A *cmd* of IPC_RMID removes a shared memory segment from the system.

We can now redo our client–server example using shared memory and the simpler semaphore operations developed in the previous section. This code uses a single shared memory segment and two binary semaphores—one named clisem and one named servsem. The server waits until it has control of servsem and then reads from the file directly into the shared memory segment. When the client gets control of its semaphore, it writes from the shared memory to the standard output. The server creates the shared memory segment and the two semaphores, with clisem initialized to one, to allow the client to start things off by reading a filename into the shared memory segment. The two semaphores then alternate between zero and one (when one semaphore value is zero the other value is one, and vice versa) as control of the shared memory toggles between the two processes.

Our header file `shm.h` is

```
#include        "mesg.h"

#define NBUFF   4       /* number of buffers in shared memory */
                        /* (for multiple buffer version) */

#define SHMKEY  ((key_t) 7890) /* base value for shmem key */

#define SEMKEY1 ((key_t) 7891) /* client semaphore key */
#define SEMKEY2 ((key_t) 7892) /* server semaphore key */

#define PERMS   0666
```

The file `mesg.h` is the same file shown earlier that defined the `Mesg` structure. (We'll discuss the `NBUFF` variable later in this section.)

Here is the server program.

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

#include        "shm.h"

int     shmid, clisem, servsem; /* shared memory and semaphore IDs */
Mesg    *mesgptr;               /* ptr to message structure, which is
                                   in the shared memory segment */

main()
{
        /*
         * Create the shared memory segment, if required,
         * then attach it.
         */

        if ( (shmid = shmget(SHMKEY, sizeof(Mesg), PERMS | IPC_CREAT)) < 0)
                err_sys("server: can't get shared memory");
        if ( (mesgptr = (Mesg *) shmat(shmid, (char *) 0, 0)) == (Mesg *) -1)
                err_sys("server: can't attach shared memory");

        /*
         * Create two semaphores.  The client semaphore starts out at 1
         * since the client process starts things going.
         */

        if ( (clisem = sem_create(SEMKEY1, 1)) < 0)
                err_sys("server: can't create client semaphore");
        if ( (servsem = sem_create(SEMKEY2, 0)) < 0)
                err_sys("server: can't create server semaphore");

        server();
```

```
        /*
         * Detach the shared memory segment and close the semaphores.
         * The client is the last one to use the shared memory, so
         * it'll remove it when it's done.
         */

        if (shmdt(mesgptr) < 0)
                err_sys("server: can't detach shared memory");

        sem_close(clisem);
        sem_close(servsem);

        exit(0);
}

server()
{
        int     n, filefd;
        char    errmesg[256], *sys_err_str();

        /*
         * Wait for the client to write the filename into shared memory.
         */

        sem_wait(servsem);      /* we'll wait here for client to start things */

        mesgptr->mesg_data[mesgptr->mesg_len] = '\0';
                                        /* null terminate filename */

        if ( (filefd = open(mesgptr->mesg_data, 0)) < 0) {
                /*
                 * Error.  Format an error message and send it back
                 * to the client.
                 */

                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(mesgptr->mesg_data, errmesg);
                mesgptr->mesg_len = strlen(mesgptr->mesg_data);
                sem_signal(clisem);             /* send to client */
                sem_wait(servsem);              /* wait for client to process */

        } else {
                /*
                 * Read the data from the file right into shared memory.
                 * The -1 in the number-of-bytes-to-read is because some
                 * Unices have a bug if you try and read into the final byte
                 * of a shared memory segment.
                 */

                while ( (n = read(filefd, mesgptr->mesg_data,
                                                MAXMESGDATA-1)) > 0) {
                        mesgptr->mesg_len = n;
                        sem_signal(clisem);     /* send to client */
                        sem_wait(servsem);      /* wait for client to process */
```

```
                }
                close(filefd);
                if (n < 0)
                        err_sys("server: read error");
        }

