# 12

# Advanced I/O

## 12.1 Introduction

This chapter covers numerous topics and functions that we lump under the term "advanced I/O." This includes nonblocking I/O, record locking, System V streams, I/O multiplexing (the `select` and `poll` functions), the `readv` and `writev` functions, and memory mapped I/O (`mmap`). We need to cover these topics before describing interprocess communication in Chapters 14 and 15, and many of the examples in later chapters.

## 12.2 Nonblocking I/O

In Section 10.5 we said that the system calls are divided into two categories: the "slow" ones, and all the others. The slow system calls are those that can block forever:

- reads from files that can block the caller forever, if data isn't present (pipes, terminal devices, and network devices),
- writes to these same files that can block forever, if the data can't be accepted immediately,
- opens of files block until some condition occurs (such as an open of a terminal device that waits until an attached modem answers the phone, or an open of a FIFO for writing-only when no other process has the FIFO open for reading),
- reads and writes of files that have mandatory record locking enabled,
- certain `ioctl` operations,
- some of the interprocess communication functions (Chapter 14).

We also said that system calls related to disk I/O are not considered slow, even though the read or write of a disk file can block the caller temporarily.

Nonblocking I/O lets us issue an I/O operation, such as an open, read, or write, and not have it block forever. If the operation cannot be completed, return is made immediately with an error noting that the operation would have blocked.

There are two ways to specify nonblocking I/O for a given descriptor.

1. If we call open to get the descriptor, we can specify the O_NONBLOCK flag (Section 3.3).
2. For a descriptor that is already open, we call fcntl to turn on the O_NONBLOCK file status flag (Section 3.13). Program 3.5 shows a function that we can call to turn on any of the file status flags for a descriptor.

Earlier versions of System V used the flag O_NDELAY to specify the nonblocking mode. These versions of System V returned a value of 0 from the read function if there wasn't any data to be read. Since this use of a return value of 0 overlapped with the normal Unix convention of 0 meaning the end of file, POSIX.1 chose to provide a nonblocking flag with a different name and different semantics. Indeed, with these older versions of System V we don't know when we get a return of 0 from read whether the call would have blocked, or if the end of file was encountered. We'll see that POSIX.1 requires that read return –1 with errno set to EAGAIN if there is no data to read from a nonblocking descriptor. SVR4 supports both the older O_NDELAY and the POSIX.1 O_NONBLOCK, but in this text we'll only use the POSIX.1 feature. The older O_NDELAY is for backward compatibility and should not be used in new applications.

4.3BSD provided the FNDELAY flag for fcntl, and its semantics were slightly different. Instead of just affecting the file status flags for the descriptor, the flags for either the terminal device or the socket were also changed to be nonblocking, affecting all users of the terminal or socket, not just the users sharing the same file table entry (4.3BSD nonblocking I/O only worked on terminals and sockets). Also, 4.3BSD returned EWOULDBLOCK if an operation on a nonblocking descriptor could not complete without blocking. 4.3+BSD provides the POSIX.1 O_NONBLOCK flag, but the semantics are similar to those for FNDELAY under 4.3BSD. A common use for nonblocking I/O is for dealing with a terminal device or a network connection, and these devices are normally used by one process at a time. This means that the change in the BSD semantics normally doesn't affect us. The different error return, EWOULDBLOCK, instead of the POSIX.1 EAGAIN, continues to be a portability difference that we must deal with. 4.3+BSD also supports FIFOs, and nonblocking I/O works with FIFOs too.

## Example

Let's look at an example of nonblocking I/O. Program 12.1 reads up to 100,000 bytes from the standard input and attempts to write it to the standard output. The standard output is first set nonblocking. The output is in a loop, with the results of each write being printed on the standard error. The function clr_fl is similar to the function set_fl that we showed in Program 3.5. This new function just clears one or more of the flag bits.

```
#include    <sys/types.h>
#include    <errno.h>
#include    <fcntl.h>
#include    "ourhdr.h"

char    buf[100000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    for (ptr = buf; ntowrite > 0; ) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */
    exit(0);
}
```

**Program 12.1** Large nonblocking `write`.

If the standard output is a regular file, we expect the `write` to be executed once.

```
$ ls -l /etc/termcap                                    print file size
-rw-rw-r--  1 root     133439 Oct 11  1990 /etc/termcap
$ a.out < /etc/termcap > temp.file                      try a regular file first
read 100000 bytes
nwrite = 100000, errno = 0                              a single write
$ ls -l temp.file                                       verify size of output file
-rw-rw-r--  1 stevens  100000 Nov 21 16:27 temp.file
```

But if the standard output is a terminal, we expect the `write` to return a partial count sometimes and an error at other times. This is what we see.

```
$ a.out < /etc/termcap 2>stderr.out          output to terminal
                                             lots of output to terminal ...
$ cat stderr.out
read 100000 bytes
nwrite = 8192, errno = 0
nwrite = 8192, errno = 0
nwrite = -1, errno = 11                      211 of these errors
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11                      658 of these errors
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11                      604 of these errors
. . .
nwrite = 4096, errno = 0
nwrite = -1, errno = 11                      1047 of these errors
. . .
nwrite = -1, errno = 11                      1046 of these errors
. . .
nwrite = 4096, errno = 0
                                             and so on ...
```

On this system the `errno` of 11 is `EAGAIN`. The terminal driver on this system always accepted 4096 or 8192 bytes at a time. On another system the first three `write`s returned 2005, 1822, and 1811, followed by 96 errors, followed by a `write` of 1846, and so on. How much data is accepted on each `write` is system dependent.

The behavior of this program under SVR4 is completely different from the preceding—when the output was to the terminal only a single `write` was needed to output the entire input file. Apparently the nonblocking mode makes no difference! A bigger input file was created and the program's buffer was increased. This behavior of the program (one `write` for the entire file) continued until the size of the input file was about 700,000 bytes. At that point every `write` returned the error `EAGAIN`. (The input file was never output to the terminal—the program just generated a continual stream of error messages.)

What's going on here is that the terminal driver in SVR4 is connected to the program through the stream I/O system. (We describe streams in detail in Section 12.4.) The streams system has its own buffers and is capable of accepting more data at a time from the program. The SVR4 behavior also depends on the type of terminal—hardwired terminal, console device, or a pseudo terminal. □

In this example the program issues thousands of `write` calls, when only around 20 are required to output the data. The rest just return an error. This type of loop, called *polling*, is a waste of CPU time on a multiuser system. In Section 12.5 we'll see that I/O multiplexing with a nonblocking descriptor is a more efficient way to do this.

We'll encounter nonblocking I/O in Chapter 17 when we output to a terminal device (a PostScript printer) and want to make certain we don't block on a `write`.

## 12.3 Record Locking

What happens when two people edit the same file at the same time? In most Unix systems the final state of the file corresponds to the last process that wrote the file. There are applications, however, such as a database system, when a process needs to be certain that it alone is writing to a file. To provide this capability for processes that need it, newer Unix systems provide record locking. (We develop a database library in Chapter 16 that uses record locking.)

*Record locking* is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file, while the first process is reading or modifying that portion of the file. Under Unix the adjective "record" is a misnomer, since the Unix kernel does not have a notion of records in a file. A better term is "range locking," since it is a range of a file (possibly the entire file) that is locked.

### History

Figure 12.1 shows the different forms of record locking provided by various Unix systems.

| System | Advisory | Mandatory | `fcntl` | `lockf` | `flock` |
|---|---|---|---|---|---|
| POSIX.1 | • | | • | | |
| XPG3 | • | | • | | |
| SVR2 | • | | • | • | |
| SVR3, SVR4 | • | • | • | • | |
| 4.3BSD | • | | | | • |
| 4.3BSD Reno | • | | • | | • |

**Figure 12.1** Forms of record locking supported by various Unix systems.

We describe the difference between advisory locking and mandatory locking later in this section. As shown in this figure, POSIX.1 selected the System V style of record locking, which is based on the `fcntl` function. This style is also supported by the latest version of 4.3BSD Reno.

Earlier Berkeley releases supported only the BSD `flock` function. This function locks only entire files, not regions of a file. But the POSIX.1 `fcntl` function can lock any region of a file, from the entire file down to a single byte within the file.

In this text we describe only the POSIX.1 `fcntl` locking. The System V `lockf` function is just an interface to the `fcntl` function.

> Record locking was originally added to Version 7 in 1980 by John Bass. The system call entry into the kernel was a function named `locking`. This function provided mandatory record locking and propagated through many vendor's versions of System III. Xenix systems picked up this function, and SVR4 still supports it in its Xenix compatibility library.
>
> SVR2 was the first release of System V to support the `fcntl` style of record locking, in 1984.

### fcntl Record Locking

Let's repeat the prototype for the fcntl function from Section 3.13.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );
```

Returns: depends on *cmd* if OK (see below), –1 on error

For record locking *cmd* is F_GETLK, F_SETLK, or F_SETLKW. The third argument (which we'll call *flockptr*) is a pointer to an flock structure.

```
struct flock {
  short  l_type;    /* F_RDLCK, F_WRLCK, or F_UNLCK */
  off_t  l_start;   /* offset in bytes, relative to l_whence */
  short  l_whence;  /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_len;     /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;     /* returned with F_GETLK */
};
```

This structure describes

- the type of lock desired: F_RDLCK (a shared read lock), F_WRLCK (an exclusive write lock), or F_UNLCK (unlocking a region),
- the starting byte offset of the region being locked or unlocked (l_start and l_whence), and
- the size of the region (l_len).

There are numerous rules about the specification of the region to be locked or unlocked.

- The two elements that specify the starting offset of the region are similar to the last two arguments of the lseek function (Section 3.6). Indeed, the l_whence member is specified as SEEK_SET, SEEK_CUR, or SEEK_END.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If the l_len is 0, it means that the lock extends to the largest possible offset of the file. This allows us to lock a region starting anywhere in the file, up through and including any data that is appended to the file. (We don't have to try to guess how many bytes might be appended to the file.)
- To lock the entire file, we set l_start and l_whence to point to the beginning of the file, and specify a length (l_len) of 0. (There are several ways to specify the beginning of the file, but most applications specify l_start as 0 and l_whence as SEEK_SET.)

We mentioned two types of locks: a shared read lock (`l_type` of `L_RDLCK`) and an exclusive write lock (`L_WRLCK`). The basic rule is that any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. Furthermore, if there are one or more read locks on a byte, there can't be any write locks on that byte, and if there is an exclusive write lock on a byte, there can't be any read locks on that byte. We show this compatibility rule in Figure 12.2.

| | | request for | |
|---|---|---|---|
| | | read lock | write lock |
| region currently has | no locks | OK | OK |
| | one or more read locks | OK | denied |
| | one write lock | denied | denied |

**Figure 12.2** Compatibility between different lock types.

To obtain a read lock the descriptor must be open for reading, and to obtain a write lock the descriptor must be open for writing.

We can now describe the three different commands for the `fcntl` function.

`F_GETLK` Determine if the lock described by *flockptr* is blocked by some other lock. If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by *flockptr*. If no lock exists that would prevent ours from being created, the structure pointed to by *flockptr* is left unchanged except for the `l_type` member, which is set to `F_UNLCK`.

`F_SETLK` Set the lock described by *flockptr*. If we are trying to obtain a read lock (`l_type` of `F_RDLCK`) or a write lock (`l_type` of `F_WRLCK`) and the compatibility rule prevents the system from giving us the lock (Figure 12.2), `fcntl` returns immediately with `errno` set to either `EACCES` or `EAGAIN`.

> SVR2 returned `EACCES`, but the manual page warned that in the future `EAGAIN` would be returned. SVR4 continues this tradition (returning `EACCES` with the same warning about the future). 4.3+BSD returns `EAGAIN`. POSIX.1 allows either error to be returned.

This command is also used to clear the lock described by *flockptr* (`l_type` of `F_UNLCK`).

`F_SETLKW` This command is a blocking version of `F_SETLK`. (The `W` in the command name means "wait.") If the requested read lock or write lock cannot be granted because another process currently has some part of the requested region locked, the calling process is put to sleep. This sleep is interrupted if a signal is caught.

Be aware that testing for a lock with F_GETLK and then trying to obtain that lock with F_SETLK or F_SETLKW is not an atomic operation. We have no guarantee that between the two fcntl calls some other process won't come in and obtain the same lock. If we don't want to block while waiting for a lock to become available to us, we must handle the possible error returns from F_SETLK.

When setting or releasing a lock on a file, the system combines or splits adjacent areas as required. For example, if we set a read lock on bytes 0 through 99 and then set a write lock on bytes 0 through 49, we then have two locked regions: bytes 0 through 49 (write locked) and bytes 50 through 99 (read locked). Similarly, if we lock bytes 100 through 199 and then unlock byte 150, the kernel still maintains the locks on bytes 100 through 149, and bytes 151 through 199.

### Example—Requesting and Releasing A Lock

To save ourselves from having to allocate an flock structure and fill in all the elements each time, the function lock_reg in Program 12.2 handles all these details.

```
#include    <sys/types.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;      /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;   /* byte offset, relative to l_whence */
    lock.l_whence = whence;  /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;        /* #bytes (0 means to EOF) */

    return( fcntl(fd, cmd, &lock) );
}
```

**Program 12.2** Function to lock or unlock a region of a file.

Since most locking calls are to lock or unlock a region (the command F_GETLK is rarely used) we normally use one of the following five macros, which are defined in ourhdr.h (Appendix B).

```
#define read_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
#define readw_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
#define write_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
#define writew_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
#define un_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)
```

We have purposely defined the first three arguments to these macros in the same order as the `lseek` function. □

### Example—Testing for A Lock

Program 12.3 defines the function `lock_test` that we'll use to test for a lock.

```
#include    <sys/types.h>
#include    <fcntl.h>
#include    "ourhdr.h"

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);      /* false, region is not locked by another proc */
    return(lock.l_pid); /* true, return pid of lock owner */
}
```

**Program 12.3** Function to test for a locking condition.

If a lock exists that would block the request specified by the arguments, this function returns the process ID of the process holding the lock. Otherwise the function returns 0 (false). We normally call this function from the following two macros (defined in `ourhdr.h`).

```
#define is_readlock(fd, offset, whence, len) \
            lock_test(fd, F_RDLCK, offset, whence, len)
#define is_writelock(fd, offset, whence, len) \
            lock_test(fd, F_WRLCK, offset, whence, len)
```

□

### Example—Deadlock

Deadlock occurs when two processes are each waiting for a resource that the other has locked. The potential for deadlock exists if a process that controls a locked region is put to sleep when it tries to lock another region that is controlled by a different process.

Program 12.4 shows an example of deadlock. The child locks byte 0 and the parent locks byte 1. Then each tries to lock the other's already locked byte. We use the parent–child synchronization routines from Section 8.8 (`TELL_xxx` and `WAIT_xxx`) so that each process can wait for the other to obtain its lock. Running Program 12.4 gives us

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

static void lockabyte(const char *, int, off_t);

int
main(void)
{
    int     fd;
    pid_t   pid;

        /* Create a file and write two bytes to it */
    if ( (fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");

    TELL_WAIT();
    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid == 0) {            /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);

    } else {                        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);

    printf("%s: got the lock, byte %d\n", name, offset);
}
```

**Program 12.4** Example of deadlock detection.

```
$ a.out
child: got the lock, byte 0
parent: got the lock, byte 1
child: fcntl error: Deadlock situation detected/avoided
parent: got the lock, byte 0
```

When a deadlock is detected, the kernel has to choose one process to receive the error return. In this example the child was chosen, but this is an implementation detail. When this program was run on another system, half the time the child received the error and half the time the parent received the error. □

### Implied Inheritance and Release of Locks

There are three rules that govern the automatic inheritance and release of record locks.

1. Locks are associated with a process and a file. This has two implications. The first is obvious: when a process terminates all its locks are released. The second is far from obvious: whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released. This means that if we do the following four steps

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = dup(fd1);
   close(fd2);
   ```

   after the `close(fd2)` the lock that was obtained on `fd1` is released. The same thing would happen if we replaced the `dup` with `open`, as in

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = open(pathname, ...)
   close(fd2);
   ```

   to open the same file on another descriptor.

2. Locks are never inherited by the child across a `fork`. This means that if a process obtains a lock and then calls `fork`, the child is considered "another process" with regard to the lock that was obtained by the parent. The child has to call `fcntl` to obtain its own locks on any descriptors that were inherited across the `fork`. This makes sense, because locks are meant to prevent multiple processes from writing to the same file at the same time. If the child inherited locks across a `fork`, both the parent and child could write to the same file at the same time.

3. Locks may be inherited by a new program across an `exec`.

   We have to say *may* here because POSIX.1 doesn't require this. Under SVR4 and 4.3+BSD, however, locks are inherited across an `exec`.

### 4.3+BSD Implementation

Let's take a brief look at the data structures used in the 4.3+BSD implementation. This should help clarify rule 1, that locks are associated with a process and a file.

Consider a process that executes the following statements (ignoring error returns):

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);     /* parent write locks byte 0 */
if (fork() > 0) {      /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
    pause();
} else {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
    pause();
}
```

Figure 12.3 shows the resulting data structures after both the parent and child have paused.

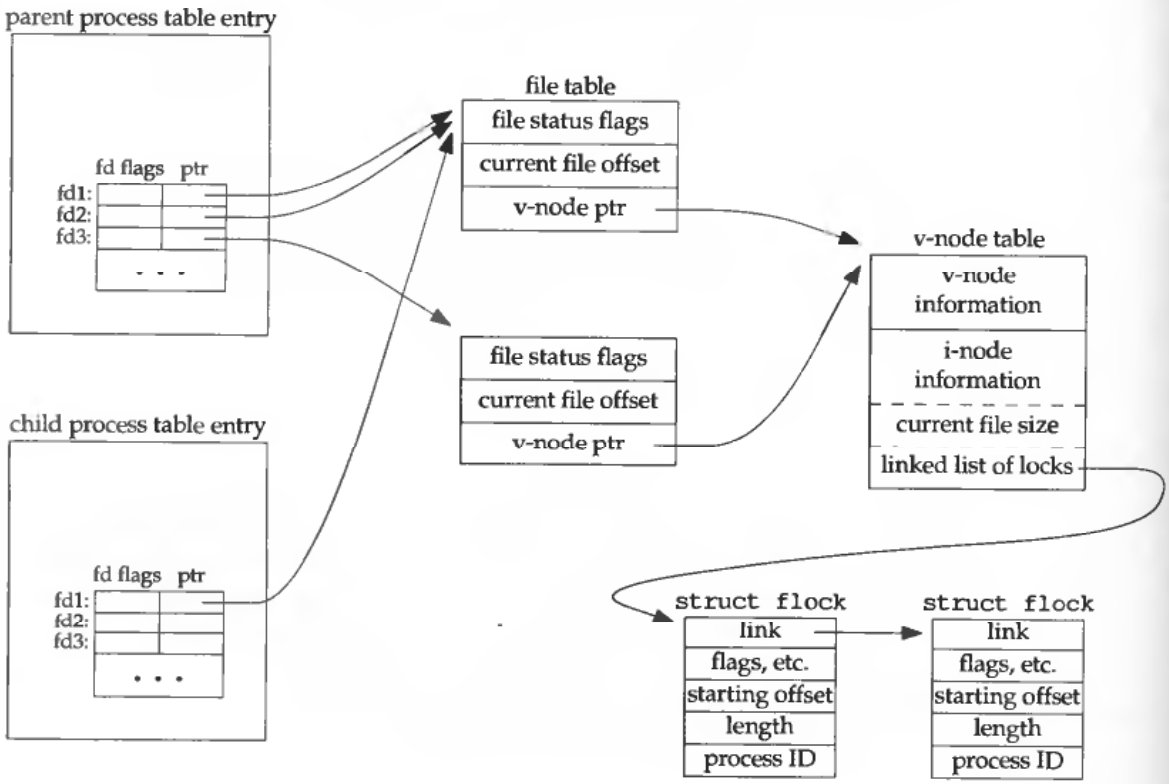


**Figure 12.3** The 4.3+BSD data structures for record locking.

We've shown the data structures that result from the `open`, `fork`, and `dup` earlier (Figures 3.4 and 8.1). What is new are the `flock` structures that are linked together from the i-node structure. Notice that each `flock` structure describes one locked region

(defined by an offset and length) for a given process. We show two of these structures, one for the parent's call to `write_lock` and one for the child's call to `read_lock`. Each structure contains the corresponding process ID.

In the parent, closing any one of `fd1`, `fd2`, or `fd3` causes the parent's lock to be released. When any one of these three descriptors is closed, the kernel goes through the linked list of locks for the corresponding i-node, and releases the locks held by the calling process. The kernel can't tell (and doesn't care) which descriptor of the three the parent's lock was obtained on.

### Example

Advisory locks can be used by a daemon to assure that only one copy of the daemon is running. When started, many daemons write their process ID to a file. This process ID can be used when it is time to shut down the system. The way to prevent multiple copies of the daemon from running is to have the daemon obtain a lock on its process ID file when it starts. If it holds the lock for as long as it runs, no more copies of itself will be started. Program 12.5 implements this technique.

We specifically truncate the file in case the file previously contained a process ID that was longer than the current process ID. If the previous contents of the file were `12345\n` and the new process ID was 654, we want the file to contain just the four bytes `654\n`, and not `654\n5\n`. Note that we call `ftruncate` after we get the lock—we cannot specify `O_TRUNC` in the call to `open`, because that could empty the file even though it was locked by another copy of the daemon. (We could use `O_TRUNC` if we were using mandatory locking, instead of advisory locking. We discuss mandatory locking later in this section.)

In this example we also set the close-on-exec flag for the descriptor. This is because daemons often `fork` and `exec` other processes, and there is no need for this file to remain open in another process. □

### Example

Use caution when locking or unlocking relative to the end of file. Most implementations convert an `l_whence` value of `SEEK_CUR` or `SEEK_END` into an absolute file offset, using `l_start` and the file's current position or current length. Often, however, we need to specify a lock relative to the file's current position or current length, because we can't call `lseek` to obtain the current file offset, since we don't have a lock on the file. (There's a chance another process could change the file's length between the call to `lseek` and the lock call.)

Program 12.6 writes a large file, one byte at a time. Each time around the loop it locks from the current end of file through any future end of file (the final argument, the length of 0), and writes one byte. It then unlocks from the current end of file through any future end of file, and writes another byte. If the system kept track of locks using the notation ("from the current end of file through any future end of file") this should work. But if the system converts this notation into absolute file offsets, we could have a problem. Running this program under SVR4 shows that we do have a problem.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <errno.h>
#include    <fcntl.h>
#include    "ourhdr.h"

#define PIDFILE     "daemon.pid"

int
main(void)
{
    int     fd, val;
    char    buf[10];

    if ( (fd = open(PIDFILE, O_WRONLY | O_CREAT, FILE_MODE)) < 0)
        err_sys("open error");

            /* try and set a write lock on the entire file */
    if (write_lock(fd, 0, SEEK_SET, 0) < 0) {
        if (errno == EACCES || errno == EAGAIN)
            exit(0);    /* gracefully exit, daemon is already running */
        else
            err_sys("write_lock error");
    }

            /* truncate to zero length, now that we have the lock */
    if (ftruncate(fd, 0) < 0)
        err_sys("ftruncate error");

            /* and write our process ID */
    sprintf(buf, "%d\n", getpid());
    if (write(fd, buf, strlen(buf)) != strlen(buf))
        err_sys("write error");

            /* set close-on-exec flag for descriptor */
    if ( (val = fcntl(fd, F_GETFD, 0)) < 0)
        err_sys("fcntl F_GETFD error");
    val |= FD_CLOEXEC;
    if (fcntl(fd, F_SETFD, val) < 0)
        err_sys("fcntl F_SETFD error");

    /* leave file open until we terminate: lock will be held */

    /* do whatever ... */

    exit(0);
}
```

**Program 12.5** Daemon start-up code to prevent multiple copies of itself from running.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
main(void)
{
    int     i, fd;

    if ( (fd = open("temp.lock", O_RDWR | O_CREAT | O_TRUNC,
                                                FILE_MODE)) < 0)
        err_sys("open error");

    for (i = 0; i < 1000000; i++) { /* try to write 2 Mbytes */
                /* lock from current EOF to EOF */
        if (writew_lock(fd, 0, SEEK_END, 0) < 0)
            err_sys("writew_lock error");

        if (write(fd, &fd, 1) != 1)
            err_sys("write error");

        if (un_lock(fd, 0, SEEK_END, 0) < 0)
            err_sys("un_lock error");

        if (write(fd, &fd, 1) != 1)
            err_sys("write error");
    }
    exit(0);
}
```

**Program 12.6** Program displaying problems with locking relative to end of file.

```
$ a.out
writew_lock error: No record locks available
$ ls -l temp.lock
-rw-r--r--   1 stevens  other    592 Nov  1 04:41 temp.lock
```

(The error ENOLCK is returned by the kernel. It indicates that the kernel's lock table is full.) It is instructive to see what the system is doing. Figure 12.4 shows the state of the file after the first call to `writew_lock` and the first call to `write`.

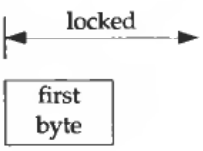


**Figure 12.4** State of file after first `writew_lock` and first `write`.

We show the locked region extending past the byte that we wrote, since we specified "through any future end of file" in the call to `writew_lock`.

We then call `un_lock`. This unlocks from the current end of file through any future end of file, which moves the right end of the arrow in Figure 12.4 back to the end of the first byte. We then write the second byte to the file. Figure 12.5 shows the state of the file after calling `un_lock` and the `write` that follows.

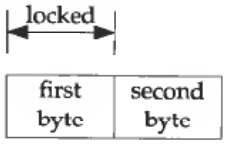


**Figure 12.5** State of file after `un_lock` and second `write`.

After going through the `for` loop one more time, we have written four bytes to the file. Figure 12.6 shows the state of the file and its locks.

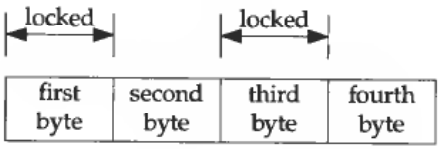


**Figure 12.6** State of file and locks after second time through `for` loop.

What happens when we run Program 12.6 is that this form of file (every other byte locked) continues until the kernel runs out of lock structures for the process. When this happens, `fcntl` returns an error of `ENOLCK`.

Since we know how many bytes we are writing to the file each time, we can correct this problem by replacing the second argument to `un_lock` (the `l_start` specifier) with the negative of the number of bytes (–1 in this case). This causes each lock to be removed by `un_lock`.

> This problem actually occurred to the author when developing the `_db_writedat` and `_db_writeidx` functions in Section 16.7. A slightly different way around the problem is shown there. □

## Advisory versus Mandatory Locking

Consider a library of database access routines. If all the functions in the library handle record locking in a consistent way, then we say that any set of processes that are using these functions to access a database are *cooperating processes*. It is feasible for these database access functions to use advisory locking if these functions are the only ones being used to access the database. But advisory locking doesn't prevent some other process that has write permission for the database file from writing whatever it wants to the database file. This rogue process would be an uncooperating process since it's not using the accepted method (the library of database functions) to access the database.

Mandatory locking causes the kernel to check every open, read, and write to verify that the calling process isn't violating a lock on the file being accessed. Mandatory locking is sometimes called enforcement-mode locking.

We saw in Figure 12.1 that SVR4 provides mandatory record locking. It is not part of POSIX.1.

Mandatory locking is enabled for a particular file by turning on the set-group-ID bit and turning off the group-execute bit. (Recall Program 4.4.) Since the set-group-ID bit makes no sense when the group-execute bit is off, the designers of SVR3 chose this way to specify that the locking for a file is to be mandatory locking and not advisory locking. (Many people consider this multiplexing of the set-group-ID bit to be a hack.)

What happens to a process that tries to read or write a file that has mandatory locking enabled and the specified part of the file is currently read or write locked by another process? The answer depends on the type of operation (read or write), the type of lock held by the other process (read lock or write lock), and whether the descriptor for the read or write is nonblocking. Figure 12.7 shows the eight possibilities.

| | Blocking descriptor, tries to | | Nonblocking descriptor, tries to | |
|---|---|---|---|---|
| | read | write | read | write |
| read lock exists on region | OK | blocks | OK | EAGAIN |
| write lock exists on region | blocks | blocks | EAGAIN | EAGAIN |

**Figure 12.7** Effect of mandatory locking on reads and writes by other processes.

In addition to the read and write functions in Figure 12.7, the open function can also be affected by mandatory record locks held by another process. Normally, open succeeds, even if the file being opened has outstanding mandatory record locks. The next read or write follows the rules listed in Figure 12.7. But if the file being opened has outstanding mandatory record locks (either read locks or write locks); and if the flags in the call to open specify either O_TRUNC, or O_CREAT, then open returns an error of EAGAIN immediately, regardless whether O_NONBLOCK is specified. (Generating the open error for O_TRUNC makes sense, because the file cannot be truncated if it is read locked or write locked by another process. Generating the error for O_CREAT, however, makes little sense, since this flag says to create the file only if it doesn't already exist, but it has to exist to be record locked by another process.)

This handling of locking conflicts with open can lead to surprising results. While developing the exercises in this section a test program was run that opened a file (whose mode specified mandatory locking), established a read lock on an entire file, then went to sleep for a while. (Recall from Figure 12.7 that a read lock should prevent writing to the file by other processes.) During this sleep period the following behavior was seen in other "normal" Unix programs.

- The same file could be edited with the ed editor, and the results written back to disk! The mandatory record locking had no effect at all. Using the system call trace feature provide by some versions of Unix it was seen that ed wrote the new contents to a temporary file, removed the original file, then renamed the

temporary file to be the original file. The mandatory record locking has no effect on the `unlink` function, which allowed this to happen.

> Under SVR4 the system call trace of a process is obtained by the `truss`(1) command. 4.3+BSD uses the `ktrace`(1) and `kdump`(1) commands.

- The `vi` editor was never able to edit the file. It could read the file's contents, but whenever we tried to write new data to the file, `EAGAIN` was returned. If we tried to append new data to the file, the `write` blocked. This behavior from `vi` is what we expect.
- Using the KornShell's > and >> operators to overwrite or append to the file resulted in the error "cannot create."
- Using the same two operators with the Bourne shell resulted in an error for >, but the >> operator just blocked until the mandatory lock was removed, and then proceeded. (The difference in the handling of the append operator is because the KornShell opens the file with `O_CREAT` and `O_APPEND`, and we mentioned above that specifying `O_CREAT` generates an error. The Bourne shell, however, doesn't specify `O_CREAT` if the file already exists, so the `open` succeeds but the next `write` blocks.)

The bottom line with this exercise is to be wary of mandatory record locking. As seen with the `ed` example, it can be circumvented.

Mandatory record locking can also be used by a malicious user to hold a read lock on a file that is publicly readable. This can prevent anyone from writing to the file. (Of course, the file has to have mandatory record locking enabled for this to occur, which may require the user be able to change the permission bits of the file.) Consider a database file that is world readable and has mandatory record locking enabled. If a malicious user were to hold a read lock on the entire file, the file could not be written to by other processes.

### Example

Program 12.7 determines whether mandatory locking is supported by a system.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <sys/wait.h>
#include    <errno.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
main(void)
{
    int             fd;
    pid_t           pid;
    char            buff[5];
    struct stat     statbuf;
```

```
    if ( (fd = open("templock", O_RDWR | O_CREAT | O_TRUNC,
                                                FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");

        /* turn on set-group-ID and turn off group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");

    TELL_WAIT();
    if ( (pid = fork()) < 0) {
        err_sys("fork error");

    } else if (pid > 0) {    /* parent */
                /* write lock entire file */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("write_lock error");
        TELL_CHILD(pid);

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

    } else {                 /* child */
        WAIT_PARENT();       /* wait for parent to set lock */

        set_fl(fd, O_NONBLOCK);

            /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1)    /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n", errno);

            /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buff, 2) < 0)
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buff = %2.2s\n", buff);
    }
    exit(0);
}
```

**Program 12.7** Determine whether mandatory locking is supported.

This program creates a file and enables mandatory locking for the file. It then splits into a parent and child, with the parent obtaining a write lock on the entire file. The child first sets its descriptor nonblocking and then attempts to obtain a read lock on the file, expecting to get an error. This lets us see if the system returns EACCES or EAGAIN. Next the child rewinds the file and tries to read from the file. If mandatory locking is

provided, the read should return EACCES or EAGAIN (since the descriptor is nonblocking). Otherwise the read returns the data that it read. Running this program under SVR4 (which supports mandatory locking) gives us

```
$ a.out
read_lock of already-locked region returns 13
read failed (mandatory locking works): No more processes
```

If we look at either the system's headers or the intro(2) manual page, we see that an errno of 13 corresponds to EACCES. We can also see from this example that the errno returned by the read (EAGAIN) has the nondescriptive message "No more processes" associated with it. Normally this error comes from fork when we are out of processes.

Under 4.3+BSD we get

```
$ a.out
read_lock of already-locked region returns 35
read OK (no mandatory locking), buff = ab
```

Here an errno of 35 corresponds to EAGAIN. Mandatory locking is not supported. □

**Example**

Let's return to the first question of this section: what happens when two people edit the same file at the same time? The normal Unix text editors do not employ record locking, so the answer is still that the final result of the file corresponds to the last process that wrote the file. (The 4.3+BSD vi editor does have a compile-time option to enable runtime advisory record locking, but this option is not enabled by default.) Even if we were to put advisory locking into one editor, say vi, it still doesn't prevent users from using another editor that doesn't employ advisory record locking.

If the system provides mandatory record locking, we could modify our favorite editor to use it (if we have the sources). Not having the source code to the editor, we might try the following. We write our own program that is a front-end to vi. This program immediately calls fork and the parent just waits for the child to complete. The child opens the file specified on the command line, enables mandatory locking, obtains a write lock on the entire file, and then execs vi. While vi is running, the file is write locked, so other users can't modify it. When vi terminates, the parent's wait returns, and our front-end terminates. Assumed in this example is that locks are inherited across an exec, which we said earlier is the case for SVR4 (the only system we've described that provides mandatory locking).

A small front-end program of this type can be written, but it doesn't work. The problem is that most editors (vi and ed, at least) read their input file and then close it. A lock is released on a file whenever a descriptor that references that file is closed. This means that when the editor closes the file after reading its contents, the lock is gone. There is no way to prevent this in the front-end program. □

We use record locking in Chapter 16 in our database library to provide concurrent access to multiple processes. In this chapter we also provide some timing measurements to see what effect record locking has on a process.

## 12.4 Streams

Streams are provided by System V as a general way to interface communication drivers into the kernel. We need to discuss streams to understand (a) the terminal interface in System V, (b) the use of the `poll` function for I/O multiplexing (Section 12.5.2), (c) the implementation of stream pipes and named stream pipes (Sections 15.2 and 15.5).

> Streams were developed by Dennis Ritchie [Ritchie 1984] as a way of cleaning up the traditional character I/O system (clists) and to accommodate networking protocols. It was later added to SVR3. Complete support for streams (i.e., a streams-based terminal I/O system) was provided with SVR4. The SVR4 implementation is described in [AT&T 1990d]. SVR4 calls the feature STREAMS. We'll just use the all lowercase name.
>
> Be careful not to confuse this usage of the word streams with our previous usage of it in the standard I/O library (Section 5.2).

A stream provides a full-duplex path between a user process and a device driver. There is no need for a stream to talk to an actual hardware device—streams can also be used with pseudo device drivers. Figure 12.8 shows the basic picture for what is called a simple stream.

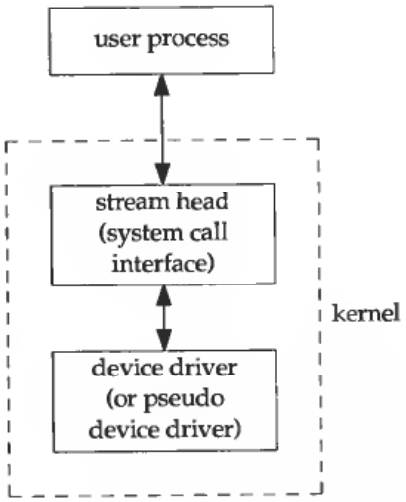


**Figure 12.8** A Simple stream.

Beneath the stream head we can push processing modules onto the stream. This is done using an `ioctl`. Figure 12.9 shows a stream with a single processing module. We also show the connection between these boxes with two arrows, to stress the full-duplex nature of streams.

Any number of processing modules can be pushed onto a stream. We use the term push, because each new module goes beneath the stream head, pushing any previously pushed modules down. (This is similar to a last-in, first-out stack.) In Figure 12.9 we have labeled the downstream and upstream sides of the stream. Data that we write to a stream head is sent downstream. Data read by the device driver is sent upstream.

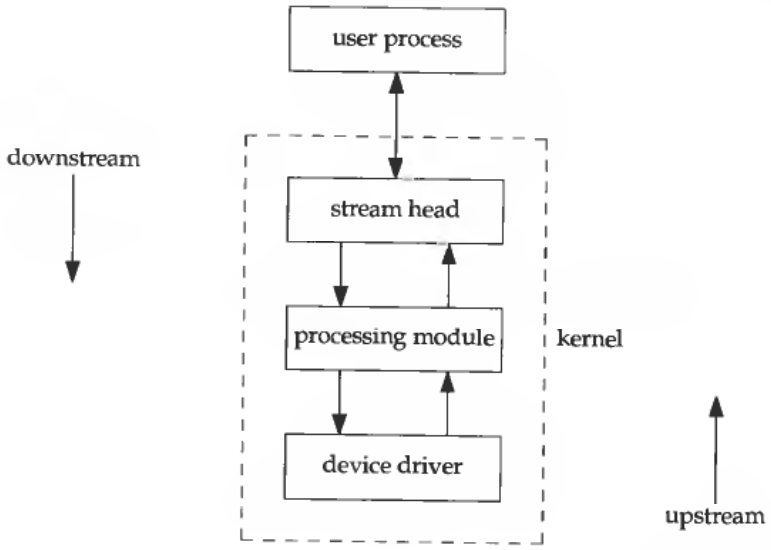


**Figure 12.9** A stream with a processing module.

Streams modules are similar to device drivers in that they execute as part of the kernel, and they are normally link edited into the kernel when the kernel is built. Most systems don't allow us to take arbitrary streams modules that have not been link edited into the kernel and try to push them onto a stream.

Figure 11.2 shows the normal picture of a streams-based terminal system. In this figure what we've labeled "read and write functions" is the stream head, and the box labeled "terminal line discipline" is a streams processing module. The actual name of this processing module is usually `ldterm`. (The manual pages for the various streams modules are found in Section 7 of [AT&T 1990d] and Section 7 of [AT&T 1991].)

We access a stream with the functions from Chapter 3: `open`, `close`, `read`, `write`, and `ioctl`. Additionally, three new functions were added to the SVR3 kernel to support streams (`getmsg`, `putmsg`, and `poll`), and another two were added with SVR4 to handle messages with different priority bands within a stream (`getpmsg` and `putpmsg`). We describe these five new functions later in this section. The *pathname* that we `open` for a stream normally lives beneath the `/dev` directory. Just looking at the device name using `ls -l`, we can't tell if the device is a streams device or not. All streams devices are character special files.

Although some streams documentation implies that we can write processing modules and push them willy-nilly onto a stream, the writing of these modules requires the same skills and care as writing a device driver. It is generally specialized applications or functions that push and pop streams modules.

> Before streams, terminals were handled with the existing clist mechanism. (Section 10.3.1 of Bach [1986] and Section 9.6 of Leffler et al. [1989] describe clists in SVR2 and 4.3BSD, respectively.) Adding other character-based devices to the kernel usually involved writing a device

driver and putting everything into the driver. Access to the new device was typically through the raw device, meaning every user `read` or `write` ended up directly in the device driver. The streams mechanism cleans up this way of interaction, allowing the data to flow between the stream head and the driver in streams messages and allowing any number of intermediate processing modules to operate on the data.

### Streams Messages

All input and output under streams is based on messages. The stream head and user process exchange messages using `read`, `write`, `ioctl`, `getmsg`, `getpmsg`, `putmsg`, and `putpmsg`. Messages are also passed up and down a stream between the stream head, the processing modules, and the device driver.

Between the user process and the stream head a message consists of (a) a message type, (b) optional control information, and (c) optional data. We show in Figure 12.10 how the different message types are generated by the various arguments to `write`, `putmsg`, and `putpmsg`. The control information and data are specified by `strbuf` structures.

```
struct strbuf
  int  maxlen;  /* size of buffer */
  int  len;     /* number of bytes currently in buffer */
  char *buf;    /* pointer to buffer */
};
```

When we send a message with `putmsg` or `putpmsg`, `len` specifies the number of bytes of data in the buffer. When we receive a message with `getmsg` or `getpmsg`, `maxlen` specifies the size of the buffer (so the kernel won't overflow the buffer) and `len` is set by the kernel to the amount of data stored in the buffer. We'll see that a zero-length message is OK, and a `len` of –1 can specify that there is no control or data.

Why do we need to pass both control information and data? Providing both allows us to implement service interfaces between a user process and a stream. Olander, McGrath, and Israel [1986] describe the original implementation of service interfaces in System V. Chapter 5 of AT&T [1990d] describes service interfaces in detail, along with a simple example. Probably the best-known service interface is the System V Transport Layer Interface (TLI), described in Chapter 7 of Stevens [1990], which provides an interface to the networking system.

Another example of control information is sending a connectionless network message (a datagram). To send the message we need to specify the contents of the message (the data) and the destination address for the message (the control information). If we couldn't send control and data together, some ad hoc scheme would be required. For example, we could specify the address using an `ioctl`, followed by a `write` of the data. Another technique would be to require that the address occupy the first $N$ bytes of the data that is written using `write`. Separating the control information from the data, and providing functions that handle both (`putmsg` and `getmsg`) is a cleaner way to handle this.

There are about 25 different types of messages, but only a few of these are used between the user process and the stream head. The rest are passed up and down a

stream within the kernel. (These are of interest to people writing streams-processing modules, but can safely be ignored by people writing user-level code.) We'll encounter only three of these message types with the functions we use (`read`, `write`, `getmsg`, `getpmsg`, `putmsg`, and `putpmsg`):

- M_DATA (user data for I/O),
- M_PROTO (protocol control information), and
- M_PCPROTO (high-priority protocol control information).

Every message on a stream has a queueing priority:

- high-priority messages (highest priority)
- priority band messages
- ordinary messages (lowest priority)

Ordinary messages are priority band messages with a band of 0. Priority band messages have a band of 1–255, with a higher band specifying a higher priority.

Each streams module has two input queues. One receives messages from the module above (messages moving downstream from the stream head toward the driver), and one receives messages from the module below (messages moving upstream from the driver toward the stream head). The messages on an input queue are arranged by priority. We show in Figure 12.10 how the different arguments to `write`, `putmsg`, and `putpmsg` cause these different priority messages to be generated.

There are other types of messages that we don't consider. For example, if the stream head receives an `M_SIG` message from below, it generates a signal. This is how a terminal line discipline module sends the terminal-generated signals to the foreground process group associated with a controlling terminal.

### `putmsg` and `putpmsg` Functions

A streams message (control information or data, or both) is written to a stream using either `putmsg` or `putpmsg`. The difference in these two functions is that the latter allows us to specify a priority band for the message.

```
#include <stropts.h>

int putmsg(int filedes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flag);

int putpmsg(int filedes, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flag);
```

Both return: 0 if OK, –1 on error

We can also `write` to a stream, and that is equivalent to a `putmsg` without any control information and with a *flag* of 0.

These two functions can generate the three different priorities of messages: ordinary, priority band, and high-priority. Figure 12.10 details the different combinations of the arguments to these two functions that generate the different types of messages.

| Function | Control? | Data? | *band* | *flag* | Message type generated |
|---|---|---|---|---|---|
| `write` | N/A | yes | N/A | N/A | M_DATA (ordinary) |
| `putmsg` | no | no | N/A | 0 | no message sent, returns 0 |
| `putmsg` | no | yes | N/A | 0 | M_DATA (ordinary) |
| `putmsg` | yes | yes or no | N/A | 0 | M_PROTO (ordinary) |
| `putmsg` | yes | yes or no | N/A | `RS_HIPRI` | M_PCPROTO (high-priority) |
| `putmsg` | no | yes or no | N/A | `RS_HIPRI` | error, `EINVAL` |
| `putpmsg` | yes or no | yes or no | 0–255 | 0 | error, `EINVAL` |
| `putpmsg` | no | no | 0–255 | `MSG_BAND` | no message sent, returns 0 |
| `putpmsg` | no | yes | 0 | `MSG_BAND` | M_DATA (ordinary) |
| `putpmsg` | no | yes | 1–255 | `MSG_BAND` | M_DATA (priority band) |
| `putpmsg` | yes | yes or no | 0 | `MSG_BAND` | M_PROTO (ordinary) |
| `putpmsg` | yes | yes or no | 1–255 | `MSG_BAND` | M_PROTO (priority band) |
| `putpmsg` | yes | yes or no | 0 | `MSG_HIPRI` | M_PCPROTO (high-priority) |
| `putpmsg` | no | yes or no | 0 | `MSG_HIPRI` | error, `EINVAL` |
| `putpmsg` | yes or no | yes or no | nonzero | `MSG_HIPRI` | error, `EINVAL` |

**Figure 12.10** Type of streams message generated for `write`, `putmsg`, and `putpmsg`.

The notation "N/A" means not applicable. In this figure a "no" for the control portion of the message corresponds to either a null *ctlptr* argument, or *ctlptr->len* being –1. A "yes" for the control portion corresponds to *ctlptr* being nonnull and *ctlptr->len* being greater than or equal to 0. The data portion of the message is handled equivalently (using *dataptr* instead of *ctlptr*).

### Streams `ioctl` Operations

We mentioned in Section 3.14 that the `ioctl` function is the catchall for anything that can't be done with the other I/O functions. The streams system continues this tradition.

Under SVR4 there are 29 different operations that can be performed on a stream using `ioctl`. These operations are documented in the `streamio`(7) manual page (part of [AT&T 1990d]) and the header `<stropts.h>` must be included in C code that uses any of these operations. The second argument for `ioctl`, *request*, specifies which of the 29 operations to perform. All the *request*s begin with `I_`. The third argument depends on the *request*. Sometimes the third argument is an integer value and sometimes it's a pointer to an integer or a structure.

### Example—`isastream` Function

We sometimes need to determine if a descriptor refers to a stream or not. This is similar to calling the `isatty` function to determine if a descriptor refers to a terminal device (Section 11.9). SVR4 provides the `isastream` function.

```
int isastream(int filedes);
```

Returns: 1 (true) if streams device, 0 (false) otherwise

(For some reason, the designers of SVR4 forgot to put the prototype for this function in a header, so we can't show an `#include` for this function.)

Like `isatty`, this is usually a trivial function that just tries an `ioctl` that is valid only on a streams device. Program 12.8 is one possible implementation of this function. We use the `I_CANPUT` `ioctl`, which checks if the band specified by the third argument (0 in the example) is writable. If the `ioctl` succeeds, the stream is not changed.

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

int
isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

**Program 12.8** Check if descriptor is a streams device.

We can use Program 12.9 to test this function.

```
#include    <sys/types.h>
#include    <sys/fcntl.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int     i, fd;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if ( (fd = open(argv[i], O_RDONLY)) < 0) {
            err_ret("%s: can't open", argv[i]);
            continue;
        }

        if (isastream(fd) == 0)
            err_ret("%s: not a stream", argv[i]);
        else
            err_msg("%s: streams device", argv[i]);
    }
    exit(0);
}
```

**Program 12.9** Test the `isastream` function.

Running this program shows the various errors returned by the `ioctl` function.

```
$ a.out /dev/tty /dev/vidadm /dev/null /etc/motd
/dev/tty: /dev/tty: streams device
/dev/vidadm: /dev/vidadm: not a stream: Invalid argument
/dev/null: /dev/null: not a stream: No such device
/etc/motd: /etc/motd: not a stream: Not a typewriter
```

`/dev/tty` is a streams device, as we expect under SVR4. `/dev/vidadm` is not a streams device, but it is a character special file that supports other `ioctl` requests. These devices return `EINVAL` when the `ioctl` request is unknown. `/dev/null` is a character special file that does not support any `ioctl` operations, so the error `ENODEV` is returned. Finally, `/etc/motd` is a regular file, not a character special file, so the classic error `ENOTTY` is returned. We never receive the error we might expect: `ENOSTR` ("Device is not a stream").

> "Not a typewriter" is a historical artifact because the Unix kernel returns `ENOTTY` whenever an `ioctl` is attempted on a descriptor that doesn't refer to a character special device. □

### Example

If the `ioctl` *request* is `I_LIST`, the system returns the names of all the modules on the stream—the ones that have been pushed onto the stream, including the topmost driver. (We say topmost because in the case of a multiplexing driver there may be more than one driver. Chapter 10 of AT&T [1990d] discusses multiplexing drivers in detail.) The third argument must be a pointer to a `str_list` structure.

```
struct str_list {
  int                sl_nmods;   /* number of entries in array */
  struct str_mlist  *sl_modlist; /* ptr to first element of array */
};
```

We have to set `sl_modlist` to point to the first element of an array of `str_mlist` structures, and set `sl_nmods` to the number of entries in the array.

```
struct str_mlist {
  char  l_name[FMNAMESZ+1];  /* null terminated module name */
};
```

The constant `FMNAMESZ` is defined in the header `<sys/conf.h>` and is often 8. The extra byte in `l_name` is for the terminating null byte.

If the third argument to the `ioctl` is 0, the count of the number of modules is returned (as the value of `ioctl`) instead of the module names. We'll use this to determine the number of modules and then allocate the required number of `str_mlist` structures.

Program 12.10 illustrates the `I_LIST` operation. Since the returned list of names doesn't differentiate between the modules and the driver, when we print the module names we know that the final entry in the list is the driver at the bottom of the stream.

If we run Program 12.10 from both a network login and a console login, to see which streams modules are pushed onto the controlling terminal, we get the following:

```
#include    <sys/conf.h>
#include    <sys/types.h>
#include    <fcntl.h>
#include    <stropts.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int                 fd, i, nmods;
    struct str_list     list;

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

    if ( (fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if (isastream(fd) == 0)
        err_quit("%s is not a stream", argv[1]);

            /* fetch number of modules */
    if ( (nmods = ioctl(fd, I_LIST, (void *) 0)) < 0)
        err_sys("I_LIST error for nmods");
    printf("#modules = %d\n", nmods);

            /* allocate storage for all the module names */
    list.sl_modlist = calloc(nmods, sizeof(struct str_mlist));
    if (list.sl_modlist == NULL)
        err_sys("calloc error");
    list.sl_nmods = nmods;

            /* and fetch the module names */
    if (ioctl(fd, I_LIST, &list) < 0)
        err_sys("I_LIST error for list");

            /* print the module names */
    for (i = 1; i <= nmods; i++)
        printf("  %s: %s\n", (i == nmods) ? "driver" : "module",
                             list.sl_modlist++);
    exit(0);
}
```

**Program 12.10** List the names of the modules on a stream.

```
$ who
stevens    console      Sep 25 06:12
stevens    pts001       Oct 12 07:12
$ a.out /dev/pts001
#modules = 4
  module: ttcompat
  module: ldterm
  module: ptem
  driver: pts
```

```
$ a.out /dev/console
#modules = 5
  module: ttcompat
  module: ldterm
  module: ansi
  module: char
  driver: cmux
```

The top two streams modules are the same for both cases (`ttcompat` and `ldterm`), but the remaining modules and the topmost driver differ. We'll return to the pseudo-terminal case (the network login) in Chapter 19. □

### `write` to Streams Devices

In Figure 12.10 we said that a `write` to a streams device generates an M_DATA message. While this is generally true, there are some additional details to consider. First, with a stream the topmost processing module specifies the minimum and maximum packet sizes that can be sent downstream. (We are unable to query the module for these values.) If we `write` more than the maximum, the stream head normally breaks the data into packets of the maximum size, with one final packet that can be smaller than the maximum.

The next thing to consider is what happens if we `write` zero bytes to a stream. Unless the stream refers to a pipe or FIFO, a zero-length message is sent downstream. With a pipe or FIFO, the default is to ignore the zero-length `write`, for compatibility with previous versions. We can change this default for pipes and FIFOs using an `ioctl` to set the write mode for the stream.

### Write Mode

There are two `ioctl`s that fetch and set the "write mode" for a stream. Setting *request* to `I_GWROPT` requires that the third argument be a pointer to an integer, and the current write mode for the stream is returned in that integer. If *request* is `I_SWROPT` then the third argument is an integer whose value becomes the new write mode for the stream. As with the file descriptor flags and the file status flags (Section 3.13) we should always fetch the current write mode value and modify it, rather than setting the write mode to some absolute value (possibly turning off some other bits that were enabled).

Currently only two write mode values are defined.

`SNDZERO` A zero-length `write` to a pipe or FIFO will cause a zero-length message to be sent downstream. By default this zero-length `write` sends no message.

`SNDPIPE` Causes `SIGPIPE` to be sent to the calling process that calls either `write` or `putmsg` after an error has occurred on a stream.

A stream also has a read mode, and we'll look at it after describing the `getmsg` and `getpmsg` functions.

### getmsg and getpmsg Functions

Streams messages are read from a stream head using read, getmsg, or getpmsg.

```
#include <stropts.h>

int getmsg(int filedes, struct strbuf *ctlptr,
           struct strbuf *dataptr, int *flagptr);

int getpmsg(int filedes, struct strbuf *ctlptr,
            struct strbuf *dataptr, int *bandptr, int *flagptr);
```

Both return: nonnegative value if OK, –1 on error

Note that *flagptr* and *bandptr* are pointers to integers. The integer pointed to by these two pointers must be set before the call to specify the type of message desired, and the integer is also set on return to the type of message that was read.

If the integer pointed to by *flagptr* is 0, getmsg returns the next message on the stream head's read queue. If the next message is a high-priority message, on return the integer pointed to by *flagptr* is set to RS_HIPRI. If we want to receive only high-priority messages, we must set the integer pointed to by *flagptr* to RS_HIPRI before calling getmsg.

A different set of constants are used by getpmsg. It can also use *bandptr* to specify a particular priority band.

These two functions have many conditions that dictate what type of message is returned to the caller, based on (a) the values pointed to by *flagptr* and *bandptr*, (b) what types of messages are on the stream's queue, (c) whether we specify a nonnull *dataptr* and *ctlptr*, and (d) the values of *ctlptr->maxlen* and *dataptr->maxlen*. We won't need all these details for our use of getmsg. Refer to the getmsg(2) manual page for all the gory details.

### Read Mode

We also need to consider what happens if we read from a streams device. There are two potential problems: (1) what happens to the record boundaries associated with the messages on a stream, and (2) what happens if we call read and the next message on the stream has control information? The default handling for condition 1 is called byte-stream mode. In this mode a read takes data from the stream until the requested number of bytes has been read or until there is no more data. The message boundaries associated with the streams messages are ignored in this mode. The default handling for condition 2 causes the read to return an error if there is a control message at the front of the queue. We can change either of these defaults.

Using ioctl, if we set *request* to I_GRDOPT the third argument is a pointer to an integer, and the current read mode for the stream is returned in that integer. A *request* of I_SRDOPT takes the integer value of the third argument and sets the read mode to that value. The read mode is specified by one of the following three constants.

| | |
|---|---|
| RNORM | Normal, byte-stream mode, as described previously. This is the default. |
| RMSGN | Message nondiscard mode. A read takes data from a stream until it reads the requested number of bytes or until a message boundary is encountered. If the read uses a partial message, the rest of the data in the message is left on the stream for a subsequent read. |
| RMSGD | Message discard mode. This is like the nondiscard mode, but if a partial message is used, the remainder of the message is discarded. |

Three additional constants can be specified in the read mode to set the behavior of read when it encounters messages containing protocol information on a stream.

| | |
|---|---|
| RPROTNORM | Protocol-normal mode: read returns an error of EBADMSG. This is the default. |
| RPROTDAT | Protocol-data mode: read returns the control portion as data to the caller. |
| RPROTDIS | Protocol-discard mode: read discards the control information but returns any data in the message. |

### Example

Program 12.11 is Program 3.3 recoded to use getmsg instead of read. If we run this program under SVR4, where both pipes and terminals are implemented using streams, we get the following output.

```
$ echo hello, world | a.out          requires pipes to be implemented using streams
flag = 0, ctl.len = -1, dat.len = 13
hello, world
flag = 0, ctl.len = 0, dat.len = 0   indicates a streams hangup
$ a.out                              requires terminals to be implemented using streams
this is line 1
flag = 0, ctl.len = -1, dat.len = 15
this is line 1
and line 2
flag = 0, ctl.len = -1, dat.len = 11
and line 2
^D                                   type our terminal EOF character
flag = 0, ctl.len = -1, dat.len = 0  tty end of file is not the same as a hangup
$ a.out < /etc/motd
getmsg error: Not a stream device
```

When the pipe is closed (when echo terminates) it appears to Program 12.11 as a streams hangup—both the control length and the data length are set to 0. (We discuss pipes in Section 14.2.) With a terminal, however, typing the end of file character only causes the data length to be returned as 0. This terminal end of file is not the same as a streams hangup. As expected, when we redirect standard input to be a nonstreams device, an error is returned by getmsg. □

```
#include    <stropts.h>
#include    "ourhdr.h"

#define BUFFSIZE    8192

int
main(void)
{
    int             n, flag;
    char            ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf   ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0;       /* return any message */
        if ( (n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("getmsg error");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
                        flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("write error");
    }
}
```

**Program 12.11** Copy standard input to standard output using getmsg.

## 12.5 I/O Multiplexing

When we read from one descriptor and write to another, we can use blocking I/O in a loop such as

```
while ( (n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");
```

We see this form of blocking I/O over and over again. What if we have to read from two descriptors? In this case we can't do a blocking read on either descriptor, as data may appear on one descriptor while we're blocked in a read on the other. A different technique is required to handle this case.

Let's skip ahead and look at the modem dialer in Chapter 18. In this program we read from the terminal (standard input) and write to the modem, and we read from the modem and write to the terminal (standard output). Figure 12.11 shows a picture of this.

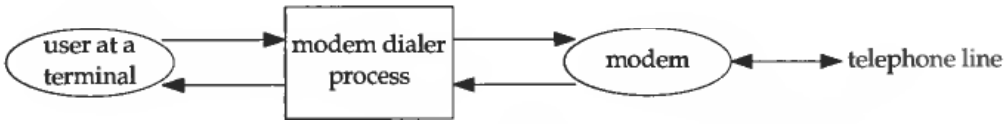


**Figure 12.11** Overview of modem dialer program.

The process has two inputs and two outputs. We can't do a blocking `read` on either of the inputs, as we never know which input will have data for us.

One way to handle this particular problem is to divide the process in two pieces (using `fork`) with each half handling one direction of data. We show this in Figure 12.12.

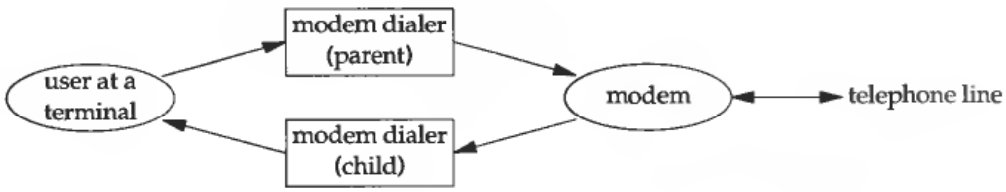


**Figure 12.12** Modem dialer using two processes.

If we use two processes we can let each process do a blocking `read`. But this leads to a problem when the operation terminates. If an end of file is received by the child (the modem is hung up by the other end of the phone line) then the child terminates and the parent is notified by the `SIGCHLD` signal. But if the parent terminates (the user enters an end of file at the terminal) then the parent has to tell the child to stop. We can use a signal for this (`SIGUSR1`, for example) but it does complicate the program somewhat.

We could use nonblocking I/O in a single process. To do this we set both descriptors nonblocking, and issue a `read` on the first descriptor. If data is present, we read it and process it. If there is no data to read, the call returns immediately. We then do the same thing with the second descriptor. After this we wait for some amount of time (a few seconds perhaps), then try to read from the first descriptor again. This type of loop is called *polling*. The problem is that it is a waste of CPU time. Most of the time there won't be data to read, so we waste the time performing the `read` system calls. We also have to guess how long to wait each time around the loop. Although polling works on any system that supports nonblocking I/O, it should be avoided on a multitasking system.

Another technique is called *asynchronous I/O*. To do this we tell the kernel to notify us with a signal when a descriptor is ready for I/O. There are two problems with this. First, not all systems support this feature (it is not yet part of POSIX, but may be in the future). SVR4 provides the `SIGPOLL` signal for this technique, but this signal works only if the descriptor refers to a streams device. 4.3+BSD has a similar signal, `SIGIO`, but it has similar limitations—it works only on descriptors that refer to terminal devices or networks. The second problem with this technique is that there is only one of these

signals per process (SIGPOLL or SIGIO). If we enable this signal for two descriptors (in the example we've been talking about, reading from two descriptors) the occurrence of the signal doesn't tell us which descriptor is ready. To determine which descriptor is ready, we still need to set each nonblocking and try them in sequence. We describe asynchronous I/O briefly in Section 12.6.

A better technique is to use *I/O multiplexing*. To do this we build a list of the descriptors that we are interested in (usually more than one descriptor) and call a function that doesn't return until one of the descriptors is ready for I/O. On return from the function we are told which descriptors are ready for I/O.

> I/O multiplexing is not yet part of POSIX. The select function is provided by both SVR4 and 4.3+BSD to do I/O multiplexing. The poll function is provided only by SVR4. SVR4 actually implements select using poll.
>
> I/O multiplexing was provided with the select function in 4.2BSD. This function has always worked with any descriptor, although its main use has been for terminal I/O and network I/O. SVR3 added the poll function when streams were added. Until SVR4, however, poll only worked with streams devices. SVR4 supports poll on any descriptor.

#### Interruptibility of select and poll

When the automatic restarting of interrupted system calls was introduced with 4.2BSD (Section 10.5), the select function was never restarted. This characteristic continues with 4.3+BSD (and most systems derived from earlier BSD systems) even if the SA_RESTART option is specified. But under SVR4, if SA_RESTART is specified, even select and poll are automatically restarted. To prevent this from catching us when we port software to SVR4, we'll always use the signal_intr function (Program 10.13) if the signal could interrupt a call to select or poll.

### 12.5.1 select Function

The select function lets us do I/O multiplexing under both SVR4 and 4.3+BSD. The arguments we pass to select tell the kernel

1. Which descriptors we're interested in.
2. What conditions we're interested in for each descriptor. (Do we want to read from a given descriptor? Do we want to write to a given descriptor? Are we interested in an exception condition for a given descriptor?)
3. How long we want to wait. (We can wait forever, wait a fixed amount of time, or not wait at all.)

On the return from select the kernel tells us

1. The total count of the number of descriptors that are ready.
2. Which descriptors are ready for each of the three conditions (read, write, or exception condition).

With this return information we can call the appropriate I/O function (usually `read` or `write`) and know that the function won't block.

```
#include <sys/types.h> /* fd_set data type */
#include <sys/time.h>  /* struct timeval */
#include <unistd.h>    /* function prototype might be here */

int select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *tvptr);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

Let's look at the last argument first. This specifies how long we want to wait.

```
struct timeval {
  long  tv_sec;   /* seconds */
  long  tv_usec;  /* and microseconds */
};
```

There are three conditions.

*tvptr* == NULL

Wait forever. This infinite wait can be interrupted if we catch a signal. Return is made when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `select` returns –1 with `errno` set to `EINTR`.

*tvptr->tv_sec* == 0 && *tvptr->tv_usec* == 0

Don't wait at all. All the specified descriptors are tested and return is made immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the `select` function.

*tvptr->tv_sec* != 0 || *tvptr->tv_usec* != 0

Wait the specified number of seconds and microseconds. Return is made when one of the specified descriptors is ready or when the time-out value expires. If the timeout expires before any of the descriptors is ready, the return value is 0. (If the system doesn't provide microsecond resolution, the *tvptr->tv_usec* value is rounded up to the nearest supported value.) As with the first condition, this wait can also be interrupted by a caught signal.

The middle three arguments, *readfds*, *writefds*, and *exceptfds*, are pointers to *descriptor sets*. These three sets specify which descriptors we're interested in and for which conditions (readable, writable, or an exception condition). A descriptor set is stored in an `fd_set` data type. This data type is chosen by the implementation so that it can hold one bit for each possible descriptor. We can consider it just a big array of bits, as shown in Figure 12.13.

The only thing we can do with the `fd_set` data type is (a) allocate a variable of this type, (b) assign a variable of this type to another variable of the same type, or (c) use one of the following four macros on a variable of this type:

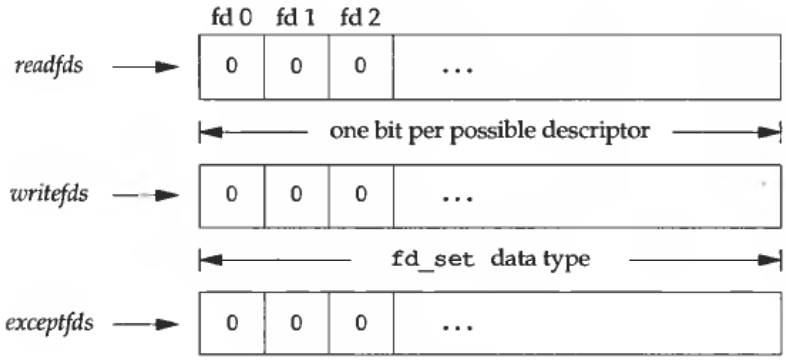


**Figure 12.13** Specifying the read, write, and exception descriptors for `select`.

