15

Interprocess Communication

15.1 Introduction

In Chapter 8, we described the process control primitives and saw how to work with multiple processes. But the only way for these processes to exchange information is by passing open files across a `fork` or an `exec` or through the file system. We'll now describe other techniques for processes to communicate with one another: interprocess communication (IPC).

In the past, UNIX System IPC was a hodgepodge of various approaches, few of which were portable across all UNIX system implementations. Through the POSIX and The Open Group (formerly X/Open) standardization efforts, the situation has since improved, but differences still exist. Figure 15.1 summarizes the various forms of IPC that are supported by the four implementations discussed in this text.

Note that the Single UNIX Specification (the "SUS" column) allows an implementation to support full-duplex pipes, but requires only half-duplex pipes. An implementation that supports full-duplex pipes will still work with correctly written applications that assume that the underlying operating system supports only half-duplex pipes. We use "(full)" instead of a bullet to show implementations that support half-duplex pipes by using full-duplex pipes.

In Figure 15.1, we show a bullet where basic functionality is supported. For full-duplex pipes, if the feature can be provided through UNIX domain sockets (Section 17.2), we show "UDS" in the column. Some implementations support the feature with pipes and UNIX domain sockets, so these entries have both "UDS" and a bullet.

The IPC interfaces introduced as part of the real-time extensions to POSIX.1 were included as options in the Single UNIX Specification. In SUSv4, the semaphore interfaces were moved from an option to the base specification.

IPC type	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•, UDS	UDS	UDS	•, UDS
named full-duplex pipes	obsolescent	UDS	UDS	UDS	•, UDS
XSI message queues	XSI	•	•	•	•
XSI semaphores	XSI	•	•	•	•
XSI shared memory	XSI	•	•	•	•
message queues (real-time)	MSG option	•	•		•
semaphores	•	•	•	•	•
shared memory (real-time)	SHM option	•	•	•	•
sockets	•	•	•	•	•
STREAMS	obsolescent				•

Figure 15.1 Summary of UNIX System IPC

Named full-duplex pipes are provided as mounted STREAMS-based pipes, but are marked obsolescent in the Single UNIX Specification.

> Although support for STREAMS on Linux is available in the "Linux Fast-STREAMS" package from the OpenSS7 project, the package hasn't been updated recently. The latest release of the package from 2008 claims to work with kernels up to Linux 2.6.26.

The first ten forms of IPC in Figure 15.1 are usually restricted to IPC between processes on the same host. The final two rows—sockets and STREAMS—are the only two forms that are generally supported for IPC between processes on different hosts.

We have divided the discussion of IPC into three chapters. In this chapter, we examine classical IPC: pipes, FIFOs, message queues, semaphores, and shared memory. In the next chapter, we take a look at network IPC using the sockets mechanism. In Chapter 17, we take a look at some advanced features of IPC.

15.2 Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

We'll see that FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

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

Returns: 0 if OK, –1 on error

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

> Originally in 4.3BSD and 4.4BSD, pipes were implemented using UNIX domain sockets. Even though UNIX domain sockets are full duplex by default, these operating systems hobbled the sockets used with pipes so that they operated in half-duplex mode only.
>
> POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

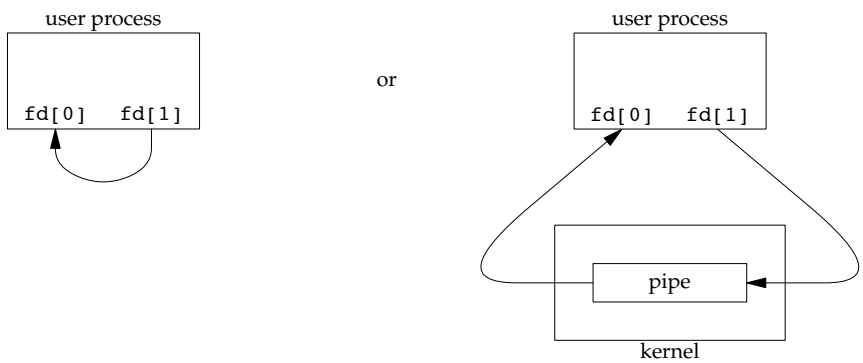

Figure 15.2 Two ways to view a half-duplex pipe

The `fstat` function (Section 4.2) returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

> POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe. This is, however, nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. Figure 15.3 shows this scenario.

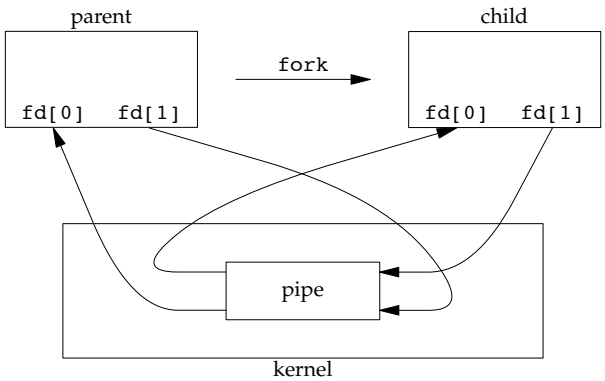

Figure 15.3 Half-duplex pipe after a `fork`

What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (`fd[0]`), and the child closes the write end (`fd[1]`). Figure 15.4 shows the resulting arrangement of descriptors.

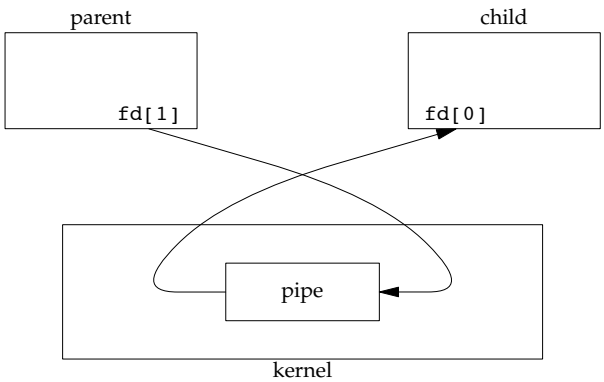

Figure 15.4 Pipe from parent to child

For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, two rules apply.

1. If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)

2. If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, `write` returns –1 with `errno` set to `EPIPE`.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A `write` of `PIPE_BUF` bytes or less will not be interleaved with the `write`s from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we `write` more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf` (recall Figure 2.12).

Example

Figure 15.5 shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Figure 15.5 Send data from parent to child over a pipe

Note that the pipe direction here matches the orientation shown in Figure 15.4. □

In the previous example, we called `read` and `write` directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Example

Consider a program that displays some output that it has created, one page at a time. Rather than reinvent the pagination done by several UNIX system utilities, we want to invoke the user's favorite pager. To avoid writing all the data to a temporary file and calling `system` to display that file, we want to pipe the output directly to the pager. To do this, we create a pipe, `fork` a child process, set up the child's standard input to be the read end of the pipe, and `exec` the user's pager program. Figure 15.6 shows how to do this. (This example takes a command-line argument to specify the name of a file to display. Often, a program of this type would already have the data to display to the terminal in memory.)

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER   "/bin/more"     /* default pager program */

int
main(int argc, char *argv[])
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                   /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
```

```
        exit(0);
    } else {                                        /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
    exit(0);
}
```

Figure 15.6 Copy file to pager program

Before calling `fork`, we create a pipe. After the `fork`, the parent closes its read end, and the child closes its write end. The child then calls `dup2` to have its standard input be the read end of the pipe. When the pager program is executed, its standard input will be the read end of the pipe.

When we duplicate one descriptor onto another (`fd[0]` onto standard input in the child), we have to be careful that the descriptor doesn't already have the desired value. If the descriptor already had the desired value and we called `dup2` and `close`, the single copy of the descriptor would be closed. (Recall the operation of `dup2` when its two arguments are equal, discussed in Section 3.12.) In this program, if standard input had not been opened by the shell, the `fopen` at the beginning of the program should have used descriptor 0, the lowest unused descriptor, so `fd[0]` should never equal standard input. Nevertheless, whenever we call `dup2` and `close` to duplicate one descriptor onto another, we'll always compare the descriptors first, as a defensive programming measure.

Note how we try to use the environment variable `PAGER` to obtain the name of the user's pager program. If this doesn't work, we use a default. This is a common usage of environment variables. □

Example

Recall the five functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Section 8.9. In Figure 10.24, we showed an implementation using signals. Figure 15.7 shows an implementation using pipes.

```
#include "apue.h"

static int  pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

Figure 15.7 Routines to let a parent and child synchronize

We create two pipes before the `fork`, as shown in Figure 15.8. The parent writes the character "p" across the top pipe when `TELL_CHILD` is called, and the child writes the character "c" across the bottom pipe when `TELL_PARENT` is called. The corresponding `WAIT_xxx` functions do a blocking `read` for the single character.

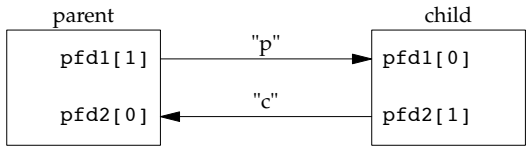

Figure 15.8 Using two pipes for parent–child synchronization

Note that each pipe has an extra reader, which doesn't matter. That is, in addition to the child reading from `pfd1[0]`, the parent has this end of the top pipe open for reading. This doesn't affect us, since the parent doesn't try to read from this pipe. □

15.3 `popen` and `pclose` Functions

Since a common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the `popen` and `pclose` functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, `fork`ing a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
                                        Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);
                                Returns: termination status of cmdstring, or –1 on error
```

The function `popen` does a `fork` and `exec` to execute the *cmdstring* and returns a standard I/O file pointer. If *type* is `"r"`, the file pointer is connected to the standard output of *cmdstring* (Figure 15.9).

Figure 15.9 Result of `fp = popen(`*cmdstring*`, "r")`

If *type* is `"w"`, the file pointer is connected to the standard input of *cmdstring*, as shown in Figure 15.10.

Figure 15.10 Result of `fp = popen(`*cmdstring*`, "w")`

One way to remember the final argument to `popen` is to remember that, like `fopen`, the returned file pointer is readable if *type* is `"r"` or writable if *type* is `"w"`.

The `pclose` function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. (We described the termination status in Section 8.6. The `system` function, described in Section 8.13, also returns the termination status.) If the shell cannot be executed, the termination status returned by `pclose` is as if the shell had executed `exit(127)`.

The *cmdstring* is executed by the Bourne shell, as in

```
sh -c cmdstring
```

This means that the shell expands any of its special characters in *cmdstring*. This allows us to say, for example,

```
fp = popen("ls *.c", "r");
```

or

```
fp = popen("cmd 2>&1", "r");
```

Example

Let's redo the program from Figure 15.6, using `popen`. This is shown in Figure 15.11.

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER   "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE    *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    /* copy argv[1] to pager */
```

```
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");

    exit(0);
}
```

Figure 15.11 Copy file to pager program using popen

Using popen reduces the amount of code we have to write.