        /*
         * Send a message with a length of 0 to signify the end.
         */

        mesgptr->mesg_len = 0;
        sem_signal(clisem);
}
```

Note that unlike the earlier versions of this program, the shared memory version defines a pointer to a `Mesg` structure (the `mesgptr` variable), not a structure itself. The actual structure is allocated by the system in the shared memory segment.

Here is the client program.

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

#include        "shm.h"

int     shmid, clisem, servsem; /* shared memory and semaphore IDs */
Mesg    *mesgptr;               /* ptr to message structure, which is
                                   in the shared memory segment */

main()
{
        /*
         * Get the shared memory segment and attach it.
         * The server must have already created it.
         */

        if ( (shmid = shmget(SHMKEY, sizeof(Mesg), 0)) < 0)
                err_sys("client: can't get shared memory segment");
        if ( (mesgptr = (Mesg *) shmat(shmid, (char *) 0, 0)) == (Mesg *) -1)
                err_sys("client: can't attach shared memory segment");

        /*
         * Open the two semaphores.  The server must have
         * created them already.
         */

        if ( (clisem = sem_open(SEMKEY1)) < 0)
                err_sys("client: can't open client semaphore");
        if ( (servsem = sem_open(SEMKEY2)) < 0)
                err_sys("client: can't open server semaphore");

        client();
```

```
        /*
         * Detach and remove the shared memory segment and
         * close the semaphores.
         */

        if (shmdt(mesgptr) < 0)
                err_sys("client: can't detach shared memory");
        if (shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0) < 0)
                err_sys("client: can't remove shared memory");

        sem_close(clisem);      /* will remove the semaphore */
        sem_close(servsem);     /* will remove the semaphore */

        exit(0);
}

client()
{
        int     n;

        /*
         * Read the filename from standard input, write it to shared memory.
         */

        sem_wait(clisem);               /* get control of shared memory */
        if (fgets(mesgptr->mesg_data, MAXMESGDATA, stdin) == NULL)
                err_sys("filename read error");

        n = strlen(mesgptr->mesg_data);
        if (mesgptr->mesg_data[n-1] == '\n')
                n--;                    /* ignore newline from fgets() */
        mesgptr->mesg_len = n;
        sem_signal(servsem);            /* wake up server */

        /*
         * Wait for the server to place something in shared memory.
         */

        sem_wait(clisem);               /* wait for server to process */
        while( (n = mesgptr->mesg_len) > 0) {
                if (write(1, mesgptr->mesg_data, n) != n)
                        err_sys("data write error");
                sem_signal(servsem);    /* wake up server */
                sem_wait(clisem);       /* wait for server to process */
        }

        if (n < 0)
                err_sys("data read error");
}
```

The two processes wait for access to the shared memory by waiting for a semaphore's value to become greater than zero. This is the most efficient way to wait for the resource, since it is the kernel that does all semaphore operations and the kernel puts a process to sleep when it has to wait for a semaphore. While the process is asleep, the kernel can allocate the CPU to other processes that are ready to run. Some books show what is called *busy–waiting* when a process needs to wait for an event. The process doesn't go to sleep, it just keeps trying to obtain the resource, over and over again, until it gets it. For example, we could add another variable to the `Mesg` structure and use this as a flag to indicate which process has control of the shared memory segment. If the flag were zero, perhaps, the client is using it, while if it were one the server could use it. Instead of executing

```
sem_signal(clisem);
sem_wait(servesm);
```

in the server loop, it could execute

```
mesgptr->mesg_flag = 0;          /* signal client */
while (mesgptr->mesg_flag == 0)
        ;                        /* wait for client to process */
```

if `mesg_flag` were the busy–wait flag in the `Mesg` structure. The corresponding code in the client loop becomes

```
mesgptr->mesg_flag = 1;          /* signal server */
while (mesgptr->mesg_flag == 1)
        ;                        /* wait for server to process */
```

While this approach works, semaphores should always be used for this type of synchronization. A busy–wait loop is a waste of CPU resources.

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 normally replaced with calls to the standard I/O functions `fgets` and `fputs`.

If we look at a process that reads data and then writes it out we have the time line shown in Figure 3.27. We have labeled the time line with numbers on the left, designating some arbitrary units of time.

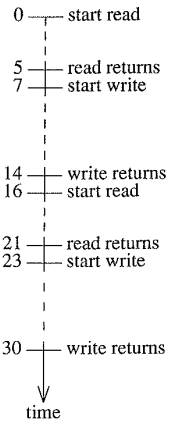


Figure 3.27 Time line for single-process read–write example.

Time increases downward. We have assumed 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.

If we divide the processing into two processes, as we've done with the client–server example in this chapter, we have the time line shown in Figure 3.28.

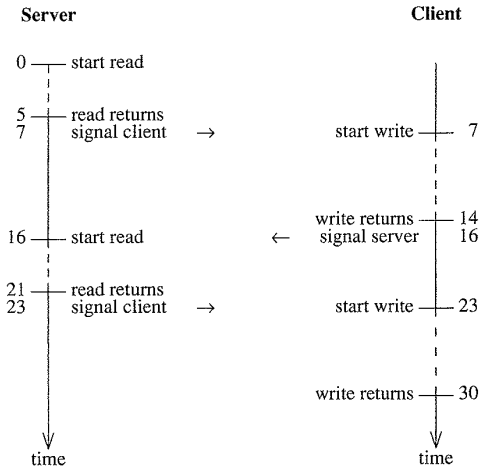


Figure 3.28 Time line for two processes doing read–write example.