```
FD_ZERO(fd_set *fdset);              /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset);       /* turn on bit for fd in fdset */
FD_CLR(int fd, fd_set *fdset);       /* turn off bit for fd in fdset */
FD_ISSET(int fd, fd_set *fdset);     /* test bit for fd in fdset */
```

After declaring a descriptor set, as in

```
fd_set    rset;
int       fd;
```

we must zero the set using `FD_ZERO`.

```
FD_ZERO(&rset);
```

We then set bits in the set for each descriptor that we're interested in:

```
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

On return from `select` we can test whether a given bit in the set is still on using `FD_ISSET`:

```
if (FD_ISSET(fd, &rset)) {
    ...
}
```

Any (or all) of the middle three arguments to `select` (the pointers to the descriptor sets) can be null pointers, if we're not interested in that condition. If all three pointers are `NULL`, then we have a higher precision timer than provided by `sleep`. (Recall from Section 10.19 that `sleep` waits for an integral number of seconds. With `select` we can wait for intervals less than 1 second; the actual resolution depending on the system's clock.) Exercise 12.6 shows such a function.

The first argument to `select`, *maxfdp1*, stands for "max fd plus 1." We calculate the highest descriptor that we're interested in, in any of the three descriptor sets, add 1, and that's the first argument. We could just set the first argument to `FD_SETSIZE`, a constant in `<sys/types.h>` that specifies the maximum number of descriptors (often

256 or 1024), but this value is too large for most applications. Indeed, most applications probably use between 3 and 10 descriptors. (There are applications that need many more descriptors, but these aren't the typical Unix program.) By specifying the highest descriptor that we're interested in, the kernel can avoid going through hundreds of unused bits in the three descriptor sets, looking for bits that are turned on.

As an example, if we write

```
fd_set  readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);

FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);

select(4, &readset, &writeset, NULL, NULL);
```

then Figure 12.14 shows what the two descriptor sets look like.

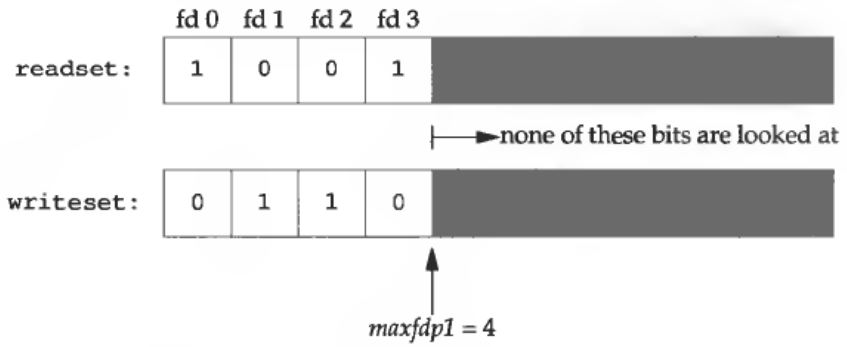


**Figure 12.14** Example descriptor sets for `select`.

The reason we have to add 1 to the maximum descriptor number is because descriptors start at 0, and the first argument is really a count of the number of descriptors to check (starting with descriptor 0).

There are three possible return values from `select`.

1. A return value of –1 means an error occurred. This can happen, for example, if a signal is caught before any of the specified descriptors are ready.
2. A return value of 0 means no descriptors are ready. This happens if the time limit expires before any of the descriptors are ready.
3. A positive return value specifies the number of descriptors that are ready. In this case the only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready.

Be careful not to check the descriptor sets on return unless the return value is greater than 0. The return state of the descriptor sets is implementation dependent if either a signal is caught or the timer expires. Indeed, if the timer expires 4.3+BSD doesn't change the descriptor sets while SVR4 clears the descriptor sets.

There is another discrepancy between the SVR4 and BSD implementations of `select`. BSD systems have always returned the sum of the number of ready descriptors in each set. If the same descriptor is ready in two sets (say the read set and the write set), that descriptor is counted twice. SVR4 unfortunately changes this and if the same descriptor is ready in multiple sets, that descriptor is counted only once. This again shows the problems we'll encounter until functions such as `select` are standardized by POSIX.

We now need to be more specific about what "ready" means.

1. A descriptor in the read set (*readfds*) is considered ready if a `read` from that descriptor won't block.
2. A descriptor in the write set (*writefds*) is considered ready if a `write` to that descriptor won't block.
3. A descriptor in the exception set (*exceptfds*) is considered ready if there is an exception condition pending on that descriptor. Currently an exception condition corresponds to (a) the arrival of out-of-band data on a network connection, or (b) certain conditions occurring on a pseudo terminal that has been placed into packet mode. (Section 15.10 of Stevens [1990] describes this latter condition.)

It is important to realize that whether a descriptor is blocking or not doesn't affect whether `select` blocks or not. That is, if we have a nonblocking descriptor that we want to read from and we call `select` with a time-out value of 5 seconds, `select` will block for up to 5 seconds. Similarly, if we specify an infinite timeout, `select` blocks until data is ready for the descriptor, or until a signal is caught.

If we encounter the end of file on a descriptor, that descriptor is considered readable by `select`. We then call `read` and it returns 0, the normal Unix way to signify end of file. (Many people incorrectly assume `select` indicates an exception condition on a descriptor when the end of file is reached.)

### 12.5.2 `poll` Function

The SVR4 `poll` function is similar to `select`, but the programmer interface is different. As we'll see, `poll` is tied to the streams system, although in SVR4 we are able to use it with any descriptor.

```
#include <stropts.h>
#include <poll.h>

int poll(struct pollfd fdarray[], unsigned long nfds, int timeout);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

Instead of building a set of descriptors for each condition (readability, writability, and exception condition), as we did with `select`, with `poll` we build an array of `pollfd` structures, with each array element specifying a descriptor number and the conditions that we're interested in for that descriptor.

```
struct pollfd {
  int    fd;       /* file descriptor to check, or <0 to ignore */
  short  events;   /* events of interest on fd */
  short  revents;  /* events that occurred on fd */
};
```

The number of elements in the *fdarray* array is specified by *nfds*.

> For some unknown reason, SVR3 specified the number of elements in the array as an `unsigned long`, which seems excessive. In the SVR4 manual [AT&T 1990d], the prototype for `poll` shows the data type of the second argument as `size_t`. (Recall the primitive system data types, Figure 2.8.) But the actual prototype in the `<poll.h>` header still shows the second argument as an `unsigned long`.
>
> The SVID for SVR4 [AT&T 1989] shows the first argument to `poll` as `struct pollfd` *fdarray*`[]`, while the SVR4 manual page [AT&T 1990d] shows this argument as `struct pollfd` **fdarray*. In the C language both declarations are equivalent. We use the first declaration to reiterate that `fdarray` points to an array of structures and not a pointer to a single structure.

We have to set the `events` member of each array element to one or more of the values in Figure 12.15. This is how we tell the kernel what events we're interested in for that descriptor. On return the `revents` member is set by the kernel, specifying which events have occurred for that descriptor. (Notice that `poll` doesn't change the `events` member—this differs from `select`, which modifies its arguments to indicate what is ready.)

| Name | Input to `events`? | Result from `revents`? | Description |
|---|---|---|---|
| POLLIN | • | • | Data other than high priority can be read without blocking. |
| POLLRDNORM | • | • | Normal data (priority band 0) can be read without blocking. |
| POLLRDBAND | • | • | Data from a nonzero priority band can be read without blocking. |
| POLLPRI | • | • | High-priority data can be read without blocking. |
| POLLOUT | • | • | Normal data can be written without blocking. |
| POLLWRNORM | • | • | Same as POLLOUT. |
| POLLWRBAND | • | • | Data for a nonzero priority band can be written without blocking. |
| POLLERR | | • | An error has occurred. |
| POLLHUP | | • | A hangup has occurred. |
| POLLNVAL | | • | The descriptor does not reference an open file. |

**Figure 12.15** The `events` and `revents` flags for `poll`.

The first four rows of Figure 12.15 test for readability, the next three test for writability, and the final three are for exception conditions.

The last three rows in Figure 12.15 are set by the kernel on return. These three values are returned in `revents` when the condition occurs, even if they weren't specified in the `events` field.

When a descriptor is hung up (POLLHUP) we can no longer write to the descriptor. There may, however, still be data to be read from the descriptor.

The final argument to poll specifies how long we want to wait. As with select, there are three different cases.

*timeout* == INFTIM

Wait forever. The constant INFTIM is defined in <stropts.h>, and its value is usually –1. Return is made when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, poll returns –1 with errno set to EINTR.

*timeout* == 0

Don't wait. All the specified descriptors are tested and return is made immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the call to poll.

*timeout* > 0

Wait *timeout* milliseconds. Return is made when one of the specified descriptors is ready or when the *timeout* expires. If the *timeout* expires before any of the descriptors is ready, the return value is 0. (If your system doesn't provide millisecond resolution, *timeout* is rounded up to the nearest supported value.)

It is important to realize the difference between an end of file and a hangup. If we're entering data from the terminal and type the end of file character, POLLIN is turned on so we can read the end of file indication (the read returns 0). POLLHUP is not turned on in revents. If we're reading from a modem and the telephone line is hung up, we'll receive the POLLHUP notification.

As with select, whether a descriptor is blocking or not doesn't affect whether poll blocks or not.

## 12.6 Asynchronous I/O

Using select and poll, as described in the previous section, is a synchronous form of notification. The system doesn't tell us anything until we ask (by calling either select or poll). As we saw in Chapter 10, signals provide an asynchronous form of notification that something has happened. Both SVR4 and 4.3+BSD provide asynchronous I/O, using a signal (SIGPOLL in SVR4, and SIGIO in 4.3+BSD) to notify the process that something of interest has happened on a descriptor.

> We saw that select and poll work with any descriptors under SVR4. Under 4BSD select has always worked with any descriptor. But with asynchronous I/O, we now encounter restrictions. Under SVR4 asynchronous I/O works only with streams devices. Under 4.3+BSD asynchronous I/O works only with terminals and networks.

One limitation of asynchronous I/O, as supported by both SVR4 and 4.3+BSD, is that there is only one signal per process. If we enable more than one descriptor for asynchronous I/O, when the signal is delivered we cannot tell which descriptor the signal corresponds to.

### 12.6.1 System V Release 4

Asynchronous I/O in SVR4 is part of the streams system. It works only with streams devices. The SVR4 asynchronous I/O signal is SIGPOLL.

To enable asynchronous I/O for a streams device we have to call ioctl with a second argument (*request*) of I_SETSIG. The third argument is an integer value formed from one or more of the constants in Figure 12.16. These constants are defined in <stropts.h>.

| Constant | Description |
|---|---|
| S_INPUT | A message other than a high-priority message has arrived. |
| S_RDNORM | An ordinary message has arrived. |
| S_RDBAND | A message with a nonzero priority band has arrived. |
| S_BANDURG | If this constant is specified with S_RDBAND, the SIGURG signal is generated instead of SIGPOLL when a nonzero priority band message has arrived. |
| S_HIPRI | A high-priority message has arrived. |
| S_OUTPUT | The write queue is no longer full. |
| S_WRNORM | Same as S_OUTPUT. |
| S_WRBAND | We can send a nonzero priority band message. |
| S_MSG | A streams signal message that contains the SIGPOLL signal has arrived. |
| S_ERROR | An M_ERROR message has arrived. |
| S_HANGUP | An M_HANGUP message has arrived. |

**Figure 12.16** Conditions for generating SIGPOLL signal.

In Figure 12.16, whenever we say "has arrived" we mean "has arrived at the stream head's read queue."

In addition to calling ioctl to specify the conditions that should generate the SIGPOLL signal, we also have to establish a signal handler for this signal. Recall from Figure 10.1 that the default action for SIGPOLL is to terminate the process, so we should establish the signal handler before calling ioctl.

### 12.6.2 4.3+BSD

Asynchronous I/O in 4.3+BSD is a combination of two different signals: SIGIO and SIGURG. The former is the general asynchronous I/O signal and the latter is used only to notify the process that out-of-band data has arrived on a network connection.

To receive the SIGIO signal we need to perform three steps.

1. Establish a signal handler for the signal, by calling either signal or sigaction.
2. Set the process ID or process group ID to receive the signal for the descriptor, by calling fcntl with a command of F_SETOWN (Section 3.13).
3. Enable asynchronous I/O on the descriptor by calling fcntl with a command of F_SETFL to set the O_ASYNC file status flag (Figure 3.5).

Step 3 can be performed only on descriptors that refer to terminals or networks, which is a fundamental limitation of the 4.3+BSD asynchronous I/O facility.

For the SIGURG signal we need only perform steps 1 and 2. This signal is generated only for descriptors that refer to network connections that support out-of-band data.

## 12.7 readv and writev Functions

The readv and writev functions let us read into and write from multiple noncontiguous buffers in a single function call. These are called *scatter read* and *gather write*.

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

ssize_t readv(int filedes, const struct iovec iov[], int iovcnt);

ssize_t writev(int filedes, const struct iovec iov[], int iovcnt);
```

Both return: number of bytes read or written, –1 on error

The second argument to both functions is a pointer to an array of iovec structures:

```
struct iovec {
  void   *iov_base;  /* starting address of buffer */
  size_t  iov_len;   /* size of buffer */
};
```

The number of elements in the *iov* array is specified by *iovcnt*.

These two functions originated in 4.2BSD. They are now in SVR4 also.

The prototypes for these two functions, and the iovec structure that they both use, exemplify the continuing differences that appear in functions that have not been standardized by either POSIX.1 or XPG3. If we compare the definitions in the SVR4 Programmer's Manual [AT&T 1990e], the SVID for SVR4 [AT&T 1989], and both the SVR4 and 4.3+BSD <sys/uio.h> headers, all are different! Part of the problem is that the SVID and the SVR4 Programmer's Manual correspond to the 1988 POSIX.1 standard, not the 1990 version. The prototype and structure definition that we show above correspond to the POSIX.1 definitions for read and write: the buffer addresses are void *, the buffer lengths are size_t, and the return value is ssize_t.

Note that we have specified the second argument to readv as const. This corresponds to the 4.3+BSD function prototype, but the SVR4 manuals omit this qualifier. The qualifier is valid with readv, since the members of the iovec structure are not modified—only the memory locations pointed to by the iov_base members are modified by the function.

4.3BSD and SVR4 limit *iovcnt* to 16. 4.3+BSD defines the constant UIO_MAXIOV, which is currently 1024. The SVID claims the constant IOV_MAX provides the System V limit, but it's not defined in any of the SVR4 headers.

Figure 12.17 shows a picture relating the arguments to these two functions and the iovec structure. writev gathers the output data from the buffers in order: *iov[0]*,

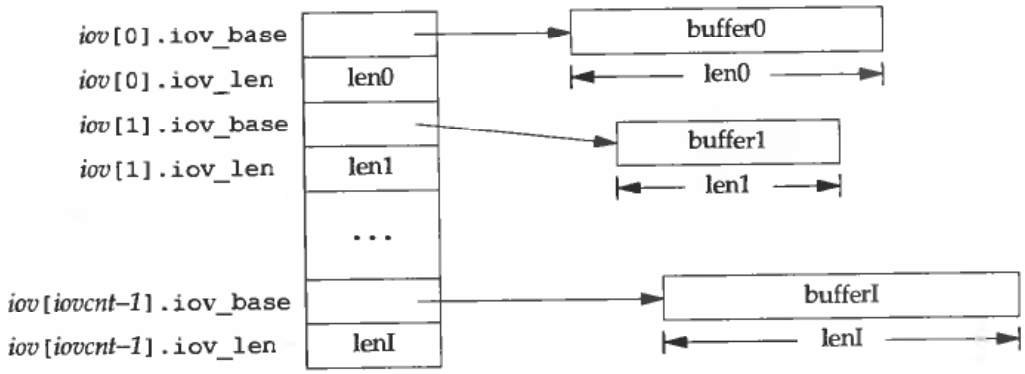


**Figure 12.17** The iovec structure for readv and writev.

*iov[1]*, through *iov[iovcnt–1]*. writev returns the total number of bytes output, which should normally equal the sum of all the buffer lengths.

readv scatters the data into the buffers in order. readv always fills one buffer before proceeding to the next. readv returns the total number of bytes that were read. A count of 0 is returned if there is no more data and the end of file is encountered.

## Example

In Section 16.7, in the function _db_writeidx, we need to write two buffers consecutively to a file. The second buffer to output is an argument passed by the caller, and the first buffer is one we create, containing the length of the second buffer and a file offset of other information in the file. There are three ways we can do this.

1. Call write twice, once for each buffer.
2. Allocate a buffer of our own that is large enough to contain both buffers, and copy both into the new buffer. We then call write once for this new buffer.
3. Call writev to output both buffers.

The solution we use in Section 16.7 is to use writev, but it's instructive to compare it to the other two solutions.

Figure 12.18 shows the results from the three different methods just described.

| Operation | SPARC | | | 80386 | | |
|---|---|---|---|---|---|---|
| | User | System | Clock | User | System | Clock |
| two writes | 0.2 | 7.2 | 17.2 | 0.5 | 13.1 | 13.7 |
| buffer copy, then one write | 0.5 | 4.4 | 17.2 | 0.7 | 7.3 | 8.1 |
| one writev | 0.3 | 4.6 | 17.1 | 0.3 | 7.8 | 8.2 |

**Figure 12.18** Timing results comparing writev and other techniques.

The test program that we measured output a 100-byte header followed by 200 bytes of data. This was done 10,000 times, generating a 3-million-byte file. Three versions of the program were written, and three times were measured for each program: the user CPU time, the system CPU time, and the clock time. All three times are in seconds.

As we expect, the system time almost doubles when we call `write` twice, compared to calling `write` once or `writev` once. This correlates with the results in Figure 3.1.

Next, note that the sum of the CPU times (user plus system) is almost constant whether we do a buffer copy followed by a single `write` or a single `writev`. The difference is whether we pay for the CPU time executing in user space (the buffer copy) or in system space (the `writev`). This sum is 4.9 seconds for the SPARC and about 8.0 seconds for the 80386.

There is one final point to note from Figure 12.18, which is unrelated to our discussion of `readv` and `writev`. The clock time for the SPARC system used for this test is dominated by the disk speed (the clock time is double the CPU time, and the tests were run on an otherwise idle system) while the clock time for the 80386 is dominated by the CPU speed (the clock time almost equals the CPU time). □

In summary, we should always use `readv` and `writev`, instead of multiple `reads` and `writes`. The timing results show that a buffer copy followed by a single `write` often takes the same amount of CPU time as a single `writev`, but usually it is more complicated to allocate the storage for a temporary buffer and do the copy, compared to calling `writev` once.

## 12.8 `readn` and `writen` Functions

Some devices, notably terminals, networks, and any SVR4 streams devices, have the following two properties.

1. A `read` operation may return less than asked for, even though we have not encountered the end of file. This is not an error, and we should just continue reading from the device.
2. A `write` operation can also return less than we specified. This may be caused by flow control constraints by downstream modules, for example. Again, it's not an error, and we should continue writing the remainder of the data. (Normally this short return from a `write` only occurs with a nonblocking descriptor or if a signal is caught.)

We'll never see this happen when reading or writing a disk file.

In Chapter 18 we'll be writing to a stream pipe (which is based on SVR4 streams or BSD Unix domain sockets) and need to take these characteristics into consideration. We can use the following two functions to `read` or `write` *N* bytes of data, letting these functions handle a possible return value that's less than requested. These two functions just call `read` or `write` as many times as required to read or write the entire *N* bytes of data.

```
#include "ourhdr.h"

ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t writen(int filedes, void *buff, size_t nbytes);
```

Both return: number of bytes read or written, –1 on error

We call `writen` anytime we're writing to one of the device types that we mentioned, but we call `readn` only when we know ahead of time that we will be receiving a certain number of bytes. (Often we issue a `read` to one of these devices and take whatever is returned.)

Program 12.12 is an implementation of `writen` that we use in later examples. Program 12.13 is an implementation of `readn`.

## 12.9 Memory Mapped I/O

Memory mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without using `read` or `write`.

To use this feature we have to tell the kernel to map a given file to a region in memory. This is done by the `mmap` function.

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

caddr_t mmap(caddr_t addr, size_t len, int prot, int flag,
             int filedes, off_t off);
```

Returns: starting address of mapped region if OK, –1 on error

> Memory mapped I/O has been in use with virtual memory systems for many years. 4.1BSD (1981) provided a different form of memory mapped I/O with its `vread` and `vwrite` functions. These two functions were then removed in 4.2BSD and were intended to be replaced with the `mmap` function. The `mmap` function, however, was not included with 4.2BSD (for reasons described in Section 2.5 of Leffler et al. [1989]). Gingell, Moran, and Shannon [1987] describe an implementation of `mmap`. The `mmap` function is now supported by both SVR4 and 4.3+BSD.

The data type `caddr_t` is often defined as `char *`. The *addr* argument lets us specify the starting address of where we want the mapped region to start. We normally set this to 0 to allow the system to choose the starting address. The return value of this function is the starting address of the mapped area.

*filedes* is the file descriptor specifying the file that is to be mapped. We have to open this file before we can map it into the address space. *len* is the number of bytes to map,

```
#include    "ourhdr.h"

ssize_t                     /* Write "n" bytes to a descriptor. */
writen(int fd, const void *vptr, size_t n)
{
    size_t      nleft, nwritten;
    const char  *ptr;

    ptr = vptr; /* can't do pointer arithmetic on void* */
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0)
            return(nwritten);       /* error */

        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n);
}
```

**Program 12.12** The writen function.

```
#include    "ourhdr.h"

ssize_t                     /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t  nleft, nread;
    char    *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0)
            return(nread);      /* error, return < 0 */
        else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}
```

**Program 12.13** The readn function.

and *off* is the starting offset in the file of the bytes to map. (There are some restrictions on the value of *off*, described later.)

Before looking at the remaining arguments, let's see what's going on here. Figure 12.19 shows a memory mapped file. (Recall the memory layout of a typical process, Figure 7.3.)

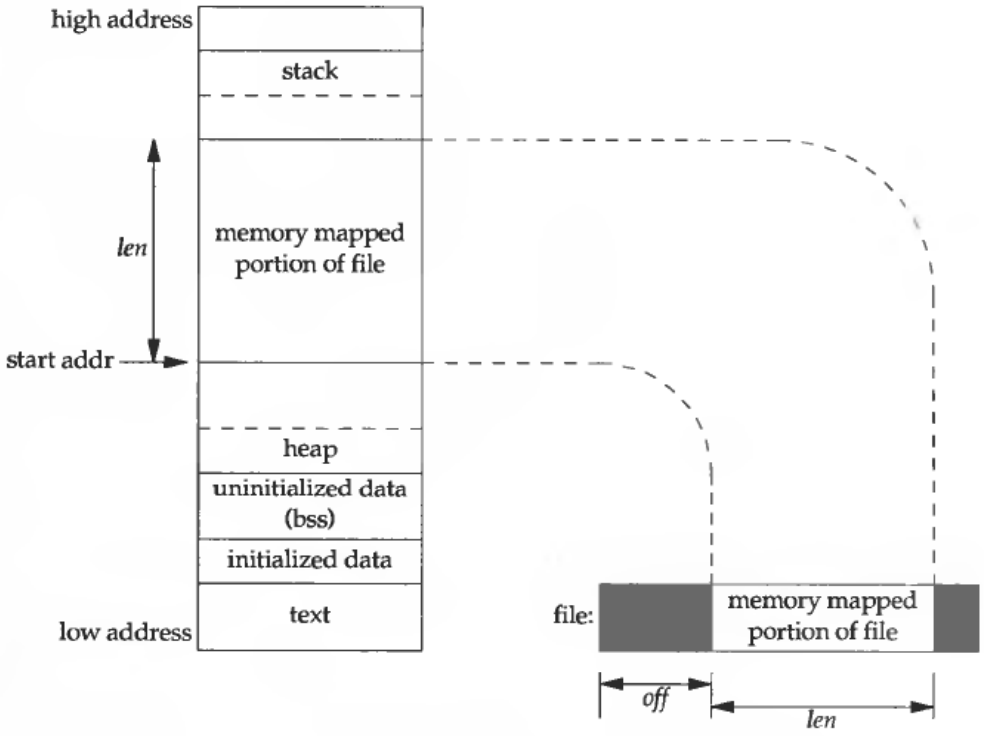


**Figure 12.19** Example of a memory mapped file.

In this figure, "start addr" is the return value from `mmap`. We have shown the mapped memory being somewhere between the heap and the stack: this is an implementation detail and may differ from one implementation to the next.

The *prot* argument specifies the protection of the mapped region.

| *prot* | Description |
|---|---|
| `PROT_READ` | region can be read |
| `PROT_WRITE` | region can be written |
| `PROT_EXEC` | region can be executed |
| `PROT_NONE` | region cannot be accessed (not in 4.3+BSD) |

**Figure 12.20** Protection of memory mapped region.

The protection specified for a region has to match the open mode of the file. For example, we can't specify `PROT_WRITE` if the file was opened read-only.

The *flag* argument affects various attributes of the mapped region.

| | |
|---|---|
| MAP_FIXED | The return value must equal *addr*. Use of this flag is discouraged, as it hinders portability.<br><br>If this flag is not specified, and *addr* is nonzero, then the kernel uses *addr* as a hint of where to place the mapped region.<br><br>Maximum portability is obtained by specifying *addr* as 0. |
| MAP_SHARED | This flag describes the disposition of store operations into the mapped region by this process. This flag specifies that store operations modify the mapped file—that is, as store operation is equivalent to a `write` to the file. Either this flag or the next (MAP_PRIVATE) must be specified. |
| MAP_PRIVATE | This flag says that store operations into the mapped region cause a copy of the mapped file to be created. All successive references to the mapped region then reference the copy. (One use of this flag is for a debugger that maps the text portion of a program file but allows the user to modify the instructions. Any modifications affect the copy, not the original program file.) |

4.3+BSD has additional MAP_xxx flag values, which are specific to that implementation. Check the 4.3+BSD mmap(2) manual page for details.

The value of *off* and the value of *addr* (if MAP_FIXED is specified) are normally required to be multiples of the system's virtual memory page size. Under SVR4 this value can be obtained from the `sysconf` function (Section 2.5.4) with an argument of SC_PAGESIZE. Under 4.3+BSD the page size is defined by the constant NBPG in the header <sys/param.h>. Since *off* and *addr* are often specified as 0, this requirement is not a problem.

Since the starting offset of the mapped file is tied to the system's virtual memory page size, what happens if the length of the mapped region isn't a multiple of the page size? Assume the file size is 12 bytes and the system's page size is 512 bytes. In this case the system normally provides a mapped region of 512 bytes and the final 500 bytes of this region are set to 0. We can modify the final 500 bytes, but any changes we make to them are not reflected in the file.

Two signals are normally used with mapped regions. SIGSEGV is the signal normally used to indicate that we have tried to access memory that is not available to us. It can also be generated if we try to store into a mapped region that we specified to mmap as read-only. The SIGBUS signal can be generated if we access a portion of the mapped region that does not make sense at the time of the access. For example, assume we map a file using the file's size, but before we reference the mapped region the file's size is truncated by some other process. If we then try to access the memory mapped region corresponding to the end portion of the file that was truncated, we'll receive SIGBUS.

A memory mapped region is inherited by a child across a `fork` (since it's part of the parent's address space), but for the same reason is not inherited by the new program across an exec.

A memory mapped region is automatically unmapped when the process terminates, or by calling munmap directly. Closing the file descriptor *filedes* does not unmap the region.

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

int munmap(caddr_t addr, size_t len);
```

Returns: 0 if OK, –1 on error

munmap does not affect the object that was mapped—that is, the call to munmap does not cause the contents of the mapped region to be written to the disk file. The updating of the disk file for a MAP_SHARED region happens automatically by the kernel's virtual memory algorithm as we store into the memory mapped region.

Some systems provide an msync function that is similar to fsync (Section 4.24), but works on memory mapped regions.

## Example

Program 12.14 copies a file (similar to the cp(1) command) using memory mapped I/O. We first open both files and then call fstat to obtain the size of the input file. We need this size for the call to mmap for the input file, plus we need to set the size of the output file. We call lseek and then write one byte to set the size of the output file. If we don't set the output file's size, the call to mmap for the output file is OK, but the first reference to the associated memory region generates SIGBUS. We might be tempted to use ftruncate to set the size of the output file, but not all systems extend the size of a file with this function. (See Section 4.13.)

We then call mmap for each file, to map the file into memory, and finally call memcpy to copy from the input buffer to the output buffer. As the bytes of data are fetched from the input buffer (src), the input file is automatically read by the kernel; and as the data is stored in the output buffer (dst), the data is automatically written to the output file.

Let's compare this memory mapped file copy to a copy that is done by calling read and write (with a buffer size of 8192). Figure 12.21 shows the results.

| Operation | SPARC | | | 80386 | | |
|---|---|---|---|---|---|---|
| | User | System | Clock | User | System | Clock |
| read/write | 0.0 | 2.6 | 11.0 | 0.0 | 5.3 | 11.2 |
| mmap/memcpy | 0.9 | 1.7 | 3.7 | 0.3 | 2.7 | 5.7 |

**Figure 12.21** Timing results comparing read/write versus mmap/memcpy.

The times are given in seconds and the size of the file being copied was almost 3 million bytes.

For the SPARC the total CPU time (user+system) is the same for both types of copies: 2.6 seconds. (This is similar to what we found for writev in Figure 12.18.) For

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <sys/mman.h>    /* mmap() */
#include    <fcntl.h>
#include    "ourhdr.h"

#ifndef MAP_FILE    /* 4.3+BSD defines this & requires it to mmap files */
#define MAP_FILE    0   /* to compile under systems other than 4.3+BSD */
#endif

int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    char        *src, *dst;
    struct stat statbuf;

    if (argc != 3)
        err_quit("usage: a.out <fromfile> <tofile>");

    if ( (fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ( (fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
                                                FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[1]);

    if (fstat(fdin, &statbuf) < 0)  /* need size of input file */
        err_sys("fstat error");

            /* set size of output file */
    if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        err_sys("lseek error");
    if (write(fdout, "", 1) != 1)
        err_sys("write error");

    if ( (src = mmap(0, statbuf.st_size, PROT_READ,
                    MAP_FILE | MAP_SHARED, fdin, 0)) == (caddr_t) -1)
        err_sys("mmap error for input");

    if ( (dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
                    MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1)
        err_sys("mmap error for output");

    memcpy(dst, src, statbuf.st_size);  /* does the file copy */

    exit(0);
}
```

**Program 12.14** Copy a file using memory mapped I/O.

the 386 the total CPU time is almost halved when we use `mmap` and `memcpy`.

When we use `mmap`, the reason that the system time decreases for both the SPARC and the 386 is because the kernel is doing I/O directly to and from the mapped memory buffers. When we call `read` and `write`, the kernel has to copy the data between our buffers and its buffers and then do I/O from its buffers.

The final point to note is that the clock time is at least halved when we use `mmap` and `memcpy`. □

Memory mapped I/O is faster, when copying one regular file to another. There are limitations. We can't use it to copy between certain devices (such as a network device or a terminal device), and we have to be careful if the size of the underlying file could change after we map it. Nevertheless, there are some applications that can benefit from memory mapped I/O, as it can often simplify the algorithms since we manipulate memory instead of reading and writing a file. One example that can benefit from memory mapped I/O is the manipulation of a frame buffer device that references a bit-mapped display.

Krieger, Stumm, and Unrau [1992] describe an alternative to the standard I/O library (Chapter 5) that uses memory mapped I/O.

We return to memory mapped I/O in Section 14.9, showing an example of how it can be used under both SVR4 and 4.3+BSD to provide shared memory between related processes.

## 12.10 Summary

In this chapter we've described numerous advanced I/O functions, most of which are used in the examples in later chapters:

- nonblocking I/O—issuing an I/O operation without letting it block (we'll need this for the PostScript printer driver in Chapter 17);
- record locking (which we'll look at in more detail through an actual example, the database library in Chapter 16);
- System V streams (which we'll need in Chapter 15 to understand SVR4 stream pipes, passing file descriptors, and SVR4 client–server connections);
- I/O multiplexing—the `select` and `poll` functions (we'll use these in many of the later examples);
- the `readv` and `writev` functions (also used in many of the later examples);
- memory mapped I/O (`mmap`).

## Exercises

12.1 Remove the second call to `write` in the `for` loop in Program 12.6. What happens and why?

12.2 Take a look at your system's `<sys/types.h>` header and examine the implementation of `select` and the four `FD_` macros.

12.3 The `<sys/types.h>` header usually has a built-in limit on the maximum number of descriptors that the `fd_set` data type can handle. Assume we need to increase this to handle up to 2048 descriptors. How can we do this?

12.4 Compare the different functions provided for signal sets (Section 10.11) and the `fd_set` descriptor sets. Also compare the implementation of the two on your system.

12.5 How many different types of information does `getmsg` return?

12.6 Implement the function `sleep_us` that is similar to `sleep`, but waits for a specified number of microseconds. Use either `select` or `poll`. Compare this function to the BSD `usleep` function.

12.7 Can you implement the functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Program 10.17 using advisory record locking instead of signals? If so, code and test your implementation.

12.8 Determine the capacity of a pipe using either `select` or `poll`. Compare this value with the value of `PIPE_BUF` from Chapter 2.

12.9 Run Program 12.14 to copy a file and determine whether the last-access time for the input file is updated.

12.10 In Program 12.14 `close` the input file after calling `mmap` to verify that closing the descriptor does not invalidate the memory mapped I/O.

# 13

# Daemon Processes

## 13.1 Introduction

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shutdown. We say they run in the background, because they don't have a controlling terminal. Unix systems have numerous daemons that perform day-to-day activities.

In this chapter we look at the process structure of daemons, and how to write a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

## 13.2 Daemon Characteristics

Let's look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions that we described in Chapter 9. The ps(1) command prints the status of various processes in the system. There are a multitude of options—consult your system's manual for all the details. We'll execute

```
ps -axj
```

under 4.3+BSD or SunOS to see the information we need for this discussion. The `-a` option shows the status of processes owned by others, and `-x` shows processes that don't have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID. Under SVR4 a similar command is `ps -efjc`. (On some Unix systems that conform to the Department of Defense security guidelines, we are not able to use ps to look at any processes other than our own.) The output from ps looks like

```
PPID   PID  PGID   SID TT TPGID  UID  COMMAND
   0     0     0     0 ?    -1    0  swapper
   0     1     0     0 ?    -1    0  /sbin/init -
   0     2     0     0 ?    -1    0  pagedaemon
   1    80    80    80 ?    -1    0  syslogd
   1    88    88    88 ?    -1    0  /usr/lib/sendmail -bd -q1h
   1   105    37    37 ?    -1    0  update
   1   108   108   108 ?    -1    0  cron
   1   114   114   114 ?    -1    0  inetd
   1   117   117   117 ?    -1    0  /usr/lib/lpd
```

We have removed a few columns that don't interest us, such as the accumulated CPU time. The columns headings, in order, are the parent process ID, process ID, process group ID, session ID, terminal name, terminal process group ID (the foreground process group associated with the controlling terminal), user ID, and actual command string.

> The system that these `ps` commands were run on (SunOS) supports the notion of a session ID, which we mentioned with the `setsid` function in Section 9.5. It is just the process ID of the session leader. A 4.3+BSD system, however, will print the address of the `session` structure corresponding to the process group that the process belongs to (Section 9.11).

Processes 0, 1, and 2 are the ones described in Section 8.2. These three are special and exist for the entire lifetime of the system. They have no parent process ID, no process group ID, and no session ID. The `syslogd` daemon is available to any program to log system messages for an operator. The messages may be printed on an actual console device and also written to a file. (We describe the `syslog` facility in Section 13.4.2.) `sendmail` is the standard mailer daemon. `update` is a program that flushes the kernel's buffer cache to disk at regular intervals (usually every 30 seconds). To do this it just calls the `sync`(2) function every 30 seconds. (We described `sync` in Section 4.24.) The `cron` daemon executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by `cron`. We talked about the `inetd` daemon in Section 9.3. It listens on the system's network interfaces for incoming requests for various network servers. The final daemon, `lpd`, handles print requests on the system.

Notice that all the daemons run with superuser privilege (a user ID of 0). None of the daemons has a controlling terminal—the terminal name is set to a question mark and the terminal foreground process group is –1. The lack of a controlling terminal is probably the result of the daemon having called `setsid`. All the daemons other than `update` are process group leaders and session leaders and are the only processes in their process group and session. `update` is the only process in its process group (37) and session (37), but the process group leader (which was probably also the session leader) has already exited. Finally, note that the parent of all these daemons is the `init` process.

## 13.3 Coding Rules

There are some basic rules to coding a daemon, to prevent unwanted interactions from happening. We state these rules and then show a function, `daemon_init`, that implements them.

1. The first thing to do is call `fork` and have the parent `exit`. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to `setsid` that is done next.
2. Call `setsid` to create a new session. The three steps listed in Section 9.5 occur. The process (1) becomes a session leader of a new session, (2) becomes the process group leader of a new process group, and (3) has no controlling terminal.

   > Under SVR4, some people recommend calling `fork` again at this point and having the parent terminate. The second child continues as the daemon. This guarantees that the daemon is not a session leader, which prevents it from acquiring a controlling terminal under the SVR4 rules (Section 9.6). Alternately, to avoid acquiring a controlling terminal be sure to specify `O_NOCTTY` whenever opening a terminal device.

3. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted filesystem. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted filesystem, that filesystem cannot be unmounted.

   Alternately, some daemons might change the current working directory to some specific location, where they will do all their work. For example, line printer spooling daemons often change to their spool directory.
4. Set the file mode creation mask to 0. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.
5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). Exactly which descriptors to close, however, depends on the daemon, so we don't show this step in our example. It can use our `open_max` function (Program 2.3) to determine the highest descriptor and close all descriptors up to that value.

### Example

Program 13.1 is a function that can be called from a program that wants to initialize itself as a daemon.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
daemon_init(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        return(-1);
    else if (pid != 0)
        exit(0);        /* parent goes bye-bye */

    /* child continues */
    setsid();           /* become session leader */

    chdir("/");         /* change working directory */

    umask(0);           /* clear our file mode creation mask */

    return(0);
}
```

**Program 13.1** Initialize a daemon process.

If the `daemon_init` function is called from a `main` program that then goes to sleep, we can check the status of the daemon with the `ps` command:

```
$ a.out
$ ps -axj
 PPID   PID  PGID   SID TT TPGID  UID  COMMAND
    1   735   735   735 ?     -1  224  a.out
```

We can see that our daemon has been initialized correctly. □

## 13.4 Error Logging

One problem a daemon has is how to handle error messages. It can't just write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations the console device runs a windowing system. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which log file and to check these files on a regular basis. A central daemon error logging facility is required.

The BSD `syslog` facility was developed at Berkeley and used widely in 4.2BSD. Most systems derived from 4.xBSD support `syslog`. We describe this facility in Section 13.4.2.

There has never been a central daemon logging facility in System V. SVR4 supports the BSD-style `syslog` facility, and the `inetd` daemon under SVR4 uses `syslog`. The basis for `syslog` in SVR4 is the `/dev/log` streams device driver, which we describe in the next section.

### 13.4.1 SVR4 Streams `log` Driver

SVR4 provides a streams device driver, documented in `log`(7) in [AT&T 1990d], with an interface for streams error logging, streams event tracing, and console logging. Figure 13.1 details the overall structure of this facility.

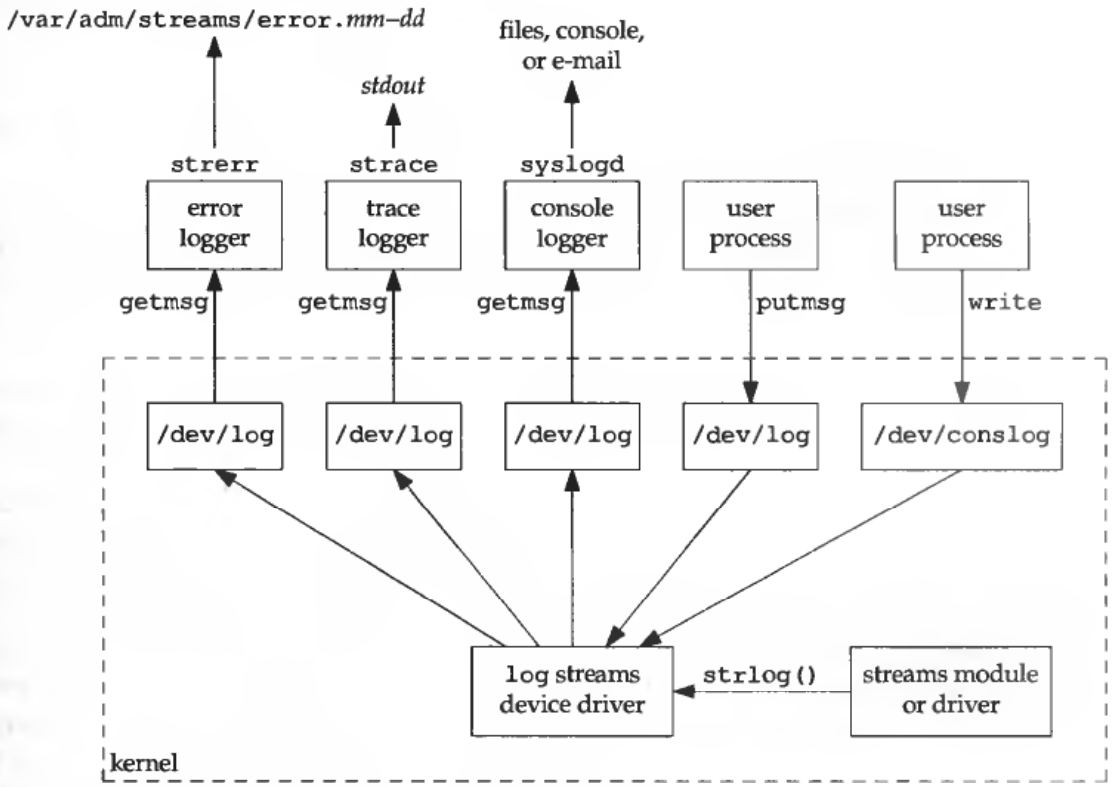


**Figure 13.1** The SVR4 `log` facility.

Each `log` message can be destined for one of three loggers: the error logger, the trace logger, or the console logger.

We show three ways to generate `log` messages and three ways to read them.

- Generating `log` messages.
  1. Routines within the kernel can call `strlog` to generate log messages. This is normally used by streams modules and streams device drivers for either error messages or trace messages. (Trace messages are often used in the

debugging of new streams modules or drivers.) We won't consider this type of message generation, since we're not interested in the coding of kernel routines.

2. A user process (such as a daemon) can `putmsg` to `/dev/log`. This message can be sent to any of the three loggers.
3. A user process (such as a daemon) can `write` to `/dev/conslog`. This message is sent only to the console logger.

- Reading `log` messages.
  4. The normal error logger is `strerr`(1M). It appends these messages to a file in the directory `/var/adm/streams`. The file's name is `error.`*mm-dd*, where *mm* is the month and *dd* is the day of the month. This program is itself a daemon, and it normally runs in the background, appending the log messages to the file.
  5. The normal trace logger is `strace`(1M). It can selectively write a specified set of trace messages to its standard output.
  6. The standard console logger is `syslogd`, a BSD-derived program that we describe in the next section. This program is a daemon that reads a configuration file and writes log messages to specified files or the console device or sends e-mail to certain users.

Not mentioned in this list, but a possibility, is for a user process to replace any of the standard system-supplied daemons: we can supply our own error logger, trace logger, or console logger.

Each `log` message has information in addition to the message itself. For example, the messages that are sent upstream by the `log` driver contain information about who generated the message (if it was generated by a streams module within the kernel), a level, a priority, some flags, and the time the message was generated. Refer to the `log`(7) manual page for all the details. If we're generating a `log` message using `putmsg`, we can also set some of these fields. If we're calling `write` to send a message to the console logger (through `/dev/conslog`), we can send only a message string.

Another possibility, not shown in Figure 13.1, is for a SVR4 daemon to call the BSD `syslog`(3) function. Doing this sends the message to the console logger, similar to a `putmsg` to `/dev/log`. With `syslog`, we can set the priority field of the message. We describe this function in the next section.

If the appropriate type of logger isn't running when a log message of that type is generated, the `log` driver just throws away the message.

> Unfortunately, in SVR4 the use of this `log` facility is haphazard. A few daemons use it, but most system-supplied daemons are hardcoded to write directly to the console.
>
> The `syslog`(3) function and `syslogd`(1M) daemon are documented in the BSD Compatibility Library [AT&T 1990c], but they are not in this library—they are in the standard C library, available to all user processes (daemons).

### 13.4.2 4.3+BSD `syslog` Facility

The BSD `syslog` facility has been widely used since 4.2BSD. Most daemons use this facility. Figure 13.2 details its organization.

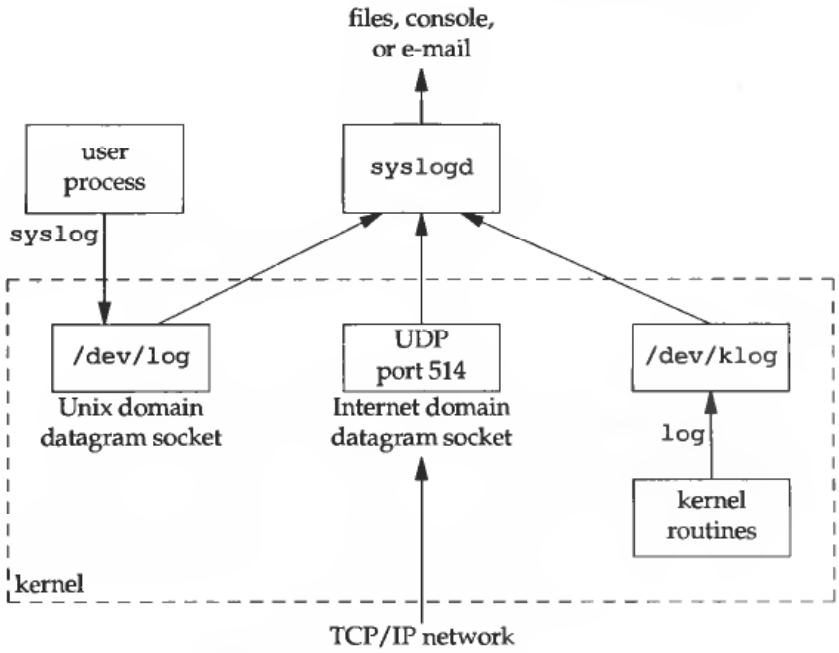


**Figure 13.2** The 4.3+BSD `syslog` facility.

There are three ways to generate log messages:

1. Kernel routines can call the `log` function. These messages can be read by any user process that opens and reads the `/dev/klog` device. We won't describe this function any further, since we're not interested in writing kernel routines.
2. Most user processes (daemons) call the `syslog`(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the Unix domain datagram socket `/dev/log`.
3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the `syslog` function never generates these UDP datagrams—they require explicit network programming by the process generating the log message.

Refer to Stevens [1990] for details on Unix domain sockets and UDP sockets.

Normally the `syslogd` daemon reads all three forms of log messages. This daemon reads a configuration file on start-up, usually `/etc/syslog.conf`, that determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator via e-mail and printed on the console, while warnings may be logged to a file.

Our interface to this facility is through the `syslog` function.

```
#include <syslog.h>

void openlog(char *ident, int option, int facility);

void syslog(int priority, char *format, ...);

void closelog(void);
```

Calling `openlog` is optional. If it's not called, the first time `syslog` is called, `openlog` is called automatically. Calling `closelog` is also optional—it just closes the descriptor that was being used to communicate with the `syslogd` daemon.

Calling `openlog` lets us specify an *ident* that is added to each log message. This is normally the name of the program (e.g., `cron`, `inetd`, etc.). Figure 13.3 describes the four possible *options*.

The *facility* argument for `openlog` is taken from Figure 13.4. The reason for the *facility* argument is to let the configuration file specify that messages from different facilities are to be handled differently. If we don't call `openlog`, or we call it with a *facility* of 0, we can still specify the facility as part of the *priority* argument to `syslog`.

We call `syslog` to generate a log message. The *priority* argument is a combination of the *facility* shown in {syslog_facility} and a *level*, shown in Figure 13.5. These *levels* are ordered by priority, from highest to lowest.

The *format* argument, and any remaining arguments, are passed to the `vsprintf` function for formatting. Any occurrence of the two characters `%m` in the *format* are first replaced with the error message string (`strerror`) corresponding to the value of `errno`.

The `logger`(1) program is also provided by both SVR4 and 4.3+BSD as a way to send log messages to the `syslog` facility. Optional arguments to this program can specify the *facility*, *level*, and *ident*. It is intended for a shell script running noninteractively that needs to generate log messages.

A form of the `logger` command is being standardized by POSIX.2.

## Example

In our PostScript printer daemon in Chapter 17 we will encounter the sequence

```
openlog("lprps", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

The first call sets the *ident* string to the program name, specifies that the process ID should always be printed, and sets the default *facility* to the line printer system. The actual call to `syslog` specifies an error condition and a message string. If we had not called `openlog`, the second call could have been

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Here we specify the *priority* argument as a combination of a *level* and a *facility*. □

| *option* | Description |
|---|---|
| LOG_CONS | If the log message can't be sent to `syslogd` via the Unix domain datagram, the message is written to the console instead. |
| LOG_NDELAY | Open the Unix domain datagram socket to the `syslogd` daemon immediately—don't wait until the first message is logged. Normally the socket is not opened until the first message is logged. |
| LOG_PERROR | Write the log message to standard error in addition to sending it to `syslogd`. This option is supported only by the 4.3BSD Reno releases and later. |
| LOG_PID | Log the process ID with each message. This is intended for daemons that `fork` a child process to handle different requests (as compared to daemons such as `syslogd` that never call `fork`). |

**Figure 13.3** The *option* argument for `openlog`.

| *facility* | Description |
|---|---|
| LOG_AUTH | authorization programs: `login`, `su`, `getty`, ... |
| LOG_CRON | `cron` and `at` |
| LOG_DAEMON | system daemons: `ftpd`, `routed`, ... |
| LOG_KERN | messages generated by the kernel |
| LOG_LOCAL0 | reserved for local use |
| LOG_LOCAL1 | reserved for local use |
| LOG_LOCAL2 | reserved for local use |
| LOG_LOCAL3 | reserved for local use |
| LOG_LOCAL4 | reserved for local use |
| LOG_LOCAL5 | reserved for local use |
| LOG_LOCAL6 | reserved for local use |
| LOG_LOCAL7 | reserved for local use |
| LOG_LPR | line printer system: `lpd`, `lpc`, ... |
| LOG_MAIL | the mail system |
| LOG_NEWS | the Usenet network news system |
| LOG_SYSLOG | the `syslogd` daemon itself |
| LOG_USER | messages from other user processes (default) |
| LOG_UUCP | the UUCP system |

**Figure 13.4** The *facility* argument for `openlog`.

| *level* | Description |
|---|---|
| LOG_EMERG | emergency (system is unusable) (highest priority) |
| LOG_ALERT | condition that must be fixed immediately |
| LOG_CRIT | critical condition (e.g., hard device error) |
| LOG_ERR | error condition |
| LOG_WARNING | warning condition |
| LOG_NOTICE | normal, but significant condition |
| LOG_INFO | informational message |
| LOG_DEBUG | debug message (lowest priority) |

**Figure 13.5** The `syslog` *levels* (ordered).

## 13.5 Client–Server Model

A common use for a daemon process is as a server process. Indeed, in Figure 13.2 we can call the `syslogd` process a server that has messages sent to it by user processes (clients) using a Unix domain datagram socket.

In general a *server* is a process that waits for a *client* to contact it, requesting some type of service. In Figure 13.2 the service being provided by the `syslogd` server is the logging of an error message.

In Figure 13.2 the communication between the client and server is one-way. The client just sends its service request to the server—the server sends nothing back to the client. In the following chapters on interprocess communication we'll see numerous examples where there is a two-way communication between the client and server. The client sends a request to the server, and the server sends a reply back to the client.

## 13.6 Summary

Daemon processes are running all the time on most Unix systems. To initialize our own process that is to run as a daemon takes some care and an understanding of the process relationships that we described in Chapter 9. In this chapter we developed a function that can be called by a daemon process to initialize itself correctly.

We also discussed the ways a daemon can log error messages, since a daemon normally doesn't have a controlling terminal. Under SVR4 the streams `log` driver is available, and under 4.3+BSD the `syslog` facility is provided. Since the BSD `syslog` facility is also provided by SVR4, in later chapters when we need to log error messages from a daemon, we'll call the `syslog` function. We'll encounter this in Chapter 17 with our PostScript printer daemon.

## Exercises

**13.1** As we might guess from Figure 13.2, when the `syslog` facility is initialized, either by calling `openlog` directly or on the first call to `syslog`, the special device file for the Unix domain datagram socket, `/dev/log`, has to be opened. What happens if the user process (the daemon) calls `chroot` before calling `openlog`?

**13.2** List all the daemons active on your system and identify the function of each one.

**13.3** Write a program that calls the `daemon_init` function in Program 13.1. After calling this function, call `getlogin` (Section 8.14) to see if the process has a login name now that it has become a daemon. Print the login name to file descriptor 3 and redirect this descriptor to a temporary file when the program is run with the notation `3>/tmp/name1` (Bourne shell or KornShell).

Now rerun the program closing descriptors 0, 1, and 2 after the call to `daemon_init`, but before the call to `getlogin`. Does this make any difference?

**13.4** Write an SVR4 daemon that establishes itself as a console logger. Refer to `log(7)` in [AT&T 1990d] for the details. Each time a message is received, print the relevant information. Also write a test program that sends console log messages to `/dev/log` to test the daemon.

**13.5** Modify Program 13.1 as we mentioned in rule 2 of Section 13.3 by doing a second `fork` so that it can never acquire a controlling terminal under SVR4. Test your function to verify that it is no longer a session leader.

# 14

# Interprocess Communication

## 14.1 Introduction

In Chapter 8 we described the process control primitives and saw how to invoke 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 filesystem. We'll now describe other techniques for processes to communicate with each other—IPC or interprocess communication.

Unix IPC has been, and continues to be, a hodgepodge of different approaches, few of which are portable across all Unix implementations. Figure 14.1 summarizes the different forms of IPC that are supported by different implementations.

| IPC type | POSIX.1 | XPG3 | V7 | SVR2 | SVR3.2 | SVR4 | 4.3BSD | 4.3+BSD |
|---|---|---|---|---|---|---|---|---|
| pipes (half duplex) | • | • | • | • | • | • | • | • |
| FIFOs (named pipes) | • | • | | • | • | • | | • |
| stream pipes (full duplex) | | | | | • | • | • | • |
| named stream pipes | | | | | • | • | • | • |
| message queues | | • | | • | • | • | | |
| semaphores | | • | | • | • | • | | |
| shared memory | | • | | • | • | • | | |
| sockets | | | | | | • | • | • |
| streams | | | | | • | • | | |

**Figure 14.1** Summary of Unix IPC.

As this figure shows, about the only form of IPC that we can count on, regardless of the Unix implementation, is half-duplex pipes. The first seven forms of IPC in this figure are usually restricted to IPC between processes on the same host. The final two rows,

sockets and streams, are the only two that are generally supported for IPC between processes on different hosts. (See Stevens [1990] for details on networked IPC.) Although the three forms of IPC in the middle of this figure (message queues, semaphores, and shared memory) are shown as being supported only by System V, in most vendor-supported Unix systems that are derived from Berkeley Unix (such as SunOS and Ultrix), support has been added by the vendors for these three forms of IPC.

> Work is underway in different POSIX groups on IPC, but the final outcome is far from clear. It appears that nothing final will come from POSIX regarding IPC until 1994 or later.

We have divided the discussion of IPC into two 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 some advanced features of IPC, supported by both SVR4 and 4.3+BSD: stream pipes, named stream pipes, and some of the things we can do with these more advanced forms of IPC.

## 14.2 Pipes

Pipes are the oldest form of Unix IPC and are provided by all Unix systems. They have two limitations:

1. They are half-duplex. Data flows only in one direction.
2. They 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 child.

We'll see that stream pipes (Section 15.2) get around the first limitation, and FIFOs (Section 14.5) and named stream pipes (Section 15.5) get around the second limitation. Despite these limitations, half-duplex pipes are still the most commonly used form of IPC.

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

```
#include <unistd.h>

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

Returns: 0 if OK, –1 on error

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

There are two ways to picture a pipe, as shown in Figure 14.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure reiterates the fact that the data in the pipe flows through the kernel.

> Under SVR4 a pipe is full duplex. Both descriptors can be written to and read from. The arrows in Figure 14.2 would have heads on both ends. We call these full-duplex pipes "stream

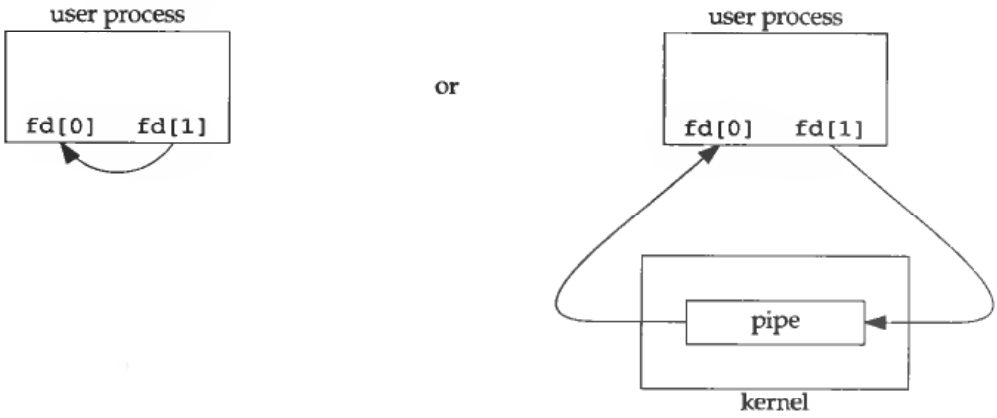


**Figure 14.2** Two ways to view a Unix pipe.

pipes" and discuss them in detail in the next chapter. Since POSIX.1 only provides half-duplex pipes, for portability we'll assume the `pipe` function creates a one-way 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 14.3 shows this scenario.

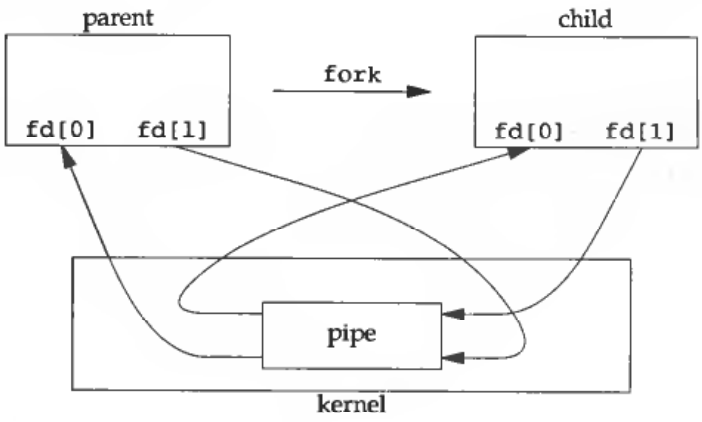


**Figure 14.3** Half-duplex pipe after a `fork`.

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 14.4 shows the resulting arrangement of descriptors.

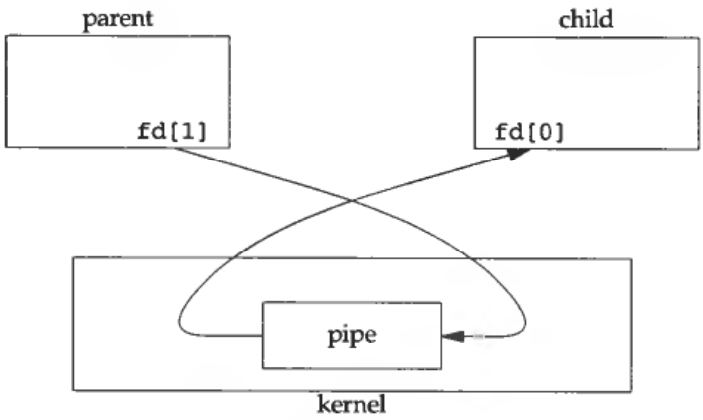


**Figure 14.4** Pipe from parent to child.

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, the following rules apply:

1. If we `read` from a pipe whose write end has been closed, after all the data has been read, `read` returns 0 to indicate an end of file. (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 an error 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 we `write` more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers.

### Example

Program 14.1 shows the code to create a pipe from the parent to the child, and send data down the pipe. □

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 `exec`s 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).

```
#include    "ourhdr.h"

int
main(void)
{
    int     n, 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);
}
```

**Program 14.1** Send data from parent to child over a pipe.

## Example

Consider a program that wants to display some output that it has created, one page at a time. Rather than reinvent the pagination done by several Unix 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. Program 14.2 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    <sys/wait.h>
#include    "ourhdr.h"

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

int
main(int argc, char *argv[])
```

```
{
    int     n, fd[2];
    pid_t   pid;
    char    line[MAXLINE], *pager, *argv0;
    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);
    }
}
```

**Program 14.2** 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 a 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` from Section 3.12 when its two arguments are equal.) 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 a descriptor onto another, as a defensive programming measure we'll always compare the descriptors first.

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 8.8. In Program 10.17 we showed an implementation using signals. Program 14.3 shows an implementation using pipes.

We create two pipes before the `fork`, as shown in Figure 14.5.

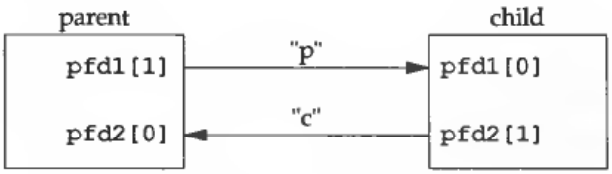


**Figure 14.5** Using two pipes for parent–child synchronization.

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.

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 also 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. □

```
#include    "ourhdr.h"

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

void
TELL_WAIT()
{
    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");
}
```

**Program 14.3** Routines to let a parent and child synchronize.

## 14.3 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: the creation of a pipe, the fork of a child, closing the unused ends of the pipe, execing a shell to execute 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 14.6).

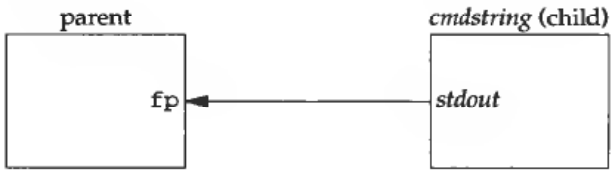


**Figure 14.6** Result of fp = popen(*command*, "r").

If *type* is "w", the file pointer is connected to the standard input of *cmdstring* (Figure 14.7).


**Figure 14.7** Result of fp = popen(*command*, "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. (The termination status is what we described in Section 8.6. This is what the system function (Section 8.12) also

returns.) 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");
```

popen and pclose are not specified by POSIX.1, since they interact with a shell, which is covered by POSIX.2. Our description of these functions corresponds to Draft 11.2 of POSIX.2. There are some differences between the proposed POSIX.2 specification and prior implementations.

```
#include    <sys/wait.h>
#include    "ourhdr.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);
}
```

**Program 14.4** Copy file to pager program using popen.

### Example

Let's redo Program 14.2 using popen. This is shown in Program 14.4. 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 nonnull, otherwise use the string more. □

### Example—popen Function

Program 14.5 shows our version of popen and pclose. 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.

Calling pipe, fork, and then duplicating the appropriate descriptors for each process is similar to what we've done earlier in this chapter.