The shell command `${PAGER:-more}` says to use the value of the shell variable PAGER if it is defined and non-null; otherwise, use the string more. □

Example — popen and pclose Functions

Figure 15.12 shows our version of popen and pclose.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Pointer to array allocated at run-time.
 */
static pid_t    *childpid = NULL;

/*
 * From our open_max(), Figure 2.17.
 */
static int      maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int     i;
    int     pfd[2];
    pid_t   pid;
    FILE    *fp;

    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;
        return(NULL);
    }
```

```
if (childpid == NULL) {     /* first time through */
    /* allocate zeroed out array for child pids */
    maxfd = open_max();
    if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
        return(NULL);
}

if (pipe(pfd) < 0)
    return(NULL);   /* errno set by pipe() */
if (pfd[0] >= maxfd || pfd[1] >= maxfd) {
    close(pfd[0]);
    close(pfd[1]);
    errno = EMFILE;
    return(NULL);
}

if ((pid = fork()) < 0) {
    return(NULL);   /* errno set by fork() */
} else if (pid == 0) {                          /* child */
    if (*type == 'r') {
        close(pfd[0]);
        if (pfd[1] != STDOUT_FILENO) {
            dup2(pfd[1], STDOUT_FILENO);
            close(pfd[1]);
        }
    } else {
        close(pfd[1]);
        if (pfd[0] != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
            close(pfd[0]);
        }
    }

    /* close all descriptors in childpid[] */
    for (i = 0; i < maxfd; i++)
        if (childpid[i] > 0)
            close(i);

    execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
    _exit(127);
}

/* parent continues... */
if (*type == 'r') {
    close(pfd[1]);
    if ((fp = fdopen(pfd[0], type)) == NULL)
        return(NULL);
} else {
    close(pfd[0]);
    if ((fp = fdopen(pfd[1], type)) == NULL)
        return(NULL);
```

```
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if (fd >= maxfd) {
        errno = EINVAL;
        return(-1);     /* invalid file descriptor */
    }
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

Figure 15.12 The popen and pclose functions

Although the core of popen is similar to the code we've used earlier in this chapter, there are many details that we need to take care of. First, each time popen is called, we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer. We choose to save the child's process ID in the array childpid, which we index by the file descriptor. This way, when pclose is called with the FILE pointer as its argument, we call the standard I/O function fileno to get the file descriptor and then have the child process ID for the call to waitpid. Since it's possible for a given process to call popen more than once, we dynamically allocate the childpid array (the first time popen is called), with room for as many children as there are file descriptors.

Note that our open_max function from Figure 2.17 can return a guess of the maximum number of open files if this value is indeterminate for the system. We need to be careful not to use a pipe file descriptor whose value is larger than (or equal to) what the open_max function returns. In popen, if the value returned by open_max happens to be too small, we close the pipe file descriptors, set errno to EMFILE to indicate too many file descriptors are open, and return –1. In pclose, if the file descriptor corresponding to the file pointer argument is larger than expected, we set errno to EINVAL and return –1.

Calling pipe and fork and then duplicating the appropriate descriptors for each process in the popen function is similar to what we did earlier in this chapter.

POSIX.1 requires that popen close any streams that are still open in the child from previous calls to popen. To do this, we go through the childpid array in the child, closing any descriptors that are still open.

What happens if the caller of pclose has established a signal handler for SIGCHLD? The call to waitpid from pclose would return an error of EINTR. Since the caller is allowed to catch this signal (or any other signal that might interrupt the call to waitpid), we simply call waitpid again if it is interrupted by a caught signal.

Note that if the application calls waitpid and obtains the exit status of the child created by popen, we will call waitpid when the application calls pclose, find that the child no longer exists, and return –1 with errno set to ECHILD. This is the behavior required by POSIX.1 in this situation.

> Some early versions of pclose returned an error of EINTR if a signal interrupted the wait. Also, some early versions of pclose blocked or ignored the signals SIGINT, SIGQUIT, and SIGHUP during the wait. This is not allowed by POSIX.1.
>
> □

Note that popen should never be called by a set-user-ID or set-group-ID program. When it executes the command, popen does the equivalent of

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

which executes the shell and *command* with the environment inherited by the caller. A malicious user can manipulate the environment so that the shell executes commands other than those intended, with the elevated permissions granted by the set-ID file mode.

One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Example

Consider an application that writes a prompt to standard output and reads a line from standard input. With the popen function, we can interpose a program between the application and its input to transform the input. Figure 15.13 shows the arrangement of processes in this situation.

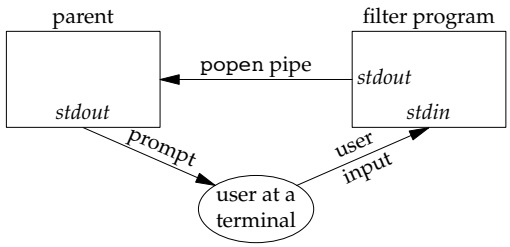

Figure 15.13 Transforming input using popen

The transformation could be pathname expansion, for example, or providing a history mechanism (remembering previously entered commands).

Figure 15.14 shows a simple filter to demonstrate this operation. The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

```
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Figure 15.14 Filter to convert uppercase characters to lowercase

We compile this filter into the executable file `myuclc`, which we then invoke from the program in Figure 15.15 using `popen`.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;
```

```
    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

Figure 15.15 Invoke uppercase/lowercase filter to read commands

We need to call `fflush` after writing the prompt, because the standard output is normally line buffered, and the prompt does not contain a newline. □

15.4 Coprocesses

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a *coprocess* when the same program generates the filter's input and reads the filter's output.

The Korn shell provides coprocesses [Bolsky and Korn 1995]. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted (see pp. 62–63 of Bolsky and Korn [1995] for all the details), coprocesses are also useful from a C program.

Whereas `popen` gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.

Example

Let's look at coprocesses with an example. The process creates two pipes: one is the standard input of the coprocess and the other is the standard output of the coprocess. Figure 15.16 shows this arrangement.

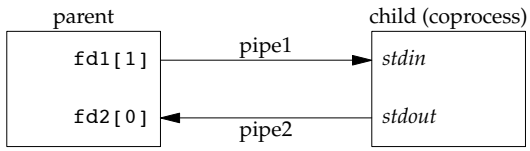

Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

The program in Figure 15.17 is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. (Coprocesses usually do more interesting work than we illustrate here. This example is admittedly contrived so that we can study the plumbing needed to connect the processes.)

```
#include "apue.h"

int
main(void)
{
    int     n, int1, int2;
    char    line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

Figure 15.17 Simple filter to add two numbers

We compile this program and leave the executable in the file `add2`.

The program in Figure 15.18 invokes the `add2` coprocess after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

```
#include "apue.h"

static void sig_pipe(int);      /* our signal handler */

int
```

```
main(void)
{
    int     n, fd1[2], fd2[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                           /* parent */
        close(fd1[0]);
        close(fd2[1]);

        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }

        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                        /* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd1[0]);
        }

        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
```

```
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

Figure 15.18 Program to drive the add2 filter

Here, we create two pipes, with the parent and the child closing the ends they don't need. We have to use two pipes: one for the standard input of the coprocess and one for its standard output. The child then calls `dup2` to move the pipe descriptors onto its standard input and standard output, before calling `execl`.

If we compile and run the program in Figure 15.18, it works as expected. Furthermore, if we `kill` the `add2` coprocess while the program in Figure 15.18 is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader. (See Exercise 15.4.) □

Example

In the coprocess `add2` (Figure 15.17), we purposely used low-level I/O (UNIX system calls): `read` and `write`. What happens if we rewrite this coprocess to use standard I/O? Figure 15.19 shows the new version.

```
#include "apue.h"

int
main(void)
{
    int     int1, int2;
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("printf error");
        } else {
            if (printf("invalid args\n") == EOF)
                err_sys("printf error");
        }
    }
    exit(0);
}
```

Figure 15.19 Filter to add two numbers, using standard I/O

If we invoke this new coprocess from the program in Figure 15.18, it no longer works. The problem is the default standard I/O buffering. When the program in Figure 15.19 is invoked, the first `fgets` on the standard input causes the standard I/O library to allocate a buffer and choose the type of buffering. Since the standard input is a pipe, the standard I/O library defaults to fully buffered. The same thing happens with the standard output. While `add2` is blocked reading from its standard input, the program in Figure 15.18 is blocked reading from the pipe. We have a deadlock.

Here, we have control over the coprocess that's being run. We can change the program in Figure 15.19 by adding the following four lines before the `while` loop:

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
```

These lines cause `fgets` to return when a line is available and cause `printf` to do an `fflush` when a newline is output (refer to Section 5.4 for the details on standard I/O buffering). Making these explicit calls to `setvbuf` fixes the program in Figure 15.19.