This time line assumes that there is a single shared memory segment for the client and server processes. We assume here that the time to process the data in the buffer, along with the signaling of the other process takes 2 units of time. The important thing to note is that dividing the reading and writing into two processes does not affect the total amount of time required to do the operation. We have not gained any speed advantage.

There are many fine points that we are ignoring in these time lines. For example, Unix systems 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 processes, and the effects of the operating system's scheduling algorithms. Our goal is to describe the concept of double buffering and see how it affects the clock time for a typical process.

Unix I/O is termed *synchronous*. A `read` or `write` system call does not return to the calling process until the operation is complete and the requested data has been copied by the kernel, either into the buffer for a `read`, or from the buffer for a `write`. The data might only be copied to or from a buffer in the kernel. There is no guarantee for typical disk I/O that the data is really read from the disk or written to the disk. Indeed, the read ahead mechanism mentioned above often causes the kernel to already have the data in one of its buffers. A similar operation on `writes`, termed *write behind*, allows the kernel to write the data to the disk at some later time.

Some operating systems provide *asynchronous I/O*. This allows a process to execute a system call to start an I/O operation and have the system call return immediately after the operation is started or queued. Another system call is required to wait for the operation to complete (or return immediately if the operation is already finished). The advantage of asynchronous I/O is that a process can overlap its execution with its I/O, or it can overlap I/O between different devices. While this typically doesn't have a great effect on a process that is reading from a disk file and writing to another disk file, asynchronous I/O can provide significant performance improvements for other types of I/O driven programs—a program that dumps a disk to a magnetic tape, a program that displays an image on an image display. Even though Unix does not provide asynchronous I/O we can gain the same performance improvements by splitting our processing into multiple processes and have them share more than one buffer.

Now consider the example from above, but with the client and server using two buffers. This is shown in Figure 3.29. The server first reads into buffer 1, then signals the client that buffer 1 is ready for processing. The server then starts reading into buffer 2, while the client is writing buffer 1. Note that we can't 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 that is the time difference between the write (7) and the read (5). The total clock time, however, will be almost halved for the double buffer case, compared to the single buffered case, for our hypothetical example.

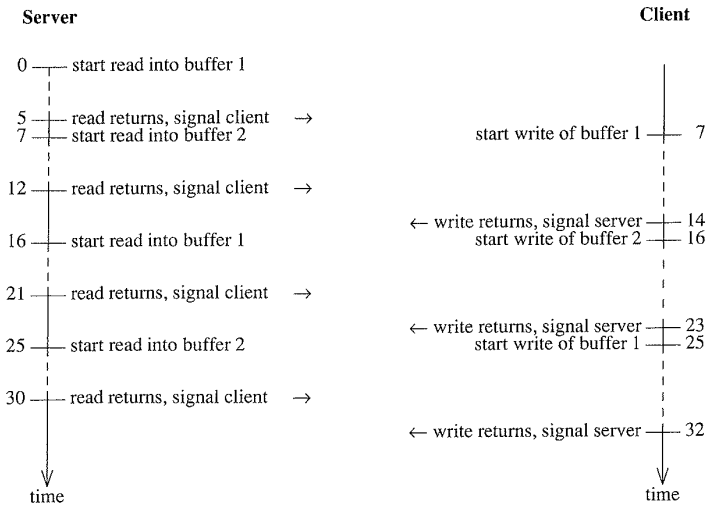


Figure 3.29 Time line for two processes doing read–write example, using two buffers.

We now show the client–server example using multiple buffers. The code was written for NBUFF buffers, not just for two buffers. Typically it is wise, when converting a single resource program to a 2-resource program, to develop an N-resource solution instead. The extra effort required to develop a solution to the N-resource case is often worth it, as many times generalizing a problem makes the solution "cleaner." By this we mean the solution is often more general and avoids magic numbers and special case handling.

The server program is

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

#include        "shm.h"

int     clisem, servsem;        /* semaphore IDs */

int     shmid[NBUFF];           /* shared memory IDs */
Mesg    *mesgptr[NBUFF];        /* ptr to message structures, which are
                                   in the shared memory segment */

main()
{
        register int    i;

        /*
         * Get the shared memory segments and attach them.
```

```
         */

        for (i = 0; i < NBUFF; i++) {
                if ( (shmid[i] = shmget(SHMKEY + i, sizeof(Mesg),
                                                PERMS | IPC_CREAT)) < 0)
                        err_sys("server: can't get shared memory %d", i);
                if ( (mesgptr[i] = (Mesg *) shmat(shmid[i], (char *) 0, 0))
                                                        == (Mesg *) -1)
                        err_sys("server: can't attach shared memory %d", i);
        }