POSIX.2 requires that popen close any streams in the child that are still open 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? waitpid 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 just call waitpid again if it is interrupted by a caught signal.

> Earlier versions of pclose returned an error of EINTR if a signal interrupted the wait.
>
> Earlier versions of pclose blocked or ignored the signals SIGINT, SIGQUIT, and SIGHUP during the wait. This is not allowed by POSIX.2. □

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

static pid_t    *childpid = NULL;
                        /* ptr to array allocated at run-time */
static int      maxfd;  /* from our open_max(), Program 2.3 */

#define SHELL   "/bin/sh"

FILE *
popen(const char *cmdstring, const char *type)
```

```
{
    int     i, pfd[2];
    pid_t   pid;
    FILE    *fp;

            /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;     /* required by POSIX.2 */
        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 ( (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(SHELL, "sh", "-c", cmdstring, (char *) 0);
        _exit(127);
    }
                                /* parent */
    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)
        return(-1);     /* popen() has never been called */

    fd = fileno(fp);
    if ( (pid = childpid[fd]) == 0)
        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 */
}
```

**Program 14.5** The popen and pclose functions.

## Example

Consider an application that writes a prompt to standard output and reads a line from standard input. With popen we can intersperse a program between the application and its input, to transform the input. Figure 14.8 shows the arrangement of processes.

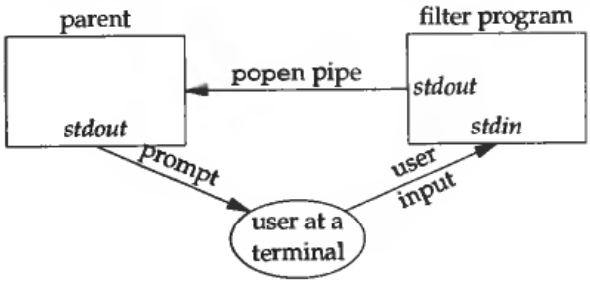


**Figure 14.8** Transforming input using popen.

```
#include    <ctype.h>
#include    "ourhdr.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);
}
```

**Program 14.6** Filter to convert uppercase characters to lowercase.

```
#include    <sys/wait.h>
#include    "ourhdr.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);
}
```

**Program 14.7** Invoke uppercase/lowercase filter to read commands.

The transformation could be pathname expansion, for example, or providing a history mechanism (remembering previously entered commands). (This example comes from the Rationale for popen in the POSIX.2 draft.)

Program 14.6 shows a simple filter to demonstrate this operation. It just 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.

We compile this filter into the executable file `myuclc`, which we then invoke from Program 14.7 using popen.

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. □

## 14.4 Coprocesses

A Unix 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 its input and reads its output.

The KornShell provides coprocesses [Bolsky and Korn 1989]. The Bourne shell and 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. 65–66 of Bolsky and Korn [1989] 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, 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 14.9 shows this arrangement.

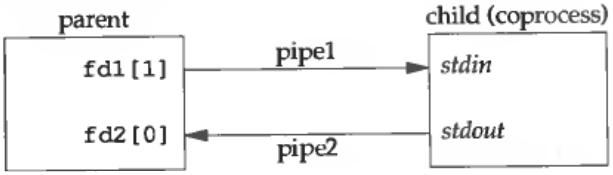


**Figure 14.9** Driving a coprocess by writing its standard input and reading its standard output.

Program 14.8 is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

```
#include    "ourhdr.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);
}
```

**Program 14.8** Simple filter to add two numbers.

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

Program 14.9 invokes the add2 coprocess, after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

```
#include    <signal.h>
#include    "ourhdr.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");
    }
}

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

**Program 14.9** Program to drive the add2 filter.

Here we create two pipes, with the parent and 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 Program 14.9, it works as expected. Furthermore, if we `kill` the add2 coprocess while Program 14.9 is waiting for our input, and then enter two numbers, when the program writes to the pipe that has no reader, the signal handler is invoked. (See Exercise 14.4.)

In Program 15.1 we provide another version of this example using a single full-duplex pipe instead of two half-duplex pipes. □

**Example**

In the coprocess add2 (Program 14.8) we purposely used Unix I/O: `read` and `write`. What happens if we rewrite this coprocess to use standard I/O? Program 14.10 shows the new version.

```
#include    "ourhdr.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);
}
```

**Program 14.10** Filter to add two numbers, using standard I/O.

If we invoke this new coprocess from Program 14.9 it no longer works. The problem is the default standard I/O buffering. When Program 14.10 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, `isatty` is false, and 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, Program 14.9 is blocked reading from the pipe. We have a deadlock.

Here we have control over the coprocess that's being `execed`. We can change Program 14.10 by adding the following four lines before the `while` loop is entered.

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

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

This causes the `fgets` to return when a line is available, and it causes `printf` to do an `fflush` when a newline is output. Making these explicit calls to `setvbuf` fixes Program 14.10.

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. □

## 14.5 FIFOs

FIFOs are sometimes called named pipes. 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 codings 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 filesystem.

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

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

Returns: 0 if OK, −1 on error

The specification of the *mode* argument for the `mkfifo` function 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.

Once we have created a FIFO using `mkfifo`, we open it using `open`. Indeed, the normal file I/O functions (`close`, `read`, `write`, `unlink`, etc.) all work with FIFOs.

The `mkfifo` function is an invention of POSIX.1. SVR3, for example, used the `mknod`(2) system call to create a FIFO. In SVR4 `mkfifo` just calls `mknod` to create the FIFO.

POSIX.2 has proposed a `mkfifo`(1) command. Both SVR4 and 4.3+BSD currently support this command. This allows a FIFO to be created using a shell command, and then accessed with the normal shell I/O redirection.

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

1. In the normal case (`O_NONBLOCK` not specified), 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.
2. If `O_NONBLOCK` is specified, an `open` for read-only returns immediately. But an `open` for write-only returns an error with an `errno` of `ENXIO` if no process has the FIFO open for reading.

Like 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 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 in a client–server application to pass data between the clients and server.

We discuss each of these 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 while pipes can be used only for linear connections between processes, since a FIFO has a name, it can be used for nonlinear connections.

Consider a procedure that needs to process a filtered input stream twice. Figure 14.10 shows this arrangement.

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 to the file named on its command line.)

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

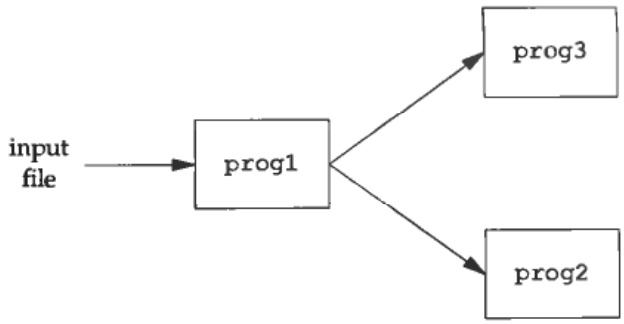


**Figure 14.10** Procedure that processes a filtered input stream twice.

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 14.11 shows the process arrangement pictorially.

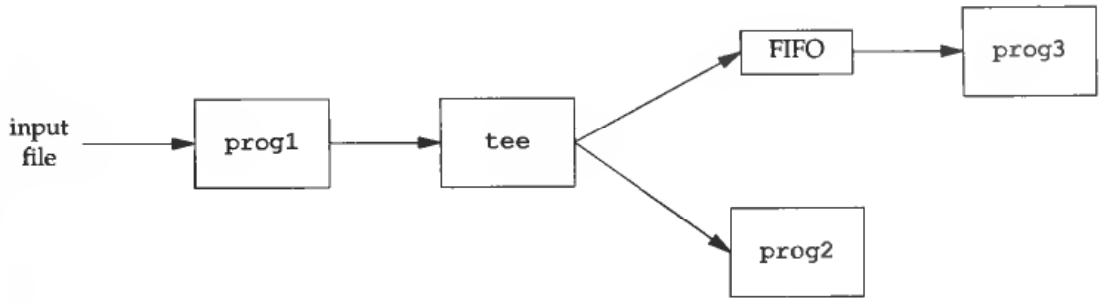


**Figure 14.11** Using a FIFO and `tee` to send a stream to two different processes.

□

## Example—Client–Server Communication Using a FIFO

Another use for FIFOs is to send data between a client and server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known 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 it.) Figure 14.12 shows this arrangement. 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

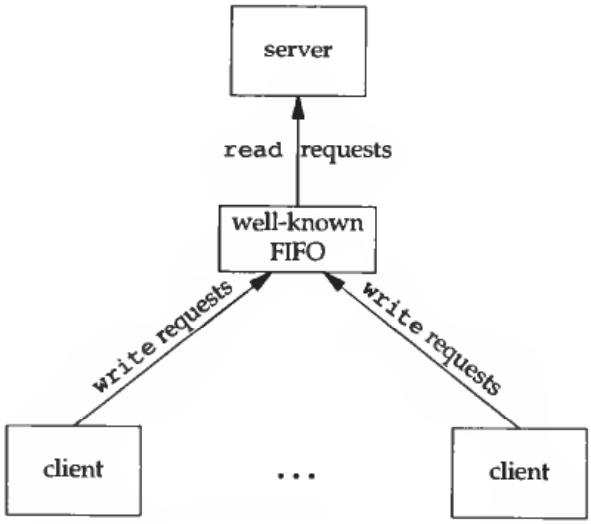


**Figure 14.12** Clients sending requests to a server using a FIFO.

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. Figure 14.13 shows this arrangement.

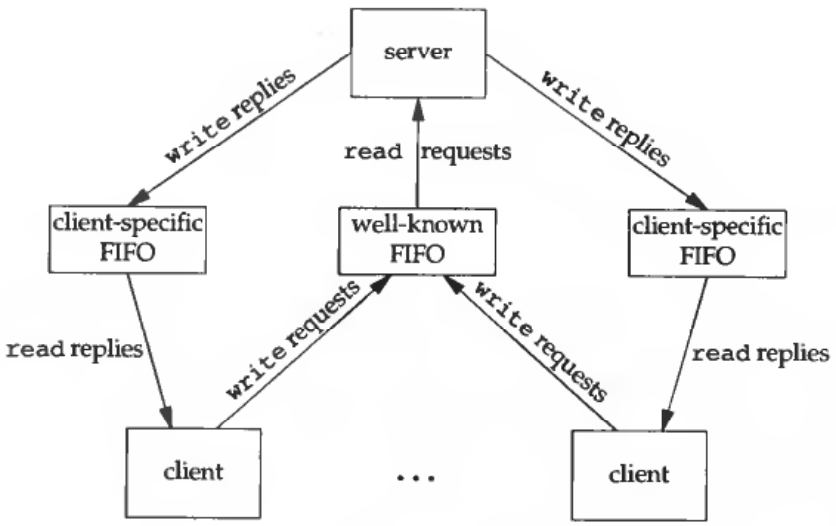


**Figure 14.13** Client–server communication using FIFOs.

This arrangement works, although it is impossible for the server to tell if a client crashes. This causes the client-specific FIFOs to be left in the filesystem. The 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 14.13, if the server opens its well-known FIFO read-only (since it only reads 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 having to handle this case, a common trick is just to have the server open its well-known FIFO for read–write. (See Exercise 14.10.) □

## 14.6 System V IPC

There are many similarities between the three types of IPC that we call System V IPC—message queues, semaphores, and shared memory. In this section we cover these similar features, before looking at the specific functions for each of the three IPC types in the following sections.

> These three types of IPC originated in the 1970s in an internal version of Unix called "Columbus Unix." These IPC features were later added to System V.

### 14.6.1 Identifiers and Keys

Each *IPC structure* (message queue, semaphore, or shared memory segment) in the kernel is referred to by a nonnegative integer *identifier*. To send or fetch a message to or 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. (This value that is remembered even after an IPC structure is deleted, and incremented each time the structure is used, is called the "slot usage sequence number." It is in the `ipc_perm` structure, which we show in the next section.)

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 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 brand new IPC structure. The disadvantage to this technique is that filesystem 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 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 server can agree on a pathname and project ID (the project ID is just a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. (The function `ftok` is described in the `stdipc`(3) manual page.) 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. Since the client and server typically share at least one header, an easier technique is to avoid using `ftok` and just store the well-known key in this header, avoiding yet another function.

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

1. *key* is `IPC_PRIVATE`, or
2. *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.)

### 14.6.2 Permission Structure

System V IPC associates an `ipc_perm` structure with each IPC structure. This structure defines the permissions and owner.

```
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 */
    ulong  seq;   /* slot usage sequence number */
    key_t  key;   /* key */
};
```

All the fields other than `seq` 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 either be the creator of the IPC structure, or it must be 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 4.4, but there is nothing corresponding to execute permission for any of the IPC structures. Also, whereas message queues and shared memory use the terms read and write, semaphores use the terms read and alter. Figure 14.14 specifies the six permissions for each form of IPC.

| Permission | Message queue | Semaphore | Shared memory |
|---|---|---|---|
| user-read | MSG_R | SEM_R | SHM_R |
| user-write (alter) | MSG_W | SEM_A | SHM_W |
| group-read | MSG_R >> 3 | SEM_R >> 3 | SHM_R >> 3 |
| group-write (alter) | MSG_W >> 3 | SEM_A >> 3 | SHM_W >> 3 |
| other-read | MSG_R >> 6 | SEM_R >> 6 | SHM_R >> 6 |
| other-write (alter) | MSG_W >> 6 | SEM_A >> 6 | SHM_W >> 6 |

**Figure 14.14** System V IPC permissions.

### 14.6.3 Configuration Limits

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

> Under SVR4 these values, and their minimum and maximum values, are in the file `/etc/conf/cf.d/mtune`.

### 14.6.4 Advantages and Disadvantages

A fundamental problem with System V 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 filesystem until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.

Another problem with System V IPC is that these IPC structures are not known by names in the filesystem. We can't access them and modify their properties with the functions we described in Chapters 3 and 4. Almost a dozen brand new system calls were added to the kernel to support them (`msgget`, `semop`, `shmat`, etc.). We can't see them 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, brand 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 with them: `select` and `poll`. 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 an actual transaction processing system built using System V IPC is given in Andrade, Carges, and Kovach [1989]. They claim that the name space used by System V IPC (the identifiers) is an advantage, and not a problem as we said earlier, because using identifiers allows a process to send a message to a message queue with just a single function call (`msgsnd`), while other forms of IPC normally require an `open`, `write`, and `close`. This argument is false. Somehow the clients still have to obtain the identifier for the server's queue, 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 14.6.1, minimally the server has to write the identifier assigned to a queue to a file for the clients to read.

Other advantages listed by these authors for message queues are that they're (a) reliable, (b) flow controlled, (c) record oriented, and (d) can be processed in other than first-in, first-out order. As we saw in Section 12.4, streams also possess all these properties, although an `open` is required before sending data to a stream, and a `close` is required when we're finished. Figure 14.15 compares some of the features of these different 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 stream socket | no | yes | yes | no | no |
| Unix datagram socket | yes | yes | no | yes | no |
| FIFOs | no | yes | yes | no | no |

**Figure 14.15** Comparison of features of different forms of IPC.

(We describe Unix stream and datagram sockets briefly in Chapter 15.) 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, the sender should automatically be awakened.

One feature that we don't show in Figure 14.15 is whether the IPC facility can automatically create a unique connection to a server for each client. We'll see in Chapter 15 that streams and Unix stream sockets provide this capability.

The next three sections describe each of the three forms of System V IPC in detail.

## 14.7 Message Queues

Message queues are 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 just a "queue ID." A new queue is created, or an existing queue is 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 nonnegative 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. This structure defines the current status of the queue.

```
struct msqid_ds {
  struct ipc_perm  msg_perm; /* see Section 14.6.2 */
  struct msg  *msg_first;  /* ptr to first message on queue */
  struct msg  *msg_last;   /* ptr to last message on queue */
  ulong        msg_cbytes; /* current # bytes on queue */
  ulong        msg_qnum;   /* # of messages on queue */
  ulong        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 */
};
```

The two pointers, `msg_first` and `msg_last` are worthless to a user process, as these point to where the corresponding messages are stored within the kernel. The remaining members of the structure are self-defining.

Figure 14.16 lists the system limits (Section 14.6.3) that affect message queues.

| Name | Description | Typical Value |
|---|---|---|
| MSGMAX | The size in bytes of the largest message we can send. | 2048 |
| MSGMNB | The maximum size in bytes of a particular queue (i.e., the sum of all the messages on the queue). | 4096 |
| MSGMNI | The maximum number of messages queues, systemwide. | 50 |
| MSGTQL | The maximum number of messages, systemwide. | 40 |

**Figure 14.16** System limits that affect message queues.

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

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

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

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

In Section 14.6.1 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 14.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the constants from Figure 14.14.
- `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 nonnegative queue ID. This value is then used with the other three message queue functions.

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

```
#include <sys/types.h>
#include <sys/ipc.h>
#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` Set the following four fields from the structure pointed to by *buf* in the 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/types.h>
#include <sys/ipc.h>
#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 nonnegative length (*nbytes*), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

*ptr* 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.

A *flag* value of `IPC_NOWAIT` can be specified. This is similar to the nonblocking I/O flag for file I/O (Section 12.2). 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 (a) there is room for the message, (b) the queue is removed from the system, or (c) a signal is caught and the signal handler returns. In the second case an error of EIDRM is returned ("identifier removed"), and in the last case the error returned is EINTR.

Notice 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 just generates errors on the next queue operation by processes still using the queue. Semaphores handle this removal in the same fashion. Removing a file doesn't delete the file's contents until the last process using the file closes it.

Messages are retrieved from a queue by msgrcv.

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

int 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*, the message is truncated if the MSG_NOERROR bit in *flag* is set. (In this case, no notification is given us that the message was truncated.) 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.

We can specify a *flag* value of IPC_NOWAIT to make the operation nonblocking. This causes msgrcv to return an error of ENOMSG if a message of the specified type is not available. If IPC_NOWAIT is not specified, we are blocked until (a) a message of the specified type is available, (b) the queue is removed from the system (an error of EIDRM is returned), or (c) a signal is caught and the signal handler returns (an error of EINTR is returned).

### Example—Timing Comparison of Message Queues versus Stream Pipes

If we need a bidirectional flow of data between a client and server, we can use either message queues or stream pipes. (We cover stream pipes in Section 15.2. They are similar to pipes but full duplex.)

Figure 14.17 shows a timing comparison of these two techniques, on two different systems. The test consisted of a program that created the IPC channel, called `fork`, and then sent 20 megabytes of data from the parent to the child. The data was sent using 10,000 calls to `msgsnd`, with a message length of 2,000 bytes, for the message queue, and 10,000 calls to `write`, with a length of 2,000 bytes, for the stream pipe. The times are all in seconds.

| | SPARC, SunOS 4.1.1 | | | 80386, SVR4 | | |
|---|---|---|---|---|---|---|
| Operation | User | System | Clock | User | System | Clock |
| message queue | 0.8 | 10.7 | 11.6 | 0.7 | 19.6 | 20.1 |
| stream pipe | 0.3 | 10.6 | 11.0 | 0.5 | 21.4 | 21.9 |

**Figure 14.17** Timing comparison of message queues and stream pipes.

On the SPARC, stream pipes are implemented using Unix domain sockets. Under SVR4 the `pipe` function provides stream pipes (using streams, as we described in Section 12.4).

What these numbers show us is that message queues, originally implemented to provide higher-than-normal speed IPC, are no longer any 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 14.6.4), we come to the conclusion that we shouldn't use them for new applications. □

## 14.8 Semaphores

A semaphore isn't really 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. 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. The process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. 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 of units of the shared resource are available for sharing.

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

1. A semaphore is not just a single nonnegative 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 System V 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.

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 14.6.2 */
  struct sem  *sem_base;  /* ptr to first semaphore in set */
  ushort       sem_nsems; /* # of semaphores in set */
  time_t       sem_otime; /* last-semop() time */
  time_t       sem_ctime; /* last-change time */
};
```

The `sem_base` pointer is worthless to a user process, since it points to memory in the kernel. What it points to is an array of `sem` structures, containing `sem_nsems` elements, one element in the array for each semaphore value in the set.

```
struct sem {
  ushort  semval;   /* semaphore value, always >= 0 */
  pid_t   sempid;   /* pid for last operation */
  ushort  semncnt;  /* # processes awaiting semval > currval */
  ushort  semzcnt;  /* # processes awaiting semval = 0 */
};
```

Figure 14.18 lists the system limits (Section 14.6.3) that affect semaphore sets.

| Name | Description | Typical Value |
|---|---|---|
| SEMVMX | The maximum value of any semaphore. | 32,767 |
| SEMAEM | The maximum value of any semaphore's adjust-on-exit value. | 16,384 |
| SEMMNI | The maximum number of semaphore sets, systemwide. | 10 |
| SEMMNS | The maximum number of semaphores, systemwide. | 60 |
| SEMMSL | The maximum number of semaphores per semaphore set. | 25 |
| SEMMNU | The maximum number of undo structures, systemwide. | 30 |
| SEMUME | The maximum number of undo entries per undo structures. | 10 |
| SEMOPN | The maximum number of operations per semop call. | 10 |

**Figure 14.18** System limits that affect semaphores.

The first function to call is `semget` to obtain a semaphore ID.

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

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

Returns: semaphore ID if OK, −1 on error

In Section 14.6.1 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 14.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the constants from Figure 14.14.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.

*nsems* is the number of semaphores in the set. If a new set is being created (typically in 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/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

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

Returns: (see following)

Notice that the final argument is the actual union, not a pointer to the union.

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

The *cmd* argument specifies one of the following 10 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 following three fields from the structure pointed to by *arg.buf* in the structure associated with this set: `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode`. 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.

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

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

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

Returns: 0 if OK, –1 on error

*semoparray* is a pointer to an array of semaphore operations.

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

*nops* 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 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 this means 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 that the resulting value for the semaphore 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):

   a. if `IPC_NOWAIT` is specified, return is made with an error of `EAGAIN`;
   b. if `IPC_NOWAIT` is not specified, the `semncnt` value for this semaphore is incremented (since we're 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 we're 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 we're no longer waiting) and the function returns an error of EINTR.

3. If sem_op is 0 this means we want 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:

   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 we're 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 we're 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 we're no longer waiting) and the function returns an error of EINTR.

The atomicity of semop is because it either does all the operations in the array or it does 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 voluntary or involuntary, the kernel checks to see if 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 versus Record Locking

If we are sharing a single resource among multiple processes, we can use either a semaphore or record locking. It's interesting to compare the timing differences between the two 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, and 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, and to release it we unlock the byte. The properties of record locking guarantee that any process that terminates while holding a lock, has the lock automatically released by the kernel.

Figure 14.19 shows the time required to perform these two locking techniques on two different systems. In each case the resource was allocated and then released 10,000 times. This was done simultaneously by three different processes. The times in Figure 14.19 are the totals in seconds for all three processes.

| Operation | SPARC, SunOS 4.1.1 | | | 80386, SVR4 | | |
|---|---|---|---|---|---|---|
| | User | System | Clock | User | System | Clock |
| semaphores with undo | 0.9 | 13.9 | 15.0 | 0.5 | 13.1 | 13.7 |
| advisory record locking | 1.1 | 15.2 | 16.5 | 2.1 | 20.6 | 22.9 |

**Figure 14.19** Timing comparison of semaphore locking and record locking.

On the SPARC, there is about a 10% penalty in the system time for record locking compared of semaphore locking. On the 80386 this penalty increases to about 50%.

Even though record locking is slightly slower than semaphore locking, if we're locking a single resource (such as a shared memory segment) and don't need all the fancy features of System V semaphores, record locking is preferred. The reasons are (a) it is much simpler to use, and (b) the system takes care of any lingering locks when a process terminates. □

## 14.9 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 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 can also be used.)

The kernel maintains the following structure for each shared memory segment.

```
struct shmid_ds {
  struct ipc_perm  shm_perm; /* see Section 14.6.2 */
  struct anon_map *shm_amp;  /* pointer in kernel */
  int      shm_segsz;   /* size of segment in bytes */
  ushort   shm_lkcnt;   /* number of times segment is being locked */
  pid_t    shm_lpid;    /* pid of last shmop() */
  pid_t    shm_cpid;    /* pid of creator */
  ulong    shm_nattch;  /* number of current attaches */
  ulong    shm_cnattch; /* used only for shminfo */
  time_t   shm_atime;   /* last-attach time */
  time_t   shm_dtime;   /* last-detach time */
  time_t   shm_ctime;   /* last-change time */
};
```

Figure 14.20 lists the system limits (Section 14.6.3) that affect shared memory.

| Name | Description | Typical Value |
|---|---|---|
| SHMMAX | The maximum size in bytes of a shared memory segment. | 131,072 |
| SHMMIN | The minimum size in bytes of a shared memory segment. | 1 |
| SHMMNI | The maximum number of shared memory segments, systemwide. | 100 |
| SHMSEG | The maximum number of shared memory segments, per process. | 6 |

**Figure 14.20** System limits that affect shared memory.

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

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

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

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

In Section 14.6.1 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 14.6.2. The mode member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the constants from Figure 14.14.
- `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` are all set to 0.
- `shm_ctime` is set to the current time.

*size* is the minimum size of the shared memory segment. If a new segment is being created (typically in the server) we must specify its *size*. If we are referencing an existing segment (a client) we can specify *size* as 0.

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

```
#include <sys/types.h>
#include <sys/ipc.h>
#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 structure associated with this 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 actually removed until the last process using the segment terminates or detaches it. Regardless whether the segment is still in use or not, 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. |
| 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/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, 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*.

1. If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
2. If *addr* is nonzero and SHM_RND is not specified, the segment is attached at the address given by *addr*.
3. 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 read-only. Otherwise the segment is attached read–write.

The value returned by shmat is the actual address that the segment is attached at, or –1 if an error occurred.

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

int shmdt(void *addr);
```

Returns: 0 if OK, –1 on error

The *addr* argument is the value that was returned by a previous call to shmat.

## Example

Where a kernel places shared memory segments that are attached with an address of 0 is highly system dependent. Program 14.11 prints some information on where one particular system places different types of data. Running this program on one particular system gives us the following output:

```
$ a.out
array[] from 18f48 to 22b88
stack around f7fffb2c
malloced from 24c28 to 3d2c8
shared memory attached from f77d0000 to f77e86a0
```

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

#define ARRAY_SIZE   40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    (SHM_R | SHM_W) /* user read/write */

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

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

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

    if ( (ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %x to %x\n", ptr, 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 %x to %x\n",
                shmptr, shmptr+SHM_SIZE);
    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

**Program 14.11** Print where different types of data are stored.

Figure 14.21 shows a picture of this, similar to what we said was a typical memory layout in Figure 7.3. Notice that the shared memory segment is placed well below the stack. In fact, there is about eight megabytes of unused address space between the shared memory segment and the stack. □

### Example—Memory Mapping of `/dev/zero`

Shared memory can be used between unrelated processes. But if the processes are related, SVR4 provides a different technique.

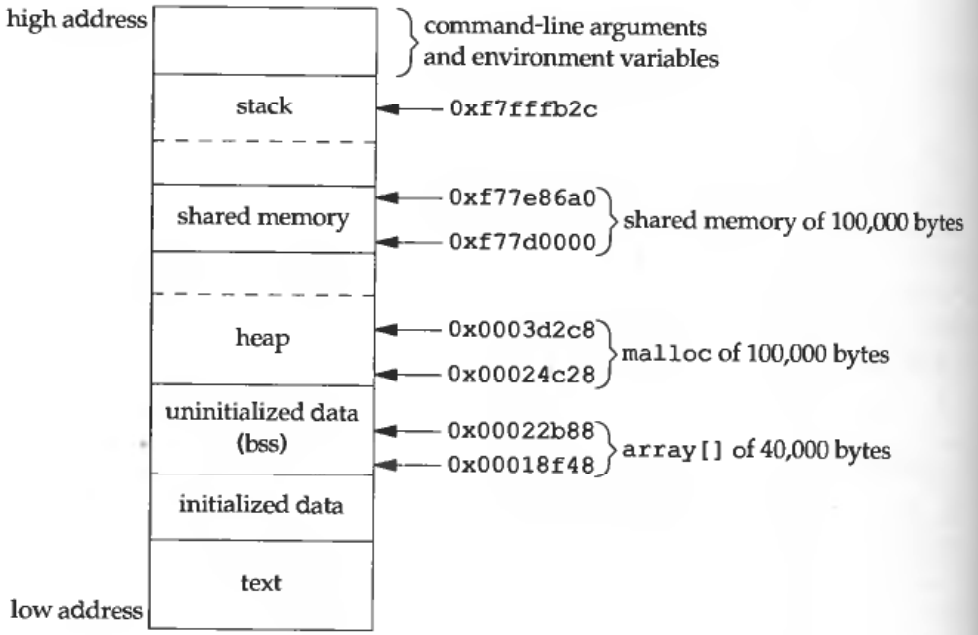


**Figure 14.21** Memory layout on one particular system.

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`.

Program 14.12 is an example that uses this special device. It opens the `/dev/zero` device and calls `mmap` specifying a size of a long integer. Notice 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 child then alternate running, incrementing a long integer in the shared memory mapped region, using the synchronization functions from Section 8.8. 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. Notice 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.

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

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

static int  update(long *);

int
main()
{
    int     fd, i, counter;
    pid_t   pid;
    caddr_t 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)) == (caddr_t) -1)
        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);
}
static int
update(long *ptr)
{
    return( (*ptr)++ ); /* return value before increment */
}
```

**Program 14.12** IPC between parent and child using memory mapped I/O of /dev/zero.

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 in this technique is that it works only between related processes. If shared memory is required between unrelated processes, the `shmXXX` functions must be used. □

### Example—Anonymous Memory Mapping

4.3+BSD provides a facility similar to the `/dev/zero` feature, called anonymous memory mapping. To use this feature 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.

To modify Program 14.12 to use this feature under 4.3+BSD we make two changes: (a) remove the open of `/dev/zero`, and (b) change the call to `mmap` to the following

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

In this call we specify the `MAP_ANON` flag, and set the file descriptor to –1. The rest of Program 14.12 is unchanged. □

## 14.10 Client–Server Properties

Let's detail some of the properties of clients and servers that are affected by the different types of IPC used between them.

The simplest type of relationship is to have the client `fork` and `exec` the desired server. Two one-way pipes can be created before the `fork` to allow data to be transferred in both directions. Figure 14.9 is an example of this. The server that is execed can be a set-user-ID program, giving it special privileges. Also, it can determine the real identity of the client by looking at its real user ID. (Recall from Section 8.9 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 in Section 15.4.) It opens files for the client, instead of client calling the `open` function. This way additional permission checking can be added, above and beyond the normal Unix 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 or not. 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 it can do is pass back the contents of the file to the parent. While this works fine for regular files, it can't be used for special device files, for example. What we would like to be able to do is have the server open the requested file and pass back the file descriptor. While 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 the next chapter).

The next type of server we showed in Figure 14.13. 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. 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 uses this form of client–server. The client is the `lp`(1) command and the server is the `lpsched` process. A single FIFO is used since the flow of data is only from the client to the server. Nothing is 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 who the message is for. 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. Alternately, 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` or `poll` work 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 shared memory is that only a single "message" can be in a shared memory segment at a time—similar to a message queue with a limit of one message per queue. For this reason shared memory IPC normally uses one shared memory segment per client.

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 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 15.5.2 to allow the server to identify the client. The same technique can be used with either FIFOs, message queues, semaphores, or shared memory. For the following description, assume FIFOs are being used, as in Figure 14.13. 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 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 assumption made by the server is 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 also 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 there are other fundamental security problems in the system.

To use this technique with System V 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 FIFO example, 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 15.5.1 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 SVR4 when file descriptors are passed between processes.

## 14.11 Summary

We've detailed numerous forms of interprocess communication: pipes, named pipes (FIFOs), and the three forms of IPC commonly called System V IPC—message queues, semaphores, and shared memory. 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 the pitfalls that can be encountered with the standard I/O library's buffering.

After comparing the timing of message queues versus stream pipes, and semaphores versus record locking, we can make the following recommendations: learn pipes and FIFOs, since there are numerous applications where these two basic techniques can still be used effectively. Avoid using message queues and semaphores in any new applications. Stream pipes and record locking should be considered instead, as they integrate with the rest of the Unix kernel far better. Shared memory still has its use, although the `mmap` function (Section 12.9) may assume some of its capabilities in future releases.

In the next chapter we look at some advanced forms of IPC that are provided with newer systems, such as SVR4 and 4.3+BSD.

## Exercises

**14.1** In Program 14.2, at the end of the parent code, remove the `close` right before the `waitpid`. Explain what happens.

**14.2** In Program 14.2, at the end of the parent code, remove the `waitpid`. Explain what happens.

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

**14.4** In Program 14.9 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`?

**14.5** In Program 14.9 use the standard I/O library for reading and writing the pipes instead of `read` and `write`.

**14.6** The Rationale for POSIX.1 gives as one of the reasons for adding the `waitpid` function the fact 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?

**14.7** Explain how `select` and `poll` handle an input descriptor that is a pipe, when the pipe is closed by the writer. Write two small test programs, one using `select` and one using `poll` to determine the answer.

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

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

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

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

**14.11** 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'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?

**14.12** 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.

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

**14.14** Draw a time line of Program 14.12 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 the child runs first after the `fork`.

**14.15** Redo Program 14.12 using the `shmXXX` functions from Section 14.9 instead of the shared memory mapped region.

**14.16** Redo Program 14.12 using the System V semaphore functions from Section 14.8 to alternate between the parent and child.

**14.17** Redo Program 14.12 using advisory record locking to alternate between the parent and child.

**14.18** Explain how the file descriptor argument for `mmap` can be used with 4.3+BSD anonymous memory mapping to allow unrelated processes to share memory.

# 15

# Advanced Interprocess Communication

## 15.1 Introduction

In the previous chapter we looked at the classical methods of IPC provided by various Unix systems: pipes, FIFOs, message queues, semaphores, and shared memory. In this chapter we look at some advanced forms of IPC and what we can do with them: stream pipes and named stream pipes. With these two forms of IPC we can pass open file descriptors between processes, and clients can rendezvous with a daemon server with the system providing a unique IPC channel per client. These advanced forms of IPC were provided with 4.2BSD and SVR3.2, but have not been widely documented or used. Many of the ideas in this chapter come from the paper by Presotto and Ritchie [1990].

## 15.2 Stream Pipes

A stream pipe is just a bidirectional (full-duplex) pipe. To obtain bidirectional data flow between a parent and child, only a single stream pipe is required. Figure 15.1 shows the two ways to view a stream pipe. The only difference between this picture and Figure 14.2 is that the arrows have heads on both ends, since the stream pipe is full duplex.

### Example

Let's redo the coprocess example, Program 14.9, with a single stream pipe. Program 15.1 is the new `main` function. The `add2` coprocess is the same (Program 14.8). We call a new function, `s_pipe`, to create a single stream pipe. (We show versions of this function for SVR4 and 4.3+BSD in the following sections.)

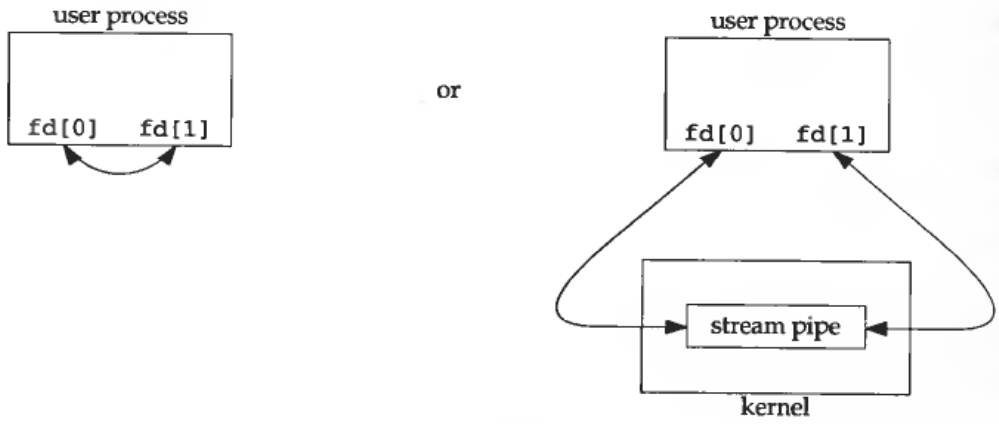


**Figure 15.1** Two ways to view a stream pipe.

```
#include    <signal.h>
#include    "ourhdr.h"

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

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

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

    if (s_pipe(fd) < 0)         /* only need a single stream pipe */
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid > 0) {                         /* parent */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("write error to pipe");
            if ( (n = read(fd[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(fd[0]);
        if (fd[1] != STDIN_FILENO) {
            if (dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
        }
        if (fd[1] != STDOUT_FILENO) {
            if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
        }
        if (execl("./add2", "add2", NULL) < 0)
            err_sys("execl error");
    }
}

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

**Program 15.1** Program to drive the add2 filter, using a stream pipe.

The parent uses only `fd[0]` and the child uses only `fd[1]`. Since each end of the stream pipe is full duplex, the parent reads and writes `fd[0]` and the child duplicates `fd[1]` to both standard input and standard output. Figure 15.2 shows the resulting descriptors.

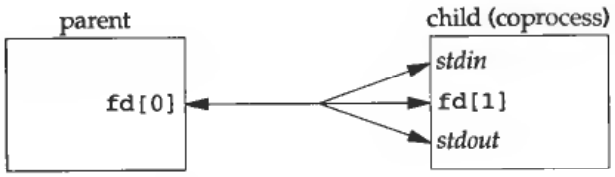


**Figure 15.2** Arrangement of descriptors for coprocess.

□

We define the function `s_pipe` to be similar to the standard `pipe` function. It takes the same argument as `pipe`, but the returned descriptors are open for reading and writing.

### Example—s_pipe Function Under SVR4

Program 15.2 shows the SVR4 version of the s_pipe function. It just calls the standard pipe function, which creates a full-duplex pipe.

```
#include    "ourhdr.h"

int
s_pipe(int fd[2]) /* two file descriptors returned in fd[0] & fd[1] */
{
    return( pipe(fd) );
}
```

**Program 15.2** SVR4 version of the s_pipe function.

Stream pipes can also be created under earlier versions of System V, but it takes more work. See Section 7.9 of Stevens [1990] for the details involved under SVR3.2.

Figure 15.3 shows what a pipe looks like under SVR4. It is just two stream heads that are connected to each other.

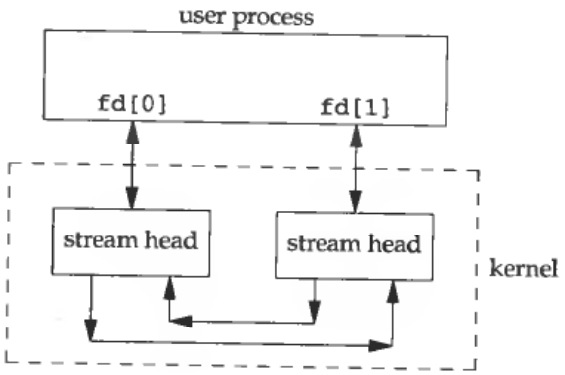


**Figure 15.3** Arrangement of a pipe under SVR4.

Since a pipe is a streams device, we can push processing modules onto either end of the pipe. In Section 15.5.1 we'll do this to provide a named stream that can be mounted. □

### Example—s_pipe Function Under 4.3+BSD

Program 15.3 shows the BSD version of the s_pipe function. This function works under 4.2BSD and any later versions. It creates a pair of connected Unix domain stream sockets.

Normal pipes have been implemented in this fashion since 4.2BSD. But when pipe is called, the write end of the first descriptor and the read end of the second descriptor are both closed. To get a full-duplex pipe we must call socketpair directly. □

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    "ourhdr.h"

int
s_pipe(int fd[2]) /* two file descriptors returned in fd[0] & fd[1] */
{
    return( socketpair(AF_UNIX, SOCK_STREAM, 0, fd) );
}
```

**Program 15.3** BSD version of the `s_pipe` function.

## 15.3 Passing File Descriptors

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing client–server applications. It allows one process (typically a server) to do everything that is required to open a file (involving details such as translation of a network name to a network address, dialing a modem, negotiating locks for the file, etc.) and just pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are transparent to the client.

> 4.2BSD supported the passing of open descriptors, but there were some bugs in the implementation. 4.3BSD fixed these bugs. SVR3.2 and above also support the passing of open descriptors.

We must be more specific about what we mean by "passing an open file descriptor" from one process to another. Recall Figure 3.3 where we showed two processes that have opened the same file. Although they share the same v-node table, each process has its own file table entry.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to also share the same file table entry. Figure 15.4 shows the desired arrangement. Technically, we are really passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.) Having two processes share an open file table is exactly what happens after a `fork` (recall Figure 8.1).

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

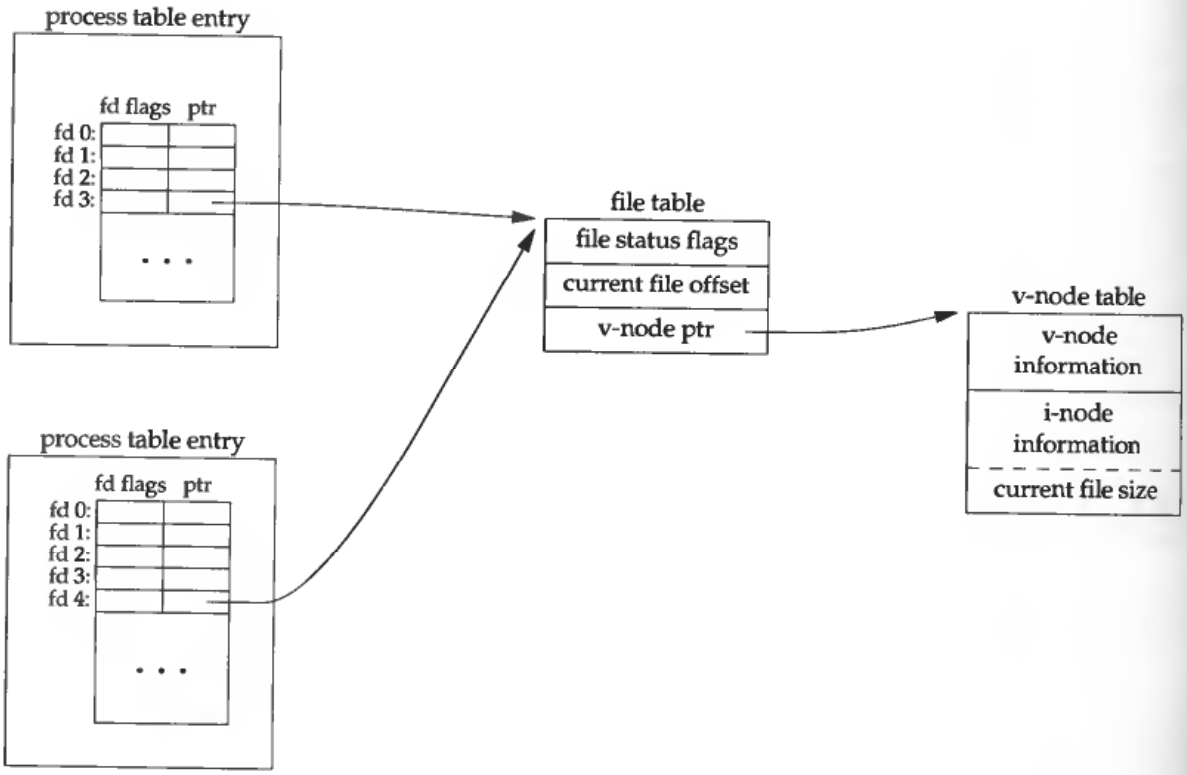


**Figure 15.4** Passing an open file from the top process to the bottom process.

We define the following three functions that we use in this chapter (and in Chapter 18) to send and receive file descriptors. Later in this section we'll show the actual code for these three functions, for both SVR4 and 4.3+BSD.

```
#include "ourhdr.h"

int send_fd(int spipefd, int filedes);

int send_err(int spipefd, int status, const char *errmsg);
```

Both return: 0 if OK, –1 on error

```
int recv_fd(int spipefd, ssize_t (*userfunc)(int, const void *, size_t));
```

Returns: file descriptor if OK, <0 on error

When a process (normally a server) wants to pass a descriptor to another process it calls either `send_fd` or `send_err`. The process waiting to receive the descriptor (the client) calls `recv_fd`.

`send_fd` sends the descriptor *filedes* across the stream pipe *spipefd*. `send_err` sends the *errmsg* across the stream pipe *spipefd*, followed by the *status* byte. The value of *status* must be in the range –1 through –255.

`recv_fd` is called by the client to receive a descriptor. If all is OK (the sender called `send_fd`), the nonnegative descriptor is returned as the value of the function. Otherwise the value returned is the *status* that was sent by `send_err` (a negative value in the range –1 through –255). Additionally, if an error message was sent by the server, the client's *userfunc* is called to process the message. The first argument to *userfunc* is the constant `STDERR_FILENO`, followed by a pointer to the error message and its length. Often the client specifies the normal Unix `write` function as the *userfunc*.

We implement our own protocol that is used by these three functions. To send a descriptor, `send_fd` sends two bytes of 0, followed by the actual descriptor. To send an error, `send_err` sends the *errmsg*, followed by a byte of 0, followed by the absolute value of the *status* byte (1–255). `recv_fd` just reads everything on the stream pipe until it encounters a null byte. Any characters read up to this point are passed to the caller's *userfunc*. The next byte read by `recv_fd` is the status byte. If the status byte is 0, a descriptor was passed, otherwise there is no descriptor to receive.

The function `send_err` just calls the `send_fd` function, after writing the error message to the stream pipe. This is shown in Program 15.4.

```
#include    "ourhdr.h"

/* Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead.  We send the error back
 * using the send_fd()/recv_fd() protocol. */

int
send_err(int clifd, int errcode, const char *msg)
{
    int     n;

    if ( (n = strlen(msg)) > 0)
        if (writen(clifd, msg, n) != n) /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;   /* must be negative */

    if (send_fd(clifd, errcode) < 0)
        return(-1);

    return(0);
}
```

**Program 15.4** The `send_err` function.

The following three sections look at the actual implementation of the two functions `send_fd` and `recv_fd` under SVR4, 4.3BSD, and 4.3+BSD.

### 15.3.1 System V Release 4

Under SVR4 file descriptors are exchanged on a stream pipe using two `ioctl` commands: `I_SENDFD` and `I_RECVFD`. To send a descriptor we just set the third argument for `ioctl` to the actual descriptor. This is shown in Program 15.5.

```
#include    <sys/types.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
    char    buf[2];      /* send_fd()/recv_fd() 2-byte protocol */

    buf[0] = 0;          /* null byte flag to recv_fd() */
    if (fd < 0) {
        buf[1] = -fd;    /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        buf[1] = 0;      /* zero status means OK */
    }

    if (write(clifd, buf, 2) != 2)
        return(-1);

    if (fd >= 0)
        if (ioctl(clifd, I_SENDFD, fd) < 0)
            return(-1);
    return(0);
}
```

**Program 15.5** The send_fd function for SVR4.

When we receive a descriptor the third argument for `ioctl` is a pointer to a `strrecvfd` structure.

```
struct strrecvfd {
  int     fd;       /* new descriptor */
  uid_t   uid;      /* effective user ID of sender */
  gid_t   gid;      /* effective group ID of sender */
  char    fill[8];
};
```

`recv_fd` just reads the stream pipe until the first byte of the two-byte protocol (the null byte) is received. When we issue the `ioctl` of `I_RECVFD` the next message at the stream's read head must be a descriptor from a `I_SENDFD`, or we get an error. This is shown in Program 15.6.

```
#include    <sys/types.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
```

```
 * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
 * 2-byte protocol for receiving the fd from send_fd(). */

int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int                 newfd, nread, flag, status;
    char                *ptr, buf[MAXLINE];
    struct strbuf       dat;
    struct strrecvfd    recvfd;

    status = -1;
    for ( ; ; ) {
        dat.buf = buf;
        dat.maxlen = MAXLINE;
        flag = 0;
        if (getmsg(servfd, NULL, &dat, &flag) < 0)
            err_sys("getmsg error");
        nread = dat.len;
        if (nread == 0) {
            err_ret("connection closed by server");
            return(-1);
        }
            /* See if this is the final data with null & status.
               Null must be next to last byte of buffer, status
               byte is last byte.  Zero status means there must
               be a file descriptor to receive. */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("message format error");
                status = *ptr & 255;
                if (status == 0) {
                    if (ioctl(servfd, I_RECVFD, &recvfd) < 0)
                        return(-1);
                    newfd = recvfd.fd;  /* new descriptor */
                } else
                    newfd = -status;
                nread -= 2;
            }
        }
        if (nread > 0)
            if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                return(-1);

        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

**Program 15.6** The recv_fd function for SVR4.

### 15.3.2 4.3BSD

Unfortunately, we have to provide different implementations for 4.3BSD (and vendor's systems built on 4.3BSD, such as SunOS and Ultrix), and later versions starting with 4.3BSD Reno.

To exchange file descriptors we call the `sendmsg`(2) and `recvmsg`(2) functions. Both functions take a pointer to a `msghdr` structure that contains all the information on what to send or receive. This structure is defined in the `<sys/socket.h>` header and under 4.3BSD it looks like

```
struct msghdr {
  caddr_t       msg_name;     /* optional address */
  int           msg_namelen;  /* size of address */
  struct iovec *msg_iov;      /* scatter/gather array */
  int           msg_iovlen;   /* # elements in msg_iov array */
  caddr_t       msg_accrights;    /* access rights sent/received */
  int           msg_accrightslen; /* size of access rights buffer */
};
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram. The next two elements allow us to specify an array of buffers (scatter read or gather write) as we described for the `readv` and `writev` functions (Section 12.7). The final two elements deal with the passing or receiving of access rights. The only access rights currently defined are file descriptors. Access rights can be passed only across a Unix domain socket (i.e., what we use as stream pipes under 4.3BSD). To send or receive a file descriptor we set `msg_accrights` to point to the integer descriptor and `msg_accrightslen` to be the length of the descriptor (i.e., the size of an integer). A descriptor is passed or received only if this length is nonzero.

Program 15.7 is the `send_fd` function for 4.3BSD.

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <errno.h>
#include    <stddef.h>
#include    "ourhdr.h"

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
```

```
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;

    if (fd < 0) {
        msg.msg_accrights    = NULL;
        msg.msg_accrightslen = 0;
        buf[1] = -fd;    /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        msg.msg_accrights    = (caddr_t) &fd;    /* addr of descriptor */
        msg.msg_accrightslen = sizeof(int);      /* pass 1 descriptor */
        buf[1] = 0;      /* zero status means OK */
    }
    buf[0] = 0;          /* null byte flag to recv_fd() */

    if (sendmsg(clifd, &msg, 0) != 2)
        return(-1);

    return(0);
}
```

**Program 15.7** The send_fd function for 4.3BSD.

In the sendmsg call we send both the two bytes of protocol data (the null and the status byte) and the descriptor.

To receive a file descriptor we read from the stream pipe until we read the null byte that precedes the final status byte. Everything up to this null byte is an error message from the sender. This is shown in Program 15.8.

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <stddef.h>
#include    "ourhdr.h"

/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
 * 2-byte protocol for receiving the fd from send_fd(). */

int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int             newfd, nread, status;
    char            *ptr, buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;
```

```
        status = -1;
        for ( ; ; ) {
            iov[0].iov_base = buf;
            iov[0].iov_len  = sizeof(buf);
            msg.msg_iov     = iov;
            msg.msg_iovlen  = 1;
            msg.msg_name    = NULL;
            msg.msg_namelen = 0;
            msg.msg_accrights = (caddr_t) &newfd;/* addr of descriptor */
            msg.msg_accrightslen = sizeof(int);  /* receive 1 descriptor */

            if ( (nread = recvmsg(servfd, &msg, 0)) < 0)
                err_sys("recvmsg error");
            else if (nread == 0) {
                err_ret("connection closed by server");
                return(-1);
            }

                /* See if this is the final data with null & status.
                   Null must be next to last byte of buffer, status
                   byte is last byte.  Zero status means there must
                   be a file descriptor to receive. */
            for (ptr = buf; ptr < &buf[nread]; ) {
                if (*ptr++ == 0) {
                    if (ptr != &buf[nread-1])
                        err_dump("message format error");
                    status = *ptr & 255;
                    if (status == 0) {
                        if (msg.msg_accrightslen != sizeof(int))
                            err_dump("status = 0 but no fd");
                        /* newfd = the new descriptor */
                    } else
                        newfd = -status;
                    nread -= 2;
                }
            }
            if (nread > 0)
                if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                    return(-1);

            if (status >= 0)    /* final data has arrived */
                return(newfd);  /* descriptor, or -status */
        }
    }
```

**Program 15.8** The `recv_fd` function for 4.3BSD.

Notice that we are always prepared to receive a descriptor (we set `msg_accrights` and `msg_accrightslen` before each call to `recvmsg`), but only if `msg_accrightslen` is nonzero on return did we receive a descriptor.

### 15.3.3 4.3+BSD

Starting with 4.3BSD Reno the definition of the msghdr structure changed. The final two elements, which were called "access rights" in previous releases, became "ancillary data." Also, a new member, msg_flags, was added to end of the structure.

```
struct msghdr {
  caddr_t       msg_name;       /* optional address */
  int           msg_namelen;    /* size of address */
  struct iovec *msg_iov;        /* scatter/gather array */
  int           msg_iovlen;     /* # elements in msg_iov array */
  caddr_t       msg_control;    /* ancillary data */
  u_int         msg_controllen; /* size of ancillary data */
  int           msg_flags;      /* flags on received message */
};
```

The msg_control field now points to a cmsghdr (control message header) structure.

```
struct cmsghdr {
  u_int  cmsg_len;   /* data byte count, including header */
  int    cmsg_level; /* originating protocol */
  int    cmsg_type;  /* protocol-specific type */
  /* followed by the actual control message data */
};
```

To send a file descriptor we set cmsg_len to the size of the cmsghdr structure, plus the size of an integer (the descriptor). cmsg_level is set to SOL_SOCKET, and cmsg_type is set to SCM_RIGHTS, to indicate that we are passing access rights. ("SCM" stands for "socket-level control message.") The actual descriptor is stored right after the cmsg_type field, using the macro CMSG_DATA to obtain the pointer to this integer. Program 15.9 shows the send_fd function for 4.3BSD Reno.

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <errno.h>
#include    <stddef.h>
#include    "ourhdr.h"

static struct cmsghdr   *cmptr = NULL;  /* buffer is malloc'ed first time */
#define CONTROLLEN  (sizeof(struct cmsghdr) + sizeof(int))
        /* size of control buffer to send/recv one file descriptor */

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */
```

```
    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    if (fd < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = (caddr_t) cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd;      /* the fd to pass */
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_fd() */

    if (sendmsg(clifd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

**Program 15.9** The send_fd function for 4.3BSD Reno.

To receive a descriptor (Program 15.10) we allocate enough room for a cmsghdr structure and a descriptor, set msg_control to point to the allocated area, and call recvmsg.

```
#include    <sys/types.h>
#include    <sys/socket.h>      /* struct msghdr */
#include    <sys/uio.h>         /* struct iovec */
#include    <stddef.h>
#include    "ourhdr.h"

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */
#define CONTROLLEN  (sizeof(struct cmsghdr) + sizeof(int))
        /* size of control buffer to send/recv one file descriptor */
/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
 * 2-byte protocol for receiving the fd from send_fd(). */
int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
```

```
{
    int             newfd, nread, status;
    char            *ptr, buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;

    status = -1;
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = (caddr_t) cmptr;
        msg.msg_controllen = CONTROLLEN;

        if ( (nread = recvmsg(servfd, &msg, 0)) < 0)
            err_sys("recvmsg error");
        else if (nread == 0) {
            err_ret("connection closed by server");
            return(-1);
        }
            /* See if this is the final data with null & status.
               Null must be next to last byte of buffer, status
               byte is last byte.  Zero status means there must
               be a file descriptor to receive. */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("message format error");
                status = *ptr & 255;
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr); /* new descriptor */
                } else
                    newfd = -status;
                nread -= 2;
            }
        }
        if (nread > 0)
            if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

**Program 15.10** The `recv_fd` function for 4.3BSD Reno.

## 15.4 An Open Server, Version 1

Using file descriptor passing, we now develop an open server: an executable program that is execed by a process to open one or more files. But instead of the server sending the file back to the calling process, it sends back an open file descriptor instead. This lets the server work with any type of file (such as a modem line or a network connection) and not just regular files. It also means that a minimum of information is exchanged using IPC—the filename and open mode from the client to the server, and the returned descriptor from the server to the client. The contents of the file are not exchanged using IPC.

There are several advantages in designing the server to be a separate executable program (either one that is execed by the client, as we develop in this section, or a daemon server, which we develop in Section 15.6).

1. The server can easily be contacted by any client, similar to a library function. We are not hardcoding a particular service into the application, but designing a general facility that others can reuse.
2. If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor). Shared libraries can simplify this updating (Section 7.7).
3. The server can be a set-user-ID program, providing it additional permissions that the client does not have. Notice that a library function (or shared library function) can't provide this capability.

The client process creates a stream pipe, and then calls `fork` and `exec` to invoke the server. The client sends requests across the stream pipe, and the server sends back responses across the pipe. We define the following application protocol between the client and server.

1. The client sends a request of the form

   open <*pathname*> <*openmode*>\0

   across the stream pipe to the server. The <*openmode*> is the numeric value, in decimal, of the second argument to the `open` function. This request string is terminated by a null byte.
2. The server sends back an open descriptor or an error by calling either `send_fd` or `send_err`.

This is an example of a process sending an open descriptor to its parent. In Section 15.6 we'll modify this example to use a single daemon server, where the server sends a descriptor to a completely unrelated process.

We first have the header, `open.h` (Program 15.11), which includes the standard system headers and defines the function prototypes.

```
#include    <sys/types.h>
#include    <errno.h>
#include    "ourhdr.h"

#define CL_OPEN "open"          /* client's request for server */

                /* our function prototypes */
int     csopen(char *, int);
```

**Program 15.11** The open.h header.

The main function (Program 15.12) is a loop that reads a pathname from standard input and copies the file to standard output. It calls the function csopen to contact the open server, and return an open descriptor.

```
#include    "open.h"
#include    <fcntl.h>

#define BUFFSIZE    8192

int
main(int argc, char *argv[])
{
    int     n, fd;
    char    buf[BUFFSIZE], line[MAXLINE];

                /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        line[strlen(line) - 1] = 0; /* replace newline with null */

                /* open the file */
        if ( (fd = csopen(line, O_RDONLY)) < 0)
            continue;   /* csopen() prints error from server */

                /* and cat to stdout */
        while ( (n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }

    exit(0);
}
```

**Program 15.12** The main function.

The function csopen (Program 15.13) does the fork and exec of the server, after creating the stream pipe.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/* Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back. */

int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      fd[2] = { -1, -1 };

    if (fd[0] < 0) {    /* fork/exec our open server first time */
        if (s_pipe(fd) < 0)
            err_sys("s_pipe error");
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid == 0) {        /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO) {
                if (dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                    err_sys("dup2 error to stdin");
            }
            if (fd[1] != STDOUT_FILENO) {
                if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                    err_sys("dup2 error to stdout");
            }
            if (execl("./opend", "opend", NULL) < 0)
                err_sys("execl error");
        }
        close(fd[1]);               /* parent */
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len)
        err_sys("writev error");

        /* read descriptor, returned errors handled by write() */
    return( recv_fd(fd[0], write) );
}
```

**Program 15.13** The csopen function.

The child closes one end of the pipe, and the parent closes the other. The child also duplicates its end of the pipe onto its standard input and standard output, for the server that it execs. (Another option would have been to pass the ASCII representation of the descriptor `fd[1]` as an argument to the server.)

The parent sends the request to the server containing the pathname and open mode. Finally the parent calls `recv_fd` to return either the descriptor or an error. If an error is returned by the server, `write` is called to output the message to standard error.

Now let's look at the open server. It is the program opend that is execed by the client in Program 15.13. First we have the `opend.h` header (Program 15.14) that includes the system headers and declares the global variables and function prototypes.

```
#include    <sys/types.h>
#include    <errno.h>
#include    "ourhdr.h"

#define CL_OPEN "open"              /* client's request for server */

            /* declare global variables */
extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open() flag: O_xxx ... */
extern char *pathname;  /* of file to open() for client */

            /* function prototypes */
int      cli_args(int, char **);
void     request(char *, int, int);
```

**Program 15.14** The `opend.h` header.

```
#include    "opend.h"

            /* define global variables */
char     errmsg[MAXLINE];
int      oflag;
char    *pathname;

int
main(void)
{
    int     nread;
    char    buf[MAXLINE];

    for ( ; ; ) {   /* read arg buffer from client, process request */
        if ( (nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break;      /* client has closed the stream pipe */

        request(buf, nread, STDIN_FILENO);
    }
    exit(0);
}
```

**Program 15.15** The `main` function.

The main function (Program 15.15) reads the requests from the client on the stream pipe (its standard input) and calls the function request.

The function request in Program 15.16 does all the work. It calls the function buf_args to break up the client's request into a standard argv-style argument list and calls the function cli_args to process the client's arguments. If all is OK, open is called to open the file, and then send_fd sends the descriptor back to the client across the stream pipe (its standard output). If an error is encountered, send_err is called to send back an error message, using the client–server protocol that we described earlier.

```
#include    "opend.h"
#include    <fcntl.h>

void
request(char *buf, int nread, int fd)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request not null terminated: %*.*s\n",
                                    nread, nread, buf);
        send_err(STDOUT_FILENO, -1, errmsg);
        return;
    }

            /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(STDOUT_FILENO, -1, errmsg);
        return;
    }

    if ( (newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                                pathname, strerror(errno));
        send_err(STDOUT_FILENO, -1, errmsg);
        return;
    }

            /* send the descriptor */
    if (send_fd(STDOUT_FILENO, newfd) < 0)
        err_sys("send_fd error");
    close(newfd);       /* we're done with descriptor */
}
```

**Program 15.16** The request function.

The client's request is a null terminated string of white-space separated arguments. The function buf_args in Program 15.17 breaks this string into a standard argv-style argument list and calls a user function to process the arguments. We'll use this function

later in this chapter and again in Chapter 18. We use the ANSI C function `strtok` to tokenize the string into separate arguments.

```
#include    "ourhdr.h"

#define MAXARGC     50  /* max number of arguments in buf */
#define WHITE   " \t\n" /* white space for tokenizing arguments */

/* buf[] contains white-space separated arguments.  We convert it
 * to an argv[] style array of pointers, and call the user's
 * function (*optfunc)() to process the argv[] array.
 * We return -1 to the caller if there's a problem parsing buf,
 * else we return whatever optfunc() returns.  Note that user's
 * buf[] array is modified (nulls placed after each token). */

int
buf_args(char *buf, int (*optfunc)(int, char **))
{
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL)     /* an argv[0] is required */
        return(-1);
    argv[argc = 0] = buf;

    while ( (ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)    /* -1 for room for NULL at end */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    return( (*optfunc)(argc, argv) );
            /* Since argv[] pointers point into the user's buf[],
               user's function can just copy the pointers, even
               though argv[] array will disappear on return. */
}
```

**Program 15.17** The `buf_args` function.

The server's function that is called by `buf_args` is `cli_args` (Program 15.18). It verifies that the client sent the right number of arguments and stores the pathname and open mode into global variables.

This completes the open server that is invoked by a `fork` and `exec` from the client. A single stream pipe is created before the `fork` and used to communicate between the client and server. With this arrangement we have one server per client.

After looking at client–server connections in the next section, we'll redo the open server in Section 15.6 to use a single daemon server that is contacted by all clients.

```
#include    "opend.h"

/* This function is called by buf_args(), which is called by
 * request().  buf_args() has broken up the client's buffer
 * into an argv[] style array, which we now process. */

int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }

    pathname = argv[1];     /* save ptr to pathname to open */
    oflag = atoi(argv[2]);
    return(0);
}
```

**Program 15.18** The `cli_args` function.

## 15.5 Client–Server Connection Functions

Stream pipes are useful for IPC between related processes, such as a parent and child. The open server in the previous section was able to pass file descriptors from a child to a parent using an unnamed stream pipe. But when we're dealing with unrelated processes (such as a server that is a daemon), a named stream pipe is required.

We can take an unnamed stream pipe (from the `s_pipe` function) and attach a pathname in the filesystem to either end. A daemon server would create just one end of a stream pipe and attach a name to that end. This way unrelated clients can rendezvous with the daemon, sending messages to the server's end of the pipe. This is similar to what we showed in Figure 14.12, where we used a well-known FIFO for the clients to send their requests to.

An even better approach is to use a technique whereby the server creates one end of a stream pipe with a well-known name, and clients *connect* to that end. Additionally, each time a new client connects to the server's named stream pipe, a brand new stream pipe is created between the client and server. This way the server is notified each time a new client connects to the server, and when any client terminates. Both SVR4 and 4.3+BSD support this form of IPC. In this section we develop three functions that can be used by a client–server to establish these per-client connections.