If we aren't able to modify the program that we're piping the output into, other techniques are required. For example, if we use `awk`(1) as a coprocess from our program (instead of the `add2` program), the following won't work:

```
#! /bin/awk -f
{ print $1 + $2 }
```

The reason this won't work is again the standard I/O buffering. But in this case, we cannot change the way `awk` works (unless we have the source code for it). We are unable to modify the executable of `awk` in any way to change the way the standard I/O buffering is handled.

The solution for this general problem is to make the coprocess being invoked (`awk` in this case) think that its standard input and standard output are connected to a terminal. That causes the standard I/O routines in the coprocess to line buffer these two I/O streams, similar to what we did with the explicit calls to `setvbuf` previously. We use pseudo terminals to do this in Chapter 19. □

15.5 FIFOs

FIFOs are sometimes called named pipes. Unnamed pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data.

We saw in Chapter 4 that a FIFO is a type of file. One of the encodings of the `st_mode` member of the `stat` structure (Section 4.2) indicates that a file is a FIFO. We can test for this with the `S_ISFIFO` macro.

Creating a FIFO is similar to creating a file. Indeed, the *pathname* for a FIFO exists in the file system.

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

int mkfifo(const char *path, mode_t mode);

int mkfifoat(int fd, const char *path, mode_t mode);
```

Both return: 0 if OK, –1 on error

The specification of the *mode* argument is the same as for the open function (Section 3.3). The rules for the user and group ownership of the new FIFO are the same as we described in Section 4.6.

The mkfifoat function is similar to the mkfifo function, except that it can be used to create a FIFO in a location relative to the directory represented by the *fd* file descriptor argument. Like the other *at functions, there are three cases:

1. If the *path* parameter specifies an absolute pathname, then the *fd* parameter is ignored and the mkfifoat function behaves like the mkfifo function.
2. If the *path* parameter specifies a relative pathname and the *fd* parameter is a valid file descriptor for an open directory, the pathname is evaluated relative to this directory.
3. If the *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD, the pathname is evaluated starting in the current working directory, and mkfifoat behaves like mkfifo.

Once we have used mkfifo or mkfifoat to create a FIFO, we open it using open. Indeed, the normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.

> Applications can create FIFOs with the mknod and mknodat functions. Because POSIX.1 originally didn't include mknod, the mkfifo function was invented specifically for POSIX.1. The mknod and mknodat functions are included in the XSI option in POSIX.1.
>
> POSIX.1 also includes support for the mkfifo(1) command. All four platforms discussed in this text provide this command. As a result, we can create a FIFO using a shell command and then access it with the normal shell I/O redirection.

When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

- In the normal case (without O_NONBLOCK), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns –1 with errno set to ENXIO if no process has the FIFO open for reading.

As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.

It is common to have multiple writers for a given FIFO. This means that we have to worry about atomic writes if we don't want the writes from multiple processes to be

interleaved. As with pipes, the constant `PIPE_BUF` specifies the maximum amount of data that can be written atomically to a FIFO.

There are two uses for FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers.

We discuss each of these uses with an example.

Example — Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file (similar to using pipes to avoid intermediate disk files). But whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

Consider a procedure that needs to process a filtered input stream twice. Figure 15.20 shows this arrangement.

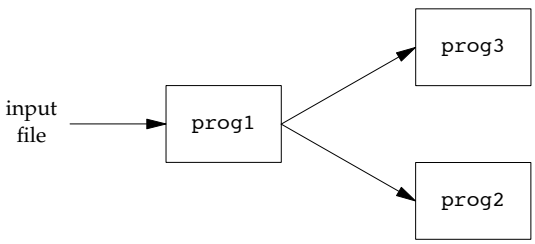

Figure 15.20 Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program `tee`(1), we can accomplish this procedure without using a temporary file. (The `tee` program copies its standard input to both its standard output and the file named on its command line.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

We create the FIFO and then start `prog3` in the background, reading from the FIFO. We then start `prog1` and use `tee` to send its input to both the FIFO and `prog2`. Figure 15.21 shows the process arrangement. □

Example — Client–Server Communication Using a FIFO

Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known

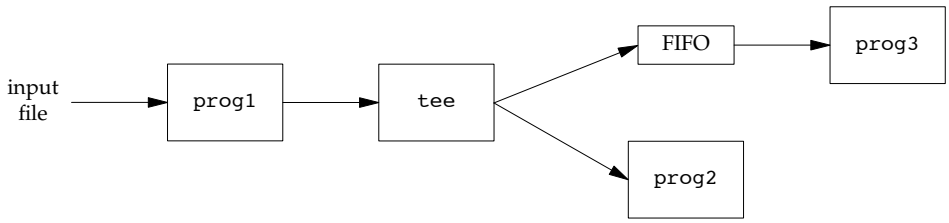

Figure 15.21 Using a FIFO and `tee` to send a stream to two different processes

FIFO that the server creates. (By "well-known," we mean that the pathname of the FIFO is known to all the clients that need to contact the server.) Figure 15.22 shows this arrangement.

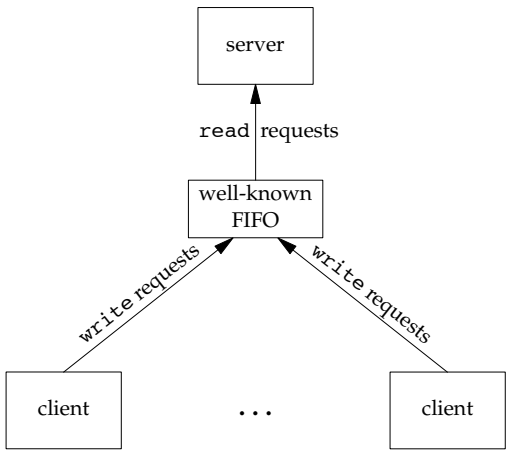

Figure 15.22 Clients sending requests to a server using a FIFO

Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than `PIPE_BUF` bytes in size. This prevents any interleaving of the client `write`s.

The problem in using FIFOs for this type of client–server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID. For example, the server can create a FIFO with the name `/tmp/serv1.XXXXX`, where `XXXXX` is replaced with the client's process ID. This arrangement is shown in Figure 15.23.

This arrangement works, although it is impossible for the server to tell whether a client crashes. A client crash leaves the client-specific FIFO in the file system. The

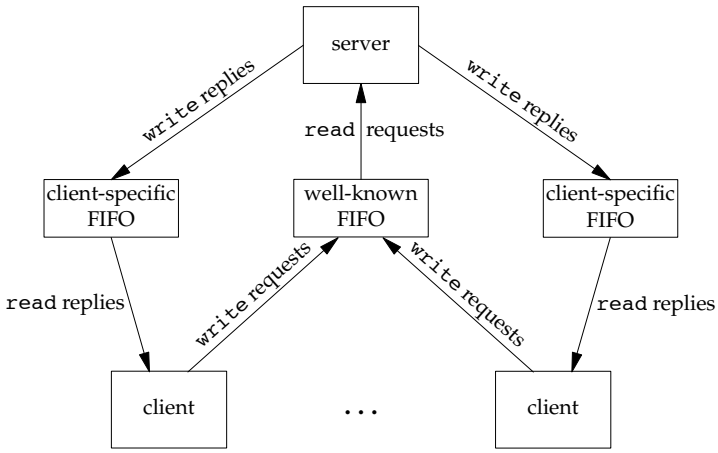

Figure 15.23 Client–server communication using FIFOs

server also must catch `SIGPIPE`, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

With the arrangement shown in Figure 15.23, if the server opens its well-known FIFO read-only (since it only `read`s from it) each time the number of clients goes from 1 to 0, the server will `read` an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server `open` its well-known FIFO for read–write. (See Exercise 15.10.) □

15.6 XSI IPC

The three types of IPC that we call XSI IPC—message queues, semaphores, and shared memory—have many similarities. In this section, we cover these similar features; in the following sections, we look at the specific functions for each of the three IPC types.

> The XSI IPC functions are based closely on the System V IPC functions. These three types of IPC originated in the 1970s in an internal AT&T version of the UNIX System called "Columbus UNIX." These IPC features were later added to System V. They are often criticized for inventing their own namespace instead of using the file system.

15.6.1 Identifiers and Keys

Each *IPC structure* (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer *identifier*. To send a message to or fetch a message from a message queue, for example, all we need know is the identifier for the queue. Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a

given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0.

The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a *key* that acts as an external name.

Whenever an IPC structure is being created (by calling `msgget`, `semget`, or `shmget`), a key must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `<sys/types.h>`. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

1. The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage of this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.

   The `IPC_PRIVATE` key is also used in a parent–child relationship. The parent creates a new IPC structure specifying `IPC_PRIVATE`, and the resulting identifier is then available to the child after the `fork`. The child can pass the identifier to a new program as an argument to one of the `exec` functions.

2. The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.

3. The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.

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

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

Returns: key if OK, `(key_t)`–1 on error

The *path* argument must refer to an existing file. Only the lower 8 bits of *id* are used when generating the key.

The key created by `ftok` is usually formed by taking parts of the `st_dev` and `st_ino` fields in the `stat` structure (Section 4.2) corresponding to the given pathname and combining them with the project ID. If two pathnames refer to two different files,

then ftok usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, information loss can occur when creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.

The three get functions (msgget, semget, and shmget) all have two similar arguments: a *key* and an integer *flag*. A new IPC structure is created (normally by a server) if either *key* is IPC_PRIVATE or *key* is not currently associated with an IPC structure of the particular type and the IPC_CREAT bit of *flag* is specified. To reference an existing queue (normally done by a client), *key* must equal the key that was specified when the queue was created, and IPC_CREAT must not be specified.

Note that it's never possible to specify IPC_PRIVATE to reference an existing queue, since this special *key* value always creates a new queue. To reference an existing queue that was created with a *key* of IPC_PRIVATE, we must know the associated identifier and then use that identifier in the other IPC calls (such as msgsnd and msgrcv), bypassing the get function.

If we want to create a new IPC structure, making sure that we don't reference an existing one with the same identifier, we must specify a *flag* with both the IPC_CREAT and IPC_EXCL bits set. Doing this causes an error return of EEXIST if the IPC structure already exists. (This is similar to an open that specifies the O_CREAT and O_EXCL flags.)

15.6.2 Permission Structure

XSI IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user ID */
  gid_t  gid;  /* owner's effective group ID */
  uid_t  cuid; /* creator's effective user ID */
  gid_t  cgid; /* creator's effective group ID */
  mode_t mode; /* access modes */
  :
};
```

Each implementation includes additional members. See <sys/ipc.h> on your system for the complete definition.