        /*
         * Create the two semaphores.
         */

        if ( (clisem = sem_create(SEMKEY1, 1)) < 0)
                err_sys("server: can't create client semaphore");
        if ( (servsem = sem_create(SEMKEY2, 0)) < 0)
                err_sys("server: can't create server semaphore");

        server();

        /*
         * Detach the shared memory segments and close the semaphores.
         * We let the client remove the shared memory segments,
         * since it'll be the last one to use them.
         */

        for (i = 0; i < NBUFF; i++) {
                if (shmdt(mesgptr[i]) < 0)
                        err_sys("server: can't detach shared memory %d", i);
        }

        sem_close(clisem);
        sem_close(servsem);

        exit(0);
}

server()
{
        register int    i, n, filefd;
        char            errmesg[256], *sys_err_str();

        /*
         * Wait for the client to write the filename into shared memory,
         * then try to open the file.
         */

        sem_wait(servsem);
        mesgptr[0]->mesg_data[mesgptr[0]->mesg_len] = '\0';
                                        /* null terminate filename */
        if ( (filefd = open(mesgptr[0]->mesg_data, 0)) < 0) {
                /*
```

```
                 * Error.  Format an error message and send it back
                 * to the client.
                 */

                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(mesgptr[0]->mesg_data, errmesg);
                mesgptr[0]->mesg_len = strlen(mesgptr[0]->mesg_data);
                sem_signal(clisem);     /* wake up client */

                sem_wait(servsem);      /* wait for client to process */
                mesgptr[1]->mesg_len = 0;
                sem_signal(clisem);     /* wake up client */

        } else {
                /*
                 * Initialize the server semaphore to the number
                 * of buffers.  We know its value is 0 now, since
                 * it was initialized to 0, and the client has done a
                 * sem_signal(), followed by our sem_wait() above.
                 * What we do is increment the semaphore value
                 * once for every buffer (i.e., the number of resources
                 * that we have).
                 */

                for (i = 0; i < NBUFF; i++)
                        sem_signal(servsem);

                /*
                 * Read the data from the file right into shared memory.
                 * The -1 in the number-of-bytes-to-read is because some
                 * Unices have a bug if you try and read into the final byte
                 * of a shared memory segment.
                 */

                for ( ; ; ) {
                        for (i = 0; i < NBUFF; i++) {
                                sem_wait(servsem);
                                n = read(filefd, mesgptr[i]->mesg_data,
                                                        MAXMESGDATA-1);
                                if (n < 0)
                                        err_sys("server: read error");
                                mesgptr[i]->mesg_len = n;
                                sem_signal(clisem);
                                if (n == 0)
                                        goto alldone;
                        }
                }
alldone:
                /* we've already written the 0-length final buffer */
                close(filefd);
        }
}
```

The client program is

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

#include        "shm.h"

int     clisem, servsem;        /* semaphore IDs */

int     shmid[NBUFF];           /* shared memory IDs */
Mesg    *mesgptr[NBUFF];        /* ptr to message structures, which are
                                   in the shared memory segment */

main()
{
        register int    i;

        /*
         * Get the shared memory segments and attach them.
         * We don't specify IPC_CREAT, assuming the server creates them.
         */

        for (i = 0; i < NBUFF; i++) {
                if ( (shmid[i] = shmget(SHMKEY + i, sizeof(Mesg), 0)) < 0)
                        err_sys("client: can't get shared memory %d", i);
                if ( (mesgptr[i] = (Mesg *) shmat(shmid[i], (char *) 0, 0))
                                                        == (Mesg *) -1)
                        err_sys("client: can't attach shared memory %d", i);
        }

        /*
         * Open the two semaphores.
         */

        if ( (clisem = sem_open(SEMKEY1)) < 0)
                err_sys("client: can't open client semaphore");
        if ( (servsem = sem_open(SEMKEY2)) < 0)
                err_sys("client: can't open server semaphore");

        client();

        /*
         * Detach and remove the shared memory segments and
         * close the semaphores.
         */

        for (i = 0; i < NBUFF; i++) {
                if (shmdt(mesgptr[i]) < 0)
                        err_sys("client: can't detach shared memory %d", i);
                if (shmctl(shmid[i], IPC_RMID, (struct shmid_ds *) 0) < 0)
                        err_sys("client: can't remove shared memory %d", i);
        }
```

```
        sem_close(clisem);
        sem_close(servsem);

        exit(0);
}

client()
{
        int     i, n;

        /*
         * Read the filename from standard input, write it to shared memory.
         */

        sem_wait(clisem);               /* wait for server to initialize */
        if (fgets(mesgptr[0]->mesg_data, MAXMESGDATA, stdin) == NULL)
                err_sys("filename read error");

        n = strlen(mesgptr[0]->mesg_data);
        if (mesgptr[0]->mesg_data[n-1] == '\n')
                n--;                    /* ignore newline from fgets() */
        mesgptr[0]->mesg_len = n;
        sem_signal(servsem);            /* wake up server */

        for ( ; ; ) {
                for (i = 0; i < NBUFF; i++) {
                        sem_wait(clisem);
                        if ( (n = mesgptr[i]->mesg_len) <= 0)
                                goto alldone;
                        if (write(1, mesgptr[i]->mesg_data, n) != n)
                                err_sys("data write error");
                        sem_signal(servsem);
                }
        }

alldone:
        if (n < 0)
                err_sys("data read error");
}
```

One change that could be made is to allocate a single shared memory segment whose size is `NBUFF * sizeof(Mesg)`, instead of `NBUFF` segments.