```
#include "ourhdr.h"

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

Returns: file descriptor to listen on if OK, <0 on error

First a server has to announce its willingness to listen for client connections on a well-known name (some pathname in the filesystem) by calling `serv_listen`. *name* is the well-known name of the server. Clients will use this name when they want to connect to the server. The return value is the file descriptor for the server's end of the named stream pipe.

Once a server has called `serv_listen`, it calls `serv_accept` to wait for a client connection to arrive.

```
#include "ourhdr.h"

int serv_accept(int listenfd, uid_t *uidptr);
```

Returns: new file descriptor if OK, <0 on error

*listenfd* is a descriptor from `serv_listen`. This function doesn't return until a client connects to the server's well-known name. When the client does connect to the server, a brand new stream pipe is automatically created, and the new descriptor is returned as the value of the function. Additionally, the effective user ID of the client is stored through the pointer *uidptr*.

A client just calls `cli_conn` to connect to a server.

```
#include "ourhdr.h"

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

Returns: file descriptor if OK, <0 on error

The *name* specified by the client must be the same name that was advertised by the server's call to `serv_listen`. The returned descriptor refers to a stream pipe that is connected to the server.

Using these three functions we can write server daemons that can manage any number of clients. The only limit is the number of descriptors available to a single process, since the server requires one descriptor for each client connection. Since these functions deal with normal file descriptors, the server can multiplex I/O requests among all its clients using either `select` or `poll`. Finally, since the client–server connections are all stream pipes, open descriptors can be passed across the connections.

In the next two sections we'll look at the implementations of these three functions under SVR4 and 4.3+BSD. Then in Section 15.6 we'll redo the open server from Section 15.4 using a single daemon server that uses these three functions. We'll also use these three functions in Chapter 18 when we develop a general connection server.

### 15.5.1 System V Release 4

SVR4 provides mounted streams and a streams processing module named `connld` that we can use to provide a named stream pipe with unique connections for the server.

> Mounted streams and the `connld` module were developed by Presotto and Ritchie [1990] for the Research Unix system. They were then picked up by SVR4.

First the server creates an unnamed stream pipe and pushes the streams processing module `connld` on one end. Figure 15.5 shows the resulting picture.

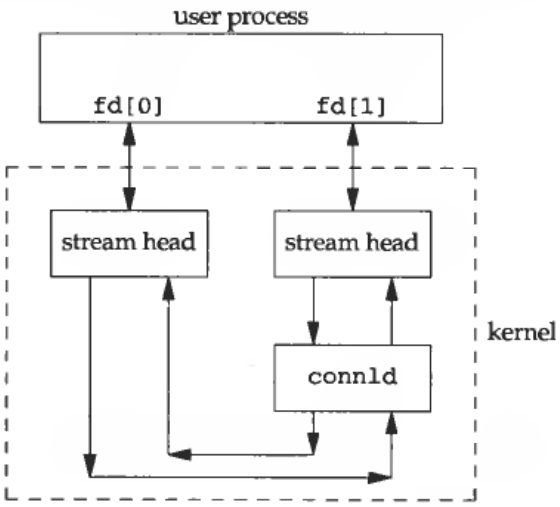


**Figure 15.5** SVR4 pipe after pushing `connld` module onto one end.

We then attach a pathname to the end of the pipe that has the `connld` pushed onto it. SVR4 provides the `fattach` function to do this. Any process that `open`s this pathname (such as a client) is referring to the named end of the pipe.

Program 15.19 shows the dozen lines of code required to implement the `serv_listen` function.

When another process calls `open` for the named end of the pipe (the end with `connld` pushed onto it), the following occurs:

1. A new pipe is created.
2. One descriptor for the new pipe is passed back to the client as the return value from `open`.
3. The other descriptor is passed to the server on the other end of the named pipe (i.e., the end that does not have `connld` pushed onto it). The server receives this new descriptor using an `ioctl` of `I_RECVFD`.

Assume that the well-known name that the server `fattaches` to its pipe is `/tmp/serv1`. Figure 15.6 shows the resulting picture, after the client's call

```
fd = open("/tmp/serv1", O_RDWR);
```

has returned. The pipe between the client and server is created by the `open`, since the pathname being opened is really a named stream that has `connld` pushed onto it. The file descriptor in the client (`fd`) is returned by the `open`. The new file descriptor in the server (`clifd1`) is received by the server using an `ioctl` of `I_RECVFD` on the descriptor `fd[0]`. Once the server has pushed `connld` onto `fd[1]` and attached a name to `fd[1]`, it never specifically uses `fd[1]` again.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <stropts.h>
#include    "ourhdr.h"

#define FIFO_MODE   (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
                    /* user rw, group rw, others rw */

int         /* returns fd if all OK, <0 on error */
serv_listen(const char *name)
{
    int     tempfd, fd[2], len;

                    /* create a file: mount point for fattach() */
    unlink(name);
    if ( (tempfd = creat(name, FIFO_MODE)) < 0)
        return(-1);
    if (close(tempfd) < 0)
        return(-2);

    if (pipe(fd) < 0)
        return(-3);
                            /* push connld & fattach() on fd[1] */
    if (ioctl(fd[1], I_PUSH, "connld") < 0)
        return(-4);
    if (fattach(fd[1], name) < 0)
        return(-5);

    return(fd[0]);  /* fd[0] is where client connections arrive */
}
```

**Program 15.19** The `serv_listen` function for SVR4.

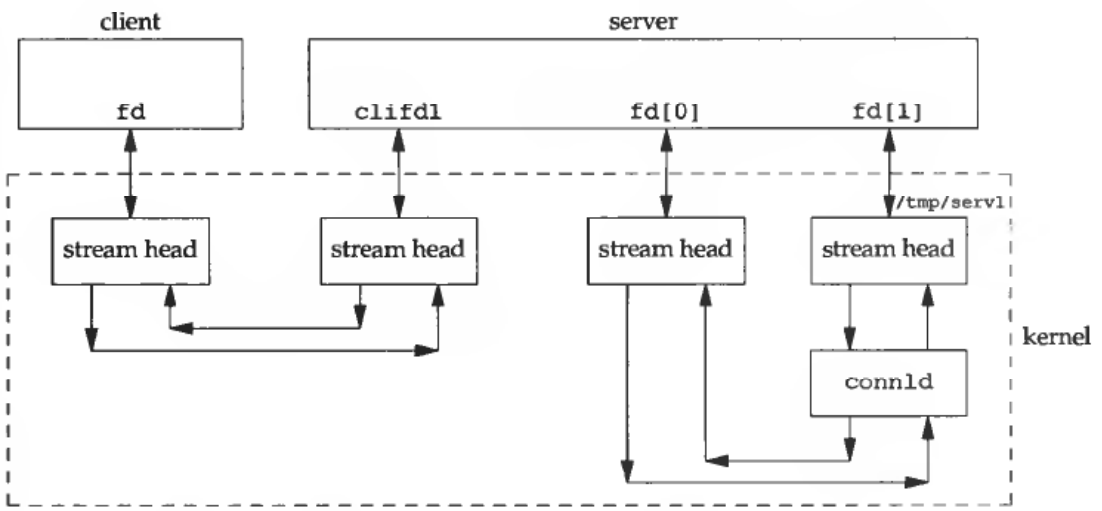


**Figure 15.6** Client–server connection on a named stream pipe.

The server waits for a client connection to arrive by calling the `serv_accept` function shown in Program 15.20.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID. */

int         /* returns new fd if all OK, -1 on error */
serv_accept(int listenfd, uid_t *uidptr)
{
    struct strrecvfd    recvfd;

    if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
        return(-1);     /* could be EINTR if signal caught */

    if (uidptr != NULL)
        *uidptr = recvfd.uid;   /* effective uid of caller */

    return(recvfd.fd);  /* return the new descriptor */
}
```

**Program 15.20** The `serv_accept` function for SVR4.

In Figure 15.6 the first argument to `serv_accept` would be the descriptor `fd[0]` and the return value from `serv_accept` would be the descriptor `clifd1`.

The client initiates the connection to the server by calling the `cli_conn` function in Program 15.21.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

/* Create a client endpoint and connect to a server. */

int         /* returns fd if all OK, <0 on error */
cli_conn(const char *name)
{
    int     fd;

                /* open the mounted stream */
    if ( (fd = open(name, O_RDWR)) < 0)
        return(-1);
    if (isastream(fd) == 0)
        return(-2);

    return(fd);
}
```

**Program 15.21** The `cli_conn` function for SVR4.

We double-check that the returned descriptor refers to a streams device, in case the server has not been started but the pathname still existed in the filesystem. (Under SVR4 there appears to be little reason to call `cli_conn` instead of just calling `open` directly. In the next section we'll see that the `cli_conn` function is more complicated under BSD systems.)

### 15.5.2 4.3+BSD

Under 4.3+BSD we have a different set of operations required to connect a client and server using Unix domain sockets. We won't go through all the details of the `socket`, `bind`, `listen`, `accept`, and `connect` functions that we use, since most of the details are for using these functions with other networking protocols. Refer to Chapter 6 of Stevens [1990] for these details.

> Since SVR4 also supports Unix domain sockets, the code shown in this section also works under SVR4.

Program 15.22 shows the `serv_listen` function. It is the first function called by the server.

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <sys/un.h>
#include    "ourhdr.h"

/* Create a server endpoint of a connection. */

int         /* returns fd if all OK, <0 on error */
serv_listen(const char *name)
{
    int                 fd, len;
    struct sockaddr_un  unix_addr;

                        /* create a Unix domain stream socket */
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    unlink(name);       /* in case it already exists */

                        /* fill in socket address structure */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    strcpy(unix_addr.sun_path, name);
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
          strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else               /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
#endif
```

```
                    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-2);

    if (listen(fd, 5) < 0)  /* tell kernel we're a server */
        return(-3);

    return(fd);
}
```

**Program 15.22** The `serv_listen` function for 4.3+BSD.

First a single Unix domain socket is created by `socket`. We then fill in a `sockaddr_un` structure with the well-known pathname to be assigned to the socket. This structure is the argument to `bind`. We then call `listen` to tell the kernel that we'll be a server awaiting connections from clients. (The second argument to `listen`, 5, is the maximum number of outstanding connection requests that the kernel will queue for this descriptor. Most implementations silently enforce an upper limit of 5 for this value.)

The client initiates the connection to the server by calling the `cli_conn` function (Program 15.23).

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <sys/stat.h>
#include    <sys/un.h>
#include    "ourhdr.h"

/* Create a client endpoint and connect to a server. */

#define CLI_PATH    "/var/tmp/"     /* +5 for pid = 14 chars */
#define CLI_PERM    S_IRWXU         /* rwx for user only */

int         /* returns fd if all OK, <0 on error */
cli_conn(const char *name)
{
    int                 fd, len;
    struct sockaddr_un  unix_addr;

                /* create a Unix domain stream socket */
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

                /* fill socket address structure w/our address */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    sprintf(unix_addr.sun_path, "%s%05d", CLI_PATH, getpid());
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
          strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else               /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
```

```
    if (len != 16)
        err_quit("length != 16");    /* hack */
#endif

    unlink(unix_addr.sun_path);      /* in case it already exists */
    if (bind(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-2);
    if (chmod(unix_addr.sun_path, CLI_PERM) < 0)
        return(-3);

                /* fill socket address structure w/server's addr */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    strcpy(unix_addr.sun_path, name);
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
          strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else               /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
#endif

    if (connect(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-4);

    return(fd);
}
```

**Program 15.23** The `cli_conn` function for 4.3+BSD.

We call `socket` to create the client's end of a Unix domain socket. We then fill in a `sockaddr_un` structure with a client-specific name. The last five characters of the pathname are the process ID of the client. (We also verify that the size of this structure is exactly 14 characters to avoid some bugs in earlier implementations of Unix domain sockets.) `unlink` is called, just in case the pathname already exists. We call `bind` to assign a name to the client's socket, and this creates the pathname in the filesystem, and the file type is a socket. `chmod` is called to turn off all permissions other than user-read, user-write, and user-execute. In `serv_accept`, the server checks these permissions and the user ID of the socket to verify the client's identity.

We then have to fill in another `sockaddr_un` structure, this time with the well-known pathname of the server. Finally the `connect` function initiates the connection with the server.

The creation of a unique connection for each client is handled in the `serv_accept` function (Program 15.24) by the `accept` function.

```
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <sys/stat.h>
#include    <sys/un.h>
#include    <stddef.h>
#include    <time.h>
```

```
#include    "ourhdr.h"

#define STALE    30  /* client's name can't be older than this (sec) */

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID from the pathname
 * that it must bind before calling us. */

int         /* returns new fd if all OK, <0 on error */
serv_accept(int listenfd, uid_t *uidptr)
{
    int                 clifd, len;
    time_t              staletime;
    struct sockaddr_un  unix_addr;
    struct stat         statbuf;

    len = sizeof(unix_addr);
    if ( (clifd = accept(listenfd, (struct sockaddr *) &unix_addr, &len)) <
        return(-1);     /* often errno=EINTR, if signal caught */

            /* obtain the client's uid from its calling address */
#ifdef  SCM_RIGHTS  /* 4.3BSD Reno and later */
    len -= sizeof(unix_addr.sun_len) - sizeof(unix_addr.sun_family);
#else               /* vanilla 4.3BSD */
    len -= sizeof(unix_addr.sun_family);    /* len of pathname */
#endif
    unix_addr.sun_path[len] = 0;            /* null terminate */

    if (stat(unix_addr.sun_path, &statbuf) < 0)
        return(-2);
#ifdef  S_ISSOCK    /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0)
        return(-3);     /* not a socket */
#endif
    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statbuf.st_mode & S_IRWXU) != S_IRWXU)
          return(-4);   /* is not rwx------ */

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
        statbuf.st_ctime < staletime ||
        statbuf.st_mtime < staletime)
          return(-5);   /* i-node is too old */

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid;   /* return uid of caller */

    unlink(unix_addr.sun_path);     /* we're done with pathname now */

    return(clifd);
}
```

**Program 15.24** The serv_accept function for 4.3+BSD.

The server blocks in the call to `accept`, waiting for a client to call `cli_conn`. When `accept` returns, its return value is a brand new descriptor that is connected to the client. (This is somewhat similar to what the `connld` module does under SVR4.) Additionally, the pathname that the client assigned to its socket (the name that contained the client's process ID) is also returned by `accept`, through the second argument (the pointer to the `sockaddr_un` structure). We null terminate this pathname and call `stat`. This lets us verify that the pathname is indeed a socket, and that the permissions allow only user-read, user-write, and user-execute. We also verify that the three times associated with the socket are no older than 30 seconds. (The `time` function returns the current time and date in seconds past the Unix Epoch.) If all these checks are OK, we assume that the identity of the client (its effective user ID) is the owner of the socket. While this check isn't perfect, it's the best we can do with current systems. (It would be better if the kernel returned the effective user ID to `accept` as the SVR4 `I_RECVFD` does.)

Figure 15.7 shows a picture of the connection, after the call to `cli_conn` has returned, assuming the server's well-known name is `/tmp/serv1`. Compare this with Figure 15.6.

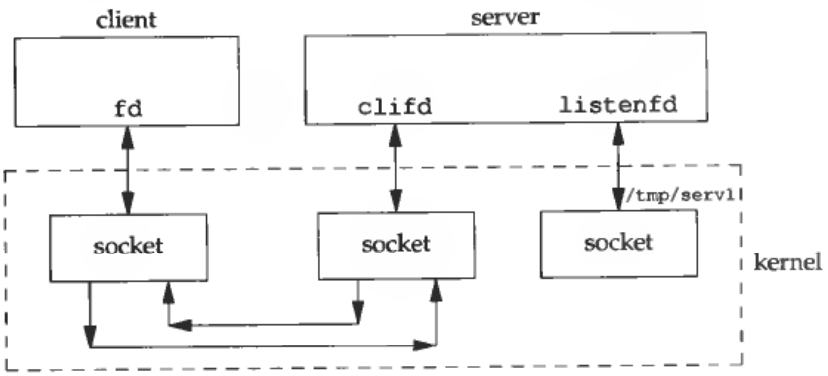


**Figure 15.7** Client–server connection on a Unix domain socket.

## 15.6 An Open Server, Version 2

In Section 15.4 we developed an open server that was invoked by a `fork` and `exec` by the client. It demonstrated how we can pass file descriptors from a child to a parent. In this section we develop an open server as a daemon process. One server handles all clients. We expect this design to be more efficient, since a `fork` and `exec` are avoided. We still use a stream pipe between the client and server and demonstrate passing file descriptors between unrelated processes. We'll use the three functions `serv_listen`, `serv_accept`, and `cli_conn` from the previous section. This server also demonstrates how a single server can handle multiple clients, using both the `select` and `poll` functions from Section 12.5.

The client is similar to the client from Section 15.4. Indeed, the file `main.c` is identical (Program 15.12). We add the following line to the `open.h` header (Program 15.11)

```
#define CS_OPEN "/home/stevens/open"  /* server's well-known name */
```

The file `open.c` does change from Program 15.13, since we now call `cli_conn`, instead of doing the `fork` and `exec`. This is shown in Program 15.25.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/* Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back. */

int
csopen(char *name, int oflag)
{
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      csfd = -1;

    if (csfd < 0) {     /* open connection to conn server */
        if ( (csfd = cli_conn(CS_OPEN)) < 0)
            err_sys("cli_conn error");
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;
                            /* null at end of buf always sent */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(csfd, &iov[0], 3) != len)
        err_sys("writev error");

                    /* read back descriptor */
                    /* returned errors handled by write() */
    return( recv_fd(csfd, write) );
}
```

**Program 15.25** The csopen function.

The protocol from the client to the server remains the same.

Let's look at the server. The header `opend.h` (Program 15.26) includes the standard headers, declares the global variables and the function prototypes.

Since this server handles all clients, it must maintain the state of each client connection. This is done with the `client` array defined in the `opend.h` header. Program 15.27 defines three functions that manipulate this array.

```
#include    <sys/types.h>
#include    <errno.h>
#include    "ourhdr.h"

#define CS_OPEN "/home/stevens/opend" /* well-known name */
#define CL_OPEN "open"                /* client's request for server */

            /* declare global variables */
extern int   debug;     /* nonzero if interactive (not daemon) */
extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open flag: O_xxx ... */
extern char *pathname;  /* of file to open for client */

typedef struct {    /* one Client struct per connected client */
  int   fd;     /* fd, or -1 if available */
  uid_t uid;
} Client;

extern Client   *client;        /* ptr to malloc'ed array */
extern int       client_size;   /* # entries in client[] array */
                    /* (both manipulated by client_XXX() functions) */

            /* function prototypes */
int      cli_args(int, char **);
int      client_add(int, uid_t);
void     client_del(int);
void     loop(void);
void     request(char *, int, int, uid_t);
```

**Program 15.26** The opend.h header.

```
#include    "opend.h"

#define NALLOC  10      /* #Client structs to alloc/realloc for */

static void
client_alloc(void)      /* alloc more entries in the client[] array */
{
    int     i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size + NALLOC) * sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

            /* have to initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;  /* fd of -1 means entry available */

    client_size += NALLOC;
}
```

```
/* Called by loop() when connection request from a new client arrives */

int
client_add(int fd, uid_t uid)
{
    int     i;

    if (client == NULL)     /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) {   /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i);  /* return index in client[] array */
        }
    }
            /* client array full, time to realloc for more */
    client_alloc();
    goto again;     /* and search again (will work this time) */
}

/* Called by loop() when we're done with a client */

void
client_del(int fd)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}
```

**Program 15.27** Functions to manipulate `client` array.

The first time `client_add` is called, it calls `client_alloc`, which calls `malloc` to allocate space for 10 entries in the array. After these 10 entries are all in use, a later call to `client_add` causes `realloc` to allocate additional space. By dynamically allocating space this way, we have not limited the size of the `client` array at compile time to some value that we guessed and put into a header.

These functions call the `log_` functions (Appendix B) if an error occurs, since we assume that the server is a daemon.

The `main` function (Program 15.28) defines the global variables, processes the command-line options, and calls the function `loop`. If we invoke the server with the `-d` option, it runs interactively instead of as a daemon. This is used when testing the server.

```
#include    "opend.h"
#include    <syslog.h>

            /* define global variables */
int       debug;
char      errmsg[MAXLINE];
int       oflag;
char     *pathname;
Client   *client = NULL;
int       client_size;

int
main(int argc, char *argv[])
{
    int     c;

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemon_init();

    loop();     /* never returns */
}
```

**Program 15.28** The `main` function.

The function `loop` is the server's infinite loop. We'll show two versions of this function. Program 15.29 shows one that uses `select` (and works under both 4.3+BSD and SVR4), then we show one that uses `poll` (for SVR4).

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

void
loop(void)
{
    int     i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;
```

```
    FD_ZERO(&allset);

                /* obtain fd to listen for client requests on */
    if ( (listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    FD_SET(listenfd, &allset);
    maxfd = listenfd;
    maxi = -1;

    for ( ; ; ) {
        rset = allset;      /* rset gets modified each time around */
        if ( (n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
            log_sys("select error");

        if (FD_ISSET(listenfd, &rset)) {
                    /* accept new client request */
            if ( (clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            FD_SET(clifd, &allset);
            if (clifd > maxfd)
                maxfd = clifd;  /* max fd for select() */
            if (i > maxi)
                maxi = i;       /* max index in client[] array */
            log_msg("new connection: uid %d, fd %d", uid, clifd);
            continue;
        }

        for (i = 0; i <= maxi; i++) {   /* go through client[] array */
            if ( (clifd = client[i].fd) < 0)
                continue;
            if (FD_ISSET(clifd, &rset)) {
                        /* read argument buffer from client */
                if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                    log_sys("read error on fd %d", clifd);
                else if (nread == 0) {
                    log_msg("closed: uid %d, fd %d",
                                        client[i].uid, clifd);
                    client_del(clifd);  /* client has closed conn */
                    FD_CLR(clifd, &allset);
                    close(clifd);
                } else          /* process client's rquest */
                    request(buf, nread, clifd, client[i].uid);
            }
        }
    }
}
```

**Program 15.29** The loop function using `select`.

This function calls `serv_listen` to create the server's endpoint for the client connections. The remainder of the function is a loop that starts with a call to `select`. Two conditions can be true after `select` returns.

1. The descriptor `listenfd` can be ready for reading, which means a new client has called `cli_conn`. To handle this we call `serv_accept` and then update the `client` array and associated bookkeeping information for the new client. (We keep track of the highest descriptor number, for the first argument to `select`. We also keep track of the highest index in the `client` array that's in use.)
2. An existing client's connection can be ready for reading. This means one of two things: (a) the client has terminated, or (b) the client has sent a new request.

   We find out about a client termination by `read` returning 0 (end of file). If `read` returns greater than 0, there is a new request to process. We call `request` to handle the new client request.

We keep track of which descriptors are currently in use in the `allset` descriptor set. As new clients connect to the server, the appropriate bit is turned on in this descriptor set. The appropriate bit is turned off when the client terminates.

We always know when a client terminates, whether the termination is voluntary or not, since all the client's descriptors (including the connection to the server) are automatically closed by the kernel. This differs from the System V IPC mechanisms.

The `loop` function that uses the `poll` function is shown in Program 15.30.

```
#include    "opend.h"
#include    <poll.h>
#include    <stropts.h>

void
loop(void)
{
    int             i, n, maxi, listenfd, clifd, nread;
    char            buf[MAXLINE];
    uid_t           uid;
    struct pollfd   *pollfd;

    if ( (pollfd = malloc(open_max() * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");

                /* obtain fd to listen for client requests on */
    if ( (listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0);     /* we use [0] for listenfd */
    pollfd[0].fd = listenfd;
    pollfd[0].events = POLLIN;
    maxi = 0;
```

```
    for ( ; ; ) {
        if ( (n = poll(pollfd, maxi + 1, INFTIM)) < 0)
            log_sys("select error");

        if (pollfd[0].revents & POLLIN) {
                    /* accept new client request */
            if ( (clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            pollfd[i].fd = clifd;
            pollfd[i].events = POLLIN;
            if (i > maxi)
                maxi = i;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i <= maxi; i++) {
            if ( (clifd = client[i].fd) < 0)
                continue;
            if (pollfd[i].revents & POLLHUP)
                goto hungup;
            else if (pollfd[i].revents & POLLIN) {
                        /* read argument buffer from client */
                if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                    log_sys("read error on fd %d", clifd);
                else if (nread == 0) {
    hungup:
                    log_msg("closed: uid %d, fd %d",
                                        client[i].uid, clifd);
                    client_del(clifd);  /* client has closed conn */
                    pollfd[i].fd = -1;
                    close(clifd);
                } else          /* process client's rquest */
                    request(buf, nread, clifd, client[i].uid);
            }
        }
    }
}
```

**Program 15.30** The `loop` function using `poll`.

To allow for as many clients as there are open descriptors, we dynamically allocate space for the array of `pollfd` structures. (The function `open_max` was shown in Program 2.3.)

We use the zeroth entry of the `client` array for the `listenfd` descriptor. That way a client's index in the `client` array is the same index that we use in the `pollfd` array. The arrival of a new client connection is indicated by a `POLLIN` on the `listenfd` descriptor. As before, we call `serv_accept` to accept the connection.

For an existing client we have to handle two different events from `poll`: a client termination is indicated by `POLLHUP` and a new request from an existing client is indicated

by POLLIN. Recall from Exercise 14.7 that the hangup message can arrive at the stream head while there is still data to be read from the stream. With a pipe we want to read all the data before processing the hangup. But with this server, when we receive the hangup from the client, we can close the connection (the stream) to the client, effectively throwing away any data still on the stream. This is because there is no reason to process any requests still on the stream, since we can't send any responses back.

As with the select version of this function, new requests from a client are handled by calling the request function (Program 15.31). This function is similar to the earlier version (Program 15.16). It calls the same function buf_args (Program 15.17) that calls cli_args (Program 15.18).

```
#include    "opend.h"
#include    <fcntl.h>

void
request(char *buf, int nread, int clifd, uid_t uid)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request from uid %d not null terminated: %*.*s\n",
                                    uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("request: %s, from uid %d", buf, uid);

            /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ( (newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                                pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

            /* send the descriptor */
    if (send_fd(clifd, newfd) < 0)
        log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd);       /* we're done with descriptor */
}
```

**Program 15.31** The request function.

This completes the open server, using a single daemon to handle all the client requests.

## 15.7 Summary

The key points in this chapter are the ability to pass file descriptors between processes and the ability of a server to accept unique connections from clients. We've seen how to do this under SVR4 and 4.3+BSD. These advanced IPC capabilities are provided by most current Unix systems. We'll use the functions that we developed in this chapter in Chapter 18 with our modem dialer.

We presented two versions of an open server. One version was invoked directly by the client, using `fork` and `exec`. The second was a daemon server that handled all client requests. Both versions used the file descriptor passing and receiving functions from Section 15.3. The final version also used the client–server connection functions from Section 15.5 and the I/O multiplexing functions from Section 12.5.

## Exercises

**15.1** Recode Program 15.1 to use the standard I/O library instead of `read` and `write` on the stream pipe.

**15.2** Write the following program using the file descriptor passing functions from this chapter and the parent–child synchronization routines from Section 8.8. The program calls `fork`, the child opens an existing file and passes the open descriptor to the parent. The child then positions the file using `lseek` and notifies the parent. The parent reads the file's current offset and prints it for verification. If the file was passed from the child to the parent as we described, the parent and child should be sharing the same file table entry, so each time the child changes the file's current offset, that change should affect the parent's descriptor also. Have the child position the file to a different offset and notify the parent again.

**15.3** In Programs 15.14 and 15.15 we differentiated between declaring and defining the global variables. What is the difference?

**15.4** Recode the `buf_args` function (Program 15.17), removing the compile-time limit on the size of the `argv` array. Use dynamic memory allocation.

**15.5** Describe ways to optimize the function `loop` in Program 15.29 and Program 15.30. Implement your optimizations.

# 16

# A Database Library

## 16.1 Introduction

During the early 1980s Unix was considered a hostile environment for running a multiuser database system. (See Stonebraker [1981] and Weinberger [1982].) Earlier systems, such as Version 7, did indeed present large obstacles, since they did not provide any form of IPC (other than half-duplex pipes) and did not provide any form of record locking. Recent Unix systems, such as SVR4 and 4.3+BSD, provide a suitable environment for running a reliable, multiuser database system. Numerous commercial firms have offered these types of systems for years.

In this chapter we develop a simple, multiuser database library. It is a library of C functions that any program can call to fetch and store records in a database. This library of C functions is usually only one part of a complete database system. We do not develop the other pieces, such as a query language, leaving these items to the many textbooks on database systems. Our interest is the interface to Unix required by a database library and how that interface relates to the topics that we've already covered (such as record locking, in Section 12.3).

## 16.2 History

One popular library of database functions in Unix has been the `dbm`(3) library. This library was developed by Ken Thompson and uses a dynamic hashing scheme. It was originally provided with Version 7, appears in all Berkeley releases, and is also provided in the Berkeley compatibility library in SVR4. Seltzer and Yigit [1991] provide a detailed history of the dynamic hashing algorithm used by the `dbm` library, and other

implementations of this library. Unfortunately, a basic limitation of all these implementations is that none allows concurrent updating of the database by multiple processes. They provide no type of concurrency controls (such as record locking).

4.3+BSD provides a new db(3) library that supports three different forms of access: (a) record oriented, (b) hashing, and (c) a B-tree. Again, no form of concurrency is provided. (This fact is plainly stated in the BUGS section of the db(3) manual page.) Recent work by Seltzer and Olson [1992], however, indicates that a future release of this library will provide concurrency features similar to most commercial database systems.

Most commercial database libraries do provide the concurrency controls required for multiple processes to update a database simultaneously. These systems typically use advisory record locking, as we described in Section 12.3. These commercial systems usually implement their database using B+ trees [Comer 1979].

## 16.3 The Library

Let's first describe the C interface to the database library, then in the next section describe the actual implementation.

When we open a database we are returned a pointer to a DB structure. This is similar to fopen returning a pointer to a FILE structure (Section 5.2) and opendir returning a pointer to a DIR structure (Section 4.21). We'll pass this pointer to the remaining database functions.

```
#include "db.h"

DB *db_open(const char *pathname, int oflag, int mode);

                                    Returns: pointer to DB structure if OK, NULL on error

void db_close(DB *db);
```

If db_open is successful, two files are created: *pathname.idx* is the index file and *pathname.dat* is the data file. The *oflag* argument is used as the second argument to open (Section 3.3) to specify how the files are to be opened (read-only, read–write, create file if it doesn't exist, etc.). *mode* is used as the third argument to open (the file access permissions) if the database files are created.

We call db_close when we're done with a database. It closes the index file and data file, and releases any memory that it allocated for internal buffers.

When we store a new record in the database we have to specify the key for the record and the data associated with the key. If the database contained personnel records, the key could be the employee ID and the data could be the employee's name, address, telephone number, date of hire, and the like. Our implementation requires that the key for each record be unique. (We can't have two different employee records with the same employee ID, for example.)

```
#include "db.h"

int db_store(DB *db, const char *key, const char *data, int flag);
```

Returns: 0 if OK, nonzero on error (see following)

*key* and *data* are null-terminated character strings. The only restriction on these two strings is that neither can contain null bytes. They may contain, for example, newlines.

*flag* is either DB_INSERT (to insert a new record) or DB_REPLACE (to replace an existing record). These two constants are defined in the db.h header. If we specify DB_REPLACE and the record does not exist, the return value is –1. If we specify DB_INSERT and the record already exists, the return value is 1.

We can fetch any record from the database by just specifying its *key*.

```
#include "db.h"

char *db_fetch(DB *db, const char *key);
```

Returns: pointer to data if OK, NULL if record not found

The return value is a pointer to the data that was stored with the *key*, if the record is found.

We can also delete a record from the database by specifying its *key*.

```
#include "db.h"

int db_delete(DB *db, const char *key);
```

Returns: 0 if OK, –1 if record not found

In addition to fetching a record by specifying its key, we can also go through the entire database, reading each record in turn. To do this we first call db_rewind to rewind the database to the first record, and then call db_nextrec to read each sequential record.

```
#include "db.h"

void db_rewind(DB *db);

char *db_nextrec(DB *db, char *key);
```

Returns: pointer to data if OK, NULL on end of file

If *key* is a nonnull pointer, db_nextrec stores the key starting at that location.

There is no order to the records returned by db_nextrec. All we're guaranteed is that we'll read each record in the database once. If we store three records with keys of

A, B, and C, in that order, we have no idea in which order `db_nextrec` will return the three records. It might return B, then A, then C, or some other (apparently random) order. The actual order depends on the implementation of the database.

These seven functions provide the interface to the database library. We now describe the actual implementation that we have chosen.

## 16.4 Implementation Overview

Most database access libraries use two files to store the information: an index file and a data file. The index file contains the actual index value (the key) and a pointer to the corresponding data record in the data file. Numerous techniques can be used to organize the index file so that it can be searched quickly and efficiently for any key—hashing and B+ trees are popular. We have chosen to use a fixed-size hash table with chaining for the index file. We mentioned in the description of `db_open` that we create two files—one with a suffix of `.idx` and a suffix of `.dat` for the other.

We store the key and index as null-terminated character strings—they cannot contain arbitrary binary data. Some database systems store numerical data in a binary format (one, two, or four bytes for an integer, for example) to save storage space. This complicates the functions and requires more work to make the database files portable between different systems. For example, if we have two systems on a network that use different formats for storing binary integers, we need to handle this if want both systems to access the database. (It is not at all uncommon today to have systems with different architectures sharing files on a network.) Storing all the records, both keys and data, as character strings simplifies everything. It does require additional disk space, but that is becoming less of a concern with the advances in disk technology.

`db_store` allows only one record to have a given key. Some database systems allow multiple records to have the same key, and then provide a way to access all the records associated with a given key. Additionally, we have only a single index file, meaning each data record can have only a single key. Some database systems allow each record to have multiple keys, and often use one index file per key. Each time a new record is inserted or deleted, each index file has to be updated accordingly. (An example of a file with multiple indexes is an employee file. We could have one index whose key is the employee ID and another whose key is the employee's Social Security number. Having an index whose key is the employee name could be a problem, as names need not be unique.)

Figure 16.1 shows a general picture of the database implementation. The index file consists of three portions: the free list pointer, the hash table, and the index records. All the fields in Figure 16.1 called *ptr* are just file offsets stored as an ASCII number.

To find a record in the database, given its key, `db_fetch` calculates the hash value of the key, which leads to one hash chain in the hash table. (The *chain ptr* field could be 0, indicating an empty chain.) We then follow this hash chain, which is a linked list of all the index records with this hash value. When we encounter a *chain ptr* value of 0, we've hit the end of the hash chain.

Let's look at an actual database file. Program 16.1 creates a new database and writes three records to it. Since we store all the fields in the database as ASCII

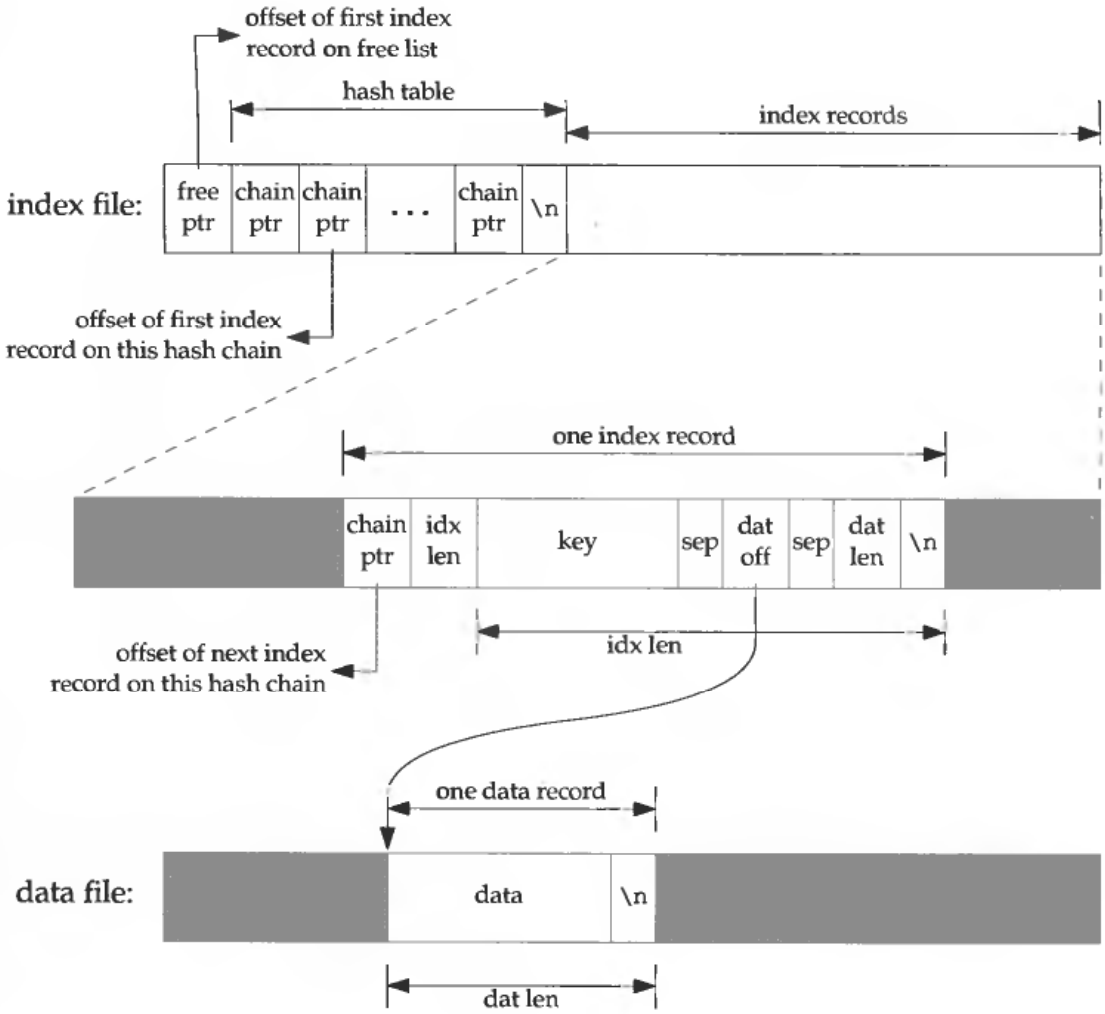


**Figure 16.1** Arrangement of index file and data file.

characters, we can look at the actual index file and data file using any of the standard Unix tools.

```
$ ls -l db4.*
-rw-r--r--  1 stevens    28 Oct 30 06:42 db4.dat
-rw-r--r--  1 stevens    72 Oct 30 06:42 db4.idx
$ cat db4.idx
   0  53  35   0
   0  10Alpha:0:6
   0  10beta:6:14
  17  11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

```
#include    "db.h"

int
main(void)
{
    DB      *db;

    if ( (db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
                                                FILE_MODE)) == NULL)
        err_sys("db_open error");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

**Program 16.1** Create a database and write three records to it.

To keep this example small, we have set the size of each *ptr* field to four ASCII characters, and the number of hash chains is three. Since each *ptr* is a file offset, a four-character field limits the total size of the index file and data file to 10,000 bytes. When we do some performance measurements of the database system in Section 16.8, we set the size of each *ptr* field to six characters (allowing file sizes up to 1 million bytes), and the number of hash chains to over 100.

The first line in the index file

```
0  53  35  0
```

is the free list pointer (0, the free list is empty), and the three hash chain pointers: 53, 35, and 0. The next line

```
0  10Alpha:0:6
```

shows the format of each index record. The first field (0) is the four-character chain pointer. This record is the end of its hash chain. The next field (10) is the four-character *idx len*, the length of the remainder of this index record. We read each index record using two `read`s: one to read the two fixed-size fields (the *chain ptr* and *idx len*), then another to read the remaining (variable-length) portion. The remaining three fields, *key*, *dat off*, and *dat len*, are delimited by a separator character (a colon in this case). We need the separator character since each of these three fields is variable length. The separator character can't appear in the key. Finally, a newline terminates the index record. The newline isn't required, since *idx len* contains the length of the record. We store the newline to separate each index record so we can use the normal Unix tools, such as `cat` and

`more` with the index file. The *key* is the value that we specified when we wrote the record to the database. The data offset (0) and data length (6) refer to the data file. We can see that the data record does start at offset 0 in the data file, and has a length of six bytes. (As with the index file, we automatically append a newline to each data record, so we can use the normal Unix tools with the file. This newline at the end is not returned to the called by `db_fetch`.)

If we follow the three hash chains in this example, we see that the first record on the first hash chain is at offset 53 (`gamma`). The next record on this chain is at offset 17 (`alpha`), and this is the last record on the chain. The first record on the second hash chain is at offset 35 (`beta`), and it's the last record on the chain. The third hash chain is empty.

Notice that the order of the keys in the index file and the order of their corresponding records in the data file is the same as the order of the calls to `db_store` in Program 16.1. Since the `O_TRUNC` flag was specified for `db_open`, the index file and data file were both truncated and the database initialized from scratch. In this case `db_store` just appends the new index records and data records to the end of the corresponding file. We'll see later that `db_store` can also reuse portions of these two files that correspond to deleted records.

The choice of a fixed-size hash table for the index is a compromise. It allows fast access as long as each hash chain isn't too long. We want to be able to search for any key quickly, but we don't want to complicate the data structures by using either a B-tree or dynamic extensible hashing. Dynamic extensible hashing has the advantage that any data record can be located with only two disk accesses (see Seltzer and Yigit [1991] for details). B-trees have the advantage of traversing the database in key order (something that we can't do with the `db_nextrec` function, using a hash table.)

## 16.5 Centralized or Decentralized?

Given multiple processes accessing the same database, there are two ways we can implement the functions.

1. Centralized. Have a single process that is the database manager and have it be the only process that accesses the database. The functions contact this central process using some form of IPC.
2. Decentralized. Have each function apply the required concurrency controls (locking) and then issue its own I/O function calls.

Database systems have been built using each of these techniques. The trend in Unix systems, however, is the decentralized approach. Given adequate locking routines, the decentralized implementation is usually faster, because IPC is avoided. Figure 16.2 depicts the operation of the centralized approach.

We purposely show the IPC going through the kernel, as most forms of message passing under Unix operate this way. (Shared memory, as described in Section 14.9,

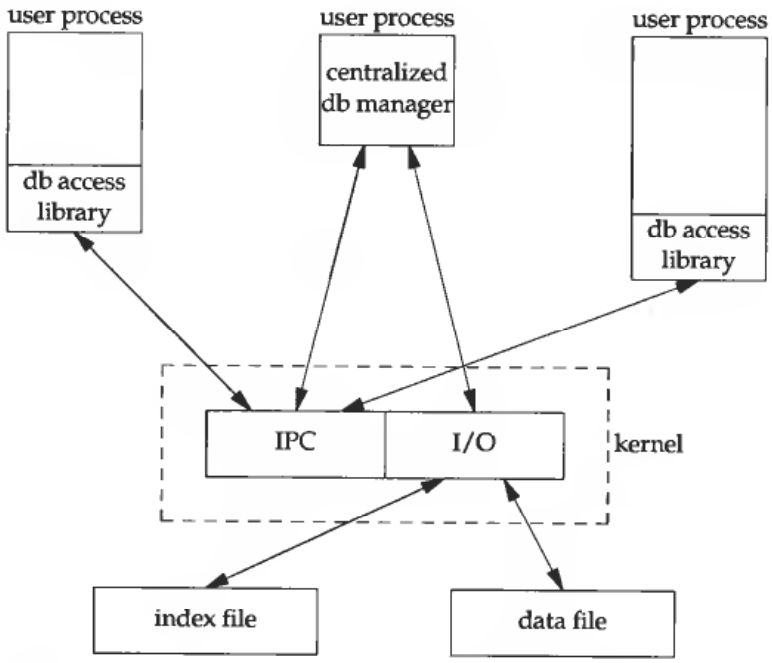


**Figure 16.2** Centralized approach for database access.

avoids this copying of the data.) We see with the centralized approach that a record is read by the central process and then passed to the requesting process using IPC. This is a disadvantage of this design. Note that the centralized database manager is the only process that does I/O with the database files.

The centralized approach has the advantage that customer tuning of its operation may be possible. For example, we might be able to assign different priorities to different processes through the centralized process. This could affect the scheduling of I/O operations by the centralized process. With the decentralized approach this is harder to do. We are usually at the mercy of the kernel's disk I/O scheduling policy and locking policy (i.e., if three processes are waiting for a lock to become available, which process gets the lock next?).

The decentralized approach is shown in Figure 16.3. This is the design that we'll implement in this chapter. The user processes that call the functions in the database library to perform I/O are considered cooperating processes, since they use record locking to provide concurrent access.

## 16.6 Concurrency

We purposely chose a two-file implementation (an index file and a data file) because that's how most systems are implemented. It requires us to handle the locking interactions of both files. But there are numerous ways to handle the locking of these two files.

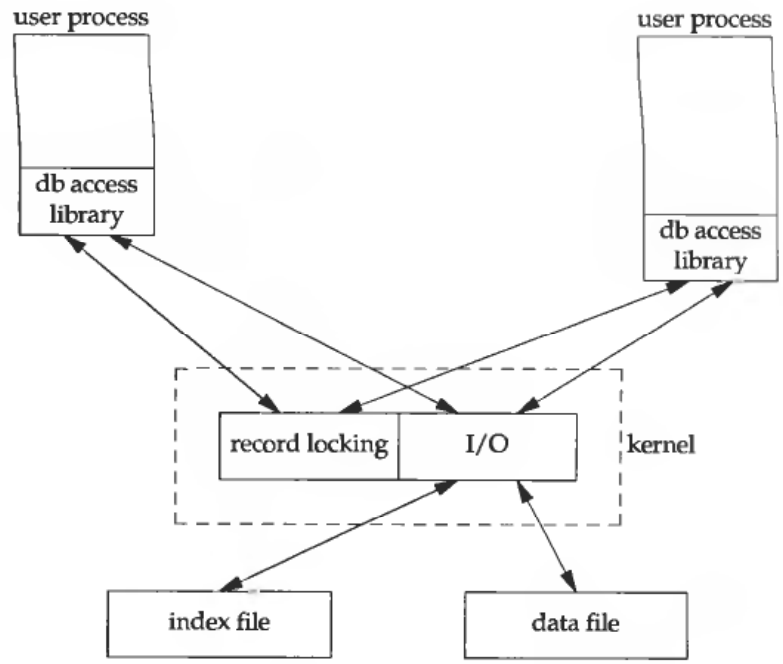


**Figure 16.3** Decentralized approach for database access.

### Coarse Locking

The simplest form of locking is to use one of the two files as a lock for the entire database and to require the caller to obtain this lock before operating on the database. We call this *coarse locking*. For example, we can say that the process with a read lock on byte 0 of the index file has read access to the entire database. A process with a write lock on byte 0 of the index file has write access to the entire database. We can use the normal Unix record locking semantics to allow any number of readers at one time, but only one writer at a time. (Recall Figure 12.2.) The functions `db_fetch` and `db_nextrec` require a read lock, and `db_delete`, `db_store`, and `db_open` all require a write lock. (The reason `db_open` requires a write lock is that if the file is being created it has to write the empty free list and hash chains at the front of the index file.)

The problem with coarse locking is that it doesn't allow the maximum amount of concurrency. If a process is adding a record to one hash chain, another process should be able to read a record on a different hash chain.

### Fine Locking

We enhance coarse locking to allow more concurrency and call this *fine locking*. We first require a reader or writer to obtain a read lock or a write lock on the hash chain for a given record. We allow any number of readers at one time on any hash chain, but only a single writer on a hash chain. Next, a writer that needs to access the free list (either `db_delete` or `db_store`) must obtain a write lock on the free list. Finally, whenever

db_store appends a new record to the end of either the index file or the data file, it has to obtain a write lock on that portion of the file.

We expect fine locking to provide more concurrency than coarse locking. In Section 16.8 we'll show some actual measurements. In Section 16.7 we show the source code to our implementation of fine locking and discuss the details of implementing locking. (Coarse locking is just a simplification of the locking that we show.)

In the source code we call read, readv, write, and writev directly. We do not use the standard I/O library. While it is possible to use record locking with the standard I/O library, it requires careful handling of buffering. We don't want an fgets, for example, to return data that was read into a standard I/O buffer 10 minutes ago, if the data was modified by another process 5 minutes ago.

Our discussion of concurrency is predicated on the simple needs of the database library. Commercial systems often have additional requirements. See Chapter 3 of Date [1982] for additional details on concurrency.

## 16.7 Source Code

We start with the db.h header in Program 16.2. This header is included by all the functions and by any user process that calls the library.

```
#include    <sys/types.h>
#include    <sys/stat.h>     /* open() & db_open() mode */
#include    <fcntl.h>        /* open() & db_open() flags */
#include    <stddef.h>       /* NULL */
#include    "ourhdr.h"

        /* flags for db_store() */
#define DB_INSERT   1        /* insert new record only */
#define DB_REPLACE  2        /* replace existing record */

        /* magic numbers */
#define IDXLEN_SZ      4     /* #ascii chars for length of index record */
#define IDXLEN_MIN     6     /* key, sep, start, sep, length, newline */
#define IDXLEN_MAX  1024     /* arbitrary */
#define SEP          ':'     /* separator character in index record */
#define DATLEN_MIN     2     /* data byte, newline */
#define DATLEN_MAX  1024     /* arbitrary */

        /* following definitions are for hash chains and free list chain
           in index file */
#define PTR_SZ         6     /* size of ptr field in hash chain */
#define PTR_MAX   999999     /* max offset (file size) = 10**PTR_SZ - 1 */
#define NHASH_DEF    137     /* default hash table size */
#define FREE_OFF       0     /* offset of ptr to free list in index file */
#define HASH_OFF  PTR_SZ     /* offset of hash table in index file */

typedef struct {     /* our internal structure */
  int   idxfd;  /* fd for index file */
  int   datfd;  /* fd for data file */
  int   oflag;  /* flags for open()/db_open(): O_xxx */
```

```
    char  *idxbuf;/* malloc'ed buffer for index record */
    char  *datbuf;/* malloc'ed buffer for data record*/
    char  *name;  /* name db was opened under */
    off_t idxoff; /* offset in index file of index record */
                  /* actual key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
    size_t idxlen;/* length of index record */
                  /* excludes IDXLEN_SZ bytes at front of index record */
                  /* includes newline at end of index record */
    off_t datoff; /* offset in data file of data record */
    size_t datlen;/* length of data record */
                  /* includes newline at end */
    off_t ptrval; /* contents of chain ptr in index record */
    off_t ptroff; /* offset of chain ptr that points to this index record */
    off_t chainoff;/* offset of hash chain for this index record */
    off_t hashoff;/* offset in index file of hash table */
    int   nhash;  /* current hash table size */
    long  cnt_delok;  /* delete OK */
    long  cnt_delerr; /* delete error */
    long  cnt_fetchok;/* fetch OK */
    long  cnt_fetcherr;/* fetch error */
    long  cnt_nextrec;/* nextrec */
    long  cnt_stor1;  /* store: DB_INSERT, no empty, appended */
    long  cnt_stor2;  /* store: DB_INSERT, found empty, reused */
    long  cnt_stor3;  /* store: DB_REPLACE, diff data len, appended */
    long  cnt_stor4;  /* store: DB_REPLACE, same data len, overwrote */
    long  cnt_storerr;/* store error */
} DB;

typedef unsigned long   hash_t; /* hash values */

            /* user-callable functions */
DB      *db_open(const char *, int, int);
void     db_close(DB *);
char    *db_fetch(DB *, const char *);
int      db_store(DB *, const char *, const char *, int);
int      db_delete(DB *, const char *);
void     db_rewind(DB *);
char    *db_nextrec(DB *, char *);
void     db_stats(DB *);

            /* internal functions */
DB      *_db_alloc(int);
int      _db_checkfree(DB *);
int      _db_dodelete(DB *);
int      _db_emptykey(char *);
int      _db_find(DB *, const char *, int);
int      _db_findfree(DB *, int, int);
int      _db_free(DB *);
hash_t   _db_hash(DB *, const char *);
char    *_db_nextkey(DB *);
char    *_db_readdat(DB *);
off_t    _db_readidx(DB *, off_t);
```

```
off_t     _db_readptr(DB *, off_t);
void      _db_writedat(DB *, const char *, off_t, int);
void      _db_writeidx(DB *, const char *, off_t, int, off_t);
void      _db_writeptr(DB *, off_t, off_t);
```

**Program 16.2** The db.h header.

Here we define the fundamental limits of the implementation. These can be changed if desired, to support bigger databases. Some of the values that we have defined as constants could also be made variable, with some added complexity in the implementation. For example, we set the size of the hash table to 137 entries. A better technique would be to let the caller specify this as an argument to db_open, based on the expected size of the database. We would then have to store this size at the beginning of the index file.

The DB structure is where we keep all the information for each open database. The DB * pointer that is returned by db_open and used by all the other functions is just a pointer to one of these structures.

We have chosen to name all the user-callable functions starting with db_ and all the internal functions start with _db_.

In Program 16.3 we show db_open. It opens the index file and data file, initializing the index file if necessary. It calls _db_alloc to allocate a DB structure and initializes it.

```
#include    "db.h"

/* Open or create a database.  Same arguments as open(). */

DB *
db_open(const char *pathname, int oflag, int mode)
{
    DB          *db;
    int         i, len;
    char        asciiptr[PTR_SZ + 1],
                hash[(NHASH_DEF + 1) * PTR_SZ + 2];
                    /* +2 for newline and null */
    struct stat statbuff;

        /* Allocate a DB structure, and the buffers it needs */
    len = strlen(pathname);
    if ( (db = _db_alloc(len)) == NULL)
        err_dump("_db_alloc error for DB");

    db->oflag = oflag;      /* save a copy of the open flags */

        /* Open index file */
    strcpy(db->name, pathname);
    strcat(db->name, ".idx");
    if ( (db->idxfd = open(db->name, oflag, mode)) < 0) {
        _db_free(db);
        return(NULL);
    }
```

```
        /* Open data file */
    strcpy(db->name + len, ".dat");
    if ( (db->datfd = open(db->name, oflag, mode)) < 0) {
        _db_free(db);
        return(NULL);
    }

        /* If the database was created, we have to initialize it */
    if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
            /* Write lock the entire file so that we can stat
               the file, check its size, and initialize it,
               as an atomic operation. */
        if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

        if (fstat(db->idxfd, &statbuff) < 0)
            err_sys("fstat error");
        if (statbuff.st_size == 0) {
                /* We have to build a list of (NHASH_DEF + 1) chain
                   ptrs with a value of 0.  The +1 is for the free
                   list pointer that precedes the hash table. */
            sprintf(asciiptr, "%*d", PTR_SZ, 0);
            hash[0] = 0;
            for (i = 0; i < (NHASH_DEF + 1); i++)
                strcat(hash, asciiptr);
            strcat(hash, "\n");

            i = strlen(hash);
            if (write(db->idxfd, hash, i) != i)
                err_dump("write error initializing index file");
        }
        if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
    }
    db->nhash    = NHASH_DEF;/* hash table size */
    db->hashoff = HASH_OFF; /* offset in index file of hash table */
                            /* free list ptr always at FREE_OFF */
    db_rewind(db);

    return(db);
}
```

**Program 16.3** The db_open function.

We encounter locking if the database is being created. Consider two processes trying to create the same database at about the same time. Assume the first process calls `fstat` and is blocked by the kernel after `fstat` returns. The second process calls db_open, finds the length of the index file is 0, and initializes the free list and hash chain. The second process continues executing and writes one record to the database. At this point the second process is blocked and the first process continues executing right after the call to `fstat`. The first process finds the size of the index file to be 0 (since `fstat` was

called before the second process initialized the index file) so the first process initializes the free list and hash chain, wiping out the record that the second process stored in the database. The way to prevent this is to use locking. We use the functions `readw_lock`, `writew_lock`, and `un_lock` from Section 12.3.

The function `_db_alloc` in Program 16.4 is called by `db_open` to allocate storage for the `DB` structure, an index buffer, and a data buffer.

```
#include    "db.h"

/* Allocate & initialize a DB structure, and all the buffers it needs */

DB *
_db_alloc(int namelen)
{
    DB      *db;

            /* Use calloc, to init structure to zero */
    if ( (db = calloc(1, sizeof(DB))) == NULL)
        err_dump("calloc error for DB");

    db->idxfd = db->datfd = -1;             /* descriptors */

        /* Allocate room for the name.
           +5 for ".idx" or ".dat" plus null at end. */

    if ( (db->name = malloc(namelen + 5)) == NULL)
        err_dump("malloc error for name");

        /* Allocate an index buffer and a data buffer.
           +2 for newline and null at end. */

    if ( (db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
        err_dump("malloc error for index buffer");
    if ( (db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
        err_dump("malloc error for data buffer");

    return(db);
}
```

**Program 16.4** The `_db_alloc` function.

The size of the index buffer and data buffer are defined in the `db.h` header. An enhancement to the database library would be to allow these buffers to expand as required. We could keep track of the size of these two buffers and call `realloc` whenever we find we need a bigger buffer.

These dynamically allocated buffers are released, and the open files closed, by `_db_free` (Program 16.5). This function is called by `db_open` if an error occurs while opening the index file or data file. `_db_free` is also called by `db_close` (Program 16.6).

`db_fetch` (Program 16.7) reads a record, given its key. It calls `_db_find` to search the database for the index record and, if found, calls `_db_readdat` to read the corresponding data record.

```
#include    "db.h"

/* Free up a DB structure, and all the malloc'ed buffers it
 * may point to.  Also close the file descriptors if still open. */
int
_db_free(DB *db)
{
    if (db->idxfd >= 0 && close(db->idxfd) < 0)
        err_dump("index close error");
    if (db->datfd >= 0 && close(db->datfd) < 0)
        err_dump("data close error");
    db->idxfd = db->datfd = -1;

    if (db->idxbuf != NULL)
        free(db->idxbuf);
    if (db->datbuf != NULL)
        free(db->datbuf);
    if (db->name != NULL)
        free(db->name);
    free(db);
    return(0);
}
```

**Program 16.5** The _db_free function.

```
#include    "db.h"

void
db_close(DB *db)
{
    _db_free(db);    /* closes fds, free buffers & struct */
}
```

**Program 16.6** The db_close function.

```
#include    "db.h"

/* Fetch a specified record.
 * We return a pointer to the null-terminated data. */
char *
db_fetch(DB *db, const char *key)
{
    char    *ptr;

    if (_db_find(db, key, 0) < 0) {
        ptr = NULL;             /* error, record not found */
        db->cnt_fetcherr++;
    } else {
        ptr = _db_readdat(db);  /* return pointer to data */
        db->cnt_fetchok++;
    }
```

```
        /* Unlock the hash chain that _db_find() locked */
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(ptr);
}
```

**Program 16.7** The db_fetch function.

Program 16.8 shows _db_find, the function that traverses a hash chain. It's called by all the functions that look up a record given a key: db_fetch, db_delete, and db_store.

```
#include    "db.h"

/* Find the specified record.
 * Called by db_delete(), db_fetch(), and db_store(). */

int
_db_find(DB *db, const char *key, int writelock)
{
    off_t   offset, nextoffset;

        /* Calculate hash value for this key, then calculate byte
           offset of corresponding chain ptr in hash table.
           This is where our search starts. */

        /* calc offset in hash table for this key */
    db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
    db->ptroff = db->chainoff;

        /* Here's where we lock this hash chain.  It's the
           caller's responsibility to unlock it when done.
           Note we lock and unlock only the first byte. */
    if (writelock) {
        if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
            err_dump("writew_lock error");
    } else {
        if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
            err_dump("readw_lock error");
    }
        /* Get the offset in the index file of first record
           on the hash chain (can be 0) */
    offset = _db_readptr(db, db->ptroff);

    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strcmp(db->idxbuf, key) == 0)
            break;      /* found a match */

        db->ptroff = offset;    /* offset of this (unequal) record */
        offset = nextoffset;    /* next one to compare */
    }
```

```
    if (offset == 0)
        return(-1);     /* error, record not found */

        /* We have a match.  We're guaranteed that db->ptroff contains
           the offset of the chain ptr that points to this matching
           index record.  _db_dodelete() uses this fact.  (The chain
           ptr that points to this matching record could be in an
           index record or in the hash table.) */
    return(0);
}
```

**Program 16.8** The _db_find function.

The last argument to _db_find specifies if we want a read lock (0) or a write lock (1). We saw that db_fetch requires a read lock, while db_delete and db_store both require a write lock. _db_find waits for the given lock before going through the hash chain.

The while loop in _db_find is where we go through each index record on the hash chain, comparing keys. The function _db_readidx is called to read each index record.

Note the final comment in _db_find. As we make our way through the hash chain, we keep track of the previous index record that points to the current index record. We'll use this when we delete a record, since we have to modify the chain pointer of the previous record when we delete the current record.

Let's start with the easy functions that are called by _db_find first. _db_hash (Program 16.9) calculates the hash value for a given key. It just multiplies each ASCII character times its 1-based index and divides the result by the number of hash table entries. The remainder from this division is the hash value for this key.

```
#include    "db.h"

/* Calculate the hash value for a key. */

hash_t
_db_hash(DB *db, const char *key)
{
    hash_t      hval;
    const char  *ptr;
    char        c;
    int         i;

    hval = 0;
    for (ptr = key, i = 1; c = *ptr++; i++)
        hval += c * i;      /* ascii char times its 1-based index */

    return(hval % db->nhash);
}
```

**Program 16.9** The _db_hash function.

The next function called by _db_find is _db_readptr (Program 16.10). It reads any one of three different chain pointers: (1) the pointer at the beginning of the index file that points to the first index record on the free list, (2) the pointers in the hash table that point to the first index record on each hash chain, and (3) the pointers that are stored at the beginning of each index record (whether the index record is part of a hash chain or on the free list). No locking is done by this function—that is up to the caller.

```
#include    "db.h"

/* Read a chain ptr field from anywhere in the index file:
 * the free list pointer, a hash table chain ptr, or an
 * index record chain ptr.  */

off_t
_db_readptr(DB *db, off_t offset)
{
    char    asciiptr[PTR_SZ + 1];

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("lseek error to ptr field");
    if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("read error of ptr field");

    asciiptr[PTR_SZ] = 0;       /* null terminate */
    return(atol(asciiptr));
}
```

**Program 16.10** The _db_readptr function.

The while loop in _db_find calls _db_readidx to read each index record. This is a larger function (Program 16.11) that reads the index record and divides it into the appropriate fields.

```
#include    "db.h"
#include    <sys/uio.h>     /* struct iovec */

/* Read the next index record.  We start at the specified offset in
 * the index file.  We read the index record into db->idxbuf and
 * replace the separators with null bytes.  If all is OK we set
 * db->datoff and db->datlen to the offset and length of the
 * corresponding data record in the data file.  */

off_t
_db_readidx(DB *db, off_t offset)
{
    int             i;
    char            *ptr1, *ptr2;
    char            asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
    struct iovec    iov[2];

        /* Position index file and record the offset.  db_nextrec()
           calls us with offset==0, meaning read from current offset.
           We still need to call lseek() to record the current offset. */
```

```
    if ( (db->idxoff = lseek(db->idxfd, offset,
                             offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
        err_dump("lseek error");

        /* Read the ascii chain ptr and the ascii length at
           the front of the index record.  This tells us the
           remaining size of the index record. */
    iov[0].iov_base = asciiptr;
    iov[0].iov_len  = PTR_SZ;
    iov[1].iov_base = asciilen;
    iov[1].iov_len  = IDXLEN_SZ;
    if ( (i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
        if (i == 0 && offset == 0)
            return(-1);     /* EOF for db_nextrec() */
        err_dump("readv error of index record");
    }

    asciiptr[PTR_SZ] = 0;           /* null terminate */
    db->ptrval = atol(asciiptr);    /* offset of next key in chain */
                        /* this is our return value; always >= 0 */
    asciilen[IDXLEN_SZ] = 0;        /* null terminate */
    if ( (db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
                           db->idxlen > IDXLEN_MAX)
        err_dump("invalid length");

        /* Now read the actual index record.  We read it into the key
           buffer that we malloced when we opened the database. */
    if ( (i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
        err_dump("read error of indexc record");
    if (db->idxbuf[db->idxlen-1] != '\n')
        err_dump("missing newline");    /* sanity checks */
    db->idxbuf[db->idxlen-1] = 0;       /* replace newline with null */

        /* Find the separators in the index record */
    if ( (ptr1 = strchr(db->idxbuf, SEP)) == NULL)
        err_dump("missing first separator");
    *ptr1++ = 0;                /* replace SEP with null */

    if ( (ptr2 = strchr(ptr1, SEP)) == NULL)
        err_dump("missing second separator");
    *ptr2++ = 0;                /* replace SEP with null */

    if (strchr(ptr2, SEP) != NULL)
        err_dump("too many separators");

        /* Get the starting offset and length of the data record */
    if ( (db->datoff = atol(ptr1)) < 0)
        err_dump("starting offset < 0");
    if ( (db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
        err_dump("invalid length");
    return(db->ptrval);     /* return offset of next key in chain */
}
```

**Program 16.11** The _db_readidx function.

We call readv to read the two fixed-length fields at the beginning of the index record: the chain pointer to the next index record and the size of the variable-length index record that follows. Once these two fields are read, the variable-length index record is read, and the three remaining fields are separated: the key, the offset of the corresponding data record, and the length of the data record. Note that the data record is not read. That is left to the caller. In db_fetch, for example, we don't read the data record until _db_find has read the index record that matches the key that we're looking for.