All the fields are initialized when the IPC structure is created. At a later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or chmod for a file.

The values in the mode field are similar to the values we saw in Figure 4.6, but there is nothing corresponding to execute permission for any of the IPC structures. Also, message queues and shared memory use the terms *read* and *write*, but semaphores use the terms *read* and *alter*. Figure 15.24 shows the six permissions for each form of IPC.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Figure 15.24 XSI IPC permissions

Some implementations define symbolic constants to represent each permission, but these constants are not standardized by the Single UNIX Specification.

15.6.3 Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

> Each platform provides its own way to report and modify a particular limit. FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 provide the `sysctl` command to view and modify kernel configuration parameters. On Solaris 10, changes to kernel IPC limits are made with the `prctl` command.
>
> On Linux, you can display the IPC-related limits by running `ipcs -l`. On FreeBSD and Mac OS X, the equivalent command is `ipcs -T`. On Solaris, you can discover the tunable parameters by running `sysdef -i`.

15.6.4 Advantages and Disadvantages

A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm`(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.

Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions we described in Chapters 3 and 4. Almost a dozen new system calls (`msgget`, `semop`, `shmat`, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an `ls` command, we can't remove them with the `rm` command, and we can't change their permissions with the `chmod` command. Instead, two new commands—`ipcs`(1) and `ipcrm`(1)—were added.

Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (`select` and `poll`) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busy–wait loop.

An overview of a transaction processing system built using System V IPC is given in Andrade, Carges, and Kovach [1989]. They claim that the namespace used by System V IPC (the identifiers) is an advantage, not a problem as we said earlier, because using identifiers allows a process to send a message to a message queue with a single function call (`msgsnd`), whereas other forms of IPC normally require an `open`, `write`, and `close`. This argument is false. Clients still have to obtain the identifier for the server's queue somehow, to avoid using a key and calling `msgget`. The identifier assigned to a particular queue depends on how many other message queues exist when the queue is created and how many times the table in the kernel assigned to the new queue has been used since the kernel was bootstrapped. This is a dynamic value that can't be guessed or stored in a header. As we mentioned in Section 15.6.1, minimally a server has to write the assigned queue identifier to a file for its clients to read.

Other advantages listed by these authors for message queues are that they're reliable, flow controlled, and record oriented, and that they can be processed in other than first-in, first-out order. Figure 15.25 compares some of the features of these various forms of IPC.

IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

Figure 15.25 Comparison of features of various forms of IPC

(We describe stream and datagram sockets in Chapter 16. We describe UNIX domain sockets in Section 17.2.) By "connectionless," we mean the ability to send a message without having to call some form of an open function first. As described previously, we don't consider message queues connectionless, since some technique is required to obtain the identifier for a queue. Since all these forms of IPC are restricted to a single host, all are reliable. When the messages are sent across a network, the possibility of messages being lost becomes a concern. "Flow control" means that the sender is put to sleep if there is a shortage of system resources (buffers) or if the receiver can't accept any more messages. When the flow control condition subsides (i.e., when there is room in the queue), the sender should automatically be awakened.

One feature that we don't show in Figure 15.25 is whether the IPC facility can automatically create a unique connection to a server for each client. We'll see in Chapter 17 that UNIX stream sockets provide this capability. The next three sections describe each of the three forms of XSI IPC in detail.

15.7 Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a *queue* and its identifier a *queue ID*.

> The Single UNIX Specification message-passing option includes an alternative IPC message queue interface derived from the POSIX real-time extensions. We do not discuss it in this text.

A new queue is created or an existing queue opened by `msgget`. New messages are added to the end of a queue by `msgsnd`. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to `msgsnd` when the message is added to a queue. Messages are fetched from a queue by `msgrcv`. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following `msqid_ds` structure associated with it:

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
  msgqnum_t        msg_qnum;     /* # of messages on queue */
  msglen_t         msg_qbytes;   /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */
  .
  .
  .
};
```

This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification. Implementations include additional fields not covered by the standard.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
size in bytes of largest message we can send	16,384	8,192	16,384	derived
maximum size in bytes of a particular queue (i.e., the sum of all the sizes of messages on the queue)	2,048	16,384	2,048	65,536
maximum number of messages queues, systemwide	40	derived	40	128
maximum number of messages, systemwide	40	derived	40	8,192

Figure 15.26 System limits that affect message queues

Figure 15.26 lists the system limits that affect message queues. We show "derived" where a limit is derived from other limits. On Linux, for example, the maximum number of messages is based on the maximum number of queues and the maximum amount of data allowed on the queues. The maximum number of queues, in turn, is based on the amount of RAM installed in the system. Note that the queue maximum byte size limit further limits the maximum size of a message to be placed on a queue.

The first function normally called is `msgget` to either open an existing queue or create a new queue.

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

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

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

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new queue is created or an existing queue is referenced. When a new queue is created, the following members of the `msqid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue. This function and the related functions for semaphores and shared memory (`semctl` and `shmctl`) are the `ioctl`-like functions for XSI IPC (i.e., the garbage-can functions).

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

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

Returns: 0 if OK, –1 on error

The *cmd* argument specifies the command to be performed on the queue specified by *msqid*.

`IPC_STAT` Fetch the `msqid_ds` structure for this queue, storing it in the structure pointed to by *buf*.

`IPC_SET` Copy the following fields from the structure pointed to by *buf* to the `msqid_ds` structure associated with this queue: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges. Only the superuser can increase the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of `EIDRM` on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges.

We'll see that these three commands (IPC_STAT, IPC_SET, and IPC_RMID) are also provided for semaphores and shared memory.

Data is placed onto a message queue by calling msgsnd.

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

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

Returns: 0 if OK, –1 on error

As we mentioned earlier, each message is composed of a positive long integer type field, a non-negative length (*nbytes*), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The *ptr* argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if *nbytes* is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg {
  long  mtype;      /* positive message type */
  char  mtext[512]; /* message data, of length nbytes */
};
```

The *ptr* argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

> Some platforms support both 32-bit and 64-bit environments. This affects the size of long integers and pointers. For example, on 64-bit SPARC systems, Solaris allows both 32-bit and 64-bit applications to coexist. If a 32-bit application were to exchange this structure over a pipe or a socket with a 64-bit application, problems would arise, because the size of a long integer is 4 bytes in a 32-bit application, but 8 bytes in a 64-bit application. This means that a 32-bit application will expect that the mtext field will start 4 bytes after the start of the structure, whereas a 64-bit application will expect the mtext field to start 8 bytes after the start of the structure. In this situation, part of the 64-bit application's mtype field will appear as part of the mtext field to the 32-bit application, and the first 4 bytes in the 32-bit application's mtext field will be interpreted as a part of the mtype field by the 64-bit application.
>
> This problem doesn't happen with XSI message queues, however. Solaris implements the 32-bit version of the IPC system calls with different entry points than the 64-bit version of the IPC system calls. The system calls know how to deal with a 32-bit application communicating with a 64-bit application, and treat the type field specially to avoid it interfering with the data portion of the message. The only potential problem is a loss of information when a 64-bit application sends a message with a value in the 8-byte type field that is larger than will fit in a 32-bit application's 4-byte type field. In this case, the 32-bit application will see a truncated type value.

A *flag* value of IPC_NOWAIT can be specified. This is similar to the nonblocking I/O flag for file I/O (Section 14.2). If the message queue is full (either the total number of messages on the queue equals the system limit, or the total number of bytes on the queue equals the system limit), specifying IPC_NOWAIT causes msgsnd to return immediately with an error of EAGAIN. If IPC_NOWAIT is not specified, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns. In the second case, an error of EIDRM is returned ("identifier removed"); in the last case, the error returned is EINTR.

Note how ungracefully the removal of a message queue is handled. Since a reference count is not maintained with each message queue (as there is for open files), the removal of a queue simply generates errors on the next queue operation by processes still using the queue. Semaphores handle this removal in the same fashion. In contrast, when a file is removed, the file's contents are not deleted until the last open descriptor for the file is closed.

When `msgsnd` returns successfully, the `msqid_ds` structure associated with the message queue is updated to indicate the process ID that made the call (`msg_lspid`), the time that the call was made (`msg_stime`), and that one more message is on the queue (`msg_qnum`).

Messages are retrieved from a queue by `msgrcv`.

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

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

Returns: size of data portion of message if OK, –1 on error

As with `msgsnd`, the *ptr* argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data. *nbytes* specifies the size of the data buffer. If the returned message is larger than *nbytes* and the `MSG_NOERROR` bit in *flag* is set, the message is truncated. (In this case, no notification is given to us that the message was truncated, and the remainder of the message is discarded.) If the message is too big and this *flag* value is not specified, an error of `E2BIG` is returned instead (and the message stays on the queue).

The *type* argument lets us specify which message we want.

type == 0 The first message on the queue is returned.

type > 0 The first message on the queue whose message type equals *type* is returned.

type < 0 The first message on the queue whose message type is the lowest value less than or equal to the absolute value of *type* is returned.

A nonzero *type* is used to read the messages in an order other than first in, first out. For example, the *type* could be a priority value if the application assigns priorities to the messages. Another use of this field is to contain the process ID of the client if a single message queue is being used by multiple clients and a single server (as long as a process ID fits in a long integer).

We can specify a *flag* value of `IPC_NOWAIT` to make the operation nonblocking, causing `msgrcv` to return –1 with `errno` set to `ENOMSG` if a message of the specified type is not available. If `IPC_NOWAIT` is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system (–1 is returned with `errno` set to `EIDRM`), or a signal is caught and the signal handler returns (causing `msgrcv` to return –1 with `errno` set to `EINTR`).