Shared Memory Limits

As with message queues and semaphores, there are certain system limits on shared memory. Typical values are shown in Figure 3.30. Some of these can be changed by a system administrator, by configuring a new kernel. Compared to the other forms of IPC described in this chapter (pipes, FIFOs, message queues, and semaphores), there is a

greater potential for writing nonportable code when using shared memory, so portability must be kept in mind.

Parameter name	AT&T VAX Sys V Rel 2	DEC VAX Ultrix	AT&T Unix PC	Xenix 386	Description
shmmax	131,072	131,072	65,536	4,194,304	maximum size in bytes of a shared memory segment
shmmin	1	1	1	1	minimum size in bytes of a shared memory segment
shmmni	100	100	100	25	max # of shared memory segments, systemwide
shmseg	6	6	6	6	max # of shared memory segments attached per process
shmbrk	8,192	32,768	8,192	0	(see text)

Figure 3.30 Typical limits for shared memory.

The `shmbrk` value needs some additional explanation. Referring to Figure 2.2, shared memory is normally allocated by the system above the heap. If shared memory starts too close to the heap, we won't be able to obtain additional memory dynamically, using either the `malloc` function in the standard C library or the `brk` or `sbrk` system calls. The `shmbrk` value specifies the number of bytes between the end of the heap (when the first shared memory segment is allocated) and the start of the first shared memory segment. If a process wants to increase the distance between the top of the heap and the first shared memory segment (to allow for future calls to `malloc` after the shared memory has been created), it can call `malloc` before creating the first shared memory segment.

3.12 Sockets and TLI

Sockets are a form of IPC provided by 4.3BSD that provide communication between processes on a single system and between processes on different systems. One type of socket—the Unix domain socket—is used for IPC between processes on a single system. Pipes are implemented in 4.3BSD using a Unix domain socket.

Describing the system calls that pertain to sockets requires additional understanding of the communication protocols that are available to the socket, so we defer our description of sockets until Chapter 6.

TLI stands for *transport layer interface* and is a form of IPC provided with System V Release 3.0. As a form of IPC, TLI is similar to Berkeley sockets in that it provides communication between processes on the same system or on different systems. We defer our discussion of TLI until Chapter 7.

3.13 Summary

IPC has traditionally been a messy area in Unix. Various solutions have been implemented, none of which are perfect. We first covered record locking and file locking, since the sharing of a single file between multiple processes is a common occurrence. We then examined numerous IPC techniques—pipes, FIFOs, message queues, semaphores, and shared memory—and will cover two more in Chapters 6 and 7. You must evaluate your needs for both IPC and portability, and choose the best technique available.

In Chapter 17 we'll examine some typical performance values for the various IPC techniques.

We'll encounter IPC in many of the examples later in the text. For example, IPC, record locking, and network programming will all come together in our discussion of the 4.3BSD line printer spooler in Chapter 13.

Exercises

3.1 Implement one of the filesystem based locking techniques from Section 3.2, say the technique that uses the `link` system call, and compare the amount of time required for this, versus actual file locking.

3.2 Under System V Release 3.2 or later, write a program that uses record locking. Use the program to verify that advisory locking is only advisory. Then modify the file's access bits to enable mandatory record locking and use the program to verify that the locking is now mandatory.

3.3 Why is a signal generated for the writer of a pipe or FIFO when the other end disappears, and not for the reader of a pipe or FIFO when its writer disappears?

3.4 Use the multiple buffer technique in Section 3.11 to write a program that copies between two different devices—a disk file and a tape drive, for example. First measure the operation when only a single buffer is used, then measure the operation for different values of `NBUFF`, say 2, 3, and 4. Compare the results. Where do you think most of the overhead is when multiple buffers are used?

3.5 What happens with the client–server example using message queues, if the file to be copied is a binary file? What happens to the version that uses the `popen` function if the file is a binary file?