We now return to db_fetch. If _db_find locates the index record with the matching key, we call _db_readdat to read the corresponding data record. This is a simple function (Program 16.12).

```
#include    "db.h"

/* Read the current data record into the data buffer.
 * Return a pointer to the null-terminated data buffer. */

char *
_db_readdat(DB *db)
{
    if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
        err_dump("lseek error");

    if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
        err_dump("read error");
    if (db->datbuf[db->datlen - 1] != '\n')     /* sanity check */
        err_dump("missing newline");
    db->datbuf[db->datlen - 1] = 0;     /* replace newline with null */

    return(db->datbuf);     /* return pointer to data record */
}
```

**Program 16.12** The _db_readdat function.

We started at db_fetch and have finally read both the index record and corresponding data record. Note that the only locking that has been done has been the read lock applied by _db_find. Since we have the hash chain read locked, we're guaranteed that no other process is proceeding down the same hash chain modifying anything.

Now let's examine the db_delete function (Program 16.13). It starts the same as db_fetch, calling _db_find to locate the record. But this time the final argument to _db_find is 1, indicating that we need the hash chain write locked.

db_delete calls _db_dodelete (Program 16.14) to do all the work. (We'll see later that db_store also calls _db_dodelete.) Most of the function just updates two linked lists, the free list and the hash chain for this key.

When a record is deleted we set its key and data record to blanks. This fact is used by db_nextrec, which we'll examine later in this section.

_db_dodelete write locks the free list. This is to prevent two processes that are deleting records at the same time, on two different hash chains, from interfering with each other. Since we'll add the deleted record to the free list, which changes the free list pointer, only one process at a time can be doing this.

```
#include    "db.h"

/* Delete the specified record */

int
db_delete(DB *db, const char *key)
{
    int     rc;

    if (_db_find(db, key, 1) == 0) {
        rc = _db_dodelete(db);  /* record found */
        db->cnt_delok++;
    } else {
        rc = -1;                /* not found */
        db->cnt_delerr++;
    }

    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}
```

**Program 16.13** The db_delete function.

```
#include    "db.h"

/* Delete the current record specified by the DB structure.
 * This function is called by db_delete() and db_store(),
 * after the record has been located by _db_find(). */

int
_db_dodelete(DB *db)
{
    int     i;
    char    *ptr;
    off_t   freeptr, saveptr;

        /* Set data buffer to all blanks */
    for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
        *ptr++ = ' ';
    *ptr = 0;   /* null terminate for _db_writedat() */

        /* Set key to blanks */
    ptr = db->idxbuf;
    while (*ptr)
        *ptr++ = ' ';

        /* We have to lock the free list */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("writew_lock error");
```

```
    /* Write the data record with all blanks */
_db_writedat(db, db->datbuf, db->datoff, SEEK_SET);

    /* Read the free list pointer.  Its value becomes the
       chain ptr field of the deleted index record.  This means
       the deleted record becomes the head of the free list. */
freeptr = _db_readptr(db, FREE_OFF);

    /* Save the contents of index record chain ptr,
       before it's rewritten by _db_writeidx(). */
saveptr = db->ptrval;

    /* Rewrite the index record.  This also rewrites the length
       of the index record, the data offset, and the data length,
       none of which has changed, but that's OK. */
_db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);

    /* Write the new free list pointer */
_db_writeptr(db, FREE_OFF, db->idxoff);

    /* Rewrite the chain ptr that pointed to this record
       being deleted.  Recall that _db_find() sets db->ptroff
       to point to this chain ptr.  We set this chain ptr
       to the contents of the deleted record's chain ptr,
       saveptr, which can be either zero or nonzero. */
_db_writeptr(db, db->ptroff, saveptr);

if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
    err_dump("un_lock error");

return(0);
}
```

**Program 16.14** The `_db_dodelete` function.

`_db_dodelete` writes the all-blank data record by calling `_db_writedat` (Program 16.15). Notice that the data file is not locked by `_db_writedat`. Since `db_delete` has write locked the hash chain for this record, we know that no other process is reading or writing this particular data record. When we cover `db_store` later in this section, we'll encounter the case where `_db_writedat` is appending to the data file and has to lock it.

`_db_writedat` calls `writev` to write the data record and newline. We can't assume that the caller's buffer has room at the end for us to append the newline to it. Recall Section 12.7, where we determined that a single `writev` is faster than two `writes`.

Then `_db_dodelete` rewrites the index record, after changing the chain pointer in the index record to point to the first record on the free list. (If the free list was empty, this new chain pointer is 0.) The free list pointer is then rewritten, to point to the index record that we just wrote (the deleted record). This means that the free list is handled on a first-in, first-out basis—deleted records are added to the front of the free list.

```
#include    "db.h"
#include    <sys/uio.h>     /* struct iovec */

/* Write a data record.  Called by _db_dodelete() (to write
   the record with blanks) and db_store(). */

void
_db_writedat(DB *db, const char *data, off_t offset, int whence)
{
    struct iovec    iov[2];
    static char     newline = '\n';

        /* If we're appending, we have to lock before doing the lseek()
           and write() to make the two an atomic operation.  If we're
           overwriting an existing record, we don't have to lock. */
    if (whence == SEEK_END)     /* we're appending, lock entire file */
        if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

    if ( (db->datoff = lseek(db->datfd, offset, whence)) == -1)
        err_dump("lseek error");
    db->datlen = strlen(data) + 1;  /* datlen includes newline */

    iov[0].iov_base = (char *) data;
    iov[0].iov_len  = db->datlen - 1;
    iov[1].iov_base = &newline;
    iov[1].iov_len  = 1;
    if (writev(db->datfd, &iov[0], 2) != db->datlen)
        err_dump("writev error of data record");

    if (whence == SEEK_END)
        if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
}
```

**Program 16.15** The `_db_writedat` function.

Notice that we don't have a separate free list for the index file and data file. When the record is deleted, the index record is added to the free list, and this index record points to the deleted data record. There are better ways to handle record deletion, in exchange for added code complexity.

Program 16.16 shows `_db_writeidx`, the function called by `_db_dodelete` to write an index record. As with `_db_writedat`, this function deals with locking only when a new index record is being appended to the index file. When `_db_dodelete` calls this function, we're rewriting an existing index record. We know in this case that the caller has write locked the hash chain, so no additional locking is required.

```
#include    "db.h"
#include    <sys/uio.h>     /* struct iovec */

/* Write an index record.
 * _db_writedat() is called before this function, to set the fields
 * datoff and datlen in the DB structure, which we need to write
 * the index record. */

void
_db_writeidx(DB *db, const char *key,
                     off_t offset, int whence, off_t ptrval)
{
    struct iovec    iov[2];
    char            asciiptrlen[PTR_SZ + IDXLEN_SZ +1];
    int             len;

    if ( (db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
        err_quit("invalid ptr: %d", ptrval);

    sprintf(db->idxbuf, "%s%c%d%c%d\n",
                key, SEP, db->datoff, SEP, db->datlen);
    if ( (len = strlen(db->idxbuf)) < IDXLEN_MIN || len > IDXLEN_MAX)
        err_dump("invalid length");
    sprintf(asciiptrlen, "%*d%*d", PTR_SZ, ptrval, IDXLEN_SZ, len);

        /* If we're appending, we have to lock before doing the lseek()
           and write() to make the two an atomic operation.  If we're
           overwriting an existing record, we don't have to lock. */
    if (whence == SEEK_END)      /* we're appending */
        if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
                                                SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

        /* Position the index file and record the offset */
    if ( (db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
        err_dump("lseek error");

    iov[0].iov_base = asciiptrlen;
    iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
    iov[1].iov_base = db->idxbuf;
    iov[1].iov_len  = len;
    if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
        err_dump("writev error of index record");

    if (whence == SEEK_END)
        if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
}
```

**Program 16.16** The _db_writeidx function.

The final function that _db_dodelete calls is _db_writeptr (Program 16.17). It is called twice—once to rewrite the free list pointer and once to rewrite the hash chain pointer (that pointed to the deleted record).

```
#include    "db.h"

/* Write a chain ptr field somewhere in the index file:
 * the free list, the hash table, or in an index record. */

void
_db_writeptr(DB *db, off_t offset, off_t ptrval)
{
    char    asciiptr[PTR_SZ + 1];

    if (ptrval < 0 || ptrval > PTR_MAX)
        err_quit("invalid ptr: %d", ptrval);
    sprintf(asciiptr, "%*d", PTR_SZ, ptrval);

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("lseek error to ptr field");
    if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("write error of ptr field");
}
```

**Program 16.17** The _db_writeptr function.

In Program 16.18 we cover the largest of the database functions, db_store. It starts by calling _db_find to see if the record already exists. It is OK if the record already exists and DB_REPLACE is specified or if the record doesn't exist and DB_INSERT is specified. If we're replacing an existing record, that implies that the keys are identical but the data records probably differ.

Note that the final argument to _db_find specifies that the hash chain must be write locked, as we will probably be modifying this hash chain.

If we are inserting a new record into the database, we call _db_findfree (Program 16.19) to search the free list for a deleted record with the same size key and the same size data.

The while loop in _db_findfree goes through the free list, looking for a record with a matching key size and matching data size. In this simple implementation we reuse a deleted record only if the key length and data length equal the lengths for the new record being inserted. There are a variety of better ways to reuse this deleted space, in exchange for added complexity.

_db_findfree needs to write lock the free list to avoid interfering with any other processes using the free list. Once the record has been removed from the free list, the write lock can be released. Recall that _db_dodelete also modified the free list.

```
#include    "db.h"

/* Store a record in the database.
 * Return 0 if OK, 1 if record exists and DB_INSERT specified,
 * -1 if record doesn't exist and DB_REPLACE specified. */

int
db_store(DB *db, const char *key, const char *data, int flag)
{
    int     rc, keylen, datlen;
    off_t   ptrval;

    keylen = strlen(key);
    datlen = strlen(data) + 1;      /* +1 for newline at end */
    if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
        err_dump("invalid data length");

        /* _db_find() calculates which hash table this new record
           goes into (db->chainoff), regardless whether it already
           exists or not.  The calls to _db_writeptr() below
           change the hash table entry for this chain to point to
           the new record.  This means the new record is added to
           the front of the hash chain. */

    if (_db_find(db, key, 1) < 0) {     /* record not found */
        if (flag & DB_REPLACE) {
            rc = -1;
            db->cnt_storerr++;
            goto doreturn;      /* error, record does not exist */
        }

            /* _db_find() locked the hash chain for us; read the
               chain ptr to the first index record on hash chain */
        ptrval = _db_readptr(db, db->chainoff);

        if (_db_findfree(db, keylen, datlen) < 0) {
                /* An empty record of the correct size was not found.
                   We have to append the new record to the ends of
                   the index and data files */
            _db_writedat(db, data, 0, SEEK_END);
            _db_writeidx(db, key, 0, SEEK_END, ptrval);
                /* db->idxoff was set by _db_writeidx().  The new
                   record goes to the front of the hash chain. */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor1++;
        } else {
```

```
                /* We can reuse an empty record.
                   _db_findfree() removed the record from the free
                   list and set both db->datoff and db->idxoff. */
            _db_writedat(db, data, db->datoff, SEEK_SET);
            _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
                /* reused record goes to the front of the hash chain. */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor2++;
        }

    } else {                        /* record found */
        if (flag & DB_INSERT) {
            rc = 1;
            db->cnt_storerr++;
            goto doreturn;      /* error, record already in db */
        }

            /* We are replacing an existing record.  We know the new
               key equals the existing key, but we need to check if
               the data records are the same size. */
        if (datlen != db->datlen) {
            _db_dodelete(db);   /* delete the existing record */

                /* Reread the chain ptr in the hash table
                   (it may change with the deletion). */
            ptrval = _db_readptr(db, db->chainoff);

                /* append new index and data records to end of files */
            _db_writedat(db, data, 0, SEEK_END);
            _db_writeidx(db, key, 0, SEEK_END, ptrval);
                /* new record goes to the front of the hash chain. */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor3++;

        } else {
                /* same size data, just replace data record */
            _db_writedat(db, data, db->datoff, SEEK_SET);
            db->cnt_stor4++;
        }
    }
    rc = 0;     /* OK */
doreturn:   /* unlock the hash chain that _db_find() locked */
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}
```

**Program 16.18** The `db_store` function.

```
#include    "db.h"

/* Try to find a free index record and accompanying data record
 * of the correct sizes.  We're only called by db_store(). */

int
_db_findfree(DB *db, int keylen, int datlen)
{
    int     rc;
    off_t   offset, nextoffset, saveoffset;

        /* Lock the free list */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("writew_lock error");

        /* Read the free list pointer */
    saveoffset = FREE_OFF;
    offset = _db_readptr(db, saveoffset);

    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
            break;      /* found a match */

        saveoffset = offset;
        offset = nextoffset;
    }

    if (offset == 0)
        rc = -1;    /* no match found */
    else {
            /* Found a free record with matching sizes.
               The index record was read in by _db_readidx() above,
               which sets db->ptrval.  Also, saveoffset points to
               the chain ptr that pointed to this empty record on
               the free list.  We set this chain ptr to db->ptrval,
               which removes the empty record from the free list. */

        _db_writeptr(db, saveoffset, db->ptrval);
        rc = 0;

            /* Notice also that _db_readidx() set both db->idxoff
               and db->datoff.  This is used by the caller, db_store(),
               to write the new index record and data record. */
    }
            /* Unlock the free list */
    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}
```

**Program 16.19** The `_db_findfree` function.

Returning to db_store, after the call to _db_find, the code divides into four cases.

1. A new record is being inserted and an empty record with the correct sizes was not found by _db_findfree. This means we have to append the new record to the ends of the index file and data file. The new record is added to the front of the hash chain by calling _db_writeptr.
2. A new record is being added and an empty record with the correct sizes was found by _db_findfree. The empty record is removed from the free list by _db_findfree, and the new data record and index record are rewritten. The new record is added to the front of the hash chain by calling _db_writeptr.
3. An existing record is being replaced and the length of the new data record differs from the length of the existing data record. We call _db_dodelete to delete the existing record and then append the new record to the ends of the index file and data file. (There are other ways to handle this case. We could try to find a deleted record that has the correct data size.) The new record is added to the front of the hash chain by calling _db_writeptr.
4. An existing record is being replaced and the length of the new data record equals the length of the existing data record. This is the easiest case—we just rewrite the data record.

We need to describe the locking when new index records or data records are appended to the end of the file. (Recall the problems we encountered in Program 12.6 with locking relative to the end of file.) In cases 1 and 3, db_store calls both _db_writeidx and _db_writedat with a third argument of 0 and a fourth argument of SEEK_END. This fourth argument is the flag to these two functions that the new record is being appended to the file. The technique used by _db_writeidx is to write lock the index file, from the end of the hash chain to the end of file. This won't interfere with any other readers or writers of the database (since they will lock a hash chain) but it does prevent other callers of db_store from trying to append at the same time. The technique used by _db_writedat is to write lock the entire data file. Again, this won't interfere with other readers or writers of the database (since they don't even try to lock the data file), but it does prevent other callers of db_store from trying to append to the data file at the same time. (See Exercise 16.3.)

We complete the tour of the source code with db_nextrec and db_rewind, the functions used to read all the records in the database. The normal use of these functions is in a loop of the form

```
db_rewind(db);
while ( (ptr = db_nextrec(db, key)) != NULL) {
    /* process record */
}
```

As we warned earlier, there is no order to the returned records—they are not in key order.

The technique for db_rewind (Program 16.20) is to position the index file to the first index record (immediately following the hash table).

```
#include    "db.h"

/* Rewind the index file for db_nextrec().
 * Automatically called by db_open().
 * Must be called before first db_nextrec().
 */

void
db_rewind(DB *db)
{
    off_t   offset;

    offset = (db->nhash + 1) * PTR_SZ;      /* +1 for free list ptr */

        /* We're just setting the file offset for this process
           to the start of the index records; no need to lock.
           +1 below for newline at end of hash table. */

    if ( (db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
        err_dump("lseek error");
}
```

**Program 16.20** The db_rewind function.

Once db_rewind has positioned the index file, db_nextrec just sequentially reads all the index records. As we see in Program 16.21, db_nextrec does not use the hash chains. Since db_nextrec reads all the deleted records along with the records on a hash chain, it has to check if a record has been deleted (its key is all blank) and ignore these deleted records.

If the database is being modified while db_nextrec is called from a loop, the records returned by db_nextrec are just a snapshot of a changing database at some point in time. db_nextrec always returns a "correct" record when it is called; that is, it won't return a record that was deleted. But it is possible for a record returned by db_nextrec to be deleted immediately after db_nextrec returns. Similarly, if a deleted record is reused right after db_nextrec skips over the deleted record, we won't see that new record unless we rewind the database and go through it again. If it's important to obtain an accurate "frozen" snapshot of the database using db_nextrec, there must be no insertions or deletions going on at the same time.

Look at the locking employed by db_nextrec. We're not going through any hash chain, and we can't determine the hash chain that a record belongs on. Therefore, it is possible for an index record to be in the process of being deleted when db_nextrec is reading the record. To prevent this, db_nextrec read locks the free list, to avoid any interactions with _db_dodelete and _db_findfree.

```
#include    "db.h"

/* Return the next sequential record.
 * We just step our way through the index file, ignoring deleted
 * records.  db_rewind() must be called before this is function
 * is called the first time.
 */

char *
db_nextrec(DB *db, char *key)
{
    char    c, *ptr;

        /* We read lock the free list so that we don't read
           a record in the middle of its being deleted. */
    if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("readw_lock error");

    do {
            /* read next sequential index record */
        if (_db_readidx(db, 0) < 0) {
            ptr = NULL;     /* end of index file, EOF */
            goto doreturn;
        }
            /* check if key is all blank (empty record) */
        ptr = db->idxbuf;
        while ( (c = *ptr++) != 0  &&  c == ' ')
            ;   /* skip until null byte or nonblank */
    } while (c == 0);   /* loop until a nonblank key is found */

    if (key != NULL)
        strcpy(key, db->idxbuf);    /* return key */
    ptr = _db_readdat(db);  /* return pointer to data buffer */

    db->cnt_nextrec++;
doreturn:
    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("un_lock error");

    return(ptr);
}
```

**Program 16.21** The `db_nextrec` function.

## 16.8 Performance

To test the database library and to obtain some timing measurements, a test program was written. This program takes two command-line arguments: the number of children to create and the number of database records (*nrec*) for each child to write to the database. The program then creates an empty database (by calling `db_open`), `fork`s

the number of child processes, and waits for all the children to terminate. Each child performs the following steps:

- Write *nrec* records to the database.
- Read the *nrec* records back by key value.
- Perform the following loop *nrec* × 5 times.
  - Read a random record.
  - Every 37 times through the loop, delete a random record.
  - Every 11 times through the loop, insert a new record and read the record back.
  - Every 17 times through the loop, replace a random record with a new record. Every other one of these replacements is a record with the same size data and the alternate is a record with a longer data portion.
- Delete all the records that this child wrote. Every time a record is deleted, 10 random records are looked up.

The actual number of operations performed on the database is counted by the `cnt_xxx` variables in the `DB` structure, which were incremented in the functions. The number of operations differs from one child to the next, since the random number generator used to select records is initialized in each child to the child's process ID. A typical count of the operations performed in each child, when *nrec* is 500, is shown in Figure 16.4.

| Operation | Count |
|---|---|
| `db_store`, `DB_INSERT`, no empty record, appended | 675 |
| `db_store`, `DB_INSERT`, empty record reused | 170 |
| `db_store`, `DB_REPLACE`, different data length, appended | 100 |
| `db_store`, `DB_REPLACE`, equal data length | 100 |
| `db_store`, record not found | 20 |
| `db_fetch`, record found | 8300 |
| `db_fetch`, record not found | 750 |
| `db_delete`, record found | 840 |
| `db_delete`, record not found | 100 |

**Figure 16.4** Typical count of operations performed by each child when *nrec* is 500.

We performed about 10 times more fetches than stores or deletions, which is probably typical of many database applications.

Each child is doing these operations (fetching, storing, and deleting), only with the records that the child wrote. All the concurrency controls are being exercised because all the children are operating on the same database (albeit different records in the same database). The total number of records in the database increases in proportion to the number of children. (With one child, *nrec* records are originally written to the database. With two children, *nrec* × 2 records are originally written, and so on.)

To test the concurrency provided by coarse locking versus fine locking and to compare the three different types of locking (no locking, advisory locking, and mandatory

locking), we ran three versions of the test program. The first version used the source code shown in Section 16.7, which we've called fine locking. The second version changed the locking calls to implement coarse locking, as described in Section 16.6. The third version had all locking calls removed, so we could measure the overhead involved in locking. We can run the first and second versions (fine locking and coarse locking) using either advisory or mandatory locking, by changing the permission bits on the database files. (In all the tests reported in this section, we measured the times for mandatory locking using only the implementation of fine locking.)

All the timing tests in this sections were done on an 80386 system running SVR4.

## Single-Process Results

Figure 16.5 shows the results when only a single child process ran, with an *nrec* of 500, 100, and 2000.

| | No locking | | | Advisory locking | | | | | | Mandatory locking | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Coarse locking | | | Fine locking | | | Fine locking | | |
| *nrec* | User | Sys | Clock | User | Sys | Clock | User | Sys | Clock | User | Sys | Clock |
| 500 | 15 | 68 | 84 | 16 | 78 | 94 | 15 | 79 | 94 | 16 | 92 | 109 |
| 1000 | 61 | 340 | 402 | 63 | 360 | 425 | 63 | 366 | 430 | 71 | 412 | 488 |
| 2000 | 157 | 906 | 1068 | 158 | 936 | 1096 | 158 | 934 | 1097 | 159 | 1081 | 1253 |

**Figure 16.5** Single child, varying *nrec*, different locking techniques.

The last 12 columns give the corresponding times in seconds. In all cases the user CPU time plus the system CPU time approximately equals the clock time. This set of tests was CPU limited and not disk limited.

The middle six columns (advisory locking, coarse and fine) are almost equal for each row. This makes sense—for a single process there is no difference between coarse locking and fine locking.

Comparing no locking versus advisory locking, we see that adding the locking calls adds between 3% and 15% to the system CPU time. Even though the locks are never used (since only a single process is running), the system call overhead in the calls to `fcntl` adds time. Also note that the user CPU time is about the same for all four versions of locking. Since the user code is almost equivalent (except for the number of calls to `fcntl`) this makes sense.

The final point to note from Figure 16.5 is that mandatory locking adds about 15% to the system CPU time, compared to advisory locking. Since the number of locking calls are the same for advisory fine locking and mandatory fine locking, the additional system call overhead must be in the `reads` and `writes`.

The final test that was run was to try the no-locking program with multiple children. The results, as expected, were random errors. Normally, records that were added to the database couldn't be found, and the test program aborted. Every time the test program was run, different errors occurred. This is a classic race condition—having multiple processes updating the same file without using any form of locking.

## Multiple-Process Results

The next set of measurements looks mainly at the differences between coarse locking and fine locking. As we said earlier, intuitively we expect fine locking to provide additional concurrency, since there is less time that portions of the database are locked from other processes. Figure 16.6 shows the results, for an *nrec* of 500, varying the number of children from 1 to 12.

| | Advisory locking | | | | | | | Mandatory locking | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Coarse locking | | | Fine locking | | | Δ | Fine locking | | | Δ |
| #Proc | User | Sys | Clock | User | Sys | Clock | Clock | User | Sys | Clock | Percent |
| 1 | 16 | 79 | 96 | 16 | 83 | 99 | 3 | 16 | 96 | 112 | 16 |
| 2 | 42 | 230 | 273 | 43 | 237 | 281 | 8 | 43 | 271 | 315 | 14 |
| 3 | 79 | 454 | 536 | 81 | 464 | 547 | 11 | 78 | 545 | 626 | 18 |
| 4 | 128 | 753 | 884 | 132 | 757 | 892 | 8 | 123 | 888 | 1015 | 17 |
| 5 | 185 | 1123 | 1315 | 196 | 1173 | 1376 | 61 | 189 | 1366 | 1560 | 16 |
| 6 | 262 | 1601 | 1870 | 270 | 1611 | 1888 | 18 | 264 | 1931 | 2205 | 20 |
| 7 | 351 | 2164 | 2526 | 354 | 2174 | 2537 | 11 | 341 | 2527 | 2877 | 16 |
| 8 | 451 | 2801 | 3264 | 454 | 2766 | 3230 | −34 | 438 | 3298 | 3750 | 19 |
| 9 | 565 | 3513 | 4092 | 569 | 3483 | 4067 | −25 | 548 | 4148 | 4712 | 19 |
| 10 | 684 | 4293 | 5000 | 688 | 4215 | 4925 | −75 | 658 | 5048 | 5732 | 20 |
| 11 | 812 | 5151 | 5987 | 811 | 5043 | 5876 | −111 | 797 | 6198 | 7020 | 23 |
| 12 | 958 | 6075 | 7058 | 960 | 5992 | 6980 | −78 | 937 | 7298 | 8265 | 22 |

**Figure 16.6** Comparison of different locking techniques, *nrec* = 500.

All the user, system, and clock times are in seconds. All these times are the total for the parent and all its children. There are many items to consider from this data.

The eighth column, labeled "Δ clock," is the difference in seconds between the clock times from advisory-coarse locking to advisory-fine locking. This is the measurement of how much concurrency we obtain by going from coarse locking to fine locking. On the system used for these tests, coarse locking is faster, until we have more than seven processes. Even after seven processes, the decrease in clock time using fine locking isn't that great (around 1%), which makes us wonder if the additional code required to implement fine locking is worth the effort.

We would like the clock time to decrease, from coarse to fine locking, as it eventually does, but we expect the system time to remain higher for fine locking, for any number of processes. The reason we expect this is because with fine locking we are issuing more `fcntl` calls than with coarse locking. If we total the number of `fcntl` calls in Figure 16.4 for coarse locking and fine locking, we have an average of 22,110 for coarse locking and 25,680 for fine locking. (To get these numbers, realize that each of the operations in Figure 16.4 requires two calls to `fcntl` for coarse locking, and the first three calls to `db_store` along with record deletion (record found) each requires four calls to `fcntl` for fine locking.) We expect this increase of 16% in the number of calls to `fcntl` to result in an increased system time for fine locking. Therefore the slight decrease in system time for fine locking, when the number of processes exceeds seven, is puzzling.

The final column, labeled "Δ percent," is the percentage increase in the system CPU time from advisory-fine locking to mandatory-fine locking. These percentages verify

what we saw in Figure 16.5, that mandatory locking adds around 15–20% to the system time.

Since the user code for all these tests is almost identical (there are some additional `fcntl` calls for both advisory-fine and mandatory-fine locking), we expect the user CPU times to be the same across any row. But the user CPU times always increase 1–3% from advisory-coarse locking to advisory-fine locking. The user CPU times always decrease 1–3% from advisory-fine locking to mandatory-fine locking. There is no apparent explanation for these differences.

The values in the first row of Figure 16.6 are similar to those for an *nrec* of 500 in Figure 16.5. We expect this.

Figure 16.7 is a graph of the data from Figure 16.6, for advisory fine locking. We plot the clock time as the number of processes goes from one to nine. (We don't plot the values for 10, 11, and 12, to avoid expanding the graph in the vertical direction.) We also plot the user CPU time divided by the number of processes and the system CPU time divided by the number of processes.

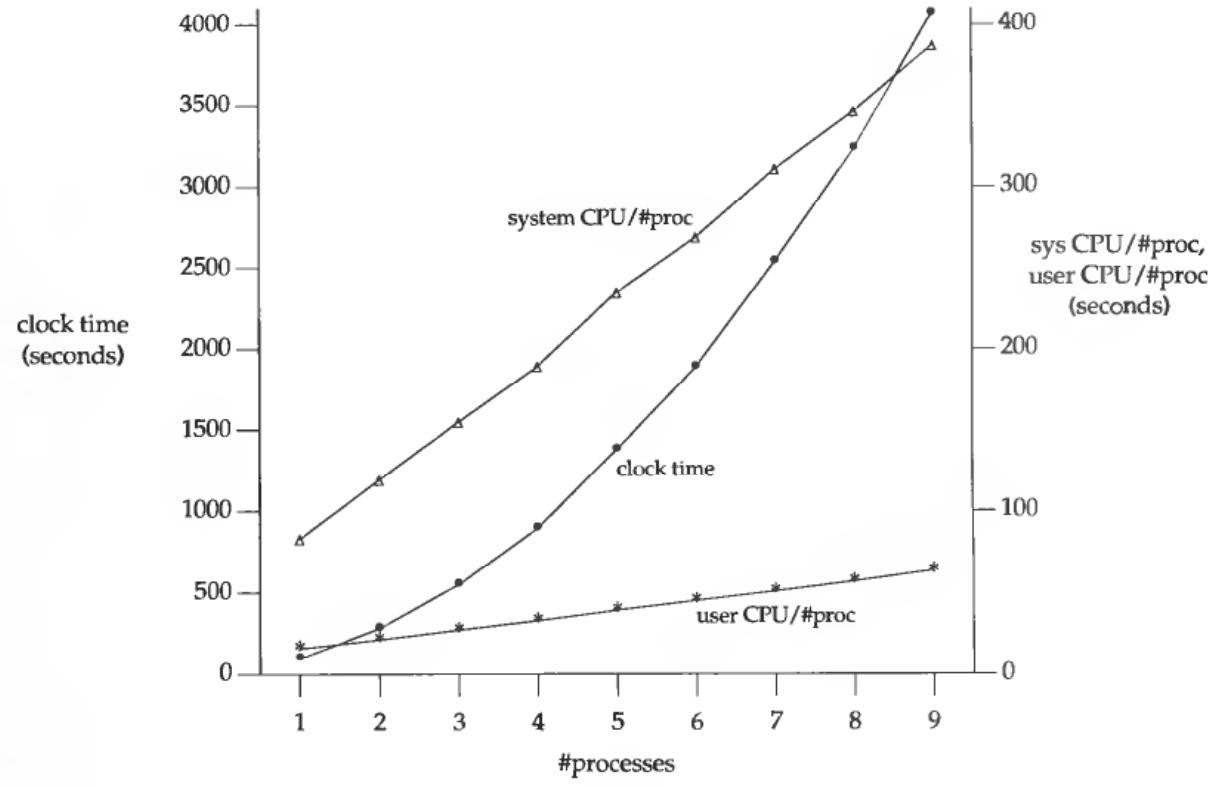


**Figure 16.7** Values from Figure 16.6 for advisory-fine locking.

Note that both CPU times, divided by the number of processes, are linear, but the plot of the clock time is nonlinear. If we sum the user CPU time and system CPU time from Figure 16.6 and compare it to the clock time for a given row, the difference between the two increases as the number of processes increases. The probable reason is the added amount of CPU time used by the operating system as the number of processes

increases. This operating system overhead would show up as an increased clock time, but shouldn't affect the CPU times of the individual processes.

The reason the user CPU time increases with the number of processes is because there are more records in the database. Each hash chain is getting longer, so it takes the `_db_find` function longer, on the average, to find a record.

## 16.9 Summary

This chapter has taken a long look at the design and implementation of a database library. Although we've kept the library small and simple, for presentation purposes, it contains the record locking required to allow concurrent access by multiple processes.

We've also looked at the performance of this library, with various number of processes, using four different types of locking: no locking, advisory locking (fine and coarse), and mandatory locking. We saw that advisory locking adds about 10% to the clock time over no locking, and mandatory locking adds another 10% over advisory locking.

## Exercises

**16.1** The locking in `_db_dodelete` is somewhat conservative. For example, we could allow more concurrency by not write locking the free list until we really need to; that is, the call to `writew_lock` could be moved between the calls to `_db_writedat` and `_db_readptr`. What happens if we do this?

**16.2** If `db_nextrec` did not read lock the free list and a record that it was reading was also in the process of being deleted, describe how `db_nextrec` could return the correct key but an all-blank (hence incorrect) data record. (Hint: look at `_db_dodelete`.)

**16.3** After the discussion of `db_store` we described the locking performed by `_db_writeidx` and `_db_writedat`. We said that this locking didn't interfere with other readers and writers except those making calls to `db_store`. Is this true if mandatory locking is being used?

**16.4** How would you integrate the `fsync` function into this database library?

**16.5** Create a new database and write some number of records to the database. Write a program that calls `db_nextrec` to read each record in the database and call `_db_hash` to calculate the hash value for each record. Print a histogram of the number of records on each hash chain. Is the hashing function in Program 16.9 adequate?

**16.6** Modify the database functions so that the number of hash chains in the index file can be specified when the database is created.

**16.7** If your systems support a network filesystem, such as Sun's Network File System (NFS) or AT&T's Remote File Sharing (RFS), compare the performance of the database functions when the database is (a) on the same host as the test program and (b) on a different host. Does the record locking provided by the database library still work?

# 17

# Communicating with a PostScript Printer

## 17.1 Introduction

We now develop a program that can communicate with a PostScript printer. PostScript printers are popular today and normally communicate with a host using an RS-232 serial interface. This gives us a chance to use some of the terminal I/O functions from Chapter 11. Also, communication with a PostScript printer is full duplex, meaning that as we send data to the printer we also have to be prepared to read status information from the printer. This gives us a chance to use the I/O multiplexing functions from Section 12.5: `select` and `poll`. The program that we develop is based on the `lprps` program written by James Clark. This program and others, making up the `lprps` package, was posted to the `comp.sources.misc` Usenet news group, Volume 21 (July 1991).

## 17.2 PostScript Communication Dynamics

The first thing to realize about printing on a PostScript printer is that we don't send a file to the printer to be printed—we send a PostScript program to the printer for it to execute. There is normally a PostScript interpreter within the printer that executes the program, generating one or more pages of printed output. If the PostScript program contains errors, the printer (actually the PostScript interpreter) returns an error message and may or may not generate any output.

The following PostScript program causes the familiar string to be printed on a page. (We won't describe PostScript programming in this text, see Adobe Systems [1985 and 1986] for these details. Our interest is in communicating with a PostScript printer.)

```
%!
/Times-Roman findfont
15 scalefont              % point size of 15
setfont                   % establish current font
300 350 moveto            % x=300, y=350 (position on page)
(hello, world) show       % output the string to current page
showpage                  % and output page to output device
```

If we change the word `setfont` to `ssetfont` in the PostScript program and send it to the printer, nothing is printed. Instead we get the following messages back from the printer

```
%%[ Error: undefined; OffendingCommand: ssetfont ]%%
%%[ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

These error messages, which can arrive from the printer at any time, are what complicate the handling of a PostScript printer. We can't just send the entire PostScript program to the printer and forget about it—we must handle these potential error messages intelligently. (Throughout this chapter we'll usually say "printer" when technically we mean the PostScript interpreter.)

PostScript printers are usually attached to a host computer using an RS-232 serial connection. This looks to the host like a terminal connection. Everything that we said about terminal I/O in Chapter 11 applies here. (There are other ways to connect PostScript printers to a host: network interfaces are becoming popular. The predominant interface these days is a serial connection.) Figure 17.2 shows the typical arrangement. A PostScript program can generate two forms of output—output on the printed page from the `showpage` operator and output to its standard output (the serial link to the host in this case) from the `print` operator.

The PostScript interpreter sends and receives seven-bit ASCII characters. A PostScript program consists entirely of printable ASCII characters. Some of the nonprinting ASCII characters have special meaning, as listed in Figure 17.1.

| Character | Octal value | Description |
|---|---|---|
| Control-C | 003 | Interrupt. Causes the PostScript `interrupt` operator to be executed. Normally this terminates the PostScript program being interpreted. |
| Control-D | 004 | End of file. |
| Line feed | 012 | End of line, the PostScript newline character. If a return and line feed are received in sequence, only a single newline character is passed to the interpreter. |
| Return | 015 | End of line. Translated to the PostScript newline character. |
| Control-Q | 021 | Start output (XON flow control). |
| Control-S | 023 | Stop output (XOFF flow control). |
| Control-T | 024 | Status query. The PostScript interpreter responds with a one-line status message. |

**Figure 17.1** Special characters sent from computer to PostScript interpreter.

The PostScript end-of-file character (Control-D) is used to synchronize the printer with the host. We send a PostScript program to the printer and then send an EOF to the

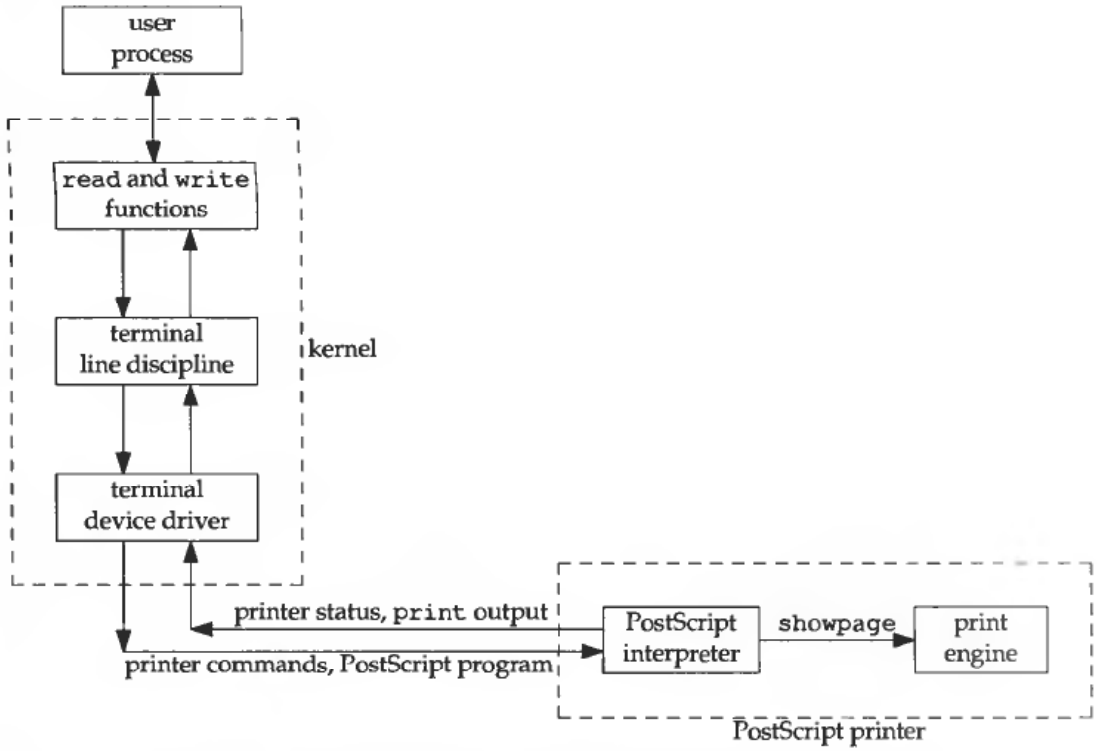


**Figure 17.2** Communicating with a PostScript printer using a serial connection.

printer. When the printer has finished executing the PostScript program, it sends an EOF back.

While the interpreter is executing a PostScript program we can send it an interrupt (Control-C). This normally causes the program being executed by the printer to terminate.

The status query message (Control-T) causes a one-line status message to be returned by the printer. All messages received from the printer have the following format:

```
%%[ key: val ]%%
```

Any number of *key: val* pairs can appear in a message, separated by semicolons. Recall the messages returned in the earlier example:

```
%%[ Error: undefined; OffendingCommand: ssetfont ]%%
%%[ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

The status messages have the form

```
%%[ status: idle ]%%
```

Other status indications, besides `idle` (no job in progress), are `busy` (executing a PostScript program), `waiting` (waiting for more of the PostScript program to execute), `printing` (paper in motion), `initializing`, and `printing test page`.

We now consider the messages that are spontaneously generated by the PostScript interpreter. We've already seen the message

```
%%[ Error: error; OffendingCommand: operator ]%%
```

About 25 different *errors* can occur. Common *errors* are `dictstackunderflow`, `invalidaccess`, `typecheck`, and `undefined`. The *operator* is the PostScript operator that generated the error.

A printer error is indicated by a message of the form

```
%%[ PrinterError: reason ]%%
```

where *reason* is often `Out Of Paper`, `Cover Open`, or `Miscellaneous Error`.

After an error has occurred, the PostScript interpreter often sends a second message

```
%%[ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

To handle these messages we have to parse the message string, looking only at the characters within a pair of the special sequences `%%[` and `]%%`. A PostScript program can also generate output from the PostScript `print` operator. This output should be sent to the user who sent the program to the printer—it is not output that our printing program should try to interpret.

Figure 17.3 lists the special characters that are sent by the PostScript interpreter to the host computer.

| Character | Octal value | Description |
|---|---|---|
| Control-D | 004 | End of file. |
| Line feed | 012 | Newline. When a newline is written to the interpreter's standard output, it is translated to a return followed by a line feed. |
| Control-Q | 021 | Start output (XON flow control). |
| Control-S | 023 | Stop output (XOFF flow control). |

**Figure 17.3** Special characters sent from PostScript interpreter to computer.

## 17.3 Printer Spooling

The program that we develop in this chapter sends a PostScript program to a PostScript printer in either stand-alone mode or through the BSD line printer spooling system. Normal usage is within a spooling system, but it is useful to provide a stand-alone (debug) mode, for testing.

SVR4 also provides a spooling system, albeit more complicated than the BSD system. Details of this spooling system can be found in all the manual pages that begin with `lp` in Section 1 of AT&T [1991]. Chapter 13 of Stevens [1990] provides details on the BSD spooling system and the pre-SVR4 System V spooling system. Our interest in this chapter is not in these spooling systems per se, but in communicating with a PostScript printer.

In the BSD spooling system we print a file with a command of the form

```
lpr -pps main.c
```

This sends the file `main.c` to the printer whose name is `ps`. If we didn't specify `-pps` the output would be sent to either the printer specified by the `PRINTER` environment variable or to the default printer `lp`. The printer is looked up in the file `/etc/printcap`. Figure 17.4 shows an entry for our PostScript printer.

```
lp|ps:\
    :br#19200:lp=/dev/ttyb:\
    :sf:sh:rw:\
    :fc#0000374:fs#0000003:xc#0:xs#0040040:\
    :af=/var/adm/psacct:lf=/var/adm/pslog:sd=/var/spool/pslpd:\
    :if=/usr/local/lib/psif:
```

**Figure 17.4** The `printcap` entry for the PostScript printer.

The first line gives the name of this entry as either `ps` or `lp`. The `br` value specifies the baud rate as 19200. `lp` specifies the pathname of the special device file for the printer. `sf` says to suppress form feeds, and `sh` says to suppress printing a burst page header at the beginning of each job. `rw` specifies that the device is to be opened for reading and writing. This is required for a PostScript printer, as described in Section 17.2.

The next four fields specify bits to turn off and turn on in the old BSD-style `sgtty` structure. (We describe these here because most BSD systems that use this form of `printcap` file support this older style of setting terminal parameters. In the source code later in this chapter we'll see how to set all the terminal parameters with the POSIX.1 functions from Chapter 11.) First the `fc` mask clears the following bits in the `sg_flags` element: `EVENP` and `ODDP` (turns off parity checking and generation), `RAW` (turns off raw mode), `CRMOD` (turns off CR/LF mapping on input and output), `ECHO` (turns off echo), and `LCASE` (turns off uppercase/lowercase mapping on input and output). Then the `fs` mask turns on the following bits: `CBREAK` (one character-at-a-time input), and `TANDEM` (host generates Control-S, Control-Q flow control). Next, the `xc` value clears bits in the local mode word. In this example the value of 0 does nothing. Finally, the `xs` value sets bits in the local mode word: `LDECCTQ` (only Control-Q restarts output that was stopped by a Control-S), and `LLITOUT` (suppress output translations).

The `af` and `lf` strings specify the accounting file and log file, respectively. `sd` specifies the spooling directory, and `if` specifies the input filter.

The input filter is invoked for every file to be printed. It is invoked as

*filter* `-n` *loginname* `-h` *hostname acctfile*

There are several optional arguments that can also appear, which can be safely ignored for a PostScript printer. The file to be printed is on the standard input, and the printer device (from the `lp` entry in the `printcap` file) is open on the standard output. The standard input can be a pipe.

With a PostScript printer, the input filter should look at the first two bytes of the input file and determine if the file is an ASCII text file or a PostScript program. The

normal convention is that the two-character sequence `%!` at the beginning of a file designates a PostScript program. If the file is a PostScript program, the `lprps` program (detailed later in this chapter) can send it to the printer. But if the file is an ASCII text file, a program is required to convert this into a PostScript program that prints the text file.

The filter `psif`, mentioned in the `printcap` file, is supplied with the `lprps` package. The program `textps` in this package converts an ASCII text file into a PostScript program that prints the file. Figure 17.5 outlines all these programs.

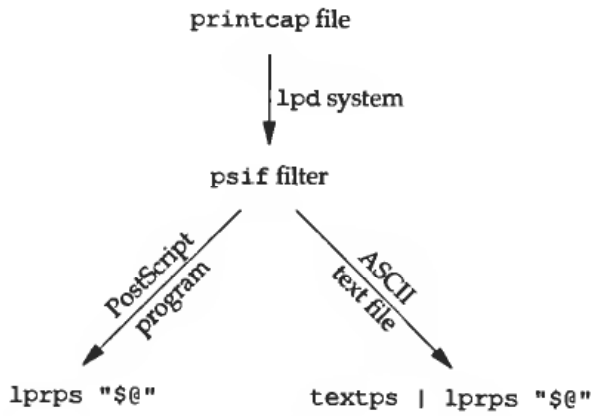


**Figure 17.5** Overview of `lprps` system.

There is another program not shown in this figure, `psrev`, that reverses the pages of output generated by a PostScript program. This can be used if the PostScript printer generates its output face up instead of face down.

Having covered all these preliminaries, we can now look at the design and source code of the `lprps` program.

## 17.4 Source Code

Let's start with an overview of the functions called by `main` and how they interact with the printer. Figure 17.6 details this interaction. The second column, labeled "Int?", specifies if the function is interruptible with a `SIGINT` signal. The third column specifies the time-out value (in seconds) set by the function. Notice that when we're sending the user's PostScript program to the printer, there is no time out. This is because a PostScript program can take any amount of time to execute. The reference to "our PostScript program" for the `get_page` function refers to the small PostScript program in Program 17.9 that fetches the current page counter.

Program 17.1 lists the header `lprps.h`. It is included by all the source files. This header includes the system headers that most files require, defines some constants, and declares the global variables and function prototypes for the global functions.

| Function | Int? | Timeout? | Send to printer | Receive from printer |
|---|---|---|---|---|
| get_status | no | 5 | Control-T | %%[ status: idle ]%% |
| get_page | no | 30 | our PostScript program<br>EOF | %%[ pagecount: *n* ]%%<br>EOF |
| send_file | yes | none | user's PostScript program<br>EOF | EOF |
| get_page | no | 30 | our PostScript program<br>EOF | %%[ pagecount: *n* ]%%<br>EOF |

**Figure 17.6** Functions called by `main`.

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    <errno.h>
#include    <signal.h>
#include    <syslog.h>      /* since we're a daemon */
#include    "ourhdr.h"

#define EXIT_SUCCESS    0   /* defined by BSD spooling system */
#define EXIT_REPRINT    1
#define EXIT_THROW_AWAY 2

#define DEF_DEVICE  "/dev/ttyb" /* defaults for debug mode */
#define DEF_BAUD    B19200

                            /* modify following as appropriate */
#define MAILCMD     "mail -s \"printer job\" %s@%s < %s"

#define OBSIZE  1024    /* output buffer */
#define IBSIZE  1024    /* input buffer */
#define MBSIZE  1024    /* message buffer */

                /* declare global variables */
extern char *loginname;
extern char *hostname;
extern char *acct_file;
extern char  eofc;      /* PS end-of-file (004) */
extern int   debug;     /* true if interactive (not a daemon) */
extern int   in_job;    /* true if sending user's PS job to printer */
extern int   psfd;      /* file descriptor for PostScript printer */
extern int   start_page;/* starting page# */
extern int   end_page;  /* ending page# */
```

```
extern volatile sig_atomic_t    intr_flag; /* set if SIGINT is caught */
extern volatile sig_atomic_t    alrm_flag; /* set if SIGALRM goes off */

extern enum status {    /* printer status */
    INVALID, UNKNOWN, IDLE, BUSY, WAITING
} status;

                /* global function prototypes */
void    do_acct(void);              /* acct.c */

void    clear_alrm(void);           /* alarm.c */
void    handle_alrm(void);
void    set_alrm(unsigned int);

void    get_status(void);           /* getstatus.c */

void    init_input(int);            /* input.c */
void    proc_input_char(int);
void    proc_some_input(void);
void    proc_upto_eof(int);

void    clear_intr(void);           /* interrupt.c */
void    handle_intr(void);
void    set_intr(void);

void    close_mailfp(void);         /* mail.c */
void    mail_char(int);
void    mail_line(const char *, const char *);

void    msg_init(void);             /* message.c */
void    msg_char(int);
void    proc_msg(void);

void    out_char(int);              /* output.c */

void    get_page(int *);            /* pagecount.c */

void    send_file(void);            /* sendfile.c */

void    block_write(const char *, int); /* tty.c */
void    tty_flush(void);
void    set_block(void);
void    set_nonblock(void);
void    tty_open(void);
```

**Program 17.1** The `lprps.h` header.

The file `vars.c` (Program 17.2) defines the global variables.

Execution starts at the `main` function, shown in Program 17.3. The `main` function calls the `log_open` function (shown in Appendix B) since this program normally runs as a daemon. We cannot write error messages to the standard error—instead we use the `syslog` facility described in Section 13.4.2.

```
#include    "lprps.h"

char    *loginname;
char    *hostname;
char    *acct_file;
char    eofc = '\004';          /* Control-D = PostScript EOF */

int     psfd = STDOUT_FILENO;
int     start_page = -1;
int     end_page = -1;
int     debug;
int     in_job;

volatile sig_atomic_t   intr_flag;
volatile sig_atomic_t   alrm_flag;

enum status     status = INVALID;
```

**Program 17.2** Declare the global variables.

```
#include    "lprps.h"

static void usage(void);

int
main(int argc, char *argv[])
{
    int         c;

    log_open("lprps", LOG_PID, LOG_LPR);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "cdh:i:l:n:x:y:w:")) != EOF) {
        switch (c) {
        case 'c':       /* control chars to be passed */
        case 'x':       /* horizontal page size */
        case 'y':       /* vertical page size */
        case 'w':       /* width */
        case 'l':       /* length */
        case 'i':       /* indent */
            break;  /* not interested in these */

        case 'd':       /* debug (interactive) */
            debug = 1;
            break;

        case 'n':       /* login name of user */
            loginname = optarg;
            break;

        case 'h':       /* host name of user */
            hostname = optarg;
            break;
```

```
            case '?':
                log_msg("unrecognized option: -%c", optopt);
                usage();
            }
        }

        if (hostname == NULL || loginname == NULL)
            usage();    /* require both hostname and loginname */

        if (optind < argc)
            acct_file = argv[optind];   /* remaining arg = acct file */

        if (debug)
            tty_open();

        if (atexit(close_mailfp) < 0)   /* register func for exit() */
            log_sys("main: atexit error");

        get_status();

        get_page(&start_page);

        send_file();            /* copies stdin to printer */

        get_page(&end_page);

        do_acct();

        exit(EXIT_SUCCESS);
}

static void
usage(void)
{
        log_msg("lprps: invalid arguments");
        exit(EXIT_THROW_AWAY);
}
```

**Program 17.3** The `main` function.

The command-line arguments are then processed, many of which can be ignored for a PostScript printer. We use the `-d` flag to indicate that the program is being run interactively, not as a daemon. If this flag is set, we need to initialize the terminal mode (`tty_open`). We describe the function `close_mailfp`, which we establish as an exit handler, later.

We then call the functions that we mentioned in Figure 17.6: fetch the printer status to assure it is ready (`get_status`), get the printer's starting page count (`get_page`), send the file (the PostScript program) to the printer (`send_file`), get the printer's ending page count (`get_page`), write an accounting record (`do_acct`), and terminate.

The file `acct.c` defines the function `do_acct` (Program 17.4). It is called at the end of `main` to write an accounting record. The name of the accounting file is taken from the `printcap` entry (Figure 17.4) and passed as the final command-line argument.

```
#include    "lprps.h"

/* Write the number of pages, hostname, and loginname to the
 * accounting file.  This function is called by main() at the end
 * if all was OK, by printer_flushing(), and by handle_intr() if
 * an interrupt is received. */

void
do_acct(void)
{
    FILE    *fp;

    if (end_page > start_page &&
        acct_file != NULL &&
        (fp = fopen(acct_file, "a")) != NULL) {
            fprintf(fp, "%7.2f %s:%s\n",
                        (double)(end_page - start_page),
                        hostname, loginname);
            if (fclose(fp) == EOF)
                log_sys("do_acct: fclose error");
    }
}
```

**Program 17.4** The do_acct function.

Historically all BSD print filters write the number of pages output to the accounting file with the `%7.2f` `printf` format. This allows raster devices to report output in feet (and fractions thereof), instead of pages.

The next file, `tty.c` (Program 17.5), contains all the terminal I/O functions. These call the functions we described in Chapter 3 (`fcntl`, `write`, and `open`), and the POSIX.1 terminal functions from Chapter 11 (`tcflush`, `tcgetattr`, `tcsetattr`, `cfsetispeed`, and `cfsetospeed`). There are times when we don't care if a `write` blocks, and we'll call `block_write` for these cases. But if we don't want to block, we call `set_nonblock` and then call `read` or `write` ourself. Since a PostScript printer is a full-duplex device, we don't want to block on a `write` if there is a chance that the printer might want to send data to us (such as an error message). If the printer sends us an error message while we're blocked trying to send it data, we can encounter a deadlock.

The kernel normally buffers terminal input and output, so if an error condition is encountered we call `tty_flush` to flush both the input and output queue.

The function `tty_open` is called from `main` if we're running interactively (not as a daemon). We need to set the terminal mode to noncanonical, set the baud rates, and set any other terminal flags. Be aware that these settings are not the same for all PostScript printers. Check your printer manuals for its settings. (The number of bits of data, seven-bit or eight-bit, the number of start and stop bits, and the parity, are most likely to change between printers.)

```
#include    "lprps.h"
#include    <fcntl.h>
#include    <termios.h>

static int      block_flag = 1;     /* default is blocking I/O */

void
set_block(void)     /* turn off nonblocking flag */
{                   /* called only by block_write() below */
    int     val;

    if (block_flag == 0) {
        if ( (val = fcntl(psfd, F_GETFL, 0)) < 0)
            log_sys("set_block: fcntl F_GETFL error");
        val &= ~O_NONBLOCK;
        if (fcntl(psfd, F_SETFL, val) < 0)
            log_sys("set_block: fcntl F_SETFL error");

        block_flag = 1;
    }
}

void
set_nonblock(void)  /* set descriptor nonblocking */
{
    int     val;

    if (block_flag) {
        if ( (val = fcntl(psfd, F_GETFL, 0)) < 0)
            log_sys("set_nonblock: fcntl F_GETFL error");
        val |= O_NONBLOCK;
        if (fcntl(psfd, F_SETFL, val) < 0)
            log_sys("set_nonblock: fcntl F_SETFL error");

        block_flag = 0;
    }
}

void
block_write(const char *buf, int n)
{
    set_block();
    if (write(psfd, buf, n) != n)
        log_sys("block_write: write error");
}

void
tty_flush(void)     /* flush (empty) tty input and output queues */
{
    if (tcflush(psfd, TCIOFLUSH) < 0)
        log_sys("tty_flush: tcflush error");
}
```

```
void
tty_open(void)
{
    struct termios  term;

    if ( (psfd = open(DEF_DEVICE, O_RDWR)) < 0)
        log_sys("tty_open: open error");

    if (tcgetattr(psfd, &term) < 0)     /* fetch attributes */
        log_sys("tty_open: tcgetattr error");
    term.c_cflag  = CS8 |               /* 8-bit data */
                    CREAD |             /* enable receiver */
                    CLOCAL;             /* ignore modem status lines */
                                        /* no parity, 1 stop bit */
    term.c_oflag &= ~OPOST;             /* turn off post processing */
    term.c_iflag  = IXON | IXOFF |      /* Xon/Xoff flow control */
                    IGNBRK |            /* ignore breaks */
                    ISTRIP |            /* strip input to 7 bits */
                    IGNCR;              /* ignore received CR */
    term.c_lflag  = 0;      /* everything off in local flag:
                               disables canonical mode, disables
                               signal generation, disables echo */
    term.c_cc[VMIN]  = 1;   /* 1 byte at a time, no timer */
    term.c_cc[VTIME] = 0;
    cfsetispeed(&term, DEF_BAUD);
    cfsetospeed(&term, DEF_BAUD);
    if (tcsetattr(psfd, TCSANOW, &term) < 0)    /* set attributes */
        log_sys("tty_open: tcsetattr error");
}
```

**Program 17.5** Terminal functions.

The program handles two signals: `SIGINT` and `SIGALRM`. Handling `SIGINT` is a requirement for any filter invoked by the BSD spooling system. This signal is sent to the filter if the printer job is removed by the `lprm`(1) command. We use `SIGALRM` for setting time outs. Both signals are handled in a similar fashion: we provide a `set_XXX` function to establish the signal handler, and a `clear_XXX` function to disable the signal handler. If the signal is delivered to the process the signal handler just sets a global flag, `intr_flag` and `alrm_flag`, and returns. It is up to the rest of the program to test these flags at the appropriate times, to see if the signal has been caught. One obvious time is after an I/O function returns an error of `EINTR`. The program then calls either `handle_intr` or `handle_alrm` to handle the condition. We call the `signal_intr` function (Program 10.13) so that either signal interrupts a slow system call. Program 17.6 shows the file `interrupt.c` that handles `SIGINT`.

When an interrupt occurs we have to send the PostScript interrupt character (Control-C) to the printer, followed by an EOF. This normally causes the PostScript interpreter to abort the program that it's interpreting. We then wait for an EOF back from the printer. (We describe the function `proc_upto_eof` later.) We finish up by reading the ending page count, writing an accounting record, and terminating.

```
#include    "lprps.h"

static void
sig_int(int signo)      /* SIGINT handler */
{
    intr_flag = 1;
    return;
}

/* This function is called after SIGINT has been delivered,
 * and the main loop has recognized it.  (It not called as
 * a signal handler, set_intr() above is the handler.) */

void
handle_intr(void)
{
    char    c;

    intr_flag = 0;
    clear_intr();           /* turn signal off */

    set_alrm(30);       /* 30 second timeout to interrupt printer */

    tty_flush();            /* discard any queued output */
    c = '\003';
    block_write(&c, 1);     /* Control-C interrupts the PS job */
    block_write(&eofc, 1);  /* followed by EOF */
    proc_upto_eof(1);       /* read & ignore up through EOF */

    clear_alrm();

    get_page(&end_page);
    do_acct();
    exit(EXIT_SUCCESS);     /* success since user lprm'ed the job */
}

void
set_intr(void)      /* enable signal handler */
{
    if (signal_intr(SIGINT, sig_int) == SIG_ERR)
        log_sys("set_intr: signal_intr error");
}

void
clear_intr(void)    /* ignore signal */
{
    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
        log_sys("clear_intr: signal error");
}
```

**Program 17.6** The `interrupt.c` file to handle interrupt signals.

Figure 17.6 noted which functions set time outs. We set a time out only when we request the printer status (get_status), when we read the printer's page count (get_page), or when we're interrupting the printer (handle_intr). If a time out does occur, we just log an error, wait for a while, and terminate. Program 17.7 shows the file alarm.c.

```
#include    "lprps.h"

static void
sig_alrm(int signo)          /* SIGALRM handler */
{
    alrm_flag = 1;
    return;
}

void
handle_alrm(void)
{
    log_ret("printer not responding");
    sleep(60);      /* it will take at least this long to warm up */

    exit(EXIT_REPRINT);
}

void        /* Establish the signal handler and set the alarm. */
set_alrm(unsigned int nsec)
{
    alrm_flag = 0;
    if (signal_intr(SIGALRM, sig_alrm) == SIG_ERR)
        log_sys("set_alrm: signal_intr error");
    alarm(nsec);
}

void
clear_alrm(void)
{
    alarm(0);
    if (signal(SIGALRM, SIG_IGN) == SIG_ERR)
        log_sys("clear_alrm: signal error");
    alrm_flag = 0;
}
```

**Program 17.7** The alarm.c file to handle time outs.

Program 17.8 shows the function get_status, which we called from main. It fetches the status by sending a Control-T to the printer. The printer should respond with a one-line message. The message that we're looking for is

```
%%[ status: idle ]%%
```

which means the printer is ready for a new job. This message is read and processed by proc_some_input, which we look at later.

```
#include    "lprps.h"

/* Called by main() before printing job.
 * We send a Control-T to the printer to fetch its status.
 * If we timeout before reading the printer's status, something
 * is wrong. */

void
get_status(void)
{
    char    c;

    set_alrm(5);            /* 5 second timeout to fetch status */

    tty_flush();
    c = '\024';
    block_write(&c, 1);     /* send Control-T to printer */

    init_input(0);
    while (status == INVALID)
        proc_some_input();  /* wait for something back */

    switch (status) {
    case IDLE:      /* this is what we're looking for ... */
        clear_alrm();
        return;

    case WAITING:   /* printer thinks it's in the middle of a job */
        block_write(&eofc, 1);  /* send EOF to printer */
        sleep(5);
        exit(EXIT_REPRINT);

    case BUSY:
    case UNKNOWN:
        sleep(15);
        exit(EXIT_REPRINT);
    }
}
```

**Program 17.8** The `get_status` function.

If we receive the message

```
%%[ status: waiting ]%%
```

it means the printer is waiting for us to send it more data for a job that it is currently printing. This means something funny happened to the previous job. To clear this state we send the printer an EOF, then terminate.

PostScript printers maintain a page counter. It is incremented each time a page is printed and is maintained even when the power is turned off. To read this counter requires us to send the printer a PostScript program. The file `pagecount.c` (Program 17.9) contains this small PostScript program (about a dozen PostScript operators) and the function `get_page` that sends this program to the printer.

```
#include    "lprps.h"

/* PostScript program to fetch the printer's pagecount.
 * Notice that the string returned by the printer:
 *      %%[ pagecount: N ]%%
 * will be parsed by proc_msg(). */

static char pagecount_string[] =
    "(%%[ pagecount: ) print "  /* print writes to current output file */
    "statusdict begin pagecount end "   /* push pagecount onto stack */
    "20 string "        /* creates a string of length 20 */
    "cvs "              /* convert to string */
    "print "            /* write to current output file */
    "( ]%%) print "
    "flush\n";          /* flush current output file */

/* Read the starting or ending pagecount from the printer.
 * The argument is either &start_page or &end_page. */

void
get_page(int *ptrcount)
{
    set_alrm(30);           /* 30 second timeout to read pagecount */

    tty_flush();
    block_write(pagecount_string, sizeof(pagecount_string) - 1);
                            /* send query to printer */
    init_input(0);
    *ptrcount = -1;
    while (*ptrcount < 0)
        proc_some_input();  /* read results from printer */

    block_write(&eofc, 1);  /* send EOF to printer */
    proc_upto_eof(0);       /* wait for EOF from printer */

    clear_alrm();
}
```

**Program 17.9** The pagecount.c file—fetch the printer's page count.

The array `pagecount_string` contains the small PostScript program. Although we could fetch the page count and print it using just

```
statusdict begin pagecount end = flush
```

we purposely format the output to look like a status message returned by the printer:

```
%%[ pagecount: N ]%%
```

By doing this the function `proc_some_input` handles the message similar to any printer status message.