When `msgrcv` succeeds, the kernel updates the `msqid_ds` structure associated with the message queue to indicate the caller's process ID (`msg_lrpid`), the time of the call (`msg_rtime`), and that one less message is on the queue (`msg_qnum`).

Example — Timing Comparison of Message Queues and Full-Duplex Pipes

If we need a bidirectional flow of data between a client and a server, we can use either message queues or full-duplex pipes. (Recall from Figure 15.1 that full-duplex pipes are available through the UNIX domain sockets mechanism [Section 17.2], although some platforms provide a full-duplex pipe mechanism through the `pipe` function.)

Figure 15.27 shows a timing comparison of three of these techniques on Solaris: message queues, full-duplex (STREAMS) pipes, and UNIX domain sockets. The tests consisted of a program that created the IPC channel, called `fork`, and then sent about 200 megabytes of data from the parent to the child. The data was sent using 100,000 calls to `msgsnd`, with a message length of 2,000 bytes for the message queue, and 100,000 calls to `write`, with a length of 2,000 bytes for the full-duplex pipe and UNIX domain socket. The times are all in seconds.

Operation	User	System	Clock
message queue	0.58	4.16	5.09
full-duplex pipe	0.61	4.30	5.24
UNIX domain socket	0.59	5.58	7.49

Figure 15.27 Timing comparison of IPC alternatives on Solaris

These numbers show us that message queues, originally implemented to provide higher-than-normal-speed IPC, are no longer that much faster than other forms of IPC. (When message queues were implemented, the only other form of IPC available was half-duplex pipes.) When we consider the problems in using message queues (Section 15.6.4), we come to the conclusion that we shouldn't use them for new applications. □

15.8 Semaphores

A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

> The Single UNIX Specification includes an alternative set of semaphore interfaces that were originally part of its real-time extensions. We discuss these interfaces in Section 15.10.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

A common form of semaphore is called a *binary semaphore*. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The *undo* feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  :
};
```

The Single UNIX Specification defines the fields shown, but implementations can define additional members in the `semid_ds` structure.

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /* # processes awaiting semval>curval */
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  :
};
```

Figure 15.28 lists the system limits that affect semaphore sets.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum value of any semaphore	32,767	32,767	32,767	65,535
maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	32,767
maximum number of semaphore sets, systemwide	10	128	87,381	128
maximum number of semaphores, systemwide	60	32,000	87,381	derived
maximum number of semaphores per semaphore set	60	250	87,381	512
maximum number of undo structures, systemwide	30	32,000	87,381	derived
maximum number of undo entries per undo structures	10	unlimited	10	derived
maximum number of operations per `semop` call	100	32	5	512

Figure 15.28 System limits that affect semaphores

When we want to use XSI semaphores, we first need to obtain a semaphore ID by calling the `semget` function.

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

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

Returns: semaphore ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new set is created or an existing set is referenced. When a new set is created, the following members of the `semid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.

The number of semaphores in the set is *nsems*. If a new set is being created (typically by the server), we must specify *nsems*. If we are referencing an existing set (a client), we can specify *nsems* as 0.

The `semctl` function is the catchall for various semaphore operations.

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

int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

Returns: (see following)

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of various command-specific arguments:

```
union semun {
  int               val;   /* for SETVAL */
  struct semid_ds  *buf;   /* for IPC_STAT and IPC_SET */
  unsigned short   *array; /* for GETALL and SETALL */
};
```

Note that the optional argument is the actual union, not a pointer to the union.

> Usually our application must define the `semun` union. However, on FreeBSD 8.0, this is defined for us in `<sys/sem.h>`.

The *cmd* argument specifies one of the following ten commands to be performed on the set specified by *semid*. The five commands that refer to one particular semaphore value use *semnum* to specify one member of the set. The value of *semnum* is between 0 and *nsems* – *1*, inclusive.

`IPC_STAT` Fetch the `semid_ds` structure for this set, storing it in the structure pointed to by *arg.buf*.

`IPC_SET` Set the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` fields from the structure pointed to by *arg.buf* in the `semid_ds` structure associated with this set. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.

`IPC_RMID` Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of `EIDRM` on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.

`GETVAL` Return the value of `semval` for the member *semnum*.

`SETVAL` Set the value of `semval` for the member *semnum*. The value is specified by *arg.val*.

`GETPID` Return the value of `sempid` for the member *semnum*.

`GETNCNT` Return the value of `semncnt` for the member *semnum*.

`GETZCNT` Return the value of `semzcnt` for the member *semnum*.

`GETALL` Fetch all the semaphore values in the set. These values are stored in the array pointed to by *arg.array*.

`SETALL` Set all the semaphore values in the set to the values pointed to by *arg.array*.

For all the `GET` commands other than `GETALL`, the function returns the corresponding value. For the remaining commands, the return value is 0 if the call succeeds. On error, the `semctl` function sets `errno` and returns –1.

The function `semop` atomically performs an array of operations on a semaphore set.

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, –1 on error

The *semoparray* argument is a pointer to an array of semaphore operations, represented by `sembuf` structures:

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

The *nops* argument specifies the number of operations (elements) in the array.

The operation on each member of the set is specified by the corresponding `sem_op` value. This value can be negative, 0, or positive. (In the following discussion, we refer to the "undo" flag for a semaphore. This flag corresponds to the `SEM_UNDO` bit in the corresponding `sem_flg` member.)

1. The easiest case is when `sem_op` is positive. This case corresponds to the returning of resources by the process. The value of `sem_op` is added to the semaphore's value. If the undo flag is specified, `sem_op` is also subtracted from the semaphore's adjustment value for this process.
2. If `sem_op` is negative, we want to obtain resources that the semaphore controls.

   If the semaphore's value is greater than or equal to the absolute value of `sem_op` (the resources are available), the absolute value of `sem_op` is subtracted from the semaphore's value. This guarantees the resulting semaphore value is greater than or equal to 0. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

   If the semaphore's value is less than the absolute value of `sem_op` (the resources are not available), the following conditions apply.

   a. If `IPC_NOWAIT` is specified, `semop` returns with an error of `EAGAIN`.
   b. If `IPC_NOWAIT` is not specified, the `semncnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
      i. The semaphore's value becomes greater than or equal to the absolute value of `sem_op` (i.e., some other process has released some resources). The value of `semncnt` for this semaphore is decremented (since the calling process is done waiting), and the absolute value of `sem_op` is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.
      ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.
      iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semncnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.
3. If `sem_op` is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.

If the semaphore's value is currently 0, the function returns immediately.

If the semaphore's value is nonzero, the following conditions apply.

a. If IPC_NOWAIT is specified, return is made with an error of EAGAIN.

b. If IPC_NOWAIT is not specified, the semzcnt value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

   i. The semaphore's value becomes 0. The value of semzcnt for this semaphore is decremented (since the calling process is done waiting).

   ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.

   iii. A signal is caught by the process, and the signal handler returns. In this case, the value of semzcnt for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of EINTR.

The semop function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on exit

As we mentioned earlier, it is a problem if a process terminates while it has resources allocated through a semaphore. Whenever we specify the SEM_UNDO flag for a semaphore operation and we allocate resources (a sem_op value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of sem_op). When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore.

If we set the value of a semaphore using semctl, with either the SETVAL or SETALL commands, the adjustment value for that semaphore in all processes is set to 0.

Example — Timing Comparison of Semaphores, Record Locking, and Mutexes

If we are sharing a single resource among multiple processes, we can use one of three techniques to coordinate access. We can use a a semaphore, record locking, or a mutex that is mapped into the address spaces of both processes. It's interesting to compare the timing differences between the three techniques.

With a semaphore, we create a semaphore set consisting of a single member and initialize the semaphore's value to 1. To allocate the resource, we call semop with a sem_op of –1; to release the resource, we perform a sem_op of +1. We also specify SEM_UNDO with each operation, to handle the case of a process that terminates without releasing its resource.

With record locking, we create an empty file and use the first byte of the file (which need not exist) as the lock byte. To allocate the resource, we obtain a write lock on the

byte; to release it, we unlock the byte. The record locking properties guarantee that if a process terminates while holding a lock, the kernel automatically releases the lock.

To use a mutex, we need both processes to map the same file into their address spaces and initialize a mutex at the same offset in the file using the `PTHREAD_PROCESS_SHARED` mutex attribute. To allocate the resource, we lock the mutex; to release the resource, we unlock the mutex. If a process terminates without releasing the mutex, recovery is difficult unless we use a robust mutex (recall the `pthread_mutex_consistent` function discussed in Section 12.4.1).

Figure 15.29 shows the time required to perform these three locking techniques on Linux. In each case, the resource was allocated and then released 1,000,000 times. This was done simultaneously by three different processes. The times in Figure 15.29 are the totals in seconds for all three processes.

Operation	User	System	Clock
semaphores with undo	0.50	6.08	7.55
advisory record locking	0.51	9.06	4.38
mutex in shared memory	0.21	0.40	0.25

Figure 15.29 Timing comparison of locking alternatives on Linux

On Linux, record locking is faster than semaphores, but mutexes in shared memory outperform both semaphores and record locking. If we're locking a single resource and don't need all the fancy features of XSI semaphores, record locking is preferred over semaphores. The reasons are that it is much simpler to use, it is faster (on this platform), and the system takes care of any lingering locks when a process terminates. Even though using a mutex in shared memory is the fastest option on this platform, we still prefer to use record locking, unless performance is the primary concern. There are two reasons for this. First, recovery from process termination is more difficult using a mutex in memory shared among multiple processes. Second, the *process-shared* mutex attribute isn't universally supported yet. In older versions of the Single UNIX Specification, it was optional. Although it is still optional in SUSv4, it is now required by all XSI-conforming implementations.

> Of the four platforms covered in this text, only Linux 3.2.0 and Solaris 10 currently support the *process-shared* mutex attribute.
>
> □

15.9 Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as we saw at the end of the previous section, record locking or mutexes can also be used.)