3.6 For the multiple buffer, shared memory example in Section 3.11, draw a time line showing which process (client or server) is using which buffer, along with the values of the two semaphores. Assume `NBUFF` is 3.

```
                        /* assure it's null terminated */

                /*
                 * We got the "localhostname.localdomainmame" string from the
                 * kernel.  Save the pointer to the "locdomainmame" part,
                 * then assure it's lower case.
                 */

                if ( (locdomptr = index(locdomname, '.')) == NULL) {
                        locdomerror = 1;        /* humm, no period */
                        return(0);              /* return not valid */
                }

                for (ptr = ++locdomptr; *ptr; ptr++)
                        if (isupper(*ptr))
                                *ptr = tolower(*ptr);
        }

        /*
         * We know the host names are identical, and the entry in the file was
         * "hostname.".  So now we compare the client's domain with the local
         * domain, and if equal, all is OK.
         */

        return(strcmp(locdomptr, clihostname + hostlen + 1) == 0);
                                        /* returns 1 if equal, 0 otherwise */
}
```

In the `_checkhost` function we call the `gethostname` system call. This system call returns a pointer to the name of the current host. This name is usually set when the system is booted, using the `hostname` program under 4.3BSD. It may be either a simple name such as `orange`, or a domain-qualified name such as `orange.foo.edu`. Refer to Chapter 18 of Comer [1988] for additional information on the domain name system.

9.3 Kerberos

Project Athena was started at M.I.T. in 1983 and its goal was to create an educational computing environment built around high-performance graphic workstations, high-speed networking, and servers of various types. The operating system for the project is based on 4.3BSD. A discussion of the project appears in seven papers published in the Conference Proceedings of the 1988 USENIX Winter Conference. The first of these papers, Treese [1988], provides an overview.

The authentication system used by Athena is called Kerberos.† It is a trusted third-party authentication service. It is worth an overview here since it is based on 4.3BSD and it provides an interesting comparison to the techniques discussed in the previous

† In Greek mythology, Kerberos is the name of the 3-headed watch dog that guards the entrance to Hades (the underground abode of the dead).

section of this chapter. This overview is based on the paper by Steiner, Neuman, and Schiller [1988].

Kerberos assumes that the workstations are not trustworthy and requires you, the client, to identify yourself every time a service is requested. For example, every time you want to send a file to a print server, you have to identify yourself to that server. Any system could implement a strict security mechanism such as this, by requiring you to enter a password of some form every time you request a network service. The techniques used by Kerberos, however, are unobtrusive. Kerberos follows these guidelines:

- You only have to identify yourself once, at the beginning of a workstation session. This involves entering your login name and password, and is automatically built in to the standard Unix `login` program.
- Passwords are never sent across the network in cleartext. They are always encrypted. Additionally, passwords are never stored on a workstation or server in cleartext.
- Every user has a password and every service has a password.
- The *only* entity that knows all passwords is the authentication server. This authentication server operates under considerable physical security.

Figure 9.2 shows all the pieces that are involved.

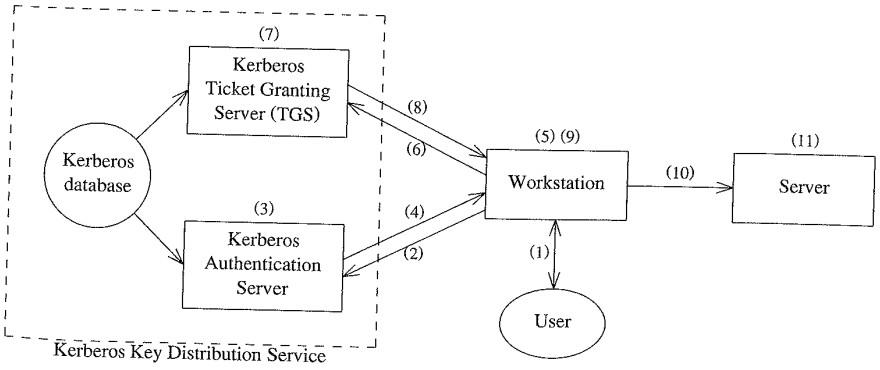


Figure 9.2 Kerberos authentication scheme.

The numbered steps, 1 to 11, in Figure 9.2 do the following:

1. You, the user, walk up to a workstation and login. You enter your login name in response to the normal Unix `login:` prompt.
2. As part of the login sequence, and *before* you're prompted for your password, a message is sent across the network to the Kerberos authentication server. This message

contains your login name along with the name of one particular Kerberos server: the Kerberos ticket-granting server (TGS).

message = { *login-name, TGS-name* }

Since this message only contains two names, it need not be encrypted. Names are not considered secret since everyone has to know other names to communicate. You need to know other login names to send mail. You have to know the names of the different network services, to use a particular service. We'll see as we progress through the steps that follow that names are the *only* entities exchanged in cleartext between the workstation and Kerberos. Everything else is encrypted.