The function `send_file` in Program 17.10 is called by `main` to send the user's PostScript program to the printer.

```
#include    "lprps.h"

void
send_file(void)     /* called by main() to copy stdin to printer */
{
    int     c;

    init_input(1);
    set_intr();             /* we catch SIGINT */

    while ( (c = getchar()) != EOF)     /* main loop of program */
        out_char(c);        /* output each character */
    out_char(EOF);          /* output final buffer */

    block_write(&eofc, 1);  /* send EOF to printer */
    proc_upto_eof(0);       /* wait for printer to send EOF back */
}
```

**Program 17.10** The `send_file` function.

This function is just a `while` loop that reads from the standard input (`getchar`) and calls the function `out_char` to output each character to the printer. When the end of file is encountered on the standard input, an EOF is sent to the printer (indicating the end of job), and we wait for an end of file back from the printer (`proc_upto_eof`).

Recall from Figure 17.2 that the output from the PostScript interpreter on the serial port can be either printer status messages or output from the PostScript `print` operator. It is possible for what we think of as a "file to be printed" to generate no printed pages at all! This file can be a PostScript program that executes and sends its results back to the host computer. PostScript is not a language that many want to program in. Nevertheless, there are times when we want to send a PostScript program to the printer and have all its output sent back to the host, not printed on a page. One example is a PostScript program to fetch the page count every day, to track printer usage.

```
%!
statusdict begin pagecount end =
```

We want any output returned to the host by the PostScript interpreter, which is not a status message, to be sent as e-mail to the user. The file `mail.c`, shown in Program 17.11, handles this.

```
#include    "lprps.h"

static FILE *mailfp;
static char temp_file[L_tmpnam];
static void open_mailfp(void);

/* Called by proc_input_char() when it encounters characters
 * that are not message characters.  We have to send these
 * characters back to the user. */

void
mail_char(int c)
{
```

```
    static int  done_intro = 0;

    if (in_job && (done_intro || c != '\n')) {
        open_mailfp();
        if (done_intro == 0) {
            fputs("Your PostScript printer job "
                  "produced the following output:\n", mailfp);
            done_intro = 1;
        }
        putc(c, mailfp);
    }
}

/* Called by proc_msg() when an "Error" or "OffendingCommand" key
 * is returned by the PostScript interpreter.  Send the key and
 * val to the user. */

void
mail_line(const char *msg, const char *val)
{
    if (in_job) {
        open_mailfp();
        fprintf(mailfp, msg, val);
    }
}

/* Create and open a temporary mail file, if not already open.
 * Called by mail_char() and mail_line() above. */

static void
open_mailfp(void)
{
    if (mailfp == NULL) {
        if ( (mailfp = fopen(tmpnam(temp_file), "w")) == NULL)
            log_sys("open_mailfp: fopen error");
    }
}

/* Close the temporary mail file and send it to the user.
 * Registered to be called on exit() by atexit() in main(). */

void
close_mailfp(void)
{
    char    command[1024];

    if (mailfp != NULL) {
        if (fclose(mailfp) == EOF)
            log_sys("close_mailfp: fclose error");
        sprintf(command, MAILCMD, loginname, hostname, temp_file);
        system(command);
        unlink(temp_file);
    }
}
```

**Program 17.11** The `mail.c` file.

The function `mail_char` is called each time a character is returned by the printer to the host, if the character is not part of a status message. (Later in this section we look at the function `proc_input_char` that calls `mail_char`.) The variable `in_job` is set only while the function `send_file` is sending a file to the printer. It is not set at other times, such as when we're fetching the printer's status or the printer's page count. The function `mail_line` is called to write a line to the mail file.

The first time the function `open_mailfp` is called, it creates a temporary file, and opens it. The function `close_mailfp` is set by `main` as an exit handler, to be called whenever `exit` is called. If the temporary mail file was created, it is closed and mailed to the user.

If we send the one-line PostScript program

```
%!
statusdict begin pagecount end =
```

to fetch the printer's page count, the mail message returned to us is

```
Your PostScript printer job produced the following output:
11185
```

The file `output.c` (Program 17.12) contains the function `out_char` that was called by `send_file` to output each character to the printer.

```
#include    "lprps.h"

static char outbuf[OBSIZE];
static int  outcnt = OBSIZE;    /* #bytes remaining */
static char *outptr = outbuf;

static void out_buf(void);

/* Output a single character.
 * Called by main loop in send_file(). */

void
out_char(int c)
{
    if (c == EOF) {
        out_buf();      /* flag that we're all done */
        return;
    }

    if (outcnt <= 0)
        out_buf();      /* buffer is full, write it first */

    *outptr++ = c;      /* just store in buffer */
    outcnt--;
}

/* Output the buffer that out_char() has been storing into.
 * We have our own output function, so that we never block on a write
 * to the printer.  Each time we output our buffer to the printer,
 * we also see if the printer has something to send us.  If so,
 * we call proc_input_char() to process each character. */
```

```
static void
out_buf(void)
{
    char    *wptr, *rptr, ibuf[IBSIZE];
    int     wcnt, nread, nwritten;
    fd_set  rfds, wfds;

    FD_ZERO(&wfds);
    FD_ZERO(&rfds);
    set_nonblock();          /* don't want the write() to block */
    wptr = outbuf;           /* ptr to first char to output */
    wcnt = outptr - wptr;    /* #bytes to output */
    while (wcnt > 0) {
        FD_SET(psfd, &wfds);
        FD_SET(psfd, &rfds);
        if (intr_flag)
            handle_intr();
        while (select(psfd + 1, &rfds, &wfds, NULL, NULL) < 0) {
            if (errno == EINTR) {
                if (intr_flag)
                    handle_intr();      /* no return */
            } else
                log_sys("out_buf: select error");
        }
        if (FD_ISSET(psfd, &rfds)) {        /* printer is readable */
            if ( (nread = read(psfd, ibuf, IBSIZE)) < 0)
                log_sys("out_buf: read error");
            rptr = ibuf;
            while (--nread >= 0)
                proc_input_char(*rptr++);
        }
        if (FD_ISSET(psfd, &wfds)) {        /* printer is writeable */
            if ( (nwritten = write(psfd, wptr, wcnt)) < 0)
                log_sys("out_buf: write error");
            wcnt -= nwritten;
            wptr += nwritten;
        }
    }
    outptr = outbuf;     /* reset buffer pointer and count */
    outcnt = OBSIZE;
}
```

**Program 17.12** The output.c file.

When the argument to out_char is EOF, that's a signal that the end of the input has been reached, and the final output buffer should be sent to the printer.

The function out_char places each character in the output buffer, calling out_buf when the buffer is full. We have to be careful writing out_buf: in addition to sending output to the printer, the printer can be sending us data also. To avoid blocking on a write, we must set the descriptor nonblocking. (Recall the example, Program 12.1.)

We use the `select` function to multiplex the two I/O directions: input and output. We set the same descriptor in the read set and the write set. There is also a chance that the `select` can be interrupted by a caught signal (`SIGINT`), so we have to check for this on any error return.

If we receive asynchronous input from the printer, we call `proc_input_char` to process each character. This input could be either a status message from the printer or output to be mailed to the user.

When we `write` to the printer we have to handle the case of the `write` returning a count less than the requested amount. Again, recall the example in Program 12.1, where we saw that a terminal device can accept any amount of data on each `write`.

The file `input.c`, shown in Program 17.13, defines the functions that handle all the input from the printer. This can be either printer status messages or output for the user.

```
#include    "lprps.h"

static int  eof_count;
static int  ignore_input;
static enum parse_state {   /* state of parsing input from printer */
    NORMAL,
    HAD_ONE_PERCENT,
    HAD_TWO_PERCENT,
    IN_MESSAGE,
    HAD_RIGHT_BRACKET,
    HAD_RIGHT_BRACKET_AND_PERCENT
} parse_state;

/* Initialize our input machine. */

void
init_input(int job)
{
    in_job = job;       /* only true when send_file() calls us */
    parse_state = NORMAL;
    ignore_input = 0;
}

/* Read from the printer until we encounter an EOF.
 * Whether or not the input is processed depends on "ignore". */

void
proc_upto_eof(int ignore)
{
    int     ec;

    ignore_input = ignore;
    ec = eof_count;     /* proc_input_char() increments eof_count */
    while (ec == eof_count)
        proc_some_input();
}

/* Wait for some data then read it.
 * Call proc_input_char() for every character read. */
```

```
void
proc_some_input(void)
{
    char    ibuf[IBSIZE];
    char    *ptr;
    int     nread;
    fd_set  rfds;

    FD_ZERO(&rfds);
    FD_SET(psfd, &rfds);
    set_nonblock();
    if (intr_flag)
        handle_intr();
    if (alrm_flag)
        handle_alrm();
    while (select(psfd + 1, &rfds, NULL, NULL, NULL) < 0) {
        if (errno == EINTR) {
            if (alrm_flag)
                handle_alrm();      /* doesn't return */
            else if (intr_flag)
                handle_intr();      /* doesn't return */
        } else
            log_sys("proc_some_input: select error");
    }
    if ( (nread = read(psfd, ibuf, IBSIZE)) < 0)
        log_sys("proc_some_input: read error");
    else if (nread == 0)
        log_sys("proc_some_input: read returned 0");

    ptr = ibuf;
    while (--nread >= 0)
        proc_input_char(*ptr++);    /* process each character */
}

/* Called by proc_some_input() above after some input has been read.
 * Also called by out_buf() whenever asynchronous input appears. */

void
proc_input_char(int c)
{
    if (c == '\004') {
        eof_count++;    /* just count the EOFs */
        return;
    } else if (ignore_input)
        return;         /* ignore everything except EOFs */

    switch (parse_state) {      /* parse the input */
    case NORMAL:
        if (c == '%')
            parse_state = HAD_ONE_PERCENT;
        else
            mail_char(c);
        break;
```

```
        case HAD_ONE_PERCENT:
            if (c == '%')
                parse_state = HAD_TWO_PERCENT;
            else {
                mail_char('%'); mail_char(c);
                parse_state = NORMAL;
            }
            break;
        case HAD_TWO_PERCENT:
            if (c == '[') {
                msg_init();      /* message starting; init buffer */
                parse_state = IN_MESSAGE;
            } else {
                mail_char('%'); mail_char('%'); mail_char(c);
                parse_state = NORMAL;
            }
            break;
        case IN_MESSAGE:
            if (c == ']')
                parse_state = HAD_RIGHT_BRACKET;
            else
                msg_char(c);
            break;
        case HAD_RIGHT_BRACKET:
            if (c == '%')
                parse_state = HAD_RIGHT_BRACKET_AND_PERCENT;
            else {
                msg_char(']'); msg_char(c);
                parse_state = IN_MESSAGE;
            }
            break;
        case HAD_RIGHT_BRACKET_AND_PERCENT:
            if (c == '%') {
                parse_state = NORMAL;
                proc_msg();      /* we have a message; process it */
            } else {
                msg_char(']'); msg_char('%'); msg_char(c);
                parse_state = IN_MESSAGE;
            }
            break;
        default:
            abort();
        }
}
```

**Program 17.13** The `input.c` file—read and process input from the printer.

The function `proc_upto_eof` is called whenever we are waiting for an EOF from the printer.

The function `proc_some_input` reads from the serial port. Note that we call `select` to determine when the descriptor is readable. This is because `select` is normally interrupted by a caught signal—it is not automatically restarted. Since the `select` can be interrupted by either `SIGALRM` or `SIGINT`, we don't want it restarted. Recall the discussion of `select` normally being interrupted in Section 12.5. Also recall from Section 10.5 that we can set `SA_RESTART` to specify that I/O functions should be automatically restarted when a certain signal occurs, but there is not always a complementary flag that lets us specify that I/O functions should not be restarted. If we don't set `SA_RESTART` then we are at the mercy of the system default, which could be to restart interrupted I/O function calls automatically. When input does arrive from the printer we `read` it in a nonblocking mode, taking whatever the printer has ready for us. The function `proc_input_char` is called to process each character.

The dirty work of processing the messages that the printer can send us is handled by `proc_input_char`. We have to look at every character and remember what state we're in. The variable `parse_state` keeps track of the state. All the characters after the sequence `%%[` are stored in the message buffer by calling `msg_char`. When we encounter the ending `]%%` we call `proc_msg` to process the message. Any characters other than the beginning `%%[`, the ending `]%%`, and the message in between are assumed to be the user's output and are mailed back to the user (by calling `mail_char`).

We now look at the functions that process the message that was accumulated by the input functions above. Program 17.14 shows the file `message.c`.

The function `msg_init` is called after the sequence `%%[` has been seen, and it just initializes the buffer counter. `msg_char` is then called for every character of the message.

The function `proc_msg` breaks up the message into separate *key: val* pairs, and looks at each *key*. The ANSI C function `strtok` is called to break the message into tokens, each *key: val* token separated by a semicolon.

A message of the form

```
%%[ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

causes the function `printer_flushing` to be called. It flushes the terminal buffers, sends an EOF to the printer, and waits for an EOF back from the printer.

If a message of the form

```
%%[ PrinterError: reason ]%%
```

is received, `log_msg` is called to log the error. Other errors with a *key* of `Error` are mailed back to the user. These usually indicate an error in the PostScript program.

If a status message is returned, denoted with a *key* of `status`, it is probably because the function `get_status` sent the printer a status request (Control-T). We look at the *val* and set the variable `status` accordingly.

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

static char msgbuf[MBSIZE];
static int  msgcnt;
static void printer_flushing(void);

/* Called by proc_input_char() after it's seen the "%%[" that
 * starts a message. */

void
msg_init(void)
{
    msgcnt = 0;      /* count of chars in message buffer */
}

/* All characters received from the printer between the starting
 * %%[ and the terminating ]%% are placed into the message buffer
 * by proc_some_input().  This message will be examined by
 * proc_msg() below. */

void
msg_char(int c)
{
    if (c != '\0' && msgcnt < MBSIZE - 1)
        msgbuf[msgcnt++] = c;
}

/* This function is called by proc_input_char() only after the final
 * percent in a "%%[ <message> ]%%" has been seen.  It parses the
 * <message>, which consists of one or more "key: val" pairs.
 * If there are multiple pairs, "val" can end in a semicolon. */

void
proc_msg(void)
{
    char    *ptr, *key, *val;
    int     n;

    msgbuf[msgcnt] = 0;      /* null terminate message */
    for (ptr = strtok(msgbuf, ";"); ptr != NULL;
                                        ptr = strtok(NULL, ";")) {
        while (isspace(*ptr))
            ptr++;           /* skip leading spaces in key */
        key = ptr;

        if ( (ptr = strchr(ptr, ':')) == NULL)
            continue;    /* missing colon, something wrong, ignore */
        *ptr++ = '\0';   /* null terminate key (overwrite colon) */

        while (isspace(*ptr))
            ptr++;       /* skip leading spaces in val */
        val = ptr;
                         /* remove trailing spaces in val */
        ptr = strchr(val, '\0');
        while (ptr > val && isspace(ptr[-1]))
```

```
            --ptr;
        *ptr = '\0';
        if (strcmp(key, "Flushing") == 0) {
            printer_flushing();     /* never returns */
        } else if (strcmp(key, "PrinterError") == 0) {
            log_msg("proc_msg: printer error: %s", val);
        } else if (strcmp(key, "Error") == 0) {
            mail_line("Your PostScript printer job "
                      "produced the error '%s'.\n", val);
        } else if (strcmp(key, "status") == 0) {
            if (strcmp(val, "idle") == 0)
                status = IDLE;
            else if (strcmp(val, "busy") == 0)
                status = BUSY;
            else if (strcmp(val, "waiting") == 0)
                status = WAITING;
            else
                status = UNKNOWN;   /* "printing", "PrinterError",
                       "initializing", or "printing test page". */
        } else if (strcmp(key, "OffendingCommand") == 0) {
            mail_line("The offending command was '%s'.\n", val);
        } else if (strcmp(key, "pagecount") == 0) {
            if (sscanf(val, "%d", &n) == 1 && n >= 0) {
                if (start_page < 0)
                    start_page = n;
                else
                    end_page = n;
            }
        }
    }
}

/* Called only by proc_msg() when the "Flushing" message
 * is received from the printer.  We exit. */

static void
printer_flushing(void)
{
    clear_intr();           /* don't catch SIGINT */
    tty_flush();            /* empty tty input and output queues */
    block_write(&eofc, 1);  /* send an EOF to the printer */
    proc_upto_eof(1);       /* this call won't be recursive,
                               since we specify to ignore input */
    get_page(&end_page);
    do_acct();
    exit(EXIT_SUCCESS);
}
```

**Program 17.14** The `message.c` file, process messages returned from the printer.

A *key* of `OffendingCommand` usually appears with other *key: val* pairs, as in

```
%%[ Error: stackunderflow; OffendingCommand: pop ]%%
```

We add another line to the mail message that is sent back to the user.

Finally, a *key* of `pagecount` is generated by the PostScript program in the `get_page` function (Program 17.9). We call `sscanf` to convert *val* to binary, and set either the starting or ending page count variable. The `while` loop in the function `get_page` is waiting for this variable to become nonnegative.

## 17.5 Summary

This chapter has examined in detail a complete program—one that sends a PostScript program to a PostScript printer over an RS-232 serial connection. It has given us a chance to see lots of functions that we described in earlier chapters used in a real program: I/O multiplexing, nonblocking I/O, terminal I/O, and signals.

## Exercises

17.1 We said that the file to be printed by `lprps` is on its standard input and could be a pipe. How would you write the `psif` program (Figure 17.5) to handle this condition, since `psif` has to look at the first two bytes of the file?

17.2 Implement the `psif` filter, handling the case outlined in the previous exercise.

17.3 Read Section 12.5 of Adobe Systems [1988] about the handling of font requests in a PostScript program. Modify the `lprps` program in this chapter to handle font requests.

# 18

# A Modem Dialer

## 18.1 Introduction

Programs that deal with modems have always had a hard time coping with the wide variety of modems that are available. On most Unix systems there are two programs that handle modems. The first is a remote login program that lets us dial some other computer, log in, and use that system. In the System V world this program is called `cu`, while Berkeley systems calls it `tip`. Both programs do similar things, and both have knowledge of many different types of modems. The other program that uses a modem is `uucico`, part of the UUCP package. The problem is that knowledge that a modem is being used is often built into these programs, and if we want to write some other program that needs a modem, we have to perform many of the same tasks. Also, if we want to change these programs to use some form of communication instead of a modem (such as a network connection), major changes are often required.

In this chapter we develop a separate program that handles all the details of modem handling. This lets us isolate all these details into a single program, instead of having it spread through multiple programs. (This program was motivated by the connection server described in Presotto and Ritchie [1990].) To use this program we have to be able to invoke it and have it pass back a file descriptor, as we described in Section 15.3. We then use this program in developing a remote login program (similar to `cu` and `tip`).

## 18.2 History

The `cu`(1) command (which stands for "call Unix") appeared in Version 7. But it handled only one particular ACU (automatic call unit). Bill Shannon at Berkeley modified `cu`, and it appeared in 4.2BSD as the `tip`(1) program. The biggest change was the use of a text file (`/etc/remote`) to contain all the information for various systems (phone

number, preferred dialer, baud rate, parity, flow control, etc.). This version of `tip` supported about six different call units and modems, but to add support for some other type of modem required source code changes.

Along with `cu` and `tip`, the UUCP system also accessed modems and automatic call units. UUCP managed locks on different modems, so that multiple instances of UUCP could be running at the same time. The `tip` and `cu` programs had to honor the UUCP locking protocol, to avoid interfering with UUCP. On the BSD systems UUCP developed its own set of dialer functions. These functions were link edited into the UUCP executable, which meant the addition of a new modem type required source code changes.

SVR2 provided a `dial`(3) function that attempted to isolate the unique features of modem dialing into a single library function. It was used by `cu`, but not by UUCP. This function was in the standard C library, so it was available to any program.

The Honey DanBer UUCP system [Redman 1989] took the modem commands out of the C source files and put them into a `Dialers` file. This allowed the addition of new modem types without having to modify the source code. But the functions used by `cu` and UUCP to access the `Dialers` file were not generally available. This means that without redeveloping all the code to process the dialing information in the `Dialers` file, programs other than `cu` and UUCP couldn't use this file.

Throughout all these versions of `cu`, `tip`, and UUCP, locking was required to assure only a single program accessed a single device at a time. Since all these programs worked across many different systems, earlier versions of which provided no record locking, a rudimentary form of file locking was used. This could lead to lock files being left around after a program crashed, and ad hoc techniques were developed to handle this. (We can't use record locking on special device files, so record locking by itself isn't the final solution.)

## 18.3 Program Design

Let's detail the features that we want the modem dialer to have.

1. It must be possible to add new modem types without requiring source code changes.

   To obtain this feature, we'll use the Honey DanBer `Dialers` file. We'll put all the code that uses this file to dial the modem into a daemon server, so any program can access it using the client–server functions from Section 15.5.

2. Some form of locking must be used so that the abnormal termination of a program holding a lock automatically releases the lock. Ad hoc techniques, such as those still used by most versions of `cu` and UUCP, should finally be discarded, since better methods exist.

   We'll let the server daemon handle all the device locking. Since the client–server functions from Section 15.5 automatically notify the server when a client terminates, the daemon can release any locks that the process had.

3. New programs must be able to use all the features that we develop. A new program that deals with a modem should not have to reinvent the wheel. Dialing any type of modem should be as simple as a function call.

   For this feature, we'll let the central server daemon do all the dialing, passing back a file descriptor.

4. Client programs, such as `cu` and `tip`, shouldn't need special privileges. They should not be set-user-ID programs.

   We'll give the special privileges to the server daemon, allowing its clients to run without any special privileges.

Obviously we can't change the existing `cu`, `tip`, and UUCP programs, but we should make it easier for others to build on this work. Also, we should take the best features of the existing Unix dialing programs.

Figure 18.1 shows the arrangement of the client and server.

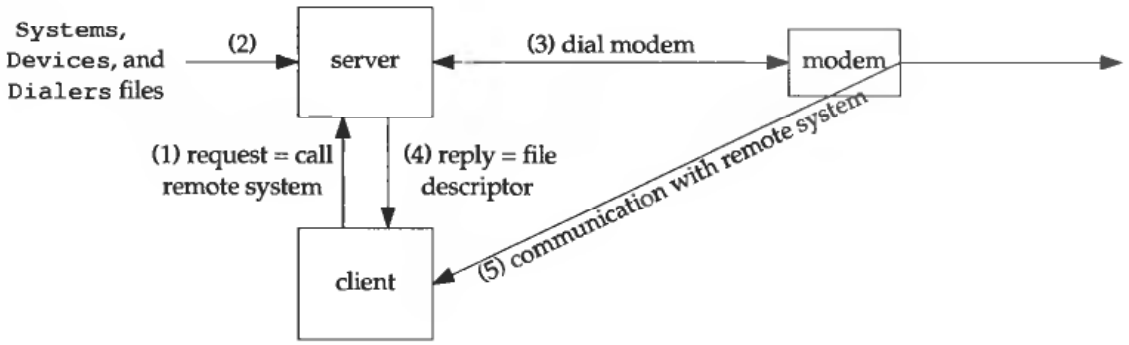


**Figure 18.1** Overview of client and server.

The steps involved in establishing communications with a remote system are as follows:

0. The server is started.
1. The client is started and opens a connection to the server, using the `cli_conn` function (Section 15.5). The client sends a request for the server to call the remote system.
2. The server reads the `Systems`, `Devices`, and `Dialers` files to determine how to call the remote system. (We describe these files in the next section.) If a modem is being used, the `Dialers` file contains all the modem-specific commands to dial the modem.
3. The server opens the modem device and dials the modem. This can take a while (typically around 15–30 seconds). The server handles all locking of this device, to avoid interfering with other users of this device.
4. If the dialing was successful, the server passes back the open file descriptor for the modem device to the client. Our functions from Section 15.3 send and receive the descriptor.

5. The client communicates directly with the remote system. The server is not involved in this communication—the client `reads` and `writes` the file descriptor returned in step 4.

The communication between the client and server (steps 1 and 4) is across a stream pipe. When the client is finished communicating with the remote system it closes this stream pipe (normally just by terminating). The server notices this close and releases the lock on the modem device.

## 18.4 Data Files

In this section we describe the three files used by the Honey DanBer UUCP system: `Systems`, `Devices`, and `Dialers`. There are many fields in these files that are used by the UUCP system. We don't describe these additional fields (or the UUCP system) in detail. Refer to Redman [1989] for additional details.

Figure 18.2 shows the six fields in the `Systems` file. We show the fields in a columnar format.

| *name* | *time* | *type* | *class* | *phone* | *login* |
|---|---|---|---|---|---|
| host1 | Any | ACU | 19200 | 5551234 | (not used) |
| host1 | Any | ACU | 9600 | 5552345 | (not used) |
| host1 | Any | ACU | 2400 | 5556789 | (not used) |
| modem | Any | modem | 19200 | - | (not used) |
| laser | Any | laser | 19200 | - | (not used) |

**Figure 18.2** The `Systems` file.

The *name* is the name of the remote system. We use this in commands of the form `cu host1`, for example. Note that we can have multiple entries for the same remote system. These entries are tried in order. The entries named `modem` and `laser` are for connecting directly to a modem and a laser printer. We don't need to dial a modem to connect to these devices, but we still need to open the appropriate terminal line, and handle the appropriate locks.

*time* specifies the time-of-day and days of the week to call this host. This is a UUCP field. The *type* field specifies which entry in the `Devices` file is to be used for this *name*. The *class* field is really the line speed to be used (baud rate). *phone* specifies the phone number for entries with a *type* of `ACU`. For other entries the `phone` field is just a hyphen. The final field, *login*, is the remainder of the line. It is a series of strings used by UUCP to log in to the remote system. We don't need this field.

The `Devices` file contains information on the modems and directly connected hosts. Figure 18.3 shows the five fields in this file. The *type* field matches an entry in the `Systems` file with an entry in the `Devices` file. The *class* field must also match the corresponding field in the `Systems` file. It normally specifies the line speed.

The actual name of a device is obtained by prefixing the *line* field with `/dev/`. In this example the actual devices are `/dev/cua0`, `/dev/ttya`, and `/dev/ttyb`. The next field, *line2*, is not used.

| *type* | *line* | *line2* | *class* | *dialer* |
|---|---|---|---|---|
| ACU | cua0 | - | 19200 | tbfast |
| ACU | cua0 | - | 9600 | tb9600 |
| ACU | cua0 | - | 2400 | tb2400 |
| ACU | cua0 | - | 1200 | tb1200 |
| modem | ttya | - | 19200 | direct |
| laser | ttyb | - | 19200 | direct |

**Figure 18.3** The `Devices` file.

The final field, *dialer*, matches the corresponding entry in the `Dialers` file. For the directly connected entries this field is `direct`.

Figure 18.4 shows the format of the `Dialers` file. This is the file that contains all the modem-specific dialing commands.

| *dialer* | *sub* | *handshake* |
|---|---|---|
| tb9600 | =W-, | "" \dA\pA\pA\pTQ0S2=255S12=255s50=6s58=2s68=255\r\c OK\r \EATDT\T\r\c CONNECT\s9600 \r\c "" |
| tbfast | =W-, | "" \dA\pA\pA\pTQ0S2=255S12=255s50=255s58=2s68=255s110=1s111=30\r\c OK\r \EATDT\T\r\c CONNECT\sFAST |

**Figure 18.4** The `Dialers` file.

We show only two entries for this file—we don't show the entries for `tbl200` and `tb2400` that were referenced in the `Devices` file. The *handshake* field is contained on a single line. We have broken it into two lines to fit on the page.

The *dialer* field is used to locate the matching entry from the `Devices` file. The *sub* field specifies substitutions to be performed for an equals sign and a minus sign that appear in a phone number. In the two entries in Figure 18.4 this field says to substitute a `W` for an equals sign, and a comma for a minus sign. This allows the phone numbers in the `Systems` file to contain an equals sign (meaning "wait for dialtone") and a minus sign (meaning "pause"). The translation of these two characters to whatever each particular modem requires is specified by the `Dialers` file.

The final field, *handshake*, contains the actual dialing instructions. It is a sequence of blank-separated strings called expect–send strings. We expect (i.e., `read` until we match) the first string and then send (i.e., `write`) the next string. Let's look at the `tbfast` entry as an example. This entry is for a Telebit Trailblazer modem in its PEP mode (packetized ensemble protocol).

1. The first expect string is empty, meaning "expect nothing." We always successfully match this empty string.
2. We send the next string. Special send sequences are specified with the backslash character. `\d` causes a delay for 2 seconds. We then send an `A`. We pause for one-half second (`\p`), send another `A`, pause, send another `A`, and pause again. We then send the remaining characters in the string, starting with `T`. These

commands all set parameters in the modem. The `\r` sends a carriage return and the final `\c` says not to write the normal newline at the end of the send string.

3. We `read` from the modem until we receive the string `OK\r`. (Again, the sequence `\r` means a carriage return.)
4. The next send string begins with `\E`. This enables echo checking: each time we send a character to the modem, we read back until the character is echoed. We then send the four characters `ATDT`. The next special character, `\T`, causes the phone number to be substituted. This is followed by a carriage return and the normal newline at the end of the send string is not sent.
5. The final expect string waits for `CONNECT FAST` to be returned by the modem. (The sequence `\s` means a single space.)

When this final expect string is received, the dialing is complete. (There are many more special sequences that can appear in the *handshake* string that we don't cover.)

Let's summarize the actions that we have to perform with these three files.

1. Using the name of the remote system, find the first entry in the `Systems` file with the same *name*.
2. Find the matching entry in the `Devices` file with a *type* and *class* that match the corresponding entries in the `Systems` file entry.
3. Find the entry in the `Dialers` file that matches the *dialer* field in the `Devices` file.
4. Dial the modem.

There are two reasons why this can fail: (1) the device corresponding to the *line* field in the `Devices` file is already is use by someone else or (2) the dialing is unsuccessful (e.g., the phone on the remote system is busy, or the remote system is down and is not answering the phone). The second case is often detected by a time out occurring when we're reading from the modem, trying to match an expect string (see Exercise 18.10). In either case, we want to go back to step 1 and search for the next entry for the same remote system. As we saw in Figure 18.2, a given host can have multiple entries, each with a different phone number (and each phone number could correspond to a different device).

There are other files in the Honey DanBer system that we don't use in the example in this chapter. The file `Dialcodes` specifies dialcode abbreviations for phone numbers in the `Systems` file. The file `Sysfiles` allows the specification of alternate copies of the three files `Systems`, `Devices`, and `Dialers`.

## 18.5 Server Design

We'll start with a description of the server. Two factors affect the design of the server:

1. Dialing can take a while (15–30 seconds), so the server has to `fork` a child process to do the actual dialing.

2. The daemon server (the parent) has to be the one process that manages all the locks.

Figure 18.5 shows the arrangement of the processes.

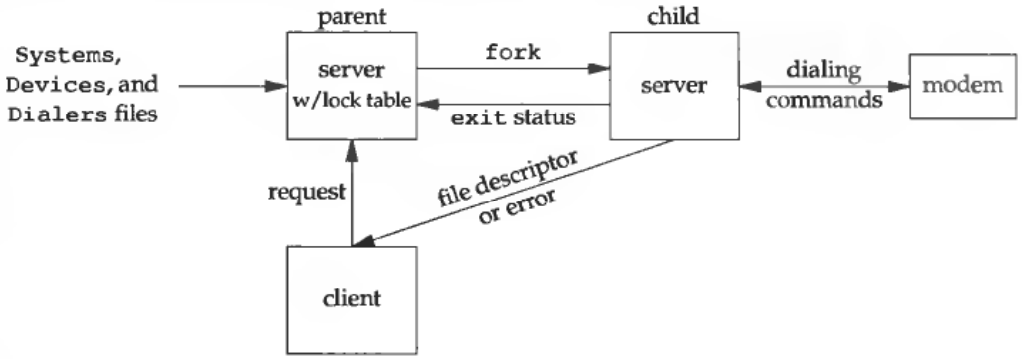


**Figure 18.5** Arrangement of processes in modem dialer.

The steps performed by the server are the following:

1. The parent receives the request from the client at the server's well-known name. As we described in Section 15.5, this creates a unique stream pipe between the client and server. This parent process has to handle multiple clients at the same time, like the open server in Section 15.6.
2. Based on the name of the remote system that the client wants to contact, the parent goes through the `Systems` file and `Devices` file, to find a match. The parent also keeps a lock table of which devices are currently in use, so it can skip those entries in the `Devices` file that are in use.
3. If a match is found, a child is `fork`ed to do the actual dialing. (The parent can handle other clients at this point.) If successful the child sends the file descriptor for the modem back to the client on the client-specific stream pipe (which got duplicated across the `fork`) and calls `exit(0)`. If an error occurs (phone line busy, no answer, etc.) the child calls `exit(1)`.
4. The parent is notified of the child termination by `SIGCHLD` and fetches its termination status (`waitpid`).

   If the child was successful there is nothing more for the parent to do. The lock must be held until the client is finished with the modem device. The client-specific stream pipe between the client and parent is left open. This way, when the client does terminate, the parent is notified, and the parent releases the lock.

   If the child was not successful, the parent picks up in the `Systems` file where it left off for this client and tries to find another match. If another entry is found for the remote system, the parent goes back to step 3 and `fork`s a new child to do the actual dialing. If no more entries exist for the remote system, the parent calls `send_err` (Program 15.4) and closes the client-specific stream pipe.

Having a unique connection to each client allows the child to send debug output back to the client, if desired. Often the client wants to see the progress of the actual dialing, if problems occur. Even though the dialing is being done by the child of an unrelated server, the unique connection allows the child to send output directly back to its client.

## 18.6 Server Source Code

We have 17 source files that constitute the server. Figure 18.6 details the files containing the various functions and specifies which are used by the parent and child. Figure 18.7 overviews the calling of the various functions.

| Source file | Parent/Child | | Functions |
|---|---|---|---|
| childdial.c | | C | child_dial |
| cliargs.c | P | | cli_args |
| client.c | P | | client_alloc, client_add, client_del, client_sigchld |
| ctlstr.c | | C | ctl_str |
| debug.c | | C | DEBUG, DEBUG_NONL |
| devfile.c | P | | dev_next, dev_rew, dev_find |
| dialfile.c | | C | dial_next, dial_rew, dial_find |
| expectstr.c | | C | expect_str, exp_read, sig_alrm |
| lock.c | P | | find_line, lock_set, lock_rel, is_locked |
| loop.c | P | | loop, cli_done, child_done |
| main.c | P | | main |
| request.c | P | | request |
| sendstr.c | | C | send_str |
| sigchld.c | P | | sig_chld |
| sysfile.c | P | | sys_next, sys_rew, sys_posn |
| ttydial.c | | C | tty_dial |
| ttyopen.c | | C | tty_open |

**Figure 18.6** Source files for server.

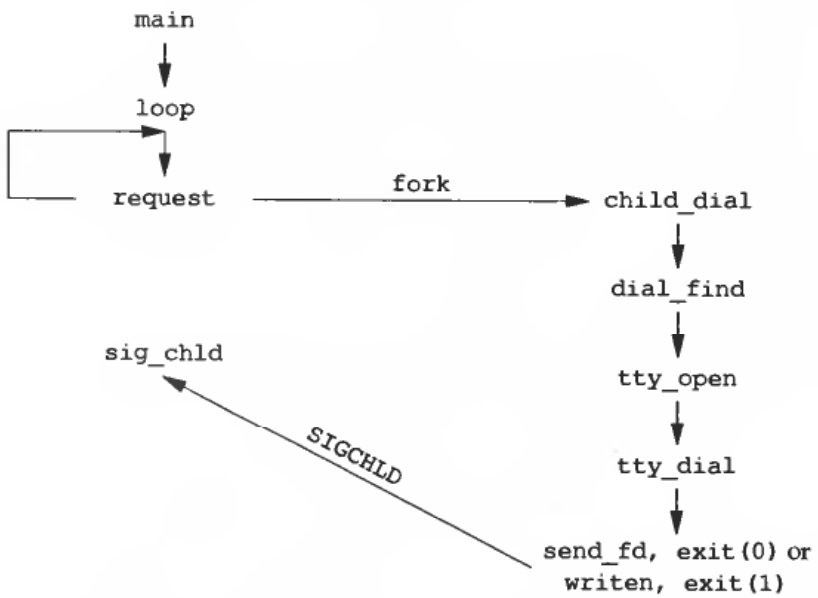


**Figure 18.7** Overview of function calling in server.

Program 18.1 shows the `calld.h` header, which is included by all the source files. It includes the standard system headers, defines some basic constants, and declares the global variables.

```
#include    <sys/types.h>
#include    <errno.h>
#include    <signal.h>
#include    "ourhdr.h"

#define CS_CALL "/home/stevens/calld"   /* well-known name */
#define CL_CALL "call"
#define MAXSYSNAME  256
#define MAXSPEEDSTR 256
#define NALLOC  10              /* #structs to alloc/realloc for */
            /* Client structs (client.c), Lock structs (lock.c) */
#define WHITE       " \t\n"             /* for separating tokens */
#define SYSTEMS     "./Systems"         /* my own copies for now */
#define DEVICES     "./Devices"
#define DIALERS     "./Dialers"

            /* declare global variables */
extern int       clifd;
extern int       debug;     /* nonzero if interactive (not daemon) */
extern int       Debug;     /* nonzero for dialing debug output */
extern char      errmsg[];  /* error message string to return to client */
extern char     *speed;     /* speed (actually "class") to use */
extern char     *sysname;   /* name of system to call */
extern uid_t     uid;       /* client's uid */
extern volatile sig_atomic_t        chld_flag;  /* when SIGCHLD occurs */
extern enum parity { NONE, EVEN, ODD } parity;  /* specified by client */

typedef struct {        /* one Client struct per connected client */
  int   fd;         /* fd, or -1 if available */
  pid_t pid;        /* child pid while dialing */
  uid_t uid;        /* client's user ID */
  int   childdone;  /* nonzero when SIGCHLD from dialing child recvd:
                       1 means exit(0), 2 means exit(1) */
  long  sysftell;   /* next line to read in Systems file */
  long  foundone;   /* true if we find a matching sysfile entry */
  int   Debug;              /* option from client */
  enum parity parity;       /* option from client */
  char  speed[MAXSPEEDSTR]; /* option from client */
  char  sysname[MAXSYSNAME];/* option from client */
} Client;

extern Client   *client;    /* ptr to malloc'ed array of Client structs */
extern int       client_size;/* # entries in client[] array */
                    /* (both manipulated by client_XXX() functions) */

typedef struct {        /* everything for one entry in Systems file */
  char  *name;      /* system name */
  char  *time;      /* (e.g., "Any") time to call (ignored) */
```

```
  char  *type;      /* (e.g., "ACU") or system name if direct connect */
  char  *class;     /* (e.g., "9600") speed */
  char  *phone;     /* phone number or "-" if direct connect */
  char  *login;     /* uucp login chat (ignored) */
} Systems;

typedef struct {        /* everything for one entry in Devices file */
  char  *type;      /* (e.g., "ACU") matched by type in Systems */
  char  *line;      /* (e.g., "cua0") without preceding "/dev/" */
  char  *line2;     /* (ignored) */
  char  *class;     /* matched by class in Systems */
  char  *dialer;    /* name of dialer in Dialers */
} Devices;

typedef struct {        /* everything for one entry in Dialers file */
  char  *dialer;    /* matched by dialer in Devices */
  char  *sub;       /* phone number substitution string (ignored) */
  char  *expsend;   /* expect/send chat */
} Dialers;

extern Systems  systems;    /* filled in by sys_next() */
extern Devices  devices;    /* filled in by dev_next() */
extern Dialers  dialers;    /* filled in by dial_next() */

                /* our function prototypes */
void     child_dial(Client *);                      /* childdial.c */

int      cli_args(int, char **);                    /* cliargs.c */

int      client_add(int, uid_t);                    /* client.c */
void     client_del(int);
void     client_sigchld(pid_t, int);

void     loop(void);                                /* loop.c */

char    *ctl_str(char);                             /* ctlstr.c */

int      dev_find(Devices *, const Systems *);      /* devfile.c */
int      dev_next(Devices *);
void     dev_rew(void);

int      dial_find(Dialers *, const Devices *);     /* dialfile.c */
int      dial_next(Dialers *);
void     dial_rew(void);

int      expect_str(int, char *);                   /* expectstr.c */

int      request(Client *);                         /* request.c */

int      send_str(int, char *, char *, int);        /* sendstr.c */

void     sig_chld(int);                             /* sigchld.c */

long     sys_next(Systems *);                       /* sysfile.c */
void     sys_posn(long);
```

```
void    sys_rew(void);

int     tty_open(char *, char *, enum parity, int);    /* ttyopen.c */

int     tty_dial(int, char *, char *, char *, char *); /* ttydial.c */

pid_t   is_locked(char *);                      /* lock.c */
void    lock_set(char *, pid_t);
void    lock_rel(pid_t);

void    DEBUG(char *, ...);                     /* debug.c */
void    DEBUG_NONL(char *, ...);
```

**Program 18.1** The `calld.h` header.

We define a `Client` structure that contains all the information for each client. This is an expansion of the similar structure in Program 15.26. In the time between `fork`ing a child to dial for a client and that child terminating, we can handle any number of other clients. This structure contains all the information that we need to try to find another `Systems` file entry for that client, and try dialing again.

We also define one structure for all the information for a single entry in the `Systems`, `Devices`, and `Dialers` files.

Program 18.2 shows the `main` function for the server. Since this program is normally run as a daemon server, we provide a `-d` command line option that lets us run the program interactively.

```
#include    "calld.h"
#include    <syslog.h>

            /* define global variables */
int         clifd;
int         debug;  /* daemon's command line flag */
int         Debug;  /* Debug controlled by client, not cmd line */
char        errmsg[MAXLINE];
char        *speed;
char        *sysname;
uid_t       uid;
Client      *client = NULL;
int         client_size;
Systems     systems;
Devices     devices;
Dialers     dialers;
volatile sig_atomic_t chld_flag;
enum parity parity = NONE;

int
main(int argc, char *argv[])
{
    int     c;

    log_open("calld", LOG_PID, LOG_USER);
```

```
    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = 1;
            break;

        case '?':
            log_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemon_init();

    loop();     /* never returns */
}
```

**Program 18.2** The `main` function.

When the `-d` option is set, all the calls to the `log_XXX` functions (Appendix B) are sent to standard error. Otherwise they are logged using `syslog`.

The function `loop` is the main loop of the server (Program 18.3). It multiplexes the various descriptors with the `select` function.

```
#include    "calld.h"
#include    <sys/time.h>
#include    <errno.h>

static void cli_done(int);
static void child_done(int);

static fd_set   allset; /* one bit per client conn, plus one for listenfd */
                        /* modified by loop() and cli_done() */

void
loop(void)
{
    int      i, n, maxfd, maxi, listenfd, nread;
    char     buf[MAXLINE];
    Client  *cliptr;
    uid_t    uid;
    fd_set   rset;

    if (signal_intr(SIGCHLD, sig_chld) == SIG_ERR)
        log_sys("signal error");

                /* obtain descriptor to listen for client requests on */
    if ( (listenfd = serv_listen(CS_CALL)) < 0)
        log_sys("serv_listen error");

    FD_ZERO(&allset);
    FD_SET(listenfd, &allset);
```

```
    maxfd = listenfd;
    maxi = -1;

    for ( ; ; ) {
        if (chld_flag)
            child_done(maxi);
        rset = allset;      /* rset gets modified each time around */
        if ( (n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0) {
            if (errno == EINTR) {
                    /* caught SIGCHLD, find entry with childdone set */
                child_done(maxi);
                continue;       /* issue the select again */
            } else
                log_sys("select error");
        }

        if (FD_ISSET(listenfd, &rset)) {
                    /* accept new client request */
            if ( (clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);

            i = client_add(clifd, uid);
            FD_SET(clifd, &allset);
            if (clifd > maxfd)
                maxfd = clifd;  /* max fd for select() */
            if (i > maxi)
                maxi = i;       /* max index in client[] array */
            log_msg("new connection: uid %d, fd %d", uid, clifd);
            continue;
        }

            /* Go through client[] array.
               Read any client data that has arrived. */

        for (cliptr = &client[0]; cliptr <= &client[maxi]; cliptr++) {
            if ( (clifd = cliptr->fd) < 0)
                continue;
            if (FD_ISSET(clifd, &rset)) {
                        /* read argument buffer from client */
                if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                    log_sys("read error on fd %d", clifd);

                else if (nread == 0) {
                    /* The client has terminated or closed the stream
                       pipe.  Now we can release its device lock. */

                    log_msg("closed: uid %d, fd %d",
                                        cliptr->uid, clifd);
                    lock_rel(cliptr->pid);
                    cli_done(clifd);
                    continue;
                }
```

```
                    /* Data has arrived from the client.  Process the
                       client's request. */

                    if (buf[nread-1] != 0) {
                        log_quit("request from uid %d not null terminated:"
                                 " %*.*s", uid, nread, nread, buf);
                        cli_done(clifd);
                        continue;
                    }
                    log_msg("starting: %s, from uid %d", buf, uid);

                            /* Parse the arguments, set options.  Since
                               we may need to try calling again for this
                               client, save options in client[] array. */
                    if (buf_args(buf, cli_args) < 0)
                        log_quit("command line error: %s", buf);
                    cliptr->Debug = Debug;
                    cliptr->parity = parity;
                    strcpy(cliptr->sysname, sysname);
                    strcpy(cliptr->speed, (speed == NULL) ? "" : speed);
                    cliptr->childdone = 0;
                    cliptr->sysftell = 0;
                    cliptr->foundone = 0;

                    if (request(cliptr) < 0) {
                            /* system not found, or unable to connect */
                        if (send_err(cliptr->fd, -1, errmsg) < 0)
                            log_sys("send_err error");
                        cli_done(clifd);
                        continue;
                    }
                    /* At this point request() has forked a child that is
                       trying to dial the remote system.  We'll find
                       out the child's status when it terminates. */
                }
            }
        }
    }

/* Go through the client[] array looking for clients whose dialing
   children have terminated.  This function is called by loop() when
   chld_flag (the flag set by the SIGCHLD handler) is nonzero. */

static void
child_done(int maxi)
{
    Client  *cliptr;

again:
    chld_flag = 0;  /* to check when done with loop for more SIGCHLDs */
```

```
        for (cliptr = &client[0]; cliptr <= &client[maxi]; cliptr++) {
            if ( (clifd = cliptr->fd) < 0)
                continue;
            if (cliptr->childdone) {
                log_msg("child done: pid %d, status %d",
                                cliptr->pid, cliptr->childdone-1);

                /* If the child was successful (exit(0)), just clear
                   the flag.  When the client terminates, we'll read
                   the EOF on the stream pipe above and release
                   the device lock. */

                if (cliptr->childdone == 1) {   /* child did exit(0) */
                    cliptr->childdone = 0;
                    continue;
                }

                /* Unsuccessful: child did exit(1).  Release the device
                   lock and try again from where we left off. */

                cliptr->childdone = 0;
                lock_rel(cliptr->pid);  /* unlock the device entry */
                if (request(cliptr) < 0) {
                            /* still unable, time to give up */
                    if (send_err(cliptr->fd, -1, errmsg) < 0)
                        log_sys("send_err error");
                    cli_done(clifd);
                    continue;
                }
                /* request() has forked another child for this client */
            }
        }
        if (chld_flag)  /* additional SIGCHLDs have been caught */
            goto again; /* need to check all childdone flags again */
    }

    /* Clean up when we're done with a client. */

    static void
    cli_done(int clifd)
    {
        client_del(clifd);      /* delete entry in client[] array */
        FD_CLR(clifd, &allset); /* turn off bit in select() set */
        close(clifd);           /* close our end of stream pipe */
    }
```

**Program 18.3** The `loop.c` file.

This function initializes the `client` array and establishes a signal handler for `SIGCHLD`. We call `signal_intr` instead of `signal` so that any slow system call is interrupted

when our signal handler returns. The `loop` function then calls `serv_listen` (Programs 15.19 and 15.22). The rest of the function is an infinite loop based on the `select` function, that tests for the following two conditions:

1. If a new client connection arrives, we call `serv_accept` (Programs 15.20 and 15.24). The function `client_add` creates an entry in the `client` array for the new client.
2. We then go through the `client` array, to see if (a) any client has terminated, or (b) any client requests have arrived.

   When a client terminates (whether voluntarily or not) its client-specific stream pipe to the server is closed, and we read an end of file from our end of the pipe. At this point we can release any device locks that the client owned and release the entry in the `client` array.

   When a request arrives from a client, we set things up and call `request`. (We showed the function `buf_args` in Program 15.17.) If the name of the remote system is valid and if an available device entry is located, `request` forks a child process and returns.

One external event that can happen at any time in this function is the termination of a child. If we're blocked in the `select` function, it returns an error of `EINTR`. Since the signal can also happen at other points in the `loop` function, we test the flag `chld_flag` each time through the loop before calling `select`. If the signal has occurred, we call the function `child_done` to process the termination.

This function goes through the `client` array, examining the `childdone` flag for each valid entry. If the child was successful, there's nothing else to do at this point. But if the child terminated with an `exit` status of 1, we call `request` to try to find another `Systems` file entry for this client.

Program 18.4 shows the function `cli_args` that is called by `buf_args` in the `loop` function, when a client request arrives. It processes the command-line arguments from the client. Note that this function sets global variables based on the command-line arguments, which `loop` then copies into the appropriate entry in the `client` array, since these options affect only a single client's request.

Program 18.5 shows the file `client.c`, which defines the functions that manipulate the `client` array. The only difference between Program 18.5 and Program 15.27 is that we now have to look up an entry based on the process ID (the function `client_sigchld`).

Program 18.6 is the file `lock.c`. These functions manage the `lock` array for the parent. As with the `client` functions, we call `realloc` to allocate space dynamically for the `lock` array, to avoid compile time limits.

```
#include    "calld.h"

/* This function is called by buf_args(), which is called by loop().
 * buf_args() has broken up the client's buffer into an argv[] style
 * array, which is now processed. */

int
cli_args(int argc, char **argv)
{
    int     c;

    if (argc < 2 || strcmp(argv[0], CL_CALL) != 0) {
        strcpy(errmsg, "usage: call <options> <hostname>");
        return(-1);
    }
    Debug = 0;      /* option defaults */
    parity = NONE;
    speed = NULL;
    opterr = 0;     /* don't want getopt() writing to stderr */
    optind = 1;     /* since we call getopt() multiple times */
    while ( (c = getopt(argc, argv, "des:o")) != EOF) {
        switch (c) {
        case 'd':
            Debug = 1;  /* client wants DEBUG() output */
            break;

        case 'e':       /* even parity */
            parity = EVEN;
            break;

        case 'o':       /* odd parity */
            parity = ODD;
            break;

        case 's':       /* speed */
            speed = optarg;
            break;

        case '?':
            sprintf(errmsg, "unrecognized option: -%c\n", optopt);
            return(-1);
        }
    }
    if (optind < argc)
        sysname = argv[optind];     /* name of host to call */
    else {
        sprintf(errmsg, "missing <hostname> to call\n");
        return(-1);
    }
    return(0);
}
```

**Program 18.4** The `cli_args` function.

```
#include    "calld.h"

static void
client_alloc(void)      /* alloc more entries in the client[] array */
{
    int     i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size + NALLOC) * sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

            /* have to initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;  /* fd of -1 means entry available */

    client_size += NALLOC;
}

/* Called by loop() when connection request from a new client arrives */

int
client_add(int fd, uid_t uid)
{
    int     i;

    if (client == NULL)     /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) {   /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i);  /* return index in client[] array */
        }
    }
            /* client array full, time to realloc for more */
    client_alloc();
    goto again;     /* and search again (will work this time) */
}

/* Called by loop() when we're done with a client */

void
client_del(int fd)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
```

```
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}

/* Find the client entry corresponding to a process ID.
 * This function is called by the sig_chld() signal
 * handler only after a child has terminated. */

void
client_sigchld(pid_t pid, int stat)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].pid == pid) {
            client[i].childdone = stat; /* child's exit() status +1 */
            return;
        }
    }
    log_quit("can't find client entry for pid %d", pid);
}
```

**Program 18.5** The `client.c` file.

```
#include    "calld.h"

typedef struct {
  char  *line;  /* points to malloc()ed area */
                /* we lock by line (device name) */
  pid_t pid;    /* but unlock by process ID */
                /* pid of 0 means available */
} Lock;
static Lock *lock = NULL;   /* the malloc'ed/realloc'ed array */
static int   lock_size;     /* #entries in lock[] */
static int   nlocks;        /* #entries currently used in lock[] */

/* Find the entry in lock[] for the specified device (line).
 * If we don't find it, create a new entry at the end of the
 * lock[] array for the new device.  This is how all the possible
 * devices get added to the lock[] array over time. */

static Lock *
find_line(char *line)
{
    int     i;
    Lock    *lptr;

    for (i = 0; i < nlocks; i++) {
        if (strcmp(line, lock[i].line) == 0)
            return(&lock[i]);   /* found entry for device */
    }
```

```
    /* Entry not found.  This device has never been locked before.
       Add a new entry to lock[] array. */

    if (nlocks >= lock_size) {  /* lock[] array is full */
        if (lock == NULL)       /* first time through */
            lock = malloc(NALLOC * sizeof(Lock));
        else
            lock = realloc(lock, (lock_size + NALLOC) * sizeof(Lock));
        if (lock == NULL)
            err_sys("can't alloc for lock array");

        lock_size += NALLOC;
    }

    lptr = &lock[nlocks++];
    if ( (lptr->line = malloc(strlen(line) + 1)) == NULL)
        log_sys("calloc error");
    strcpy(lptr->line, line);   /* copy caller's line name */
    lptr->pid  = 0;
    return(lptr);
}

void
lock_set(char *line, pid_t pid)
{
    Lock    *lptr;

    log_msg("locking %s for pid %d", line, pid);
    lptr = find_line(line);
    lptr->pid  = pid;
}

void
lock_rel(pid_t pid)
{
    Lock    *lptr;

    for (lptr = &lock[0]; lptr < &lock[nlocks]; lptr++) {
        if (lptr->pid == pid) {
            log_msg("unlocking %s for pid %d", lptr->line, pid);
            lptr->pid  = 0;
            return;
        }
    }
    log_msg("can't find lock for pid = %d", pid);
}

pid_t
is_locked(char *line)
{
    return( find_line(line)->pid ); /* nonzero pid means locked */
}
```

**Program 18.6** Functions for managing client device locks.

Each entry in the `lock` array is associated with a single *line* (the second field in the `Devices` file). Since these locking functions don't know all the different *line* values in this data file, new entries in the `lock` array are created whenever a new *line* is locked the first time. The function `find_line` handles this.

The next three source files handle the three data files: `Systems`, `Devices`, and `Dialers`. Each file has a `XXX_next` function that reads the next line of the file and breaks it up into fields. The ANSI C function `strtok` is called to break the lines into fields. Program 18.7 handles the `Systems` file.

```
#include    "calld.h"

static FILE *fpsys = NULL;
static int   syslineno;          /* for error messages */
static char  sysline[MAXLINE];
        /* can't be automatic; sys_next() returns pointers into here */

/* Read and break apart a line in the Systems file. */

long                            /* return >0 if OK, -1 on EOF */
sys_next(Systems *sysptr)       /* structure is filled in with pointers */
{
    if (fpsys == NULL) {
        if ( (fpsys = fopen(SYSTEMS, "r")) == NULL)
            log_sys("can't open %s", SYSTEMS);
        syslineno = 0;
    }

again:
    if (fgets(sysline, MAXLINE, fpsys) == NULL)
        return(-1);      /* EOF */
    syslineno++;

    if ( (sysptr->name = strtok(sysline, WHITE)) == NULL) {
        if (sysline[0] == '\n')
            goto again;      /* ignore empty line */
        log_quit("missing 'name' in Systems file, line %d", syslineno);
    }
    if (sysptr->name[0] == '#')
        goto again;          /* ignore comment line */

    if ( (sysptr->time = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'time' in Systems file, line %d", syslineno);

    if ( (sysptr->type = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'type' in Systems file, line %d", syslineno);

    if ( (sysptr->class = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'class' in Systems file, line %d", syslineno);

    if ( (sysptr->phone = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'phone' in Systems file, line %d", syslineno);

    if ( (sysptr->login = strtok(NULL, "\n")) == NULL)
        log_quit("missing 'login' in Systems file, line %d", syslineno);
```

```
    return(ftell(fpsys));   /* return the position in Systems file */
}

void
sys_rew(void)
{
    if (fpsys != NULL)
        rewind(fpsys);
    syslineno = 0;
}

void
sys_posn(long posn)      /* position Systems file */
{
    if (posn == 0)
        sys_rew();
    else if (fseek(fpsys, posn, SEEK_SET) != 0)
        log_sys("fseek error");
}
```

**Program 18.7** Functions to read `Systems` file.

The function `sys_next` is called by `request` to read the next entry in the file.

We have to remember our position in this file for each client (the `sysftell` member of the `Client` structure). This is so that if a child fails to dial the remote system, we can pick up where we left off in the `Systems` file (for that client), to try to find another entry for the remote system. The position is obtained by calling the standard I/O function `ftell` and reset using `fseek`.

Program 18.8 contains the functions for reading the `Devices` file.

```
#include    "calld.h"

static FILE *fpdev = NULL;
static int  devlineno;          /* for error messages */
static char devline[MAXLINE];
        /* can't be automatic; dev_next() returns pointers into here */

/* Read and break apart a line in the Devices file. */

int
dev_next(Devices *devptr)       /* pointers in structure are filled in */
{
    if (fpdev == NULL) {
        if ( (fpdev = fopen(DEVICES, "r")) == NULL)
            log_sys("can't open %s", DEVICES);
        devlineno = 0;
    }

again:
    if (fgets(devline, MAXLINE, fpdev) == NULL)
        return(-1);     /* EOF */
    devlineno++;
```

```
    if ( (devptr->type = strtok(devline, WHITE)) == NULL) {
        if (devline[0] == '\n')
            goto again;     /* ignore empty line */
        log_quit("missing 'type' in Devices file, line %d", devlineno);
    }
    if (devptr->type[0] == '#')
        goto again;         /* ignore comment line */

    if ( (devptr->line = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'line' in Devices file, line %d", devlineno);

    if ( (devptr->line2 = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'line2' in Devices file, line %d", devlineno);

    if ( (devptr->class = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'class' in Devices file, line %d", devlineno);

    if ( (devptr->dialer = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'dialer' in Devices file, line %d", devlineno);

    return(0);
}

void
dev_rew(void)
{
    if (fpdev != NULL)
        rewind(fpdev);
    devlineno = 0;
}

/* Find a match of type and class */

int
dev_find(Devices *devptr, const Systems *sysptr)
{
    dev_rew();
    while (dev_next(devptr) >= 0) {
        if (strcmp(sysptr->type, devptr->type) == 0 &&
            strcmp(sysptr->class, devptr->class) == 0)
                return(0);      /* found a device match */
    }
    sprintf(errmsg, "device '%s'/'%s' not found\n",
                            sysptr->type, sysptr->class);
    return(-1);
}
```

**Program 18.8** Functions for reading `Devices` file.

We'll see that the `request` function calls `dev_find` to locate an entry with *type* and *class* fields that match an entry in the `Systems` file.

Program 18.9 contains the functions for reading the `Dialers` file.

```
#include    "calld.h"

static FILE *fpdial = NULL;
static int  diallineno;          /* for error messages */
static char dialline[MAXLINE];
        /* can't be automatic; dial_next() returns pointers into here */

/* Read and break apart a line in the Dialers file. */

int
dial_next(Dialers *dialptr) /* pointers in structure are filled in */
{
    if (fpdial == NULL) {
        if ( (fpdial = fopen(DIALERS, "r")) == NULL)
            log_sys("can't open %s", DIALERS);
        diallineno = 0;
    }

again:
    if (fgets(dialline, MAXLINE, fpdial) == NULL)
        return(-1);      /* EOF */
    diallineno++;

    if ( (dialptr->dialer = strtok(dialline, WHITE)) == NULL) {
        if (dialline[0] == '\n')
            goto again;      /* ignore empty line */
        log_quit("missing 'dialer' in Dialers file, line %d", diallineno);
    }
    if (dialptr->dialer[0] == '#')
        goto again;          /* ignore comment line */

    if ( (dialptr->sub = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'sub' in Dialers file, line %d", diallineno);

    if ( (dialptr->expsend = strtok(NULL, "\n")) == NULL)
        log_quit("missing 'expsend' in Dialers file, line %d", diallineno);

    return(0);
}

void
dial_rew(void)
{
    if (fpdial != NULL)
        rewind(fpdial);
    diallineno = 0;
}

/* Find a dialer match */

int
dial_find(Dialers *dialptr, const Devices *devptr)
{
```

```
    dial_rew();
    while (dial_next(dialptr) >= 0) {
        if (strcmp(dialptr->dialer, devptr->dialer) == 0)
            return(0);      /* found a dialer match */
    }
    sprintf(errmsg, "dialer '%s' not found\n", dialptr->dialer);
    return(-1);
}
```

**Program 18.9** Functions for reading `Dialers` file.

We'll see that the `child_dial` function calls `dial_find` to find an entry with a *dialer* field that matches a particular device.

Notice from Figure 18.6 that the `Systems` and `Devices` files are handled by the parent, while the `Dialers` file is handled by the child. This was one of the design goals—the parent finds a matching device that is not locked and `fork`s a child to do the actual dialing.

We look at the `request` function in Program 18.10. It was called by the `loop` function to try to locate an unlocked device for the specified remote host. To do this it goes through the `Systems` file, then the `Devices` file. If a match is found, a child is `fork`ed. We allow the client to specify a speed, in addition to the name of the remote system. For example, with the `Systems` file in Figure 18.2, the client's request can look like

```
call -s 9600 host1
```

which causes us to ignore the other two entries for `host1` in Figure 18.2.

Notice that we can't record the device lock using `lock_set` until we know the process ID of the child (i.e., after the `fork`), but we have to test whether the device is locked before the `fork`. Since we don't want the child starting until we have set the lock, we use the `TELL_WAIT` functions (Program 10.17) to synchronize the parent and child. Also note that although the test `is_locked` and the actual setting of the lock by `set_lock` are two separate operations (i.e., not a single atomic operation) we do not have a race condition. This is because `request` is called only by the single parent server daemon—it is not called by multiple processes.

If `request` returns 0, a child was `fork`ed to start the dial, otherwise it returns –1 to indicate that either the name of the remote system wasn't valid or all the possible devices for the remote system were locked.

```
#include    "calld.h"

int                         /* return 0 if OK, -1 on error */
request(Client *cliptr)
{
    pid_t   pid;

    errmsg[0] = 0;
        /* position where this client left off last (or rewind) */
    sys_posn(cliptr->sysftell);
```

```
        while ( (cliptr->sysftell = sys_next(&systems)) >= 0) {
            if (strcmp(cliptr->sysname, systems.name) == 0) {
                        /* system match */
                    /* if client specified a speed, it must match too */
                if (cliptr->speed[0] != 0 &&
                    strcmp(cliptr->speed, systems.class) != 0)
                        continue;    /* speeds don't match */

                DEBUG("trying sys: %s, %s, %s, %s", systems.name,
                        systems.type, systems.class, systems.phone);
                cliptr->foundone++;

                if (dev_find(&devices, &systems) < 0)
                    break;
                DEBUG("trying dev: %s, %s, %s, %s", devices.type,
                        devices.line, devices.class, devices.dialer);
                if ( (pid = is_locked(devices.line)) != 0) {
                    sprintf(errmsg, "device '%s' already locked by pid %d\n",
                                                        devices.line, pid);
                    continue;    /* look for another entry in Systems file */
                }

                    /* We've found a device that's not locked.
                       fork() a child to to the actual dialing. */
                TELL_WAIT();
                if ( (cliptr->pid = fork()) < 0)
                    log_sys("fork error");
                else if (cliptr->pid == 0) {    /* child */
                    WAIT_PARENT();      /* let parent set lock */
                    child_dial(cliptr); /* never returns */
                }
                /* parent */
                lock_set(devices.line, cliptr->pid);
                        /* let child resume, now that lock is set */
                TELL_CHILD(cliptr->pid);
                return(0);  /* we've started a child */
            }
        }
        /* reached EOF on Systems file */
        if (cliptr->foundone == 0)
            sprintf(errmsg, "system '%s' not found\n", cliptr->sysname);
        else if (errmsg[0] == 0)
            sprintf(errmsg, "unable to connect to system '%s'\n",
                                                    cliptr->sysname);
        return(-1);     /* also, cliptr->sysftell is -1 */
}
```

**Program 18.10** The request function.

The last of the parent-specific functions is `sig_chld`, the signal handler for the `SIGCHLD` signal. This is shown in Program 18.11.