The Single UNIX Specification shared memory objects option includes alternative interfaces, originally real-time extensions, to access shared memory. We don't discuss them in this text.

We've already seen one form of shared memory when multiple processes map the same file into their address spaces. The XSI shared memory differs from memory-mapped files in that there is no associated file. The XSI shared memory segments are anonymous segments of memory.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  .
  .
  .
};
```

(Implementations add other structure members to support shared memory segments.)

The type `shmatt_t` is defined to be an unsigned integer at least as large as an `unsigned short`. Figure 15.30 lists the system limits that affect shared memory.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum size in bytes of a shared memory segment	33,554,432	32,768	4,194,304	derived
minimum size in bytes of a shared memory segment	1	1	1	1
maximum number of shared memory segments, systemwide	192	4,096	32	128
maximum number of shared memory segments, per process	128	4,096	8	128

Figure 15.30 System limits that affect shared memory

The first function called is usually `shmget`, to obtain a shared memory identifier.

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

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

Returns: shared memory ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and whether a new segment is created or an existing segment is referenced. When a new segment is created, the following members of the `shmid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.

- `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are all set to 0.
- `shm_ctime` is set to the current time.
- `shm_segsz` is set to the *size* requested.

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up this size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use. If a new segment is being created (typically by the server), we must specify its *size*. If we are referencing an existing segment (a client), we can specify *size* as 0. When a new segment is created, the contents of the segment are initialized with zeros.

The `shmctl` function is the catchall for various shared memory operations.

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

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

Returns: 0 if OK, –1 on error

The *cmd* argument specifies one of the following five commands to be performed, on the segment specified by *shmid*.

`IPC_STAT` Fetch the `shmid_ds` structure for this segment, storing it in the structure pointed to by *buf*.

`IPC_SET` Set the following three fields from the structure pointed to by *buf* in the `shmid_ds` structure associated with this shared memory segment: `shm_perm.uid`, `shm_perm.gid`, and `shm_perm.mode`. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.

`IPC_RMID` Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the `shm_nattch` field in the `shmid_ds` structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that `shmat` can no longer attach the segment. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

`SHM_LOCK` Lock the shared memory segment in memory. This command can be executed only by the superuser.

`SHM_UNLOCK` Unlock the shared memory segment. This command can be executed only by the superuser.

Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat`.

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

void *shmat(int shmid, const void *addr, int flag);
```

Returns: pointer to shared memory segment if OK, –1 on error

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the `SHM_RND` bit is specified in *flag*.

- If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
- If *addr* is nonzero and `SHM_RND` is not specified, the segment is attached at the address given by *addr*.
- If *addr* is nonzero and `SHM_RND` is specified, the segment is attached at the address given by (*addr* – (*addr* modulus `SHMLBA`)). The `SHM_RND` command stands for "round." `SHMLBA` stands for "low boundary address multiple" and is always a power of 2. What the arithmetic does is round the address down to the next multiple of `SHMLBA`.

Unless we plan to run the application on only a single type of hardware (which is highly unlikely today), we should not specify the address where the segment is to be attached. Instead, we should specify an *addr* of 0 and let the system choose the address.

If the `SHM_RDONLY` bit is specified in *flag*, the segment is attached as read-only. Otherwise, the segment is attached as read–write.

The value returned by `shmat` is the address at which the segment is attached, or –1 if an error occurred. If `shmat` succeeds, the kernel will increment the `shm_nattch` counter in the `shmid_ds` structure associated with the shared memory segment.

When we're done with a shared memory segment, we call `shmdt` to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling `shmctl` with a command of `IPC_RMID`.

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

int shmdt(const void *addr);
```

Returns: 0 if OK, –1 on error

The *addr* argument is the value that was returned by a previous call to `shmat`. If successful, `shmdt` will decrement the `shm_nattch` counter in the associated `shmid_ds` structure.

Example

Where a kernel places shared memory segments that are attached with an address of 0 is highly system dependent. Figure 15.31 shows a program that prints some information on where one particular system places various types of data.

```
#include "apue.h"
#include <sys/shm.h>

#define ARRAY_SIZE  40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    0600    /* user read/write */

char    array[ARRAY_SIZE];  /* uninitialized data = bss */

int
main(void)
{
    int     shmid;
    char    *ptr, *shmptr;

    printf("array[] from %p to %p\n", (void *)&array[0],
      (void *)&array[ARRAY_SIZE]);
    printf("stack around %p\n", (void *)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %p to %p\n", (void *)ptr,
      (void *)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
    printf("shared memory attached from %p to %p\n", (void *)shmptr,
      (void *)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

Figure 15.31 Print where various types of data are stored

Running this program on a 64-bit Intel-based Linux system gives us the following output:

```
$ ./a.out
array[] from 0x6020c0 to 0x60bd00
stack around 0x7fff957b146c
malloced from 0x9e3010 to 0x9fb6b0
shared memory attached from 0x7fba578ab000 to 0x7fba578c36a0
```

Figure 15.32 shows a picture of this, similar to what we said was a typical memory layout in Figure 7.6. Note that the shared memory segment is placed well below the stack. □

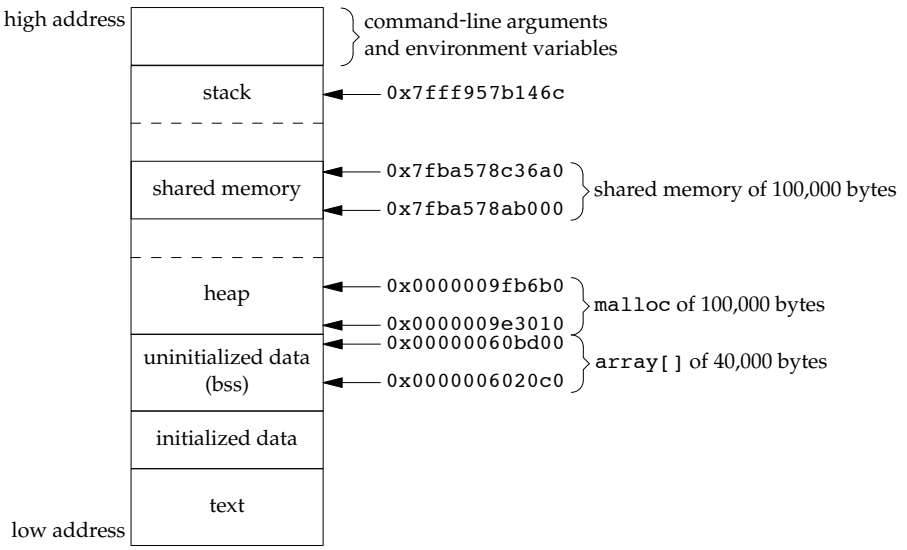

Figure 15.32 Memory layout on an Intel-based Linux system

Recall that the `mmap` function (Section 14.8) can be used to map portions of a file into the address space of a process. This is conceptually similar to attaching a shared memory segment using the `shmat` XSI IPC function. The main difference is that the memory segment mapped with `mmap` is backed by a file, whereas no file is associated with an XSI shared memory segment.

Example—Memory Mapping of `/dev/zero`

Shared memory can be used between unrelated processes. But if the processes are related, some implementations provide a different technique.

> The following technique works on FreeBSD 8.0, Linux 3.2.0, and Solaris 10. Mac OS X 10.6.8 currently doesn't support the mapping of character devices into the address space of a process.

The device `/dev/zero` is an infinite source of 0 bytes when read. This device also accepts any data that is written to it, ignoring the data. Our interest in this device for IPC arises from its special properties when it is memory mapped.

- An unnamed memory region is created whose size is the second argument to `mmap`, rounded up to the nearest page size on the system.
- The memory region is initialized to 0.
- Multiple processes can share this region if a common ancestor specifies the `MAP_SHARED` flag to `mmap`.

The program in Figure 15.33 is an example that uses this special device.

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE        sizeof(long)    /* size of shared memory area */

static int
update(long *ptr)
{
    return((*ptr)++);   /* return value before increment */
}

int
main(void)
{
    int     fd, i, counter;
    pid_t   pid;
    void    *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {           /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {                        /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("child: expected %d, got %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Figure 15.33 IPC between parent and child using memory mapped I/O of `/dev/zero`

The program opens the /dev/zero device and calls mmap, specifying a size of a long integer. Note that once the region is mapped, we can close the device. The process then creates a child. Since MAP_SHARED was specified in the call to mmap, writes to the memory-mapped region by one process are seen by the other process. (If we had specified MAP_PRIVATE instead, this example wouldn't work.)

The parent and the child then alternate running, incrementing a long integer in the shared memory-mapped region, using the synchronization functions from Section 8.9. The memory-mapped region is initialized to 0 by mmap. The parent increments it to 1, then the child increments it to 2, then the parent increments it to 3, and so on. Note that we have to use parentheses when we increment the value of the long integer in the update function, since we are incrementing the value and not the pointer.

The advantage of using /dev/zero in the manner that we've shown is that an actual file need not exist before we call mmap to create the mapped region. Mapping /dev/zero automatically creates a mapped region of the specified size. The disadvantage of this technique is that it works only between related processes. With related processes, however, it is probably simpler and more efficient to use threads (Chapters 11 and 12). Note that no matter which technique is used, we still need to synchronize access to the shared data. □

Example — Anonymous Memory Mapping

Many implementations provide anonymous memory mapping, a facility similar to the /dev/zero feature. To use this facility, we specify the MAP_ANON flag to mmap and specify the file descriptor as –1. The resulting region is anonymous (since it's not associated with a pathname through a file descriptor) and creates a memory region that can be shared with descendant processes.

> The anonymous memory-mapping facility is supported by all four platforms discussed in this text. Note, however, that Linux defines the MAP_ANONYMOUS flag for this facility, but defines the MAP_ANON flag to be the same value for improved application portability.