3. The authentication server looks up your login name and the service name (TGS) in the Kerberos database, and obtains an encryption key for each. The encryption keys used by Kerberos are one-way encrypted passwords, similar to what is stored in the password field entry of the normal Unix password file.

4. The authentication server forms a response to send back to the login program on the workstation. This response contains a *ticket* that grants you access to the requested server (the TGS). The concept of a ticket and what makes up a ticket is central to the Kerberos system. Tickets are always sent across the network encrypted—they are never sent in cleartext. The ticket is a 4-tuple containing

   ticket = { *login-name, TGS-name, WS-net-address, TGS-session-key* }

   (At the end of this section we'll describe how a ticket also contains a time stamp and an expiration date, in addition to the fields shown above.) The *WS-net-address* is the Internet address of the workstation. The *TGS-session-key* is a random number generated by the authentication server. The authentication server then encrypts this ticket using the encryption key of the TGS server (which it obtained in step 3). This produces what is called a *sealed ticket.* A message is then formed consisting of the sealed ticket and the *TGS-session-key*

   message = { *TGS-session-key, sealed-ticket* }

   This message is then encrypted using your encryption key—your encrypted password, which is contained in the Kerberos database. Note that the TGS session key appears twice in the message: once within the ticket and again as part of the message itself.

5. The `login` program receives the encrypted message and only then prompts you for your password. This cleartext password that you enter is first processed through the standard Unix one-way encryption algorithm and the result, what we've called the user's encryption key, is used to decrypt the message (TGS session key and sealed ticket) that was received. Your cleartext password is then erased from memory. All that the workstation has now is a sealed ticket that has been encrypted using the TGS encryption key along with a TGS session key. This sealed ticket is incomprehensible to the workstation since it was encrypted using the TGS encryption key, that is known only to the TGS and Kerberos. The TGS session key is comprehensible to the

workstation, but it is just some random bit string that the authentication server created.

At this point the workstation software saves a copy of the sealed ticket and the TGS session key. Every workstation contains similar information: a sealed ticket for the ticket-granting service along with the corresponding random TGS session key. Even though all the sealed tickets are encrypted using the same encryption key (that of the ticket-granting server), since each ticket contains the name of the user the ticket was granted to, the workstation address, along with a different random TGS session key, each is different.

Now consider the scenario when you request some network service. You might want to read your mail from the mail server, print a file on a print server, or access a file from a file server. We'll call the requested server the *end-server*, to differentiate it from the Kerberos ticket-granting server and the Kerberos authentication server described above. *Every* service request of this form requires first obtaining a ticket for the particular service. To obtain the ticket, the workstation software has to contact the Kerberos ticket-granting server. The following steps, however, are transparent to you when you access the service. This was a goal of Kerberos.

6. The workstation builds a message to be sent to the ticket-granting service. This message is a 3-tuple, consisting of

   message = { *sealed-ticket, sealed-authenticator, end-server-name* }

   The *authenticator* is created by the workstation software, and is a 3-tuple consisting of

   authenticator = { *login-name, WS-net-address, current-time* }

   To seal the authenticator, the workstation encrypts it using the TGS session key that it obtained in step 5. This message is sent to the ticket-granting service. Note that the first two elements of the message are encrypted (sealed) and the last element is a name that need not be encrypted.

7. The ticket-granting service receives the message and first decrypts the sealed ticket using its TGS encryption key. Recall that this ticket was originally sealed by the Kerberos authentication server in step 4 using this same key. From the unencrypted ticket the TGS obtains the TGS session key. It uses this session key to decrypt the sealed-authenticator. There are multiple items for the TGS to check for validity: the login name in both the ticket and the authenticator, and the TGS server name in the ticket. It also compares the network address in the ticket, the authenticator, and in the received message, and all must be equal. Finally, it compares the current time in the authenticator to make certain the message is recent. This requires that all the workstations and servers maintain their clocks within some prescribed tolerance. (We describe some techniques for clock synchronization on an internet in Chapter 10.) The TGS now looks up the *end-server-name* from the message in the Kerberos database, and obtains the encryption key for the specified service.

8. The TGS forms a new random session key and then creates a new ticket based on the requested end service name and the new session key.

   ticket = { *login-name, end-server-name, WS-net-address, new-session-key* }

   This ticket format contains the same fields as the one we showed in step 4 earlier. We have just labelled the session key with the qualifier "new" to differentiate it from the TGS session key that was created by the Kerberos authentication service. This ticket is sealed using the encryption key for the requested end server. The TGS forms a message and sends it to the workstation.

   message = { *new-session-key, sealed-ticket* }

   This message is then sealed using the TGS session key that the workstation knows, and sent to the workstation.