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

/* SIGCHLD handler, invoked when a child terminates. */

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

    errno_save = errno;     /* log_msg() might change errno */
    chld_flag = 1;
    if ( (pid = waitpid(-1, &stat, 0)) <= 0)
        log_sys("waitpid error");

    if (WIFEXITED(stat) != 0)
                    /* set client's childdone status for loop() */
        client_sigchld(pid, WEXITSTATUS(stat)+1);
    else
        log_msg("child %d terminated abnormally: %04x", pid, stat);

    errno = errno_save;
    return;     /* probably interrupts accept() in serv_accept() */
}
```

**Program 18.11** The `sig_chld` signal handler.

When a child terminates we must record its termination status and process ID in the appropriate entry in the `client` array. We call the function `client_sigchld` (Program 18.5) to do this.

Note that we are violating one of our earlier rules from Chapter 10—a signal handler should only set a global variable and nothing else. Here we call `waitpid` and the function `client_sigchld` (Program 18.5). This latter function is signal safe. All it does is record information in an entry in the `client` array—it doesn't create or delete entries (which would be nonreentrant) and it doesn't call any system functions.

`waitpid` is defined by POSIX.1 to be signal safe (Figure 10.3). If we didn't call `waitpid` from the signal handler, the parent would have to call it when the flag `chld_flag` was nonzero. But since numerous children can terminate before the main loop gets a chance to look at `chld_flag`, we would either need to increment `chld_flag` each time a child terminated (so the main loop would know how many times to call `waitpid`) or call `waitpid` in a loop, with the `WNOHANG` flag (Figure 8.3). The simplest solution is to call `waitpid` from the signal handler, and record the information in the `client` array.

We now proceed to the functions that are called by the child as part of its attempt to dial the remote system. Everything starts for the child after the `fork` when `request` calls `child_dial` (Program 18.12).

```
#include    "calld.h"

/* The child does the actual dialing and sends the fd back to
 * the client.  This function can't return to caller, must exit.
 * If successful, exit(0), else exit(1).
 * The child uses the following global variables, which are just
 * in the copy of the data space from the parent:
 *      cliptr->fd (to send DEBUG() output and fd back to client),
 *      cliptr->Debug (for all DEBUG() output), childptr->parity,
 *      systems, devices, dialers. */

void
child_dial(Client *cliptr)
{
    int     fd, n;

    Debug = cliptr->Debug;
    DEBUG("child, pid %d", getpid());

    if (strcmp(devices.dialer, "direct") == 0) { /* direct tty line */
        fd = tty_open(systems.class, devices.line, cliptr->parity, 0);
        if (fd < 0)
            goto die;
    } else {                    /* else assume dialing is needed */
        if (dial_find(&dialers, &devices) < 0)
            goto die;
        fd = tty_open(systems.class, devices.line, cliptr->parity, 1);
        if (fd < 0)
            goto die;
        if (tty_dial(fd, systems.phone, dialers.dialer,
                                dialers.sub, dialers.expsend) < 0)
            goto die;
    }

    DEBUG("done");
            /* send the open descriptor to client */
    if (send_fd(cliptr->fd, fd) < 0)
        log_sys("send_fd error");
    exit(0);    /* parent will see this */

die:
    /* The child can't call send_err() as that would send the final
       2-byte protocol to the client.  We just send our error message
       back to the client.  If the parent finally gives up, it'll
       call send_err(). */

    n = strlen(errmsg);
    if (writen(cliptr->fd, errmsg, n) != n) /* send error to client */
        log_sys("send_err error");
    exit(1);    /* parent will see this, release lock, and try again */
}
```

**Program 18.12** The `child_dial` function.

If the device being used is directly connected, just the function `tty_open` is called to open the terminal device and set all the appropriate terminal parameters. But if the device is a modem, three functions are called: `dial_find` (to locate the appropriate entry in the `Dialers` file), `tty_open`, and `tty_dial` (to do the actual dialing).

If `child_dial` is successful, it returns the file descriptor to the client by calling `send_fd` (Programs 15.5 and 15.9) and calls `exit(0)`. Otherwise it sends an error message back to the client across the stream pipe and calls `exit(1)`. The client-specific stream pipe is duplicated across the `fork`, so the child can send either the descriptor or error message directly back to the client.

```
#include    "calld.h"
#include    <stdarg.h>

/* Note that all debug output goes back to the client. */

void
DEBUG(char *fmt, ...)       /* debug output, newline at end */
{
    va_list args;
    char    line[MAXLINE];
    int     n;

    if (Debug == 0)
        return;
    va_start(args, fmt);
    vsprintf(line, fmt, args);
    strcat(line, "\n");
    va_end(args);

    n = strlen(line);
    if (writen(clifd, line, n) != n)
        log_sys("writen error");
}

void
DEBUG_NONL(char *fmt, ...)  /* debug output, NO newline at end */
{
    va_list args;
    char    line[MAXLINE];
    int     n;

    if (Debug == 0)
        return;
    va_start(args, fmt);
    vsprintf(line, fmt, args);
    va_end(args);

    n = strlen(line);
    if (writen(clifd, line, n) != n)
        log_sys("writen error");
}
```

**Program 18.13** Debugging functions.

The client can send a -d option in its command to the server, and this sets the client-specific variable Debug. This flag is used in Program 18.13 by the two functions DEBUG and DEBUG_NONL to send debugging information back to the client. This information is useful when dialing problems are encountered for a particular system. These two functions are called predominantly by the child, although the parent also called them from the request function (Program 18.10).

Program 18.14 shows the tty_open function. It is called for both modem devices and direct connect devices, to open the terminal and set its modes. The *class* field of the Systems and Devices file specified the line speed, and the client can specify the parity.

```
#include    "calld.h"
#include    <fcntl.h>
#include    <termios.h>

/* Open the terminal line */

int
tty_open(char *class, char *line, enum parity parity, int modem)
{
    int             fd, baud;
    char            devname[100];
    struct termios  term;

            /* first open the device */
    strcpy(devname, "/dev/");
    strcat(devname, line);
    if ( (fd = open(devname, O_RDWR | O_NONBLOCK)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                                devname, strerror(errno));
        return(-1);
    }
    if (isatty(fd) == 0) {
        sprintf(errmsg, "%s is not a tty\n", devname);
        return(-1);
    }

            /* fetch then set modem's terminal status */
    if (tcgetattr(fd, &term) < 0)
        log_sys("tcgetattr error");

    if (parity == NONE)
        term.c_cflag = CS8;
    else if (parity == EVEN)
        term.c_cflag = CS7 | PARENB;
    else if (parity == ODD)
        term.c_cflag = CS7 | PARENB | PARODD;
    else
        log_quit("unknown parity");
    term.c_cflag |= CREAD |         /* enable receiver */
                    HUPCL;          /* lower modem lines on last close */
                                    /* 1 stop bit (since CSTOPB off) */
```

```
    if (modem == 0)
        term.c_cflag |= CLOCAL;      /* ignore modem status lines */

    term.c_oflag  = 0;               /* turn off all output processing */
    term.c_iflag  = IXON | IXOFF |   /* Xon/Xoff flow control (default) */
                    IGNBRK |         /* ignore breaks */
                    ISTRIP |         /* strip input to 7 bits */
                    IGNPAR;          /* ignore input parity errors */
    term.c_lflag  = 0;       /* everything off in local flag:
                                disables canonical mode, disables
                                signal generation, disables echo */
    term.c_cc[VMIN]  = 1;    /* 1 byte at a time, no timer */
    term.c_cc[VTIME] = 0;    /* (See Figure 18.10) */

    if      (strcmp(class, "38400") == 0)   baud = B38400;
    else if (strcmp(class, "19200") == 0)   baud = B19200;
    else if (strcmp(class, "9600") == 0)    baud = B9600;
    else if (strcmp(class, "4800") == 0)    baud = B4800;
    else if (strcmp(class, "2400") == 0)    baud = B2400;
    else if (strcmp(class, "1800") == 0)    baud = B1800;
    else if (strcmp(class, "1200") == 0)    baud = B1200;
    else if (strcmp(class, "600") == 0)     baud = B600;
    else if (strcmp(class, "300") == 0)     baud = B300;
    else if (strcmp(class, "200") == 0)     baud = B200;
    else if (strcmp(class, "150") == 0)     baud = B150;
    else if (strcmp(class, "134") == 0)     baud = B134;
    else if (strcmp(class, "110") == 0)     baud = B110;
    else if (strcmp(class, "75") == 0)      baud = B75;
    else if (strcmp(class, "50") == 0)      baud = B50;
    else {
        sprintf(errmsg, "invalid baud rate: %s\n", class);
        return(-1);
    }
    cfsetispeed(&term, baud);
    cfsetospeed(&term, baud);

    if (tcsetattr(fd, TCSANOW, &term) < 0)  /* set attributes */
        log_sys("tcsetattr error");

    DEBUG("tty open");
    clr_fl(fd, O_NONBLOCK);      /* turn off nonblocking */
    return(fd);
}
```

**Program 18.14** The `tty_open` function.

We open the terminal device with the nonblocking flag, as sometimes the open of a terminal connected to a modem doesn't return until the modem's carrier is present. Since we are dialing out and not dialing in, we don't want to wait. At the end of the function we call the `clr_fl` function to clear the nonblocking mode. The only difference between a modem and a direct connect line in the `tty_open` function is for a direct connect line we set the `CLOCAL` bit.

The details of dialing a modem takes place in the `tty_dial` function (Program 18.15). This function is only called for modem lines, not for direct connect lines.

```
#include    "calld.h"

int
tty_dial(int fd, char *phone, char *dialer, char *sub, char *expsend)
{
    char    *ptr;

    ptr = strtok(expsend, WHITE);    /* first expect string */
    for ( ; ; ) {
        DEBUG_NONL("expect = %s\nread: ", ptr);
        if (expect_str(fd, ptr) < 0)
            return(-1);

        if ( (ptr = strtok(NULL, WHITE)) == NULL)
            return(0);      /* at the end of the expect/send */
        DEBUG_NONL("send = %s\nwrite: ", ptr);
        if (send_str(fd, ptr, phone, 0) < 0)
            return(-1);

        if ( (ptr = strtok(NULL, WHITE)) == NULL)
            return(0);      /* at the end of the expect/send */
    }
}
```

**Program 18.15** The `tty_dial` function.

The function just calls one function to handle the expect string and another to handle the send string. We are done when there are no more send or expect strings. (Note that we do not handle the *sub* string from Figure 18.4.)

Program 18.16 shows the function `send_str` that outputs the send strings. To keep the size of this example manageable, we have not implemented every special escape sequence—we have implemented enough to use the program with the `Dialers` files shown in Figure 18.4.

```
#include    "calld.h"

int
send_str(int fd, char *ptr, char *phone, int echocheck)
{
    char    c, tempc;

        /* go though send string, converting escape sequences on the fly */
    while ( (c = *ptr++) != 0) {
        if (c == '\\') {
            if (*ptr == 0) {
                sprintf(errmsg, "backslash at end of send string\n");
                return(-1);
            }
```

```
        c = *ptr++;     /* char following backslash */

        switch (c) {
        case 'c':       /* no CR, if at end of string */
            if (*ptr == 0)
                goto returnok;
            continue;   /* ignore if not at end of string */

        case 'd':       /* 2 second delay */
            DEBUG_NONL("<delay>");
            sleep(2);
            continue;

        case 'p':       /* 0.25 second pause */
            DEBUG_NONL("<pause>");
            sleep_us(250000);   /* Exercise 12.6 */
            continue;

        case 'e':
            DEBUG_NONL("<echo check off>");
            echocheck = 0;
            continue;

        case 'E':
            DEBUG_NONL("<echo check on>");
            echocheck = 1;
            continue;

        case 'T':       /* output phone number */
            send_str(fd, phone, phone, echocheck);  /* recursive */
            continue;

        case 'r':
            c = '\r';
            break;

        case 's':
            c = ' ';
            break;


            /* room for lots more case statements ... */


        default:
            sprintf("errmsg, unknown send escape char: \\%s\n",
                                                    ctl_str(c));
            return(-1);
        }
    }

    DEBUG_NONL("%s", ctl_str(c));
    if (write(fd, &c, 1) != 1)
        log_sys("write error");
```

```
        if (echocheck) {        /* wait for char to be echoed */
            do {
                if (read(fd, &tempc, 1) != 1)
                    log_sys("read error");
                DEBUG_NONL("{%s}", ctl_str(tempc));
            } while (tempc != c);
        }
    }
    c = '\r';   /* if no \c at end of string, CR written at end */
    DEBUG_NONL("%s", ctl_str(c));
    if (write(fd, &c, 1) != 1)
        log_sys("write error");
returnok:
    DEBUG("");
    return(0);
}
```

**Program 18.16** The send_str function.

send_str calls the function ctl_str to convert ASCII control characters into a printable version. Program 18.17 shows the ctl_str function.

```
#include    "calld.h"

/* Make a printable string of the character "c", which may be a
 * control character.  Works only with ASCII. */

char *
ctl_str(char c)
{
    static char tempstr[6];     /* biggest is "\177" + null */

    c &= 255;
    if (c == 0)
        return("\\0");          /* really shouldn't see a null */
    else if (c < 040)
        sprintf(tempstr, "^%c", c + 'A' - 1);
    else if (c == 0177)
        return("DEL");
    else if (c > 0177)
        sprintf(tempstr, "\\%03o", c);
    else
        sprintf(tempstr, "%c", c);
    return(tempstr);
}
```

**Program 18.17** The ctl_str function.

The hardest part of dialing the modem is recognizing the expect strings. Program 18.18 shows the function expect_str that does this. (As with the send strings, we have implemented only a subset of all the possible features provided by the Dialers file.)

```
#include    "calld.h"

#define EXPALRM     45          /* alarm time to read expect string */

static int      expalarm = EXPALRM;
static void     sig_alrm(int);
static volatile sig_atomic_t    caught_alrm;

static size_t   exp_read(int, char *);

int                     /* return 0 if got it, -1 if not */
expect_str(int fd, char *ptr)
{
    char    expstr[MAXLINE], inbuf[MAXLINE];
    char    c, *src, *dst, *inptr, *cmpptr;
    int     i, matchlen;

    if (strcmp(ptr, "\"\"") == 0)
        goto returnok;      /* special case of "" (expect nothing) */

            /* copy expect string, converting escape sequences */
    for (src = ptr, dst = expstr; (c = *src++) != 0; ) {
        if (c == '\\') {
            if (*src == 0) {
                sprintf(errmsg, "invalid expect string: %s\n", ptr);
                return(-1);
            }
            c = *src++;     /* char following backslash */
            switch (c) {
            case 'r':   c = '\r'; break;
            case 's':   c = ' '; break;
                /* room for lots more case statements ... */
            default:
                sprintf(errmsg, "unknown expect escape char: \\%s\n",
                                                        ctl_str(c));
                return(-1);
            }
        }
        *dst++ = c;
    }
    *dst = 0;
    matchlen = strlen(expstr);

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        log_quit("signal error");
    caught_alrm = 0;
    alarm(expalarm);

    do {
        if (exp_read(fd, &c) < 0)
            return(-1);
    } while (c != expstr[0]);   /* skip until first chars equal */
```

```
    cmpptr = inptr = inbuf;
    *inptr = c;

    for (i = 1; i < matchlen; i++) {    /* read matchlen chars */
        inptr++;
        if (exp_read(fd, inptr) < 0)
            return(-1);
    }

    for ( ; ; ) {       /* keep reading until we have a match */
        if (strncmp(cmpptr, expstr, matchlen) == 0)
            break;      /* have a match */
        inptr++;
        if (exp_read(fd, inptr) < 0)
            return(-1);
        cmpptr++;
    }
returnok:
    alarm(0);
    DEBUG("\nexpect: got it");
    return(0);
}

size_t          /* read one byte, handle timeout errors & DEBUG */
exp_read(int fd, char *buf)
{
    if (caught_alrm) {  /* test flag before blocking in read */
        DEBUG("\nread timeout");
        return(-1);
    }
    if (read(fd, buf, 1) == 1) {
        DEBUG_NONL("%s", ctl_str(*buf));
        return(1);
    }
    if (errno == EINTR && caught_alrm) {
        DEBUG("\nread timeout");
        return(-1);
    }
    log_sys("read error");
}

static void
sig_alrm(int signo)
{
    caught_alrm = 1;
    return;
}
```

**Program 18.18** Functions to read and recognize expect strings.

We first copy the expect string, converting the special characters. Our matching technique is to read characters from the modem until the character matches the first

character of the expect string. We then read enough characters to equal the number of characters in the expect string. From that point we continually read characters from the modem into the buffer, comparing them against the expect string, until we have a match or until the alarm goes off. (There are better algorithms for string matching—ours was chosen to simplify the coding. The number of characters returned by the modem that are compared to the expect string is usually on the order of 50, and the size of the expect string is often around 10–20 characters.)

Note that we have to set an alarm each time we try to match an expect string, as the alarm is the only way we can determine that we didn't receive what we were waiting for.

This completes the server daemon. All it does is open a terminal device and dial a modem. What happens with the terminal device after it is opened depends on the client. We'll now examine a client that provides an interface similar to `cu` and `tip`, allowing us to dial a remote system and log in.

## 18.7 Client Design

The interface between the client and server is only about a dozen lines of code. The client formats a command line, sends it to the server, and receives back either a file descriptor or an error indication. The rest of the client design depends on what the client wants to do with the returned descriptor. In this section we'll outline the design of the `call` client that works like the familiar `cu` and `tip` programs. It allows us to call a remote system and log in to it. The remote system need not be a Unix system—we can use it to communicate with any system or device that's connected to the host with an RS-232 serial connection.

### Terminal Line Disciplines

In Figures 12.11 and 12.12 we gave an overview of the modem dialer. Figure 18.8 is an expansion of Figure 12.11, recognizing the fact that there are two line disciplines between the user and the modem and assuming that we're using the program to dial into a remote Unix host. (Recall from the output of Program 12.10 that for a streams-based terminal system, Figure 18.8 is a simplification. There may be multiple streams modules making up the line discipline and multiple modules making up the terminal device driver. We also don't explicitly show the stream head.)

The two dashed boxes in Figure 18.8 above the modem on the local system were established by the server's `tty_open` function (Program 18.14). That function set the dashed terminal line discipline module to noncanonical (i.e., raw) mode. The modem on the local system was dialed by the server's `tty_dial` function (Program 18.15). The two arrows between the dashed terminal line discipline box and the `call` process correspond to the descriptor returned by the server. (We show the single descriptor as two arrows, to reiterate the fact that it's a full-duplex descriptor.)

The line discipline box beneath the shell on the remote system is set by the login process on that system to be in the canonical mode. After we have dialed the remote

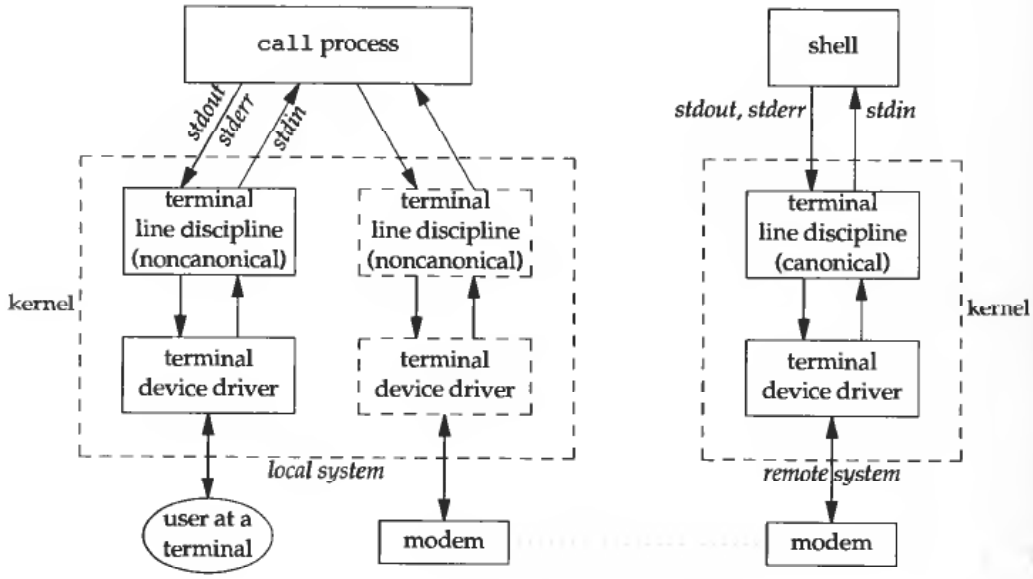


**Figure 18.8** Overview of modem dialer process to log in to remote Unix host.

system we want the special terminal input characters (end of file, erase line, etc. from Section 11.3) recognized by the line discipline module on the remote host. That means we have to set the mode of the line discipline module above the terminal (standard input, standard output, and standard error of the `call` process) to noncanonical mode.

## One Process or Two?

In Figure 18.8 we show the `call` process as a single process. Doing so requires support for an I/O multiplexing function such as `select` or `poll`, since two descriptors are being read from and two descriptors are being written to. We could also design the client as two processes, a parent and a child, as we showed in Figure 12.12. Figure 18.9 shows only these two processes and the line disciplines beneath them. Historically, `cu` and `tip` have always been two processes, as in Figure 18.9. This is because early Unix systems didn't support an I/O multiplexing function.

We choose to use a single process for the following two reasons.

1. Having two processes complicates the termination of the client. If we terminate the connection by entering `~.` (a tilde followed by a period) at the beginning of a line, the child recognizes this and terminates. The parent then has to catch the `SIGCHLD` signal so that the parent can terminate too.

   If the connection is terminated by the remote system or if the line is dropped, the parent will detect this by reading an end of file from the modem descriptor. The parent then has to notify the child, so that the child can also terminate.

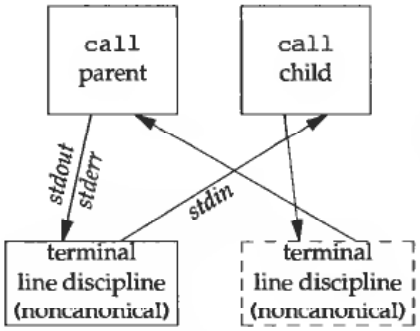


**Figure 18.9** The `call` client as two processes.

Using a single process obviates the need for one process notifying the other when it terminates.

2. We are going to implement a file transfer function in the client, similar to the put and take commands of `cu` and `tip`. We enter these commands on the standard input, on a line that begins with a tilde (the default escape character). These commands are recognized by the child if two processes are being used (Figure 18.9). But the file that's received by the client, in the case of a take command, comes across the modem descriptor, which is being `read` by the parent. This means, to implement the take command, the child has to notify the parent so that the parent stops reading from the modem. The parent is probably blocked in a `read` on this descriptor, so a signal is required to interrupt the parent's `read`. When the child is done, another notification is required to tell the parent to resume reading from the modem. While possible, this scenario quickly becomes messy.

A single process simplifies the entire client. By using a single process, however, we lose the ability to job-control stop just the child. The BSD `tip` program supports this feature. It allows us to stop the child while the parent continues running. This means all the terminal input is directed back to our shell instead of the child, letting us work on the local system, but we'll still see any output generated by the remote system. This is handy if we start a long running job on the remote system and want to see any output that it generates, while working on the local system.

We now look at the source code to implement the client.

## 18.8 Client Source Code

The client is smaller than the server, since the client doesn't handle all the details of connecting the remote system—the server from Section 18.6 handles this. About one-half of the client is to handle commands such as take and put.

Program 18.19 shows the `call.h` header that is included by all the source files.

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    <errno.h>
#include    <termios.h>
#include    "ourhdr.h"

#define CS_CALL "/home/stevens/calld"   /* well-known server name */
#define CL_CALL "call"                  /* command for server */

                /* declare global variables */
extern char  escapec;   /* tilde for local commands */
extern char *src;       /* for take and put commands */
extern char *dst;       /* for take and put commands */

                /* function prototypes */
int     call(const char *);
int     doescape(int);
void    loop(int);
int     prompt_read(char *, int (*)(int, char **));
void    put(int);
void    take(int);
int     take_put_args(int, char **);
```

**Program 18.19** The `call.h` header.

The command for the server and the server's well-known name have to correspond to the values in Program 18.1.

Program 18.20 shows the `main` function.

```
#include    "call.h"

            /* define global variables */
char     escapec = '~';
char    *src;
char    *dst;

static void usage(char *);

int
main(int argc, char *argv[])
{
    int         c, remfd, debug;
    char        args[MAXLINE];

    args[0] = 0;    /* build arg list for conn server here */
    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "des:o")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = 1;
            strcat(args, "-d ");
            break;
```

```
        case 'e':       /* even parity */
            strcat(args, "-e ");
            break;

        case 'o':       /* odd parity */
            strcat(args, "-o ");
            break;

        case 's':       /* speed */
            strcat(args, "-s ");
            strcat(args, optarg);
            strcat(args, " ");
            break;

        case '?':
            usage("unrecognized option");
        }
    }
    if (optind < argc)
        strcat(args, argv[optind]); /* name of host to call */
    else
        usage("missing <hostname> to call");

    if ( (remfd = call(args)) < 0)  /* place the call */
        exit(1);    /* call() prints reason for failure */
    printf("Connected\n");

    if (tty_raw(STDIN_FILENO) < 0)  /* user's tty to raw mode */
        err_sys("tty_raw error");
    if (atexit(tty_atexit) < 0)     /* reset user's tty on exit */
        err_sys("atexit error");

    loop(remfd);                    /* and do it */

    printf("Disconnected\n\r");
    exit(0);
}

static void
usage(char *msg)
{
    err_quit("%s\nusage: call -d -e -o -s<speed> <hostname>", msg);
}
```

**Program 18.20** The `main` function.

It processes the command-line arguments, saving them in the array `args`, which is sent to the server. The function `call` contacts the server and returns the file descriptor to the remote system.

The line discipline module above the terminal (Figure 18.8) is set to noncanonical mode using the `tty_raw` function (Program 11.10). To reset the terminal when we're done we establish the function `tty_atexit` as an exit handler.

The function `loop` is then called to copy everything that we enter to the modem and everything from the modem to the terminal.

The `call` function in Program 18.21 contacts the server to obtain a file descriptor for the modem. As we said earlier, it takes only a dozen lines of code to contact the server to obtain the descriptor.

```
#include    "call.h"
#include    <sys/uio.h>     /* struct iovec */

/* Place the call by sending the "args" to the calling server,
 * and reading a file descriptor back. */

int
call(const char *args)
{
    int             csfd, len;
    struct iovec    iov[2];

                    /* create connection to conn server */
    if ( (csfd = cli_conn(CS_CALL)) < 0)
        err_sys("cli_conn error");

    iov[0].iov_base = CL_CALL " ";
    iov[0].iov_len  = strlen(CL_CALL) + 1;
    iov[1].iov_base = (char *) args;
    iov[1].iov_len  = strlen(args) + 1;
                            /* null at end of args always sent */
    len = iov[0].iov_len + iov[1].iov_len;
    if (writev(csfd, &iov[0], 2) != len)
        err_sys("writev error");

                    /* read back descriptor */
                    /* returned errors handled by write() */
    return( recv_fd(csfd, write) );
}
```

**Program 18.21** The `call` function.

The function `loop` handles the I/O multiplexing between the two input streams and the two output streams. We can use either `poll` or `select`, depending what the local system provides. Program 18.22 shows an implementation using `poll`.

```
#include    "call.h"
#include    <poll.h>
#include    <stropts.h>

/* Copy everything from stdin to "remfd",
 * and everything from "remfd" to stdout. */

#define BUFFSIZE    512

void
loop(int remfd)
```

```
{
    int             bol, n, nread;
    char            c, buff[BUFFSIZE];
    struct pollfd   fds[2];

    setbuf(stdout, NULL);       /* set stdout unbuffered */
                                /* (for printfs in take() and put() */
    fds[0].fd = STDIN_FILENO;   /* user's terminal input */
    fds[0].events = POLLIN;
    fds[1].fd = remfd;          /* input from remote (modem) */
    fds[1].events = POLLIN;

    for ( ; ; ) {
        if (poll(fds, 2, INFTIM) <= 0)
            err_sys("poll error");

        if (fds[0].revents & POLLIN) {  /* data to read on stdin */
            if (read(STDIN_FILENO, &c, 1) != 1)
                err_sys("read error from stdin");

            if (c == escapec && bol) {
                if ( (n = doescape(remfd)) < 0)
                    break;      /* user wants to terminate */
                else if (n == 0)
                    continue;   /* escape seq has been processed */

                /* else, char following escape was not special,
                   so it's returned and echoed below */
                c = n;
            }
            if (c == '\r' || c == '\n')
                bol = 1;
            else
                bol = 0;

            if (write(remfd, &c, 1) != 1)
                err_sys("write error");
        }
        if (fds[0].revents & POLLHUP)
            break;      /* stdin hangup -> done */

        if (fds[1].revents & POLLIN) {  /* data to read from remote */
            if ( (nread = read(remfd, buff, BUFFSIZE)) <= 0)
                break;      /* error or EOF, terminate */

            if (writen(STDOUT_FILENO, buff, nread) != nread)
                err_sys("writen error to stdout");
        }
        if (fds[1].revents & POLLHUP)
            break;      /* modem hangup -> done */
    }
}
```

**Program 18.22** The `loop` function using the `poll` function.

The basic loop of this function just waits for data to appear from either the terminal or the modem. When data is `read` from the terminal, it's just copied to the modem and vice versa. The only complication is to recognize the escape character (the tilde) as the first character of a line.

Note that we `read` one character at a time from the terminal (standard input), but up to one buffer at a time from the modem. One reason for the single character at a time from the terminal is because we have to look at every character to know when a new line begins, to recognize the special commands. Although this character-at-a-time I/O is expensive in terms of CPU time (recall Figure 3.1), there is usually far less input from the terminal than from the remote system. (In remote login sessions using this program measured by the author, there are around 100 characters output by the remote host for every character input.)

When the escape character is seen, `doescape` is called to process the command (Program 18.23). We support only five commands. Simple commands are handled directly in this function, while the more complicated take and put commands are handled by separate functions (`take` and `put`).

- A period terminates the client. For some devices, such as a laser printer, this is the only way to terminate the client. When we're logged into a remote system, such as in Figure 18.8, logging out from that system usually causes the remote modem to drop the phone line, causing a hangup to be received on the modem descriptor by the `loop` function.
- If the system supports job control we recognize the job-control suspend character and suspend the client. Note that it is simpler for us to recognize this character directly and stop ourselves than to have the line discipline recognize the character and generate the `SIGSTOP` signal (compare with Program 10.22). We have to reset the terminal mode before stopping ourselves, and reset it when we're continued.
- A pound sign generates a BREAK on the modem descriptor. We use the POSIX.1 `tcsendbreak` function to do this (Section 11.8). The BREAK condition often causes the remote system's `getty` or `ttymon` program to switch line speeds (Section 9.2).
- The take and put commands require separate functions to be called. The way to distinguish between the two commands is to remember that the command describes what the client is doing on the local system: taking a file from the remote system or putting a file to the remote system.

Program 18.24 shows the code required to handle the take command. The function `take` first calls `prompt_read` (which we show in Program 18.25) to echo `~[take]:` in response to the `~t` command. The `prompt_read` function then reads a line of input from the terminal, containing the source pathname (the file on the remote host) and the destination pathname (the file on the local host). The results are stored in the global variables `src` and `dst`.

```
#include    "call.h"
#include    <signal.h>

/* Called when first character of a line is the escape character
 * (tilde).  Read the next character and process.  Return -1
 * if next character is "terminate" charcter, 0 if next character
 * is valid command character (that's been processed), or next
 * character itself (if the next character is not special). */

int
doescape(int remfd)
{
    char    c;

    if (read(STDIN_FILENO, &c, 1) != 1)     /* next input char */
        err_sys("read error from stdin");

    if (c == escapec)       /* two in a row -> process as one */
        return(escapec);

    else if (c == '.') {    /* terminate */
        write(STDOUT_FILENO, "~.\n\r", 4);
        return(-1);

#ifdef  VSUSP
    } else if (c == tty_termios()->c_cc[VSUSP]) { /* suspend client */
        tty_reset(STDIN_FILENO);    /* restore tty mode */
        kill(getpid(), SIGTSTP);    /* suspend ourself */

        tty_raw(STDIN_FILENO);      /* and reset tty to raw */
        return(0);
#endif

    } else if (c == '#') {  /* generate break */
        tcsendbreak(remfd, 0);
        return(0);

    } else if (c == 't') {  /* take a file from remote host */
        take(remfd);
        return(0);

    } else if (c == 'p') {  /* put a file to remote host */
        put(remfd);
        return(0);
    }

    return(c);          /* not a special character */
}
```

**Program 18.23** The escape function.

```
#include    "call.h"

#define CTRLA   001     /* eof designator for take */

static int       rem_read(int);
static char      rem_buf[MAXLINE];
static char     *rem_ptr;
static int       rem_cnt = 0;

/* Copy a file from remote to local. */

void
take(int remfd)
{
    int     n, linecnt;
    char    c, cmd[MAXLINE];
    FILE    *fpout;

    if (prompt_read("~[take] ", take_put_args) < 0) {
        printf("usage: [take] <sourcefile> <destfile>\n\r");
        return;
    }

            /* open local output file */
    if ( (fpout = fopen(dst, "w")) == NULL) {
        err_ret("can't open %s for writing", dst);
        putc('\r', stderr);
        fflush(stderr);
        return;
    }

            /* send cat/echo command to remote host */
    sprintf(cmd, "cat %s; echo %c\r", src, CTRLA);
    n = strlen(cmd);
    if (write(remfd, cmd, n) != n)
        err_sys("write error");

            /* read echo of cat/echo command line from remote host */
    rem_cnt = 0;        /* initialize rem_read() */
    for ( ; ; ) {
        if ( (c = rem_read(remfd)) == 0)
            return;     /* line has dropped */
        if (c == '\n')
            break;      /* end of echo line */
    }

            /* read file from remote host */
    linecnt = 0;
    for ( ; ; ) {
        if ( (c = rem_read(remfd)) == 0)
            break;          /* line has dropped */
        if (c == CTRLA)
            break;          /* all done */
```

```
            if (c == '\r')
                continue;           /* ignore returns */
            if (c == '\n')          /* but newlines are written to file */
                printf("\r%d", ++linecnt);
            if (putc(c, fpout) == EOF)
                break;              /* output error */
        }
        if (ferror(fpout) || fclose(fpout) == EOF) {
            err_msg("output error to local file");
            putc('\r', stderr);
            fflush(stderr);
        }
        c = '\n';
        write(remfd, &c, 1);
}

/* Read from remote.  Read up to MAXLINE, but parcel out one
 * character at a time. */

int
rem_read(int remfd)
{
    if (rem_cnt <= 0) {
        if ( (rem_cnt = read(remfd, rem_buf, MAXLINE)) < 0)
            err_sys("read error");
        else if (rem_cnt == 0)
            return(0);
        rem_ptr = rem_buf;
    }
    rem_cnt--;
    return(*rem_ptr++ & 0177);
}
```

**Program 18.21** Processing the take command.

After the `take` function opens the local file for writing it sends the following command to the remote host:

`cat` *sourcefile* `; echo ^A`

This causes the remote host to execute the `cat` command, followed by an echo of the ASCII Control-A character. We look for this Control-A in all the characters that are returned by the remote host, and when we encounter it, we know the file transfer is complete. Note that we also have to read back the echo of the command line that we send to the remote host. Only after that echo do we start receiving the output of the `cat` command.

While we're reading the remote file we look for newline characters and count the lines returned. We display these at the left margin, overwriting each line number with the next (since we terminate the line in the `printf` with a carriage return only and not a newline). This provides a visual display on the terminal of the progress of the file transfer and a final line count at the end.

This source file also contains the function `rem_read`, which is called to read each character from the remote host. We `read` up to one buffer at a time, but return only one character at a time to the caller.

Originally the take command was written to `read` one character at a time, similar to what `cu` and `tip` have historically done. Ten years ago, when 1200 baud modems were considered fast, this was OK. But with today's much faster modems, delivering characters to the terminal device driver at 9600 baud and above, characters get lost, even on the faster CPUs found today. The author encountered this with both `cu` and `tip`, using a Telebit T2500 modem in PEP mode, even when both the local host and remote host use flow control. When transferring a large text file (about 75,000 bytes) about half the time characters were lost, requiring the transfer to be done again.

The solution was just to code the `rem_read` function to read up to a buffer at a time. Doing this reduced the system CPU time by a factor of three (from 16 seconds to 5 seconds, to transfer the 75,000 byte file) and provided a reliable transfer every time. A counter was temporarily added to the `rem_read` function, to see how many bytes were returned by each call to `read`. Figure 18.10 shows the results.

| #bytes | Count | #bytes | Count | #bytes | Count | #bytes | Count |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 28 | 2 | 39 | 1 | 55 | 1 |
| 13 | 1 | 29 | 1 | 40 | 1 | 56 | 9 |
| 16 | 1 | 32 | 1 | 46 | 1 | 57 | 751 |
| 17 | 1 | 33 | 1 | 48 | 2 | 58 | 530 |
| 22 | 1 | 34 | 1 | 51 | 2 | 59 | 2 |
| 24 | 1 | 35 | 1 | 52 | 2 | 114 | 1 |
| 25 | 4 | 37 | 1 | 53 | 1 | 115 | 1 |
| 26 | 3 | 38 | 1 | 54 | 1 | 194 | 1 |

**Figure 18.10** Number of bytes returned by `read` during file transfer.

Only once was a single byte returned; 99% of the time either 57 or 58 bytes were returned by `read`. Making this small change reduced the number of `read`s from more than 75,000 to 1,329.

Note that the number of bytes returned by `read` in Figure 18.10 occurred even though the line discipline module for the modem had its MIN set to 1 and TIME set to 0 by the `tty_open` function (Program 18.14). This is case B from Section 11.11. This reiterates the fact that MIN is only a minimum. If we ask for more than the minimum, and the bytes are ready to be read, they're returned. We are not restricted to character-at-a-time input when we set MIN to 1.

Program 18.25 shows the two ancillary functions `take_put_args` and `prompt_read`. The latter is called from both the `take` and `put` functions, with the former as an argument (that is then called by the `buf_args` function, Program 15.17).

```
#include    "call.h"

/* Process the argv-style arguments for take or put commands. */

int
take_put_args(int argc, char **argv)
```

```
{
    if (argc == 1) {
        src = dst = argv[0];
        return(0);
    } else if (argc == 2) {
        src = argv[0];
        dst = argv[1];
        return(0);
    }
    return(-1);
}

static char cmdargs[MAXLINE];
            /* can't be automatic; src/dst point into here */

/* Read a line from the user.  Call our buf_args() function to
 * break it into an argv-style array, and call userfunc() to
 * process the arguments. */

int
prompt_read(char *prompt, int (*userfunc)(int, char **))
{
    int     n;
    char    c, *ptr;

    tty_reset(STDIN_FILENO);    /* allow user's editing chars */

    n = strlen(prompt);
    if (write(STDOUT_FILENO, prompt, n) != n)
        err_sys("write error");

    ptr = cmdargs;
    for ( ; ; ) {
        if ( (n = read(STDIN_FILENO, &c, 1)) < 0)
            err_sys("read error");
        else if (n == 0)
            break;
        if (c == '\n')
            break;
        if (ptr < &cmdargs[MAXLINE-2])
            *ptr++ = c;
    }
    *ptr = 0;       /* null terminate */

    tty_raw(STDIN_FILENO);      /* reset tty mode to raw */

    return( buf_args(cmdargs, userfunc) );
                /* return whatever userfunc() returns */
}
```

**Program 18.25** The `take_put_args` and `prompt_read` functions.

The function prompt_read reads a line of input from the terminal, and then calls buf_args to split the line into a standard argument list that is processed by take_put_args. Note that the terminal is reset to canonical mode to read the arguments, allowing the use of the standard editing characters while entering the line.

The final client function is put, shown in Program 18.26. It is called to copy a local file to the remote host.

```
#include    "call.h"

/* Copy a file from local to remote. */

void
put(int remfd)
{
    int     i, n, linecnt;
    char    c, cmd[MAXLINE];
    FILE    *fpin;

    if (prompt_read("~[put] ", take_put_args) < 0) {
        printf("usage: [put] <sourcefile> <destfile>\n\r");
        return;
    }

            /* open local input file */
    if ( (fpin = fopen(src, "r")) == NULL) {
        err_ret("can't open %s for reading", src);
        putc('\r', stderr);
        fflush(stderr);
        return;
    }

            /* send stty/cat/stty command to remote host */
    sprintf(cmd, "stty -echo; cat >%s; stty echo\r", dst);
    n = strlen(cmd);
    if (write(remfd, cmd, n) != n)
        err_sys("write error");
    tcdrain(remfd);     /* wait for our output to be sent */
    sleep(4);           /* and let stty take effect */

            /* send file to remote host */
    linecnt = 0;
    for ( ; ; ) {
        if ( (i = getc(fpin)) == EOF)
            break;          /* all done */
        c = i;
        if (write(remfd, &c, 1) != 1)
            break;          /* line has probably dropped */
        if (c == '\n')      /* but newlines are written to file */
            printf("\r%d", ++linecnt);
    }
```

```
            /* send EOF to remote, to terminate cat */
    c = tty_termios()->c_cc[VEOF];
    write(remfd, &c, 1);
    tcdrain(remfd);             /* wait for our output to be sent */
    sleep(2);
    tcflush(remfd, TCIOFLUSH);  /* flush echo of stty/cat/stty */
    c = '\n';
    write(remfd, &c, 1);

    if (ferror(fpin)) {
        err_msg("read error of local file");
        putc('\r', stderr);
        fflush(stderr);
    }
    fclose(fpin);
}
```

**Program 18.26** The put function.

As with the take command, we send a command string to the remote system. This time the command is

```
stty -echo; cat > destfile ; stty echo
```

We have to turn echo off, otherwise the entire file would also be sent back to us. To terminate the `cat` command we send the end-of-file character (often Control-D). This requires that the same end-of-file character be used on both the local system and the remote system. Additionally, the file cannot contain the ERASE or KILL characters in use on the remote system.

## 18.9 Summary

In this chapter we've looked at two different programs: a daemon server that dials a modem and a remote login program that uses the server to contact a remote system that's connected through a terminal port. The server can be used by other programs that need to contact remote systems or hardware devices connected through asynchronous terminal ports.

The design of the server was similar to the open server in Section 15.6 and required the use of stream pipes, unique per-client connections to the server, and the passing of file descriptors. These advanced IPC features allow us to build client–server applications with many desirable features, as described in Section 18.3.

The client is similar to the `cu` and `tip` programs provided by many Unix systems, but in our example we didn't have to worry about dialing a modem, interfering with UUCP lock files, setting the characteristics of the modem's line discipline module, and the like. The server handles all these details. It let us concentrate on the real issues of the client, such as providing a reliable file transfer mechanism.

## Exercises

**18.1** How can we avoid step 0 (starting the server by hand) in Section 18.3?

**18.2** What happens if we don't set `optind` to 1 in Program 18.4?

**18.3** What happens if someone edits the `Systems` file between the time `request` (Program 18.10) `fork`s a child and the time the child terminates with a status of 1?

**18.4** In Section 7.8 we said to be careful any time we use pointers into a region that gets `realloc`ed, since the region can move around in memory on each call to `realloc`. Why can we use the pointer `cliptr` in Program 18.3 when the `client` array is manipulated by `realloc`?

**18.5** What happens if either of the pathname arguments to the take and put commands contain a semicolon?

**18.6** Modify the server to read its three data files once when it starts, storing them in memory. If the files are modified, how should the server handle this?

**18.7** In Program 18.21 why do we cast the argument `args` when filling in the structure for `writev`?

**18.8** Implement Program 18.22 using `select` instead of `poll`.

**18.9** How can you verify that the file being sent with the put command does not contain characters that will be interpreted by the line discipline on the remote system?

**18.10** The faster the dialing function recognizes that a dial has failed, the faster it can proceed to the next possible entry in the `Systems` file. For example, if we can determine that the remote phone is busy and terminate before the timer in `expect_str` expires, we can save 15 or 20 seconds. To handle these types of errors, the 4.3BSD UUCP expect–send strings allow an expect string of `ABORT`, followed by a string that if matched, aborts the current dial. For example, right before the final expect string `CONNECT\sFAST` in Figure 18.4 we would like to add

```
ABORT BUSY
```

Implement this feature.

# 19

# Pseudo Terminals

## 19.1 Introduction

In Chapter 9 we saw that terminal logins come in through a terminal device, automatically providing terminal semantics. There is a terminal line discipline (Figure 11.2) between the terminal and the programs that we run, so we can set the terminal's special characters (backspace, line erase, interrupt, etc.) and the like. When a login arrives on a network connection, however, a terminal line discipline is not automatically provided between the incoming network connection and the login shell. Figure 9.5 showed that a *pseudo-terminal* device driver is used to provide terminal semantics.

In addition to network logins, pseudo terminals have other uses that we explore in this chapter. We start by providing functions to create pseudo terminals under SVR4 and 4.3+BSD and then use these functions to write a program that we call `pty`. We'll show various uses of this program: making a transcript of all the character input and output on the terminal (the BSD `script` program) and running coprocesses to avoid the buffering problems we encountered in Program 14.10.

## 19.2 Overview

The term *pseudo terminal* implies that it looks like a terminal to an application program, but it's not a real terminal. Figure 19.1 shows the typical arrangement of the processes involved when a pseudo terminal is being used. The key points in this figure are the following.

1. Normally a process opens the pseudo-terminal master and then calls `fork`. The child establishes a new session, opens the corresponding pseudo-terminal slave, duplicates it to be standard input, standard output, and standard error, and then

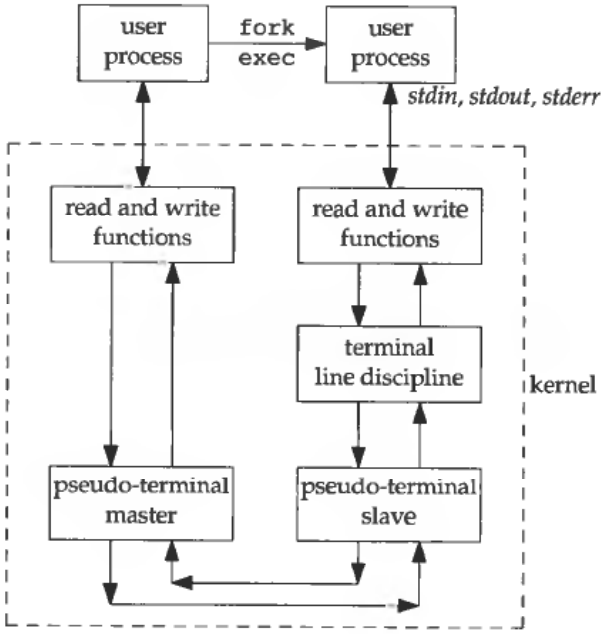


**Figure 19.1** Typical arrangement of processes using a pseudo terminal.

calls `exec`. The pseudo-terminal slave becomes the controlling terminal for the child process.

2. It appears to the user process above the slave that its standard input, standard output, and standard error are a terminal device. It can issue all the terminal I/O functions from Chapter 11 on these descriptors. But since there is not an actual terminal device beneath the slave, functions that don't make sense (change the baud rate, send a break character, set odd parity, etc.) are just ignored.
3. Anything written to the master appears as input to the slave and vice versa. Indeed all the input to the slave comes from the user process above the pseudo-terminal master. This looks like a stream pipe (Figure 15.3) but with the terminal line discipline module above the slave we have additional capabilities over a plain pipe.

Figure 19.1 shows what a pseudo terminal looks like on a BSD system. In Section 19.3.2 we show how to open these devices.

Under SVR4 a pseudo terminal is built using the streams system (Section 12.4). Figure 19.2 details the arrangement of the pseudo-terminal streams modules under SVR4. The two streams modules that are shown as dashed boxes are optional. Note that the three streams modules above the slave are the same as the output from Program 12.10 for a network login. In Section 19.3.1 we show how to build this arrangement of streams modules.

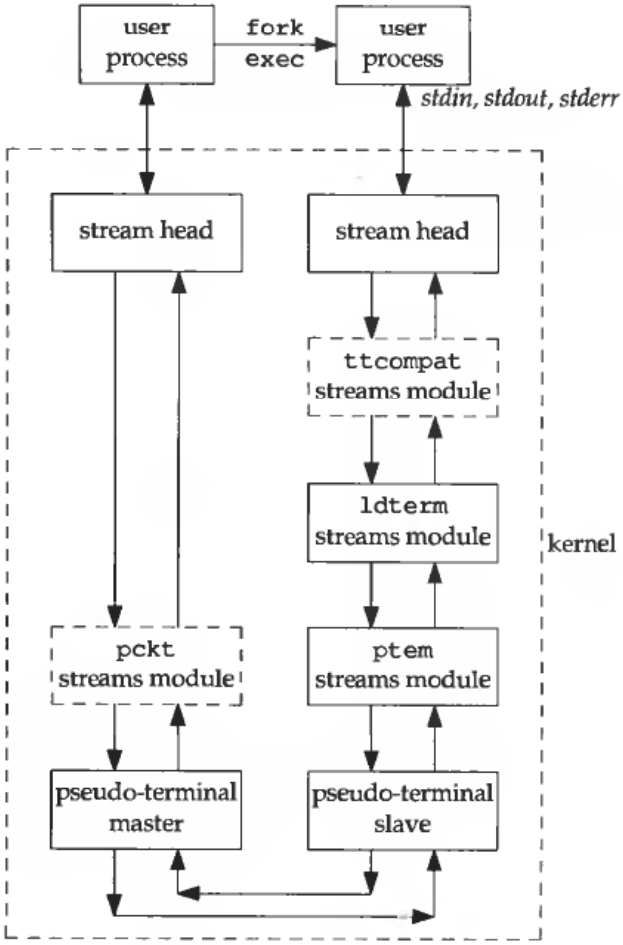


**Figure 19.2** Arrangement of pseudo terminals under SVR4.

From this point on we'll simplify the figures by not showing the "read and write functions" from Figure 19.1 or the "stream head" from Figure 19.2. We'll also use the abbreviation "pty" for pseudo terminal and lump all the streams modules above the slave pty in Figure 19.2 into a box called "terminal line discipline" as in Figure 19.1.

We'll now examine some of the typical uses of pseudo terminals.

### Network Login Servers

Pseudo terminals are built into servers that provide network logins. The typical examples are the `telnetd` and `rlogind` servers. Chapter 15 of Stevens [1990] details the steps involved in the `rlogin` service. Once the login shell is running on the remote host we have the arrangement shown in Figure 19.3. A similar arrangement is used by the `telnetd` server.

We show two calls to `exec` between the `rlogind` server and the login shell, because the `login` program is usually between the two to validate the user.

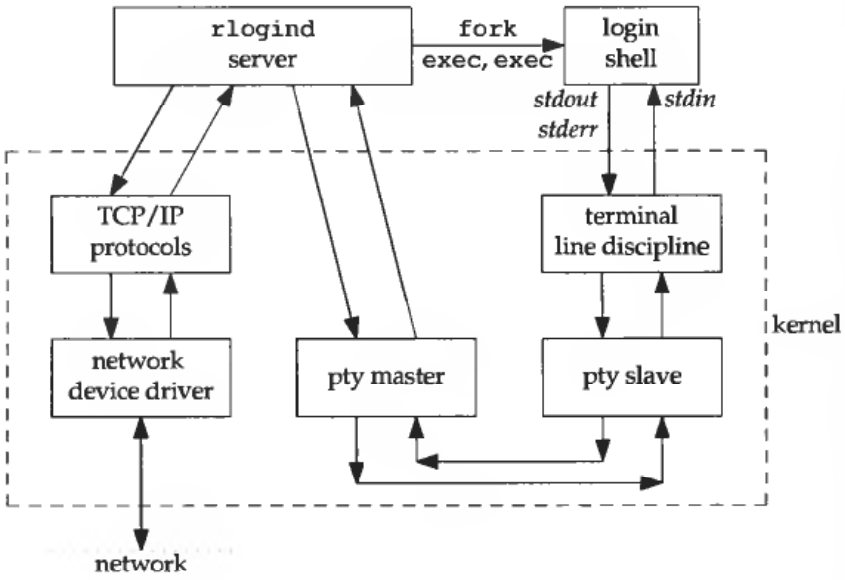


**Figure 19.3** Arrangement of processes for `rlogind` server.

A key point in this figure is that the process driving the pty master is normally reading and writing another I/O stream at the same time. In this example the other I/O stream is the TCP/IP protocol box. This implies that the process must be using some form of I/O multiplexing (Section 12.5), such as `select` or `poll` or must be divided into two processes. Recall the discussion of one process versus two in Section 18.7.

### `script` Program

The `script`(1) program that is supplied with SVR4 and 4.3+BSD makes a copy in a file of everything that is input and output during a terminal session. It does this by placing itself between the terminal and a new invocation of our login shell. Figure 19.4 details the interactions involved in the `script` program. Here we specifically show that the `script` program is normally run from a login shell, which then waits for `script` to terminate.

While `script` is running, everything output by the terminal line discipline above the pty slave is copied to the script file (usually called `typescript`). Since our keystrokes are normally echoed by that line discipline module, the script file also contains our input. The script file won't contain any passwords that we enter, however, since passwords aren't echoed.

> All the examples in this text that consist of running a program and displaying its output were generated with the `script` program. This avoids typographical errors that could occur when copying program output by hand.

After developing the general `pty` program in Section 19.5 we'll see that a trivial shell script turns it into a version of the `script` program.

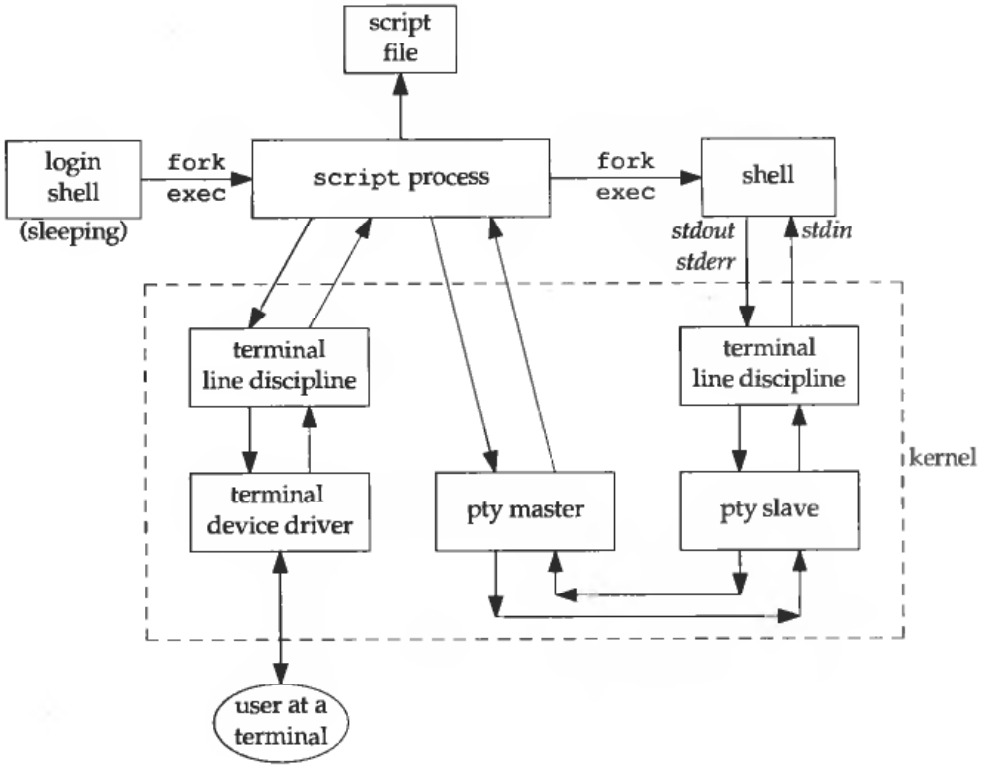


**Figure 19.4** The `script` program.

## `expect` Program

Pseudo terminals can be used to drive interactive programs in noninteractive modes. Numerous programs are hardwired to require a terminal to run. The `call` process in Section 18.7 is an example. It assumes that standard input is a terminal and sets it to raw mode when it starts up (Program 18.20). This program cannot be run from a shell script to automatically dial a remote system, log in, fetch some information, and log out.

Rather than modify all the interactive programs to support a batch mode of operation, a better solution is to provide a way to drive any interactive program from a script. The `expect` program [Libes 1990; 1991] provides a way to do this. It uses pseudo terminals to run other programs, similar to the `pty` program in Section 19.5. But `expect` also provides a programming language to examine the output of the program being run to make decisions about what to send the program as input. When an interactive program is being run from a script, we can't just copy everything from the script to the program and vice versa. Instead we have to send the program some input, look at its output, and decide what to send it next.

## Running Coprocesses

In the coprocess example in Program 14.10 we couldn't invoke a coprocess that used the standard I/O library for its input and output, because when we talked to the coprocess

across a pipe, the standard I/O library fully buffered the standard input and standard output, leading to a deadlock. If the coprocess is a compiled program for which we don't have the source code, we can't add `fflush` statements to solve this problem. Figure 14.9 showed a process driving a coprocess. What we need to do is place a pseudo terminal between the two processes, as shown in Figure 19.5.

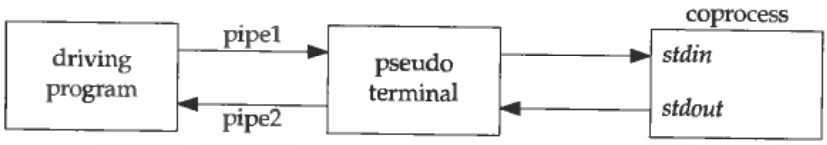


**Figure 19.5** Driving a coprocess using a pseudo terminal.

Now the standard input and standard output of the coprocess look like a terminal device, so the standard I/O library will set these two streams to be line buffered.

There are two different ways for the parent to obtain a pseudo terminal between itself and the coprocess. (The parent in this case could be either Program 14.9, which used two pipes to communicate with the coprocess, or Program 15.1, which used a single stream pipe.) One way is for the parent to call the `pty_fork` function directly (Section 19.4), instead of calling `fork`. Another is to `exec` the `pty` program (Section 19.5) with the coprocess as its argument. We'll look at these two solutions after showing the `pty` program.

### Watching the Output of Long Running Programs

If we have a program that runs for a long time we can easily run it in the background using any of the standard shells. But if we redirect its standard output to a file, and if it doesn't generate much output, we can't easily monitor its progress because the standard I/O library will fully buffer its standard output. All that we'll see are blocks of output written by the standard I/O library to the output file, possibly in chunks as large as 8192 bytes.

If we have the source code we can insert calls to `fflush`. Alternatively, we can run the program under the `pty` program, making its standard I/O library think that its standard output is a terminal. Figure 19.6 shows this arrangement, where we have called the slow output program `slowout`. The `fork/exec` arrow from the login shell to the `pty` process is shown as a dashed arrow to reiterate that the `pty` process is running as a background job.

## 19.3 Opening Pseudo-Terminal Devices

Opening a pseudo-terminal device differs between SVR4 and 4.3+BSD. We provide two functions that handle all the details: `ptym_open` to open the next available pty master device and `ptys_open` to open the corresponding slave device.

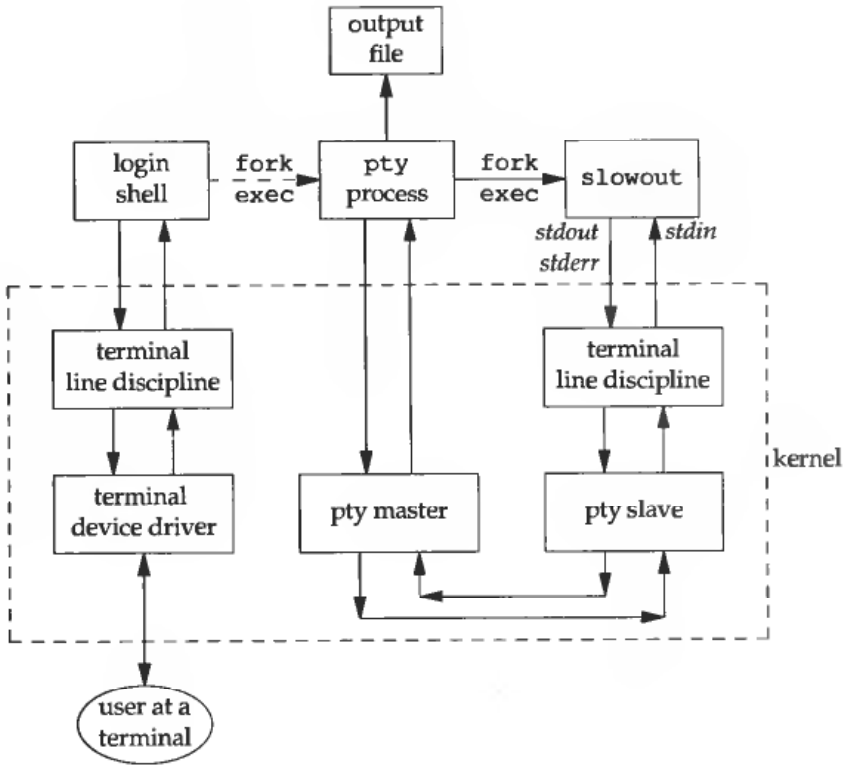


**Figure 19.6** Running a slow output program using a pseudo terminal.

```
#include "ourhdr.h"

int ptym_open(char *pts_name);
```

Returns: file descriptor of pty master if OK, –1 on error

```
int ptys_open(int fdm, char *pts_name);
```

Returns: file descriptor of pty slave if OK, –1 on error

Normally we don't call these two functions directly—the function `pty_fork` (Section 19.4) calls them and also `fork`s a child process.

`ptym_open` determines the next available pty master and opens the device. The caller must allocate an array to hold the name of either the master or slave, and if the call succeeds the name of the corresponding slave is returned through *pts_name*. This name and the file descriptor returned by `ptym_open` are then passed to `ptys_open`, which opens the slave device.

The reason for providing two functions to open the two devices will become obvious when we show the `pty_fork` function. Normally a process calls `ptym_open` to open the master and obtain the name of the slave. The process then `fork`s and the child calls `ptys_open` to open the slave after calling `setsid` to establish a new session. This is how the slave becomes the controlling terminal for the child.

### 19.3.1 System V Release 4

All the details of the streams implementation of pseudo terminals under SVR4 are covered in Chapter 12 of AT&T [1990d]. Three functions are also described there: `grantpt`(3), `unlockpt`(3), and `ptsname`(3).

The pty master device is `/dev/ptmx`. It is a streams *clone device*. This means that when we open the clone device, its open routine automatically determines the first unused pty master device and opens that unused device. (We'll see in the next section that under Berkeley systems we have to find the first unused pty master ourselves.)