To modify the program in Figure 15.33 to use this facility, we make three changes: (a) remove the open of /dev/zero, (b) remove the close of fd, and (c) change the call to mmap to the following:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

In this call, we specify the MAP_ANON flag and set the file descriptor to –1. The rest of the program from Figure 15.33 is unchanged. □

The last two examples illustrate sharing memory among multiple related processes. If shared memory is required between unrelated processes, there are two alternatives. Applications can use the XSI shared memory functions, or they can use mmap to map the same file into their address spaces using the MAP_SHARED flag.

15.10 POSIX Semaphores

The POSIX semaphore mechanism is one of three IPC mechanisms that originated with the real-time extensions to POSIX.1. The Single UNIX Specification placed the three mechanisms (message queues, semaphores, and shared memory) in option classes. Prior to SUSv4, the POSIX semaphore interfaces were included in the semaphores option. In SUSv4, these interfaces were moved to the base specification, but the message queue and shared memory interfaces remained optional.

The POSIX semaphore interfaces were meant to address several deficiencies with the XSI semaphore interfaces:

- The POSIX semaphore interfaces allow for higher-performance implementations compared to XSI semaphores.
- The POSIX semaphore interfaces are simpler to use: there are no semaphore sets, and several of the interfaces are patterned after familiar file system operations. Although there is no requirement that they be implemented in the file system, some systems do take this approach.
- The POSIX semaphores behave more gracefully when removed. Recall that when an XSI semaphore is removed, operations using the same semaphore identifier fail with `errno` set to `EIDRM`. With POSIX semaphores, operations continue to work normally until the last reference to the semaphore is released.

POSIX semaphores are available in two flavors: named and unnamed. They differ in how they are created and destroyed, but otherwise work the same. Unnamed semaphores exist in memory only and require that processes have access to the memory to be able to use the semaphores. This means they can be used only by threads in the same process or threads in different processes that have mapped the same memory extent into their address spaces. Named semaphores, in contrast, are accessed by name and can be used by threads in any processes that know their names.

To create a new named semaphore or use an existing one, we call the `sem_open` function.

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode,
                unsigned int value */ );
```

Returns: Pointer to semaphore if OK, `SEM_FAILED` on error

When using an existing named semaphore, we specify only two arguments: the name of the semaphore and a zero value for the *oflag* argument. When the *oflag* argument has the `O_CREAT` flag set, we create a new named semaphore if it does not yet exist. If it already exists, it is opened for use, but no additional initialization takes place.

When we specify the `O_CREAT` flag, we need to provide two additional arguments. The *mode* argument specifies who can access the semaphore. It can take on the same values as the permission bits for opening a file: user-read, user-write, user-execute, group-read, group-write, group-execute, other-read, other-write, and other-execute. The resulting permissions assigned to the semaphore are modified by the caller's file

creation mask (Sections 4.5 and 4.8). Note, however, that only read and write access matter, but the interfaces don't allow us to specify the mode when we open an existing semaphore. Implementations usually open semaphores for both reading and writing.

The *value* argument is used to specify the initial value for the semaphore when we create it. It can take on any value from 0 to `SEM_VALUE_MAX` (Figure 2.9).

If we want to ensure that we are creating the semaphore, we can set the *oflag* argument to `O_CREAT|O_EXCL`. This will cause `sem_open` to fail if the semaphore already exists.

To promote portability, we must follow certain conventions when selecting a semaphore name.

- The first character in the name should be a slash (/). Although there is no requirement that an implementation of POSIX semaphores uses the file system, if the file system *is* used, we want to remove any ambiguity as to the starting point from which the name is interpreted.
- The name should contain no other slashes to avoid implementation-defined behavior. For example, if the file system is used, the names `/mysem` and `//mysem` would evaluate to the same filename, but if the implementation doesn't use the file system, the two names could be treated as different (consider what would happen if the implementation hashed the name to an integer value used to identify the semaphore).
- The maximum length of the semaphore name is implementation defined. The name should be no longer than `_POSIX_NAME_MAX` (Figure 2.8) characters, because this is the minimum acceptable limit to the maximum name length if the implementation does use the file system.

The `sem_open` function returns a semaphore pointer that we can pass to other semaphore functions when we want to operate on the semaphore. When we are done with the semaphore, we can call the `sem_close` function to release any resources associated with the semaphore.

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns: 0 if OK, –1 on error

If our process exits without having first called `sem_close`, the kernel will close any open semaphores automatically. Note that this doesn't affect the state of the semaphore value—if we have incremented its value, this doesn't change just because we exit. Similarly, if we call `sem_close`, the semaphore value is unaffected. There is no mechanism equivalent to the `SEM_UNDO` flag found with XSI semaphores.

To destroy a named semaphore, we can use the `sem_unlink` function.

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Returns: 0 if OK, –1 on error

The sem_unlink function removes the name of the semaphore. If there are no open references to the semaphore, then it is destroyed. Otherwise, destruction is deferred until the last open reference is closed.

Unlike with XSI semaphores, we can only adjust the value of a POSIX semaphore by one with a single function call. Decrementing the count is analogous to locking a binary semaphore or acquiring a resource associated with a counting semaphore.

> Note that there is no distinction of semaphore type with POSIX semaphores. Whether we use a binary semaphore or a counting semaphore depends on how we initialize and use the semaphore. If a semaphore only ever has a value of 0 or 1, then it is a binary semaphore. When a binary semaphore has a value of 1, we say that it is "unlocked;" when it has a value of 0, we say that it is "locked."

To decrement the value of a semaphore, we can use either the sem_wait or sem_trywait function.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);

int sem_wait(sem_t *sem);
```

Both return: 0 if OK, –1 on error

With the sem_wait function, we will block if the semaphore count is 0. We won't return until we have successfully decremented the semaphore count or are interrupted by a signal. We can use the sem_trywait function to avoid blocking. If the semaphore count is zero when we call sem_trywait, it will return –1 with errno set to EAGAIN instead of blocking.

A third alternative is to block for a bounded amount of time. We can use the sem_timedwait function for this purpose.

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict tsptr);
```

Returns: 0 if OK, –1 on error

The *tsptr* argument specifies the absolute time when we want to give up waiting for the semaphore. The timeout is based on the CLOCK_REALTIME clock (recall Figure 6.8). If the semaphore can be decremented immediately, then the value of the timeout doesn't matter—even though it might specify some time in the past, the attempt to decrement the semaphore will still succeed. If the timeout expires without being able to decrement the semaphore count, then sem_timedwait will return –1 with errno set to ETIMEDOUT.

To increment the value of a semaphore, we call the sem_post function. This is analogous to unlocking a binary semaphore or releasing a resource associated with a counting semaphore.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns: 0 if OK, –1 on error

If a process is blocked in a call to `sem_wait` (or `sem_timedwait`) when we call `sem_post`, the process is awakened and the semaphore count that was just incremented by `sem_post` is decremented by `sem_wait` (or `sem_timedwait`).

When we want to use POSIX semaphores within a single process, it is easier to use unnamed semaphores. This only changes the way we create and destroy the semaphore. To create an unnamed semaphore, we call the `sem_init` function.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Returns: 0 if OK, –1 on error

The *pshared* argument indicates if we plan to use the semaphore with multiple processes. If so, we set it to a nonzero value. The *value* argument specifies the initial value of the semaphore.

Instead of returning a pointer to the semaphore like `sem_open` does, we need to declare a variable of type `sem_t` and pass its address to `sem_init` for initialization. If we plan to use the semaphore between two processes, we need to ensure that the *sem* argument points into the memory extent that is shared between the processes.

When we are done using the unnamed semaphore, we can discard it by calling the `sem_destroy` function.

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Returns: 0 if OK, –1 on error

After calling `sem_destroy`, we can't use any of the semaphore functions with *sem* unless we reinitialize it by calling `sem_init` again.

One other function is available to allow us to retrieve the value of a semaphore. We call the `sem_getvalue` function for this purpose.

```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict valp);
```

Returns: 0 if OK, –1 on error

On success, the integer pointed to by the *valp* argument will contain the value of the semaphore. Be aware, however, that the value of the semaphore can change by the time that we try to use the value we just read. Unless we use additional synchronization mechanisms to avoid this race, the `sem_getvalue` function is useful only for debugging.

The `sem_getvalue` function is not supported by Mac OS X 10.6.8.

Example

One of the motivations for introducing the POSIX semaphore interfaces was that they can be made to perform significantly better than the existing XSI semaphore interfaces. It is instructive to see if this goal was reached in existing systems, even though these systems were not designed to support real-time applications.

In Figure 15.34, we compare the performance of using XSI semaphores (without `SEM_UNDO`) and POSIX semaphores when 3 processes compete to allocate and release the semaphore 1,000,000 times on two platforms (Linux 3.2.0 and Solaris 10).

Operation	Solaris 10			Linux 3.2.0		
	User	System	Clock	User	System	Clock
XSI semaphores	11.85	15.85	27.91	0.33	5.93	7.33
POSIX semaphores	13.72	10.52	24.44	0.26	0.75	0.41

Figure 15.34 Timing comparison of semaphore implementations

In Figure 15.34, we can see that POSIX semaphores provide only a 12% improvement over XSI semaphores on Solaris, but on Linux the improvement is 94% (almost 18 times faster)! If we trace the programs, we find that the Linux implementation of POSIX semaphores maps the file into the process address space and performs individual semaphore operations without using system calls. □

Example

Recall from Figure 12.5 that the Single UNIX Specification doesn't define what happens when one thread locks a normal mutex and a different thread tries to unlock it, but that error-checking mutexes and recursive mutexes generate errors in this case. Because a binary semaphore can be used like a mutex, we can use a semaphore to create our own locking primitive to provide mutual exclusion.

Assuming we were to create our own lock that could be locked by one thread and unlocked by another, our lock structure might look like

```
struct slock {
  sem_t *semp;
  char   name[_POSIX_NAME_MAX];
};
```

The program in Figure 15.35 shows an implementation of a semaphore-based mutual exclusion primitive.