9. The workstation receives this sealed message and decrypts it using the TGS session key that it knows. From this message it again receives a sealed ticket that it cannot decrypt. This sealed ticket is what it has to send to the end server. It also obtains the new session key that it uses in the next step.

10. The workstation builds an authenticator

    authenticator = { *login-name, WS-net-address, current-time* }

    and seals it using the new session key. Finally, the workstation is able to send a message to the end-server. This message is of the same format as the message it sent to the TGS.

    message = { *sealed-ticket, sealed-authenticator, end-server-name* }

    This message isn't encrypted, since both the ticket and authenticator within the message are sealed, and the name of the end server isn't a secret.

11. The end-server receives this message and first decrypts the sealed-ticket using its encryption key, which only it and Kerberos know. It then uses the new session key contained in the ticket to decrypt the authenticator and does the same validation process that we described in step 7.

Tickets and *authenticators* are the key points to understand in the implementation of Kerberos. Let's review some of the central points to this scheme and some of the details that we've ignored.

- In order for the workstation to use any end-server, a ticket is required. All tickets, other than the first ticket, are obtained from the TGS. The first ticket is special: it is a ticket for the TGS and is obtained from the Kerberos authentication server.
- The tickets held by a workstation are not comprehensible to the workstation. They are encrypted using the key of the server for whom the ticket is to be used for.

- Every ticket is associated with a session key that is assigned every time a ticket is allocated.
- Tickets are reusable. We didn't mention this above, but once a ticket has been allocated to a workstation for a particular service, that ticket can be reused. Every ticket has a lifetime, typically eight hours. After a ticket has expired, you have to identify yourself to Kerberos again, entering your login name and password. To accomplish this expiration of tickets, each ticket also contains the time the ticket was issued and the amount of time the ticket is valid for.
- Unlike the ticket, which can be reused, a new authenticator is required every time the client initiates a new connection with a server. The authenticator carries a time stamp within it, and the authenticator expires a few minutes after it is issued. This is why we mentioned that all the workstations and servers have to maintain synchronized clocks. The preciseness of this synchronization and the size of the network determines the maximum reasonable lifetime for an authenticator.
- A server should maintain a history of previous client requests for which the time stamp in the authenticator is still valid (i.e., a history of all requests within the past few minutes). This way a server can reject duplicate requests that could arise from a stolen ticket and authenticator.
- Since both the ticket and authenticator contain the network address of the client, another workstation can't use stolen copies of these without changing their system to impersonate the owner's network address. Furthermore, since an authenticator has a short lifetime and is only valid once, the impersonating system has to do all the following before the authenticator expires: steal the ticket and authenticator, assure that the original copies of the ticket and authenticator don't arrive at the specified end-server, and modify its network address to look like the original client.
- Once a server has validated a client's request for service, that client and server share a private encryption key (the session key that is part of the ticket for that client and server) that no one else besides Kerberos knows. If desired, the client and server can encrypt all their session data using this key, or they can choose not to encrypt the session data at all. Kerberos provides the private key, but it is the application's choice whether to use it or not for all the session data.
- During a workstation session, a list of

  { *server-name, sealed-ticket, session-key* }

  3-tuples is constructed by the workstation on behalf of the user. The first entry will be for the ticket-granting server, and the others for each end-server contacted by the workstation on your behalf. Someone else cannot legitimately login to the workstation and use entries from this list (should they find this list in memory), since your name (the client to whom the tickets were issued) is contained in the sealed ticket. Also, when you log out from a workstation, this list of 3-tuples is destroyed.

- Once a server has validated a client, an option exists for the client to validate the server. This handles the problem of an intruder impersonating as a server. The client in this case requires that the server send back a message consisting of the time stamp from the client's authenticator, with one added to the time stamp value. This message is encrypted using the session key that was passed from the client to the server. If the server is bogus, it won't know the real server's encryption key, so it won't be able to obtain the session key from the sealed ticket or the time stamp from the sealed authenticator.

We described Kerberos as a trusted third-party authentication scheme. The adjective "third-party" is because Kerberos is a third party to the client and end-server. Trust is inherent throughout the system: You trust Kerberos, if it correctly provides your encryption key. The server trusts you, the client, if you pass the server a ticket that was sealed using the server's encryption key. As with most contemporary computer security schemes, an intruder that successfully obtains a legitimate user's login name and password can impersonate that user.

9.4 Summary

In this chapter we've covered two types of security that can be provided in a network environment. We'll encounter the techniques used by 4.3BSD in most of the remaining chapters of this text. We'll also refer to Figure 9.1 as we discuss each of the servers shown in this figure. These techniques aren't foolproof, but they are in widespread use.

Next, we provided an overview of the Kerberos system. Kerberos provides additional security, compared to 4.3BSD. It is the type of security system that will probably be used in post–4.3BSD systems.