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <errno.h>
#include    <fcntl.h>
#include    <stropts.h>
#include    "ourhdr.h"

extern char *ptsname(int);  /* prototype not in any system header */

int
ptym_open(char *pts_name)
{
    char    *ptr;
    int     fdm;

    strcpy(pts_name, "/dev/ptmx");  /* in case open fails */
    if ( (fdm = open(pts_name, O_RDWR)) < 0)
        return(-1);

    if (grantpt(fdm) < 0) {     /* grant access to slave */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) {    /* clear slave's lock flag */
        close(fdm);
        return(-3);
    }
    if ( (ptr = ptsname(fdm)) == NULL) {    /* get slave's name */
        close(fdm);
        return(-4);
    }

    strcpy(pts_name, ptr);  /* return name of slave */
    return(fdm);            /* return fd of master */
}
```

```
int
ptys_open(int fdm, char *pts_name)
{
    int     fds;

            /* following should allocate controlling terminal */
    if ( (fds = open(pts_name, O_RDWR)) < 0) {
        close(fdm);
        return(-5);
    }
    if (ioctl(fds, I_PUSH, "ptem") < 0) {
        close(fdm);
        close(fds);
        return(-6);
    }
    if (ioctl(fds, I_PUSH, "ldterm") < 0) {
        close(fdm);
        close(fds);
        return(-7);
    }
    if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
        close(fdm);
        close(fds);
        return(-8);
    }

    return(fds);
}
```

**Program 19.1** Pseudo-terminal open functions for SVR4.

We first open the clone device `/dev/ptmx` and obtain the file descriptor for the pty master. Opening this master device automatically locks out the corresponding slave device.

We then call `grantpt` to change permissions of the slave device. It does the following: (a) changes the ownership of the slave to the effective user ID, (b) changes the group ownership to the group `tty`, and (c) changes the permissions to allow only user-read, user-write, and group-write. The reason for setting the group ownership to `tty` and enabling group-write permission is that the programs `wall`(1) and `write`(1) are set-group-ID to the group `tty`. Calling the `grantpt` function executes the program `/usr/lib/pt_chmod`. This program is set-user-ID to root so that it can modify the ownership and permissions of the slave.

The function `unlockpt` is called to clear an internal lock on the slave device. We have to do this before we can open the slave. Additionally we must call `ptsname` to obtain the name of the slave device. This name is of the form `/dev/pts/`*NNN*.

The next function in the file is `ptys_open`, which does the actual `open` of the slave device. Under SVR4, if the caller is a session leader that does not already have a controlling terminal, this `open` allocates the pty slave as the controlling terminal. If we didn't want this to happen, we could specify the `O_NOCTTY` flag for `open`.

After opening the slave device we push three streams modules onto the slave's stream. `ptem` stands for "pseudo-terminal emulation module" and `ldterm` is the terminal line discipline module. Together these two modules act like a real terminal. `ttcompat` provides compatibility for older V7, 4BSD, and Xenix `ioctl` calls. It's an optional module but since it's automatically pushed for console logins and network logins (see the output from Program 12.10), we push it onto the slave's stream.

The result of calling these two functions is a file descriptor for the master and a file descriptor for the slave.

### 19.3.2 4.3+BSD

Under 4.3+BSD we have to determine the first available pty master device ourself. To do this we start at `/dev/ptyp0` and keep trying until we successfully open a pty master or until we run out of devices. We can get two different errors from open: `EIO` means that the device is already in use, while `ENOENT` means that the device doesn't exist. In the latter case we can terminate the search as all pseudo terminals are in use. Once we are able to open a pty master, say `/dev/pty`*MN*, the name of the corresponding slave is `/dev/tty`*MN*.

The function `ptys_open` in Program 19.2 opens the slave device. We call `chown` and `chmod` but realize that these two functions won't work unless the calling process has superuser permissions. If it is important that the ownership and protection be changed, these two function calls need to placed into a set-user-ID root executable, similar to the SVR4 `grantpt` function.

The open of the slave pty under 4.3+BSD does not have the side effect of allocating the device as the controlling terminal. We'll see in the next section how to allocate the controlling terminal under 4.3+BSD.

This function tries 16 different groups of 16 pty master devices: `/dev/ptyp0` through `/dev/ptyTf`. The actual number of pty devices available depends on two factors: (a) the number configured into the kernel, and (b) the number of special device files that have been created in the `/dev` directory. The number available to any program is the lesser of (a) or (b). Also, even if the lesser of (a) or (b) is greater than 64, many existing BSD applications (`telnetd`, `rlogind`, etc.) search in the first `for` loop in Program 19.2 only through `"pqrs"`.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <errno.h>
#include    <fcntl.h>
#include    <grp.h>
#include    "ourhdr.h"

int
ptym_open(char *pts_name)
{
    int     fdm;
    char    *ptr1, *ptr2;

    strcpy(pts_name, "/dev/ptyXY");
      /* array index: 0123456789 (for references in following code) */
```

```
    for (ptr1 = "pqrstuvwxyzPQRST"; *ptr1 != 0; ptr1++) {
        pts_name[8] = *ptr1;
        for (ptr2 = "0123456789abcdef"; *ptr2 != 0; ptr2++) {
            pts_name[9] = *ptr2;

                        /* try to open master */
            if ( (fdm = open(pts_name, O_RDWR)) < 0) {
                if (errno == ENOENT)    /* different from EIO */
                    return(-1);         /* out of pty devices */
                else
                    continue;           /* try next pty device */
            }

            pts_name[5] = 't';  /* change "pty" to "tty" */
            return(fdm);        /* got it, return fd of master */
        }
    }
    return(-1);     /* out of pty devices */
}

int
ptys_open(int fdm, char *pts_name)
{
    struct group    *grptr;
    int             gid, fds;

    if ( (grptr = getgrnam("tty")) != NULL)
        gid = grptr->gr_gid;
    else
        gid = -1;       /* group tty is not in the group file */

        /* following two functions don't work unless we're root */
    chown(pts_name, getuid(), gid);
    chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP);

    if ( (fds = open(pts_name, O_RDWR)) < 0) {
        close(fdm);
        return(-1);
    }
    return(fds);
}
```

**Program 19.2** Pseudo-terminal open functions for 4.3+BSD.

## 19.4 pty_fork Function

We now use the two functions from the previous section, `ptym_open` and `ptys_open`, to write a new function that we call `pty_fork`. This new function combines the opening of the master and slave with a call to `fork`, establishing the child as a session leader with a controlling terminal.

```
#include <sys/types.h>
#include <termios.h>
#include <sys/ioctl.h> /* 4.3+BSD defines struct winsize here */
#include "ourhdr.h"

pid_t pty_fork(int *ptrfdm, char *slave_name,
               const struct termios *slave_termios,
               const struct winsize *slave_winsize);
```

Returns: 0 in child, process ID of child in parent, –1 on error

The file descriptor of the pty master is returned through the *ptrfdm* pointer.

If *slave_name* is nonnull, the name of the slave device is stored at that location. The caller has to allocate the storage pointed to by this argument.

If the pointer *slave_termios* is nonnull, the referenced structure initializes the terminal line discipline of the slave. If this pointer is null, the system initializes the slave's `termios` structure to an implementation-defined initial state. Similarly, if the *slave_winsize* pointer is nonnull, the referenced structure initializes the slave's window size. If this pointer is null, the `winsize` structure is normally initialized to 0.

Program 19.3 shows the code for this function. This function works under both SVR4 and 4.3+BSD, calling the appropriate `ptym_open` and `ptys_open` functions.

After opening the pty master, `fork` is called. As we mentioned before, we want to wait to call `ptys_open` until in the child, and after calling `setsid` to establish a new session. When it calls `setsid` the child is not a process group leader (why?) so the three steps listed in Section 9.5 occur: (a) a new session is created with the child as the session leader, (b) a new process group is created for the child, and (c) the child has no controlling terminal. Under SVR4 the slave becomes the controlling terminal of this new session when `ptys_open` is called. Under 4.3+BSD we have to call `ioctl` with an argument of `TIOCSCTTY` to allocate the controlling terminal. The two structures `termios` and `winsize` are then initialized in the child. Finally the slave file descriptor is duplicated onto standard input, standard output, and standard error in the child. This means that whatever process the caller `exec`s from the child will have these three descriptors connected to the slave pty (its controlling terminal).

After the call to `fork` the parent just returns the pty master descriptor and returns. In the next section we use the `pty_fork` function in the `pty` program.

```
#include    <sys/types.h>
#include    <termios.h>
#ifndef TIOCGWINSZ
#include    <sys/ioctl.h>   /* 4.3+BSD requires this too */
#endif
#include    "ourhdr.h"

pid_t
pty_fork(int *ptrfdm, char *slave_name,
         const struct termios *slave_termios,
         const struct winsize *slave_winsize)
{
```

```
    int     fdm, fds;
    pid_t   pid;
    char    pts_name[20];

    if ( (fdm = ptym_open(pts_name)) < 0)
        err_sys("can't open master pty: %s", pts_name);

    if (slave_name != NULL)
        strcpy(slave_name, pts_name);   /* return name of slave */

    if ( (pid = fork()) < 0)
        return(-1);

    else if (pid == 0) {        /* child */
        if (setsid() < 0)
            err_sys("setsid error");

                    /* SVR4 acquires controlling terminal on open() */
        if ( (fds = ptys_open(fdm, pts_name)) < 0)
            err_sys("can't open slave pty");
        close(fdm);     /* all done with master in child */
#if defined(TIOCSCTTY) && !defined(CIBAUD)
                /* 4.3+BSD way to acquire controlling terminal */
                /* !CIBAUD to avoid doing this under SunOS */
        if (ioctl(fds, TIOCSCTTY, (char *) 0) < 0)
            err_sys("TIOCSCTTY error");
#endif
                /* set slave's termios and window size */
        if (slave_termios != NULL) {
            if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                err_sys("tcsetattr error on slave pty");
        }
        if (slave_winsize != NULL) {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
                err_sys("TIOCSWINSZ error on slave pty");
        }
                /* slave becomes stdin/stdout/stderr of child */
        if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
            err_sys("dup2 error to stderr");
        if (fds > STDERR_FILENO)
            close(fds);
        return(0);      /* child returns 0 just like fork() */

    } else {                    /* parent */
        *ptrfdm = fdm;  /* return fd of master */
        return(pid);    /* parent returns pid of child */
    }
}
```

**Program 19.3** The pty_fork function.

## 19.5 pty Program

The goal in writing the pty program is to be able to type

```
pty prog arg1 arg2
```

instead of

```
prog arg1 arg2
```

When we use pty to execute another program, that program is executed in a session of its own, connected to a pseudo terminal.

Let's look at the source code for the pty program. The first file (Program 19.4) contains the main function. It calls the pty_fork function from the previous section.

```
#include    <sys/types.h>
#include    <termios.h>
#ifndef TIOCGWINSZ
#include    <sys/ioctl.h>   /* 4.3+BSD requires this too */
#endif
#include    "ourhdr.h"

static void set_noecho(int);     /* at the end of this file */
void        do_driver(char *);   /* in the file driver.c */
void        loop(int, int);      /* in the file loop.c */

int
main(int argc, char *argv[])
{
    int             fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t           pid;
    char            *driver, slave_name[20];
    struct termios  orig_termios;
    struct winsize  size;

    interactive = isatty(STDIN_FILENO);
    ignoreeof = 0;
    noecho = 0;
    verbose = 0;
    driver = NULL;

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d:einv")) != EOF) {
        switch (c) {
        case 'd':       /* driver for stdin/stdout */
            driver = optarg;
            break;

        case 'e':       /* noecho for slave pty's line discipline */
            noecho = 1;
            break;

        case 'i':       /* ignore EOF on standard input */
            ignoreeof = 1;
            break;
```

```
            case 'n':       /* not interactive */
                interactive = 0;
                break;

            case 'v':       /* verbose */
                verbose = 1;
                break;

            case '?':
                err_quit("unrecognized option: -%c", optopt);
            }
        }
        if (optind >= argc)
            err_quit("usage: pty [ -d driver -einv ] program [ arg ... ]");

        if (interactive) {  /* fetch current termios and window size */
            if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
                err_sys("tcgetattr error on stdin");
            if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
                err_sys("TIOCGWINSZ error");
            pid = pty_fork(&fdm, slave_name, &orig_termios, &size);
        } else
            pid = pty_fork(&fdm, slave_name, NULL, NULL);

        if (pid < 0)
            err_sys("fork error");

        else if (pid == 0) {        /* child */
            if (noecho)
                set_noecho(STDIN_FILENO);   /* stdin is slave pty */

            if (execvp(argv[optind], &argv[optind]) < 0)
                err_sys("can't execute: %s", argv[optind]);
        }

        if (verbose) {
            fprintf(stderr, "slave name = %s\n", slave_name);
            if (driver != NULL)
                fprintf(stderr, "driver = %s\n", driver);
        }

        if (interactive && driver == NULL) {
            if (tty_raw(STDIN_FILENO) < 0)  /* user's tty to raw mode */
                err_sys("tty_raw error");
            if (atexit(tty_atexit) < 0)     /* reset user's tty on exit */
                err_sys("atexit error");
        }

        if (driver)
            do_driver(driver);  /* changes our stdin/stdout */

        loop(fdm, ignoreeof);   /* copies stdin -> ptym, ptym -> stdout */

        exit(0);
    }
```

```
static void
set_noecho(int fd)      /* turn off echo (for slave pty) */
{
    struct termios  stermios;

    if (tcgetattr(fd, &stermios) < 0)
        err_sys("tcgetattr error");

    stermios.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    stermios.c_oflag &= ~(ONLCR);
            /* also turn off NL to CR/NL mapping on output */

    if (tcsetattr(fd, TCSANOW, &stermios) < 0)
        err_sys("tcsetattr error");
}
```

**Program 19.4** The `main` function for the `pty` program.

We'll look at the various command-line options when we examine different uses of the `pty` program in the next section.

Before calling `pty_fork` we fetch the current values for the `termios` and `winsize` structures, passing these as arguments to `pty_fork`. This way the pty slave assumes the same initial state as the current terminal.

After returning from `pty_fork` the child optionally turns off echoing for the slave pty and then calls `execvp` to execute the program specified on the command line. All remaining command-line arguments are passed as arguments to this program.

The parent optionally sets the user's terminal to raw mode setting an exit handler to reset the terminal state when `exit` is called. We describe the `do_driver` function in the next section.

The function `loop` (Program 19.5) is then called by the parent. It just copies everything received from the standard input to the pty master and everything from the pty master to standard output. We have the same decision as we had in Section 18.7—one process or two? For variety we have coded it in two processes this time, although a single process using either `select` or `poll` would also work.

```
#include    <sys/types.h>
#include    <signal.h>
#include    "ourhdr.h"

#define BUFFSIZE    512

static void sig_term(int);
static volatile sig_atomic_t    sigcaught;  /* set by signal handler */

void
loop(int ptym, int ignoreeof)
{
    pid_t   child;
    int     nread;
    char    buff[BUFFSIZE];
```

```
    if ( (child = fork()) < 0) {
        err_sys("fork error");

    } else if (child == 0) {    /* child copies stdin to ptym */
        for ( ; ; ) {
            if ( (nread = read(STDIN_FILENO, buff, BUFFSIZE)) < 0)
                err_sys("read error from stdin");
            else if (nread == 0)
                break;      /* EOF on stdin means we're done */

            if (writen(ptym, buff, nread) != nread)
                err_sys("writen error to master pty");
        }

            /* We always terminate when we encounter an EOF on stdin,
               but we only notify the parent if ignoreeof is 0. */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM);   /* notify parent */
        exit(0);    /* and terminate; child can't return */
    }

        /* parent copies ptym to stdout */
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
        err_sys("signal_intr error for SIGTERM");

    for ( ; ; ) {
        if ( (nread = read(ptym, buff, BUFFSIZE)) <= 0)
            break;      /* signal caught, error, or EOF */

        if (writen(STDOUT_FILENO, buff, nread) != nread)
            err_sys("writen error to stdout");
    }

    /* There are three ways to get here: sig_term() below caught the
     * SIGTERM from the child, we read an EOF on the pty master (which
     * means we have to signal the child to stop), or an error. */

    if (sigcaught == 0) /* tell child if it didn't send us the signal */
        kill(child, SIGTERM);
    return;     /* parent returns to caller */
}

/* The child sends us a SIGTERM when it receives an EOF on
 * the pty slave or encounters a read() error. */

static void
sig_term(int signo)
{
    sigcaught = 1;      /* just set flag and return */
    return;             /* probably interrupts read() of ptym */
}
```

**Program 19.5** The loop function.

Note that, with two processes, when one terminates it has to notify the other. We use the SIGTERM signal for this notification.

## 19.6 Using the pty Program

We'll now look at various examples with the pty program, seeing the need for the various command-line options.

If our shell is the KornShell we can execute

```
pty ksh
```

and get a brand new invocation of the shell, running under a pseudo terminal.

If the file ttyname is the program we showed in Program 11.7, then we can run the pty program as follows:

```
$ who
stevens  console Feb  6 10:43
stevens  ttyp0   Feb  6 15:00
stevens  ttyp1   Feb  6 15:00
stevens  ttyp2   Feb  6 15:00
stevens  ttyp3   Feb  6 15:48
stevens  ttyp4   Feb  7 14:28      ttyp4 is the highest pty currently in use
$ pty ttyname                      run Program 11.7 from pty
fd 0: /dev/ttyp5                   ttyp5 is the next available pty
fd 1: /dev/ttyp5
fd 2: /dev/ttyp5
```

### utmp File

In Section 6.7 we described the utmp file that records all users currently logged into a Unix system. The question is whether a user running a program on a pseudo terminal is considered logged in or not. In the case of remote logins, telnetd and rlogind, obviously an entry should be made in the utmp file for the user logged in on the pseudo terminal. There is little agreement, however, whether users running a shell on a pseudo terminal, from a window system or from a program such as script, should have entries made in the utmp file. Some systems record these and some don't. If a system doesn't record these in the utmp file, the who(1) program normally won't show the corresponding pseudo terminals as being used.

Unless the utmp file has other-write permission enabled, random programs that use pseudo terminals won't be able to write to this file. Some systems, however, deliver the utmp file with all write permissions enabled.

### Job-Control Interaction

If we run a job-control shell under pty it works normally. For example,

```
pty ksh
```

runs the KornShell under pty. We can run programs under this new shell and use job control just as our login shell. But if we run an interactive program other than a job-control shell under pty, as in

```
pty cat
```

everything is fine until we type our job-control suspend character. At that point under SVR4 and 4.3+BSD the job-control character is echoed as ^Z and is ignored. Under SunOS 4.1.2 the cat process terminates, the pty process terminates, and we're back to our original shell.

To understand what's going on here we need to examine all the processes involved, their process groups, and sessions. Figure 19.7 shows the arrangement when pty cat is running.

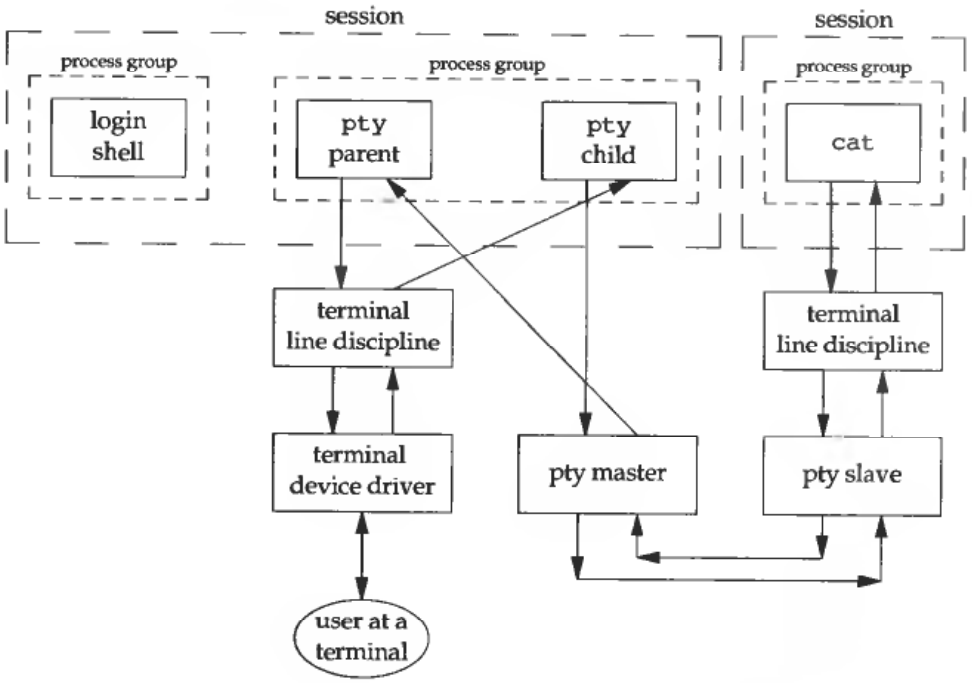


**Figure 19.7** Process groups and sessions for pty cat.

When we type the suspend character (Control-Z) it is recognized by the line discipline module beneath the cat process, since pty puts the terminal (beneath the pty parent) into a raw mode. But the kernel won't stop the cat process because it belongs to an orphaned process group (Section 9.10). The parent of cat is the pty parent, and it belongs to another session.

Different systems handle this condition differently. POSIX.1 just says that the SIGTSTP signal can't be delivered to the process. Earlier Berkeley-derived systems deliver SIGKILL instead, which the process can't even catch. This is what we see under SunOS 4.1.2. (The POSIX.1 Rationale suggests SIGHUP as a better alternative, since the process can at least catch it.) Enabling process accounting and looking at the termination status of the cat process with Program 8.17 shows that it is indeed terminated by a SIGKILL signal.

Under SVR4 and 4.3+BSD we use a modification to Program 10.22 to see what's going on. The modification has the signal handler for SIGTSTP print when the signal is caught and print again when the SIGCONT signal is sent and the process resumes. Doing this shows that SIGTSTP is caught by the process but when the process tries to send that signal to itself using kill (to really suspend itself), the kernel immediately sends SIGCONT to resume the process. The kernel will not let the process be job-control stopped. This handling of the signal by SVR4 and 4.3+BSD is less drastic than sending SIGKILL.

When we use pty to run a job-control shell, the jobs invoked by this new shell are never members of an orphaned process group because the job-control shell always belongs to the same session. In that case the Control-Z that we type is sent to the process invoked by the shell, not to the shell itself.

The only way to avoid this inability of the process invoked by pty to handle job-control signals is to add yet another command-line flag to pty telling it to recognize the job control suspend character itself (in the pty child) instead of letting the character get all the way through to the other line discipline.

### Watching the Output of Long Running Programs

Another example of job-control interaction with the pty program is with the example in Figure 19.6. If we run the program that generates output slowly as

```
pty slowout > file.out &
```

the pty process is stopped immediately when the child tries to read from its standard input (the terminal). This is because the job is a background job and gets job-control stopped when it tries to access the terminal. If we redirect standard input so that pty doesn't try to read from the terminal, as in

```
pty slowout < /dev/null > file.out &
```

then the pty program stops immediately because it reads an end of file on its standard input and terminates. The solution for this problem is the -i option, which says to ignore an end of file on the standard input:

```
pty -i slowout < /dev/null > file.out &
```

This flag causes the pty child in Program 19.5 to terminate when the end of file is encountered, but the child doesn't tell the parent to terminate. Instead the parent continues copying the pty slave output to standard output (the file file.out in the example).

### script Program

Using the pty program we can implement the BSD script(1) program as the following shell script.

```
#!/bin/sh
pty "${SHELL:-/bin/sh}" | tee typescript
```

Once we run this shell script we can execute the ps command to see all the process relationships. Figure 19.8 details these relationships.

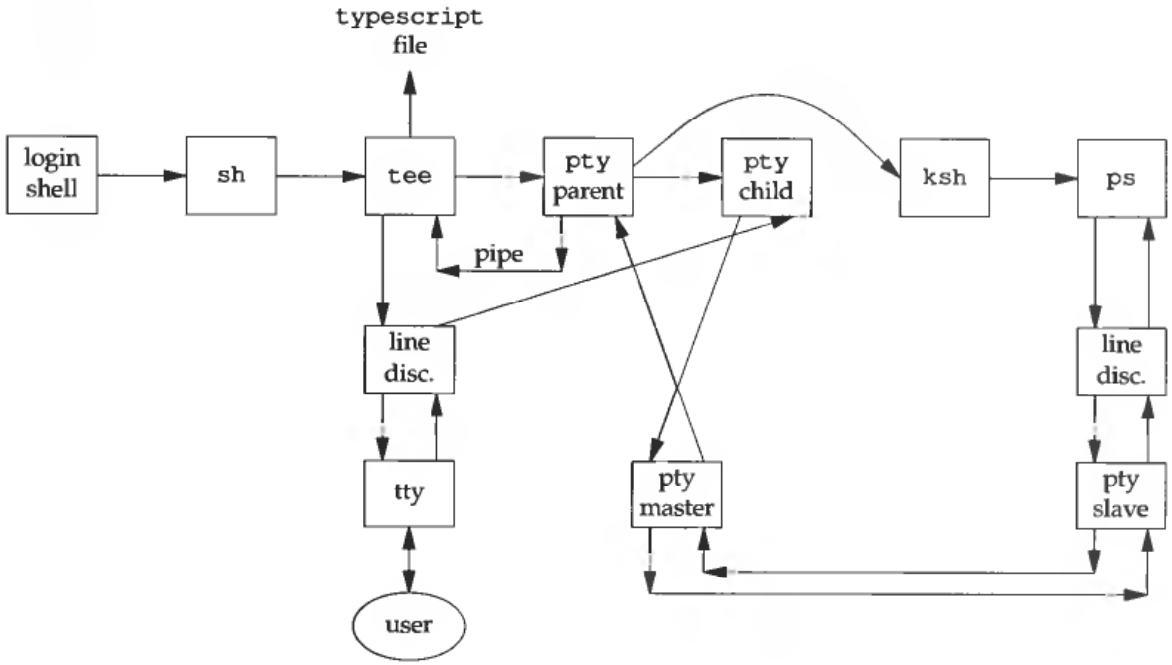


**Figure 19.8** Arrangement of process for `script` shell script.

In this example we assume that the SHELL variable is the KornShell (probably /bin/ksh). As we mentioned earlier, `script` only copies what is output by the new shell (and any processes that it invokes) but since the line discipline module above the pty slave normally has echo enabled, most of what we type also gets written to the `typescript` file.

## Running Coprocesses

In Program 14.9 we couldn't have the coprocess use the standard I/O functions because they set the standard input and standard output fully buffered, since the two descriptors do not refer to a terminal. If we run the coprocess under `pty` by replacing the line

```
if (execl("./add2", "add2", (char *) 0) < 0)
```

with

```
if (execl("./pty", "pty", "-e", "add2", (char *) 0) < 0)
```

the program now works, even if the coprocess uses standard I/O.

Figure 19.9 shows the arrangement of processes when we run the coprocess with a pseudo terminal as its input and output. The box labeled "driving program" is Program 14.9 with the `execl` changed as described previously. This figure is an expansion of Figure 19.5 showing all the process connections and data flow.

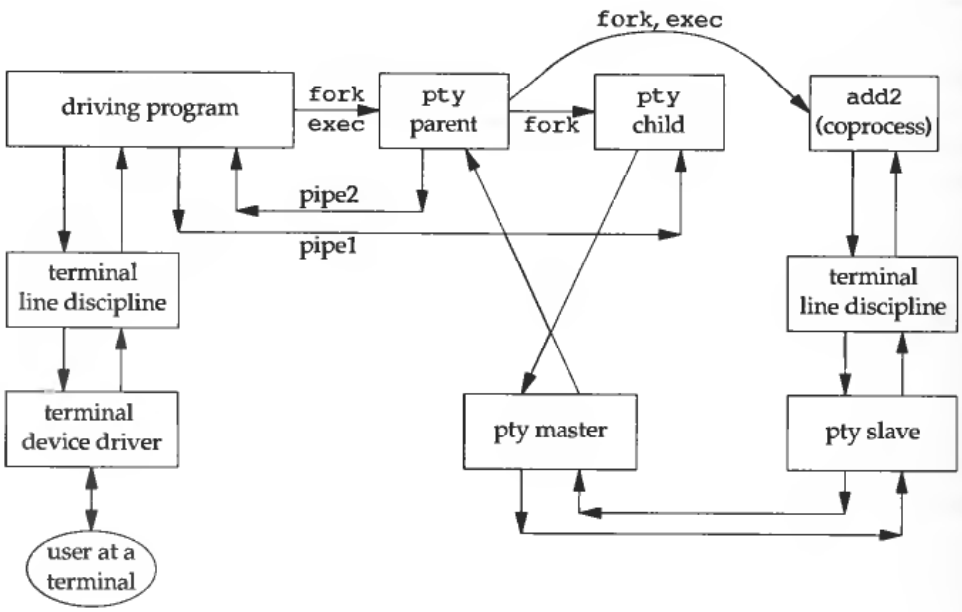


**Figure 19.9** Running a coprocess with a pseudo terminal as its input and output.

This example shows the need for the `-e` (no echo) option for the `pty` program. `pty` is not running interactively because its standard input is not connected to a terminal. In Program 19.4 the `interactive` flag defaults to false since the call to `isatty` returns false. This means that the line discipline above the actual terminal remains in a canonical mode with echo enabled. By specifying the `-e` option we turn off echo in the line discipline module above the pty slave. If we don't do this, everything we type is echoed twice—by both line discipline modules.

We also have the `-e` option turn off the `ONLCR` flag in the `termios` structure to prevent all the output from the coprocess from being terminated with a carriage return and a newline.

Testing this example on different systems showed another problem that we alluded to in Section 12.8 when we described the `readn` and `writen` functions. The amount of data returned by a `read`, when the descriptor refers to something other than an ordinary disk file, can differ between implementations. This coprocess example using `pty` gave unexpected results that were tracked down to the `read` function on the pipe in Program 14.9 returning less than a line. The solution was to not use Program 14.9 but to use the version of this program from Exercise 14.5 that was modified to use the standard I/O library, with the standard I/O streams for the both pipes set to line buffering. By doing this the `fgets` function does as many `reads` as required to obtain a complete line. The `while` loop in Program 14.9 assumes that each line sent to the coprocess causes one line to be returned.

### Driving Interactive Programs Noninteractively

Although it's tempting to think that pty can run any coprocess, even a coprocess that is interactive, it doesn't work. The problem is that pty just copies everything on its standard input to the pty and everything from the pty to its standard output. It never looks at what it sends or what it gets back.

As an example, we can run the call client from Section 18.7 under pty talking directly to the modem.

```
pty call t2500
```

Doing this provides no benefit over just typing call t2500, but we would like to run the call program from a script, perhaps to fetch the contents of the modem's internal registers. If the file t2500.cmd contains the two lines

```
aatn?
~.
```

the first line prints all the modem's registers and the second line terminates the call program. But if we run this script as

```
pty -i < t2500.cmd call t2500
```

the output isn't what we want. What happens is that the contents of the file t2500.cmd are sent to the modem before it has a chance to say that it's ready. When we run the call program interactively we wait for the modem to say Connected, but the pty program doesn't know to do this. This is why it takes a more sophisticated program than pty, such as expect, to drive an interactive program from a script file.

Even running pty from Program 14.9 as we showed earlier doesn't help, because Program 14.9 assumes that each line that it writes to the pipe generates exactly one line on the other pipe. With an interactive program one line of input may generate many lines of output. Furthermore Program 14.9 always sent a line to the coprocess before reading from it. In the case of the preceding modem example, we want to read from the coprocess (the call program) to receive the line Connected before sending it anything.

There are a few ways to proceed from here to be able to drive an interactive program from a script. We could add a command language and interpreter to pty, but a reasonable command language would probably be 10 times larger than the pty program. Another option is to take a command language and use the pty_fork function to invoke interactive programs. This is what the expect program does.

We'll take a different path and just provide an option (-d) to allow pty to be connected to a driver process for its input and output. The standard output of the driver is pty's standard input and vice versa. This is similar to a coprocess, but on "the other side" of pty. The resulting arrangement of processes is almost identical to Figure 19.9 but in the current scenario pty does the fork and exec of the driver process. Also we'll use a single stream pipe between pty and the driver process, instead of two half-duplex pipes.

Program 19.6 shows the source for the do_driver function that is called by the main function of pty (Program 19.4) when the -d option is specified.

```
#include    <sys/types.h>
#include    <signal.h>
#include    "ourhdr.h"

void
do_driver(char *driver)
{
    pid_t   child;
    int     pipe[2];

        /* create a stream pipe to communicate with the driver */
    if (s_pipe(pipe) < 0)
        err_sys("can't create stream pipe");

    if ( (child = fork()) < 0)
        err_sys("fork error");

    else if (child == 0) {          /* child */
        close(pipe[1]);

                /* stdin for driver */
        if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");

                /* stdout for driver */
        if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        close(pipe[0]);

                /* leave stderr for driver alone */

        execlp(driver, driver, (char *) 0);
        err_sys("execlp error for: %s", driver);
    }

    close(pipe[0]);     /* parent */

    if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
        err_sys("dup2 error to stdin");

    if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("dup2 error to stdout");
    close(pipe[1]);

    /* Parent returns, but with stdin and stdout connected
       to the driver. */
}
```

**Program 19.6** The do_driver function for the pty program.

By writing our own driver program that is invoked by `pty` we can drive interactive programs in any way desired. Even though the driver process has its standard input and standard output connected to `pty`, it can still interact with the user by reading and writing `/dev/tty`. This solution still isn't as general as the `expect` program, but it provides a useful option to `pty` for less than 50 lines of code.

## 19.7 Advanced Features

Pseudo terminals have some additional capabilities that we briefly mention here. These are further documented in AT&T [1990d] and the 4.3+BSD `pty`(4) manual page.

### Packet Mode

Packet mode lets the pty master learn of state changes in the pty slave. Under SVR4 this mode is enabled by pushing the streams module `pckt` onto the pty master side. We showed this optional module in Figure 19.2. Under 4.3+BSD this mode is enabled with an `ioctl` of `TIOCPKT`.

The details of packet mode differ between SVR4 and 4.3+BSD. Under SVR4 the process reading the pty master has to call `getmsg` to fetch the messages from the stream head, because the `pckt` module converts certain events into non-data streams messages. With 4.3+BSD each `read` from the pty master returns a status byte followed by optional data.

Regardless of the implementation details, the purpose of packet mode is to inform the process reading the pty master when the following events occur at the line discipline module above the pty slave: when the read queue is flushed, when the write queue is flushed, whenever output is stopped (e.g., Control-S), whenever output is restarted, whenever XON/XOFF flow control is enabled after being disabled, and whenever XON/XOFF flow control is disabled after being enabled. These events are used, for example, by the `rlogin` client and `rlogind` server.

### Remote Mode

A pty master can set the pty slave into remote mode by issuing an `ioctl` of `TIOCREMOTE`. Although both SVR4 and 4.3+BSD use the same command to enable and disable this feature, under SVR4 the third argument to `ioctl` is an integer while with 4.3+BSD it is a pointer to an integer.

When the pty master sets this mode it is telling the pty slave's line discipline module not to perform any processing of the data that it receives from the pty master, regardless of the canonical/noncanonical flag in the slave's `termios` structure. Remote mode is intended for an application such as a window manager that does its own line editing.

### Window Size Changes

The process above the pty master can issue the `ioctl` of `TIOCSWINSZ` to set the window size of the slave. If the new size differs from the current size, a `SIGWINCH` signal is sent to the foreground process group of the pty slave.

### Signal Generation

The process reading and writing the pty master can send signals to the process group of the pty slave. Under SVR4 this is done with an `ioctl` of `TIOCSIGNAL` with the third argument being the actual signal number. With 4.3+BSD the `ioctl` is `TIOCSIG` and the third argument is a pointer to the integer signal number.

## 19.8 Summary

We started this chapter by examining the code required to open a pseudo terminal under both SVR4 and 4.3+BSD. We then used this code to provide the generic `pty_fork` function that can be used by many different applications. We used this function as the basis for a small program (`pty`), which we then used to explore many of the properties of pseudo terminals.

Pseudo terminals are used daily on most Unix systems to provide network logins. We've examined other uses for pseudo terminals, from the `script` program to driving interactive programs from a batch script.

## Exercises

**19.1** When we remotely log in to a BSD system using either `telnet` or `rlogin`, the ownership of the pty slave and its permissions are set, as we described in Section 19.3.2. How does this happen?

**19.2** Modify the 4.3+BSD function `ptys_open` to invoke a set-user-ID program to change the ownership and protection of the pty slave device (similar to what the SVR4 `grantpt` function does).

**19.3** Use the `pty` program to determine the values used by your system to initialize a slave pty's `termios` structure and `winsize` structure.

**19.4** Recode the `loop` function (Program 19.5) as a single process using either `select` or `poll`.

**19.5** In the child process after `pty_fork` returns, standard input, standard output, and standard error are all open for read–write. Can you change standard input to be read-only and the other two to be write-only?

**19.6** In Figure 19.7 identify which process groups are foreground and which are background, and identify the session leaders.

**19.7** In Figure 19.7 in what order do the processes terminate when we type our end-of-file character? Verify this with process accounting, if possible.

**19.8** The `script(1)` program normally adds a line to the beginning of the output file with the starting time, and another line at the end of the output file with the ending time. Add these features to the simple shell script that we showed.

**19.9** Explain why the contents of the file `data` are output to the terminal in the following example, when the program `ttyname` only generates output and never reads its input.

```
$ cat data                      a file with two lines
hello,
world
$ pty -i < data ttyname         -i says ignore eof on stdin
hello,                          where did these two lines come from?
world
fd 0: /dev/ttyp5                we expect these three lines from ttyname
fd 1: /dev/ttyp5
fd 2: /dev/ttyp5
```

**19.10** Write a program that calls `pty_fork` and have the child `exec` another program that you must write. The new program that the child `exec`s must catch `SIGTERM` and `SIGWINCH`. When it catches a signal it should print that it did, and for the latter signal it should also print the terminal's window size. Then have the parent process send the `SIGTERM` signal to the process group of the pty slave with the `ioctl` we described in Section 19.7. Read back from the slave to verify that the signal was caught. Follow this with the parent setting the window size of the pty slave and read back the slave's output again. Have the parent `exit` and determine if the slave process also terminates, and if so, how does it terminate?

# Appendix A

# Function Prototypes

This appendix contains the function prototypes for the standard Unix, POSIX, and ANSI C functions described in the text. Often we want to see just the arguments to a function ("which argument is the file pointer for `fgets`?") or just the return value ("does `sprintf` return a pointer or a count?").

These prototypes also show which headers need to be included to obtain the definitions of any special constants, and to obtain the ANSI C function prototype to help detect any compile-time errors.

The page number reference with each prototype gives the page containing the actual prototype for the function. That page should be consulted for additional information on the function.

```
void      _exit(int status);
                    <unistd.h>                          p. 162
                    This function never returns

void      abort(void);
                    <stdlib.h>                          p. 309
                    This function never returns

int       access(const char *pathname, int mode);
                    <unistd.h>                          p. 82
                    mode: R_OK, W_OK, X_OK, F_OK
                    Returns: 0 if OK, -1 on error

unsigned
int       alarm(unsigned int seconds);
                    <unistd.h>                          p. 285
                    Returns: 0 or #seconds until previously set alarm
```

```
char    *asctime(const struct tm *tmptr);
                    <time.h>                          p. 157
                    Returns: pointer to null terminated string

int     atexit(void (*func)(void));
                    <stdlib.h>                        p. 163
                    Returns: 0 if OK, nonzero on error

void    *calloc(size_t nobj, size_t size);
                    <stdlib.h>                        p. 170
                    Returns: nonnull pointer if OK, NULL on error

speed_t cfgetispeed(const struct termios *termptr);
                    <termios.h>                       p. 343
                    Returns: baud rate value

speed_t cfgetospeed(const struct termios *termptr);
                    <termios.h>                       p. 343
                    Returns: baud rate value

int     cfsetispeed(struct termios *termptr, speed_t speed);
                    <termios.h>                       p. 343
                    Returns: 0 if OK, –1 on error

int     cfsetospeed(struct termios *termptr, speed_t speed);
                    <termios.h>                       p. 343
                    Returns: 0 if OK, –1 on error

int     chdir(const char *pathname);
                    <unistd.h>                        p. 112
                    Returns: 0 if OK, –1 on error

int     chmod(const char *pathname, mode_t mode);
                    <sys/types.h>                     p. 85
                    <sys/stat.h>
                    mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, –1 on error

int     chown(const char *pathname, uid_t owner, gid_t group);
                    <sys/types.h>                     p. 89
                    <unistd.h>
                    Returns: 0 if OK, –1 on error

void    clearerr(FILE *fp);
                    <stdio.h>                         p. 129

int     close(int filedes);
                    <unistd.h>                        p. 51
                    Returns: 0 if OK, –1 on error

int     closedir(DIR *dp);
                    <sys/types.h>                     p. 107
                    <dirent.h>
                    Returns: 0 if OK, –1 on error

void    closelog(void);
                    <syslog.h>                        p. 422
```

```
int     creat(const char *pathname, mode_t mode);
                        <sys/types.h>                   p. 50
                        <sys/stat.h>
                        <fcntl.h>
                        mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                        Returns: file descriptor opened for write-only if OK, –1 on error

char    *ctermid(char *ptr);
                        <stdio.h>                       p. 345
                        Returns: pathname of controlling terminal

char    *ctime(const time_t *calptr);
                        <time.h>                        p. 157
                        Returns: pointer to null terminated string

int     dup(int filedes);
                        <unistd.h>                      p. 61
                        Returns: new file descriptor if OK, –1 on error

int     dup2(int filedes, int filedes2);
                        <unistd.h>                      p. 61
                        Returns: new file descriptor if OK, –1 on error

void    endgrent(void);
                        <sys/types.h>                   p. 150
                        <grp.h>

void    endpwent(void);
                        <sys/types.h>                   p. 147
                        <pwd.h>

int     execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );
                        <unistd.h>                      p. 207
                        Returns: –1 on error, no return on success

int     execle(const char *pathname, const char *arg0, ... /* (char *) 0,
              char *const envp[] */ );
                        <unistd.h>                      p. 207
                        Returns: –1 on error, no return on success

int     execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );
                        <unistd.h>                      p. 207
                        Returns: –1 on error, no return on success

int     execv(const char *pathname, char *const argv[]);
                        <unistd.h>                      p. 207
                        Returns: –1 on error, no return on success

int     execve(const char *pathname, char *const argv[], char *const envp[]);
                        <unistd.h>                      p. 207
                        Returns: –1 on error, no return on success

int     execvp(const char *filename, char *const argv[]);
                        <unistd.h>                      p. 207
                        Returns: –1 on error, no return on success
```

```
void     exit(int status);
                    <stdlib.h>                                  p. 162
                    This function never returns

int      fchdir(int filedes);
                    <unistd.h>                                  p. 112
                    Returns: 0 if OK, –1 on error

int      fchmod(int filedes, mode_t mode);
                    <sys/types.h>                               p. 85
                    <sys/stat.h>
                    mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, –1 on error

int      fchown(int filedes, uid_t owner, gid_t group);
                    <sys/types.h>                               p. 89
                    <unistd.h>
                    Returns: 0 if OK, –1 on error

int      fclose(FILE *fp);
                    <stdio.h>                                   p. 127
                    Returns: 0 if OK, EOF on error

int      fcntl(int filedes, int cmd, ... /* int arg */ );
                    <sys/types.h>                               p. 63
                    <unistd.h>
                    <fcntl.h>
                    cmd: F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL
                    Returns: depends on cmd if OK, –1 on error

FILE    *fdopen(int filedes, const char *type);
                    <stdio.h>                                   p. 125
                    type: "r", "w", "a", "r+", "w+", "a+",
                    Returns: file pointer if OK, NULL on error

int      feof(FILE *fp);
                    <stdio.h>                                   p. 129
                    Returns: nonzero (true) if end of file on stream, 0 (false) otherwise

int      ferror(FILE *fp);
                    <stdio.h>                                   p. 129
                    Returns: nonzero (true) if error on stream, 0 (false) otherwise

int      fflush(FILE *fp);
                    <stdio.h>                                   p. 125
                    Returns: 0 if OK, EOF on error

int      fgetc(FILE *fp);
                    <stdio.h>                                   p. 128
                    Returns: next character if OK, EOF on end of file or error

int      fgetpos(FILE *fp, fpos_t *pos);
                    <stdio.h>                                   p. 136
                    Returns: 0 if OK, nonzero on error
```

```
char    *fgets(char *buf, int n, FILE *fp);
                        <stdio.h>                          p. 130
                        Returns: buf if OK, NULL on end of file or error

int      fileno(FILE *fp);
                        <stdio.h>                          p. 138
                        Returns: file descriptor associated with the stream

FILE    *fopen(const char *pathname, const char *type);
                        <stdio.h>                          p. 125
                        type: "r", "w", "a", "r+", "w+", "a+",
                        Returns: file pointer if OK, NULL on error

pid_t    fork(void);
                        <sys/types.h>                      p. 188
                        <unistd.h>
                        Returns: 0 in child, process ID of child in parent, -1 on error

long     fpathconf(int filedes, int name);
                        <unistd.h>                         p. 35
                        name: _PC_CHOWN_RESTRICTED, _PC_LINK_MAX, _PC_MAX_CANON,
                              _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
                              _PC_PATH_MAX, _PC_PIPE_BUF, _PC_VDISABLE
                        Returns: corresponding value if OK, -1 on error

int      fprintf(FILE *fp, const char *format, ...);
                        <stdio.h>                          p. 136
                        Returns: #characters output if OK, negative value if output error

int      fputc(int c, FILE *fp);
                        <stdio.h>                          p. 130
                        Returns: c if OK, EOF on error

int      fputs(const char *str, FILE *fp);
                        <stdio.h>                          p. 131
                        Returns: nonnegative value if OK, EOF on error

size_t   fread(void *ptr, size_t size, size_t nobj, FILE *fp);
                        <stdio.h>                          p. 134
                        Returns: number of objects read

void     free(void *ptr);
                        <stdlib.h>                         p. 170

FILE    *freopen(const char *pathname, const char *type, FILE *fp);
                        <stdio.h>                          p. 125
                        type: "r", "w", "a", "r+", "w+", "a+",
                        Returns: file pointer if OK, NULL on error

int      fscanf(FILE *fp, const char *format, ...);
                        <stdio.h>                          p. 137
                        Returns: #input items assigned, EOF if input error or EOF before any conversion

int      fseek(FILE *fp, long offset, int whence);
                        <stdio.h>                          p. 135
                        whence: SEEK_SET, SEEK_CUR, SEEK_END
                        Returns: 0 if OK, nonzero on error
```

```
int      fsetpos(FILE *fp, const fpos_t *pos);
                 <stdio.h>                                  p. 136
                 Returns: 0 if OK, nonzero on error

int      fstat(int filedes, struct stat *buf);
                 <sys/types.h>                              p. 73
                 <sys/stat.h>
                 Returns: 0 if OK, -1 on error

int      fsync(int filedes);
                 <unistd.h>                                 p. 116
                 Returns: 0 if OK, -1 on error

long     ftell(FILE *fp);
                 <stdio.h>                                  p. 135
                 Returns: current file position indicator if OK, -1L on error

int      ftruncate(int filedes, off_t length);
                 <sys/types.h>                              p. 92
                 <unistd.h>
                 Returns: 0 if OK, -1 on error

size_t   fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
                 <stdio.h>                                  p. 134
                 Returns: number of objects written

int      getc(FILE *fp);
                 <stdio.h>                                  p. 128
                 Returns: next character if OK, EOF on end of file or error

int      getchar(void);
                 <stdio.h>                                  p. 128
                 Returns: next character if OK, EOF on end of file or error

char    *getcwd(char *buf, size_t size);
                 <unistd.h>                                 p. 113
                 Returns: buf if OK, NULL on error

gid_t    getegid(void);
                 <sys/types.h>                              p. 188
                 <unistd.h>
                 Returns: effective group ID of calling process

char    *getenv(const char *name);
                 <stdlib.h>                                 p. 172
                 Returns: pointer to value associated with name, NULL if not found

uid_t    geteuid(void);
                 <sys/types.h>                              p. 188
                 <unistd.h>
                 Returns: effective user ID of calling process

gid_t    getgid(void);
                 <sys/types.h>                              p. 188
                 <unistd.h>
                 Returns: real group ID of calling process
```

```
struct
group   *getgrent(void);
                        <sys/types.h>                   p. 150
                        <grp.h>
                        Returns: pointer if OK, NULL on error or end of file

struct
group   *getgrgid(gid_t gid);
                        <sys/types.h>                   p. 150
                        <grp.h>
                        Returns: pointer if OK, NULL on error

struct
group   *getgrnam(const char *name);
                        <sys/types.h>                   p. 150
                        <grp.h>
                        Returns: pointer if OK, NULL on error

int      getgroups(int gidsetsize, gid_t grouplist[]);
                        <sys/types.h>                   p. 151
                        <unistd.h>
                        Returns: number of supplementary group IDs if OK, –1 on error

int      gethostname(char *name, int namelen);
                        <unistd.h>                      p. 154
                        Returns: 0 if OK, –1 on error

char    *getlogin(void);
                        <unistd.h>                      p. 232
                        Returns: pointer to string giving login name if OK, NULL on error

int      getmsg(int filedes, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagptr);
                        <stropts.h>                     p. 392
                        *flagptr: 0, RS_HIPRI
                        Returns: nonnegative value if OK, –1 on error

pid_t    getpgrp(void);
                        <sys/types.h>                   p. 243
                        <unistd.h>
                        Returns: process group ID of calling process

pid_t    getpid(void);
                        <sys/types.h>                   p. 188
                        <unistd.h>
                        Returns: process ID of calling process

int      getpmsg(int filedes, struct strbuf *ctlptr, struct strbuf *dataptr, int *bandptr,
                 int *flagptr);
                        <stropts.h>                     p. 392
                        *flagptr: 0, MSG_HIPRI, MSG_BAND, MSG_ANY
                        Returns: nonnegative value if OK, –1 on error

pid_t    getppid(void);
                        <sys/types.h>                   p. 188
                        <unistd.h>
                        Returns: parent process ID of calling process
```

```
struct
passwd  *getpwent(void);
                        <sys/types.h>                    p. 147
                        <pwd.h>
                        Returns: pointer if OK, NULL on error or end of file

struct
passwd  *getpwnam(const char *name);
                        <sys/types.h>                    p. 147
                        <pwd.h>
                        Returns: pointer if OK, NULL on error

struct
passwd  *getpwuid(uid_t uid);
                        <sys/types.h>                    p. 147
                        <pwd.h>
                        Returns: pointer if OK, NULL on error

int      getrlimit(int resource, struct rlimit *rlptr);
                        <sys/time.h>                     p. 180
                        <sys/resource.h>
                        Returns: 0 if OK, nonzero on error

char    *gets(char *buf);
                        <stdio.h>                        p. 130
                        Returns: buf if OK, NULL on end of file or error

uid_t    getuid(void);
                        <sys/types.h>                    p. 188
                        <unistd.h>
                        Returns: real user ID of calling process

struct
tm      *gmtime(const time_t *calptr);
                        <time.h>                         p. 156
                        Returns: pointer to broken-down time

int      initgroups(const char *username, gid_t basegid);
                        <sys/types.h>                    p. 151
                        <unistd.h>
                        Returns: 0 if OK, –1 on error

int      ioctl(int filedes, int request, ...);
                        <unistd.h>      /* SVR4 */       p. 68
                        <sys/ioctl.h>   /* 4.3+BSD */
                        Returns: –1 on error, something else if OK

int      isastream(int filedes);                         p. 388
                        Returns: 1 (true) if streams device, 0 (false) otherwise

int      isatty(int filedes);
                        <unistd.h>                       p. 346
                        Returns: 1 (true) if terminal device, 0 (false) otherwise
```

```
int       kill(pid_t pid, int signo);
                        <sys/types.h>                    p. 284
                        <signal.h>
                        Returns: 0 if OK, –1 on error

int       lchown(const char *pathname, uid_t owner, gid_t group);
                        <sys/types.h>                    p. 89
                        <unistd.h>
                        Returns: 0 if OK, –1 on error

int       link(const char *existingpath, const char *newpath);
                        <unistd.h>                       p. 95
                        Returns: 0 if OK, –1 on error

struct
tm       *localtime(const time_t *calptr);
                        <time.h>                         p. 156
                        Returns: pointer to broken-down time

void      longjmp(jmp_buf env, int val);
                        <setjmp.h>                       p. 176
                        This function never returns

off_t     lseek(int filedes, off_t offset, int whence);
                        <sys/types.h>                    p. 51
                        <unistd.h>
                        whence: SEEK_SET, SEEK_CUR, SEEK_END
                        Returns: new file offset if OK, –1 on error

int       lstat(const char *pathname, struct stat *buf);
                        <sys/types.h>                    p. 73
                        <sys/stat.h>
                        Returns: 0 if OK, –1 on error

void     *malloc(size_t size);
                        <stdlib.h>                       p. 170
                        Returns: nonnull pointer if OK, NULL on error

int       mkdir(const char *pathname, mode_t mode);
                        <sys/types.h>                    p. 106
                        <sys/stat.h>
                        mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                        Returns: 0 if OK, –1 on error

int       mkfifo(const char *pathname, mode_t mode);
                        <sys/types.h>                    p. 445
                        <sys/stat.h>
                        mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                        Returns: 0 if OK, –1 on error

time_t    mktime(struct tm *tmptr);
                        <time.h>                         p. 157
                        Returns: calendar time if OK, –1 on error
```

```
caddr_t  mmap(caddr_t addr, size_t len, int prot, int flag, int filedes, off_t off);
                        <sys/types.h>                                   p. 407
                        <sys/mman.h>
                        prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
                        flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
                        Returns: starting address of mapped region if OK, -1 on error

int      msgctl(int msqid, int cmd, struct msqid_ds *buf);
                        <sys/types.h>                                   p. 454
                        <sys/ipc.h>
                        <sys/msg.h>
                        cmd: IPC_STAT, IPC_SET, IPC_RMID
                        Returns: 0 if OK, -1 on error

int      msgget(key_t key, int flag);
                        <sys/types.h>                                   p. 454
                        <sys/ipc.h>
                        <sys/msg.h>
                        flag: 0, IPC_CREAT, IPC_EXCL
                        Returns: message queue ID if OK, -1 on error

int      msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                        <sys/types.h>                                   p. 456
                        <sys/ipc.h>
                        <sys/msg.h>
                        flag: 0, IPC_NOWAIT, MSG_NOERROR
                        Returns: size of data portion of message if OK, -1 on error

int      msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
                        <sys/types.h>                                   p. 455
                        <sys/ipc.h>
                        <sys/msg.h>
                        flag: 0, IPC_NOWAIT
                        Returns: 0 if OK, -1 on error

int      munmap(caddr_t addr, size_t len);
                        <sys/types.h>                                   p. 411
                        <sys/mman.h>
                        Returns: 0 if OK, -1 on error

int      open(const char *pathname, int oflag, ... /* , mode_t mode */ );
                        <sys/types.h>                                   p. 48
                        <sys/stat.h>
                        <fcntl.h>
                        oflag: O_RDONLY, O_WRONLY, O_RDWR;
                               O_APPEND, O_CREAT, O_EXCL, O_TRUNC,
                               O_NOCTTY, O_NONBLOCK, O_SYNC
                        mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                        Returns: file descriptor if OK, -1 on error

DIR     *opendir(const char *pathname);
                        <sys/types.h>                                   p. 107
                        <dirent.h>
                        Returns: pointer if OK, NULL on error
```

```
void     openlog(char *ident, int option, int facility);
                 <syslog.h>                                   p. 422
                 option: LOG_CONS, LOG_NDELAY, LOG_PERROR, LOG_PID
                 facility: LOG_AUTH, LOG_CRON, LOG_DAEMON, LOG_KERN,
                        LOG_LOCAL[0-7], LOG_LPR, LOG_MAIL, LOG_NEWS,
                        LOG_SYSLOG, LOG_USER, LOG_UUCP

long     pathconf(const char *pathname, int name);
                 <unistd.h>                                   p. 35
                 name: _PC_CHOWN_RESTRICTED, _PC_LINK_MAX, _PC_MAX_CANON,
                       _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
                       _PC_PATH_MAX, _PC_PIPE_BUF, _PC_VDISABLE
                 Returns: corresponding value if OK, –1 on error

int      pause(void);
                 <unistd.h>                                   p. 285
                 Returns: –1 with errno set to EINTR

int      pclose(FILE *fp);
                 <stdio.h>                                    p. 435
                 Returns: termination status of cmdstring, or –1 on error

void     perror(const char *msg);
                 <stdio.h>                                    p. 15

int      pipe(int filedes[2]);
                 <unistd.h>                                   p. 428
                 Returns: 0 if OK, –1 on error

int      poll(struct pollfd fdarray[], unsigned long nfds, int timeout);
                 <stropts.h>                                  p. 400
                 <poll.h>
                 Returns: count of ready descriptors, 0 on timeout, –1 on error

FILE    *popen(const char *cmdstring, const char *type);
                 <stdio.h>                                    p. 435
                 type: "r", "w"
                 Returns: file pointer if OK, NULL on error

int      printf(const char *format, ...);
                 <stdio.h>                                    p. 136
                 Returns: # characters output if OK, negative value if output error

void     psignal(int signo, const char *msg);
                 <signal.h>                                   p. 322

int      putc(int c, FILE *fp);
                 <stdio.h>                                    p. 130
                 Returns: c if OK, EOF on error

int      putchar(int c);
                 <stdio.h>                                    p. 130
                 Returns: c if OK, EOF on error

int      putenv(const char *str);
                 <stdlib.h>                                   p. 173
                 Returns: 0 if OK, nonzero on error
```

```
int      putmsg(int filedes, const struct strbuf *ctlptr, const struct strbuf *dataptr,
                int flag);
                        <stropts.h>                          p. 386
                        flag: 0, RS_HIPRI
                        Returns: 0 if OK, -1 on error

int      putpmsg(int filedes, const struct strbuf *ctlptr, const struct strbuf *dataptr,
                 int band, int flag);
                        <stropts.h>                          p. 386
                        flag: 0, MSG_HIPRI, MSG_BAND
                        Returns: 0 if OK, -1 on error

int      puts(const char *str);
                        <stdio.h>                            p. 131
                        Returns: nonnegative value if OK, EOF on error

int      raise(int signo);
                        <sys/types.h>                        p. 284
                        <signal.h>
                        Returns: 0 if OK, -1 on error

ssize_t  read(int filedes, void *buff, size_t nbytes);
                        <unistd.h>                           p. 54
                        Returns: #bytes read if OK, 0 if end of file, -1 on error

struct
dirent   *readdir(DIR *dp);
                        <sys/types.h>                        p. 107
                        <dirent.h>
                        Returns: pointer if OK, NULL on error

int      readlink(const char *pathname, char *buf, int bufsize);
                        <unistd.h>                           p. 102
                        Returns: #bytes read if OK, -1 on error

ssize_t  readv(int filedes, const struct iovec iov[], int iovcnt);
                        <sys/types.h>                        p. 404
                        <sys/uio.h>
                        Returns: #bytes read if OK, -1 on error

void     *realloc(void *ptr, size_t newsize);
                        <stdlib.h>                           p. 170
                        Returns: nonnull pointer if OK, NULL on error

int      remove(const char *pathname);
                        <stdio.h>                            p. 98
                        Returns: 0 if OK, -1 on error

int      rename(const char *oldname, const char *newname);
                        <stdio.h>                            p. 98
                        Returns: 0 if OK, -1 on error

void     rewind(FILE *fp);
                        <stdio.h>                            p. 135
```

```
void     rewinddir(DIR *dp);
                    <sys/types.h>                    p. 107
                    <dirent.h>

int      rmdir(const char *pathname);
                    <unistd.h>                       p. 107
                    Returns: 0 if OK, –1 on error

int      scanf(const char *format, ...);
                    <stdio.h>                        p. 137
                    Returns: #input items assigned, EOF if input error or EOF before any conversion

int      select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
                struct timeval *tvptr);
                    <sys/types.h>                    p. 397
                    <sys/time.h>
                    <unistd.h>
                    Returns: count of ready descriptors, 0 on timeout, –1 on error
                    FD_ZERO(fd_set *fdset);
                    FD_SET(int filedes, fd_set *fdset);
                    FD_CLR(int filedes, fd_set *fdset);
                    FD_ISSET(int filedes, fd_set *fdset);

int      semctl(int semid, int semnum, int cmd, union semun arg);
                    <sys/types.h>                    p. 459
                    <sys/ipc.h>
                    <sys/sem.h>
                    cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT, GETZCNT,
                         GETVAL, SETVAL, GETALL, SETALL
                    Returns: (depends on command)

int      semget(key_t key, int nsems, int flag);
                    <sys/types.h>                    p. 459
                    <sys/ipc.h>
                    <sys/sem.h>
                    flag: 0, IPC_CREAT, IPC_EXCL
                    Returns: semaphore ID if OK, –1 on error

int      semop(int semid, struct sembuf semoparray[], size_t nops);
                    <sys/types.h>                    p. 461
                    <sys/ipc.h>
                    <sys/sem.h>
                    Returns: 0 if OK, –1 on error

void     setbuf(FILE *fp, char *buf);
                    <stdio.h>                        p. 124

int      setegid(gid_t gid);
                    <sys/types.h>                    p. 216
                    <unistd.h>
                    Returns: 0 if OK, –1 on error

int      setenv(const char *name, const char *value, int rewrite);
                    <stdlib.h>                       p. 173
                    Returns: 0 if OK, nonzero on error
```

```
int     seteuid(uid_t uid);
                <sys/types.h>                                   p. 216
                <unistd.h>
                Returns: 0 if OK, –1 on error

int     setgid(gid_t gid);
                <sys/types.h>                                   p. 213
                <unistd.h>
                Returns: 0 if OK, –1 on error

void    setgrent(void);
                <sys/types.h>                                   p. 150
                <grp.h>

int     setgroups(int ngroups, const gid_t grouplist[]);
                <sys/types.h>                                   p. 151
                <unistd.h>
                Returns: 0 if OK, –1 on error

int     setjmp(jmp_buf env);
                <setjmp.h>                                      p. 176
                Returns: 0 if called directly, nonzero if returning from a call to longjmp

int     setpgid(pid_t pid, pid_t pgid);
                <sys/types.h>                                   p. 244
                <unistd.h>
                Returns: 0 if OK, –1 on error

void    setpwent(void);
                <sys/types.h>                                   p. 147
                <pwd.h>

int     setregid(gid_t rgid, gid_t egid);
                <sys/types.h>                                   p. 215
                <unistd.h>
                Returns: 0 if OK, –1 on error

int     setreuid(uid_t ruid, uid_t euid);
                <sys/types.h>                                   p. 215
                <unistd.h>
                Returns: 0 if OK, –1 on error

int     setrlimit(int resource, const struct rlimit *rlptr);
                <sys/time.h>                                    p. 180
                <sys/resource.h>
                Returns: 0 if OK, nonzero on error

pid_t   setsid(void);
                <sys/types.h>                                   p. 245
                <unistd.h>
                Returns: process group ID if OK, –1 on error

int     setuid(uid_t uid);
                <sys/types.h>                                   p. 213
                <unistd.h>
                Returns: 0 if OK, –1 on error
```

```
int       setvbuf(FILE *fp, char *buf, int mode, size_t size);
                  <stdio.h>                                    p. 124
                  mode: _IOFBF, _IOLBF, _IONBF
                  Returns: 0 if OK, nonzero on error

void     *shmat(int shmid, void *addr, int flag);
                  <sys/types.h>                                p. 465
                  <sys/ipc.h>
                  <sys/shm.h>
                  flag: 0, SHM_RND, SHM_RDONLY
                  Returns: pointer to shared memory segment if OK, -1 on error

int       shmctl(int shmid, int cmd, struct shmid_ds *buf);
                  <sys/types.h>                                p. 465
                  <sys/ipc.h>
                  <sys/shm.h>
                  cmd: IPC_STAT, IPC_SET, IPC_RMID,
                       SHM_LOCK, SHM_UNLOCK
                  Returns: 0 if OK, -1 on error

int       shmdt(void *addr);
                  <sys/types.h>                                p. 466
                  <sys/ipc.h>
                  <sys/shm.h>
                  Returns: 0 if OK, -1 on error

int       shmget(key_t key, int size, int flag);
                  <sys/types.h>                                p. 464
                  <sys/ipc.h>
                  <sys/shm.h>
                  flag: 0, IPC_CREAT, IPC_EXCL
                  Returns: shared memory ID if OK, -1 on error

int       sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
                  <signal.h>                                   p. 296
                  Returns: 0 if OK, -1 on error

int       sigaddset(sigset_t *set, int signo);
                  <signal.h>                                   p. 291
                  Returns: 0 if OK, -1 on error

int       sigdelset(sigset_t *set, int signo);
                  <signal.h>                                   p. 291
                  Returns: 0 if OK, -1 on error

int       sigemptyset(sigset_t *set);
                  <signal.h>                                   p. 291
                  Returns: 0 if OK, -1 on error

int       sigfillset(sigset_t *set);
                  <signal.h>                                   p. 291
                  Returns: 0 if OK, -1 on error

int       sigismember(const sigset_t *set, int signo);
                  <signal.h>                                   p. 291
                  Returns: 1 if true, 0 if false
```

```
void      siglongjmp(sigjmp_buf env, int val);
                        <setjmp.h>                          p. 300
                        This function never returns

void    (*signal(int signo, void (*func)(int)))(int);
                        <signal.h>                          p. 270
                        Returns: previous disposition of signal, SIG_ERR on error

int       sigpending(sigset_t *set);
                        <signal.h>                          p. 293
                        Returns: 0 if OK, –1 on error

int       sigprocmask(int how, const sigset_t *set, sigset_t *oset);
                        <signal.h>                          p. 293
                        how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
                        Returns: 0 if OK, –1 on error

int       sigsetjmp(sigjmp_buf env, int savemask);
                        <setjmp.h>                          p. 300
                        Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

int       sigsuspend(const sigset_t *sigmask);
                        <signal.h>                          p. 303
                        Returns: –1 with errno set to EINTR

unsigned
int       sleep(unsigned int seconds);
                        <unistd.h>                          p. 317
                        Returns: 0 or number of unslept seconds

int       sprintf(char *buf, const char *format, ...);
                        <stdio.h>                           p. 136
                        Returns: #characters stored in array

int       sscanf(const char *buf, const char *format, ...);
                        <stdio.h>                           p. 137
                        Returns: #input items assigned, EOF if input error or EOF before any conversion

int       stat(const char *pathname, struct stat *buf);
                        <sys/types.h>                       p. 73
                        <sys/stat.