```
#include "slock.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

struct slock *
```

```
s_alloc()
{
    struct slock *sp;
    static int cnt;

    if ((sp = malloc(sizeof(struct slock))) == NULL)
        return(NULL);
    do {
        snprintf(sp->name, sizeof(sp->name), "/%ld.%d", (long)getpid(),
          cnt++);
        sp->semp = sem_open(sp->name, O_CREAT|O_EXCL, S_IRWXU, 1);
    } while ((sp->semp == SEM_FAILED) && (errno == EEXIST));
    if (sp->semp == SEM_FAILED) {
        free(sp);
        return(NULL);
    }
    sem_unlink(sp->name);
    return(sp);
}

void
s_free(struct slock *sp)
{
    sem_close(sp->semp);
    free(sp);
}

int
s_lock(struct slock *sp)
{
    return(sem_wait(sp->semp));
}

int
s_trylock(struct slock *sp)
{
    return(sem_trywait(sp->semp));
}

int
s_unlock(struct slock *sp)
{
    return(sem_post(sp->semp));
}
```

Figure 15.35 Mutual exclusion using a POSIX semaphore

We create a name based on the process ID and a counter. We don't bother to protect the counter with a mutex, because if two racing threads call `s_alloc` at the same time and end up with the same name, using the `O_EXCL` flag in the call to `sem_open` will cause one to succeed and one to fail with `errno` set to `EEXIST`, so we just retry if this happens. Note that we unlink the semaphore after opening it. This destroys the name so that no other process can access it and simplifies cleanup when the process ends. □

15.11 Client–Server Properties

Let's detail some of the properties of clients and servers that are affected by the various types of IPC used between them. The simplest type of relationship is to have the client `fork` and `exec` the desired server. Two half-duplex pipes can be created before the `fork` to allow data to be transferred in both directions. Figure 15.16 is an example of this arrangement. The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID. (Recall from Section 8.10 that the real user ID and real group ID don't change across an `exec`.)

With this arrangement, we can build an *open server*. (We show an implementation of this client–server mechanism in Section 17.5.) It opens files for the client instead of the client calling the `open` function. This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions. We assume that the server is a set-user-ID program, giving it additional permissions (root permission, perhaps). The server uses the real user ID of the client to determine whether to give it access to the requested file. This way, we can build a server that allows certain users permissions that they don't normally have.

In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent. Although this works fine for regular files, it can't be used for special device files, for example. We would like to be able to have the server open the requested file and pass back the file descriptor. Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent (unless special programming techniques are used, which we cover in Chapter 17).

We showed the next type of server in Figure 15.23. The server is a daemon process that is contacted using some form of IPC by all clients. We can't use pipes for this type of client–server arrangement. A form of named IPC is required, such as FIFOs or message queues. With FIFOs, we saw that an individual per-client FIFO is also required if the server is to send data back to the client. If the client–server application sends data only from the client to the server, a single well-known FIFO suffices. (The System V line printer spooler used this form of client–server arrangement. The client was the `lp`(1) command, and the server was the `lpsched` daemon process. A single FIFO was used, since the flow of data was only from the client to the server. Nothing was sent back to the client.)

Multiple possibilities exist with message queues.

1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient. For example, the clients can send their requests with a type field of 1. Included in the request must be the client's process ID. The server then sends the response with the type field set to the client's process ID. The server receives only the messages with a type field of 1 (the fourth argument for `msgrcv`), and the clients receive only the messages with a type field equal to their process IDs.

2. Alternatively, an individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue

with a key of IPC_PRIVATE. The server also has its own queue, with a key or identifier known to all clients. The client sends its first request to the server's well-known queue, and this request must contain the message queue ID of the client's queue. The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

One problem with this technique is that each client-specific queue usually has only a single message on it: a request for the server or a response for a client. This seems wasteful of a limited systemwide resource (a message queue), and a FIFO can be used instead. Another problem is that the server has to read messages from multiple queues. Neither select nor poll works with message queues.

Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).

The problem with this type of client–server relationship (the client and the server being unrelated processes) is for the server to identify the client accurately. Unless the server is performing a nonprivileged operation, it is essential that the server know who the client is. This is required, for example, if the server is a set-user-ID program. Although all these forms of IPC go through the kernel, there is no facility provided by them to have the kernel identify the sender.

With message queues, if a single queue is used between the client and the server (so that only a single message is on the queue at a time, for example), the msg_lspid of the queue contains the process ID of the other process. But when writing the server, we want the effective user ID of the client, not its process ID. There is no portable way to obtain the effective user ID, given the process ID. (Naturally, the kernel maintains both values in the process table entry, but other than rummaging around through the kernel's memory, we can't obtain one, given the other.)

We'll use the following technique in Section 17.2 to allow the server to identify the client. The same technique can be used with FIFOs, message queues, semaphores, and shared memory. For the following description, assume that FIFOs are being used, as in Figure 15.23. The client must create its own FIFO and set the file access permissions of the FIFO so that only user-read and user-write are on. We assume that the server has superuser privileges (or else it probably wouldn't care about the client's true identity), so the server can still read and write to this FIFO. When the server receives the client's first request on the server's well-known FIFO (which must contain the identity of the client-specific FIFO), the server calls either stat or fstat on the client-specific FIFO. The server assumes that the effective user ID of the client is the owner of the FIFO (the st_uid field of the stat structure). The server verifies that only the user-read and user-write permissions are enabled. As another check, the server should look at the three times associated with the FIFO (the st_atime, st_mtime, and st_ctime fields of the stat structure) to verify that they are recent (no older than 15 or 30 seconds, for example). If a malicious client can create a FIFO with someone else as the owner and set the file's permission bits to user-read and user-write only, then the system has other fundamental security problems.

To use this technique with XSI IPC, recall that the `ipc_perm` structure associated with each message queue, semaphore, and shared memory segment identifies the creator of the IPC structure (the `cuid` and `cgid` fields). As with the example using FIFOs, the server should require the client to create the IPC structure and have the client set the access permissions to user-read and user-write only. The times associated with the IPC structure should also be verified by the server to be recent (since these IPC structures hang around until explicitly deleted).

We'll see in Section 17.3 that a far better way of doing this authentication is for the kernel to provide the effective user ID and effective group ID of the client. This is done by the socket subsystem when file descriptors are passed between processes.

15.12 Summary

We've detailed numerous forms of interprocess communication: pipes, named pipes (FIFOs), the three forms of IPC commonly called XSI IPC (message queues, semaphores, and shared memory), and an alternative semaphore mechanism provided by POSIX. Semaphores are really a synchronization primitive, not true IPC, and are often used to synchronize access to a shared resource, such as a shared memory segment. With pipes, we looked at the implementation of the `popen` function, at coprocesses, and at the pitfalls that can be encountered with the standard I/O library's buffering.

After comparing the timing of message queues versus full-duplex pipes, and semaphores versus record locking, we can make the following recommendations: learn pipes and FIFOs, since these two basic techniques can still be used effectively in numerous applications. Avoid using message queues and semaphores in any new applications. Full-duplex pipes and record locking should be considered instead, as they are far simpler. Shared memory still has its use, although the same functionality can be provided through the use of the `mmap` function (Section 14.8).

In the next chapter, we will look at network IPC, which can allow processes to communicate across machine boundaries.

Exercises

15.1 In the program shown in Figure 15.6, remove the `close` right before the `waitpid` at the end of the parent code. Explain what happens.

15.2 In the program in Figure 15.6, remove the `waitpid` at the end of the parent code. Explain what happens.

15.3 What happens if the argument to `popen` is a nonexistent command? Write a small program to test this.

15.4 In the program shown in Figure 15.18, remove the signal handler, execute the program, and then terminate the child. After entering a line of input, how can you tell that the parent was terminated by `SIGPIPE`?

15.5 In the program in Figure 15.18, use the standard I/O library for reading and writing the pipes instead of `read` and `write`.

15.6 The Rationale for POSIX.1 gives as one of the reasons for adding the `waitpid` function that most pre-POSIX.1 systems can't handle the following:

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

What happens in this code if `waitpid` isn't available and `wait` is used instead?

15.7 Explain how `select` and `poll` handle an input descriptor that is a pipe, when the pipe is closed by the writer. To determine the answer, write two small test programs: one using `select` and one using `poll`.

Redo this exercise, looking at an output descriptor that is a pipe, when the read end is closed.

15.8 What happens if the *cmdstring* executed by `popen` with a *type* of `"r"` writes to its standard error?

15.9 Since `popen` invokes a shell to execute its *cmdstring* argument, what happens when `cmdstring` terminates? (Hint: Draw all the processes involved.)

15.10 POSIX.1 specifically states that opening a FIFO for read–write is undefined. Although most UNIX systems allow this, show another method for opening a FIFO for both reading and writing, without blocking.

15.11 Unless a file contains sensitive or confidential data, allowing other users to read the file causes no harm. (It is usually considered antisocial, however, to go snooping around in other people's files.) But what happens if a malicious process reads a message from a message queue that is being used by a server and several clients? What information does the malicious process need to know to read the message queue?

15.12 Write a program that does the following. Execute a loop five times: create a message queue, print the queue identifier, delete the message queue. Then execute the next loop five times: create a message queue with a key of `IPC_PRIVATE`, and place a message on the queue. After the program terminates, look at the message queues using `ipcs`(1). Explain what is happening with the queue identifiers.

15.13 Describe how to build a linked list of data objects in a shared memory segment. What would you store as the list pointers?

15.14 Draw a timeline of the program in Figure 15.33 showing the value of the variable `i` in both the parent and child, the value of the long integer in the shared memory region, and the value returned by the `update` function. Assume that the child runs first after the `fork`.

15.15 Redo the program in Figure 15.33 using the XSI shared memory functions from Section 15.9 instead of the shared memory-mapped region.

15.16 Redo the program in Figure 15.33 using the XSI semaphore functions from Section 15.8 to alternate between the parent and the child.

15.17 Redo the program in Figure 15.33 using advisory record locking to alternate between the parent and the child.

15.18 Redo the program in Figure 15.33 using the POSIX semaphore functions from Section 15.10 to alternate between the parent and the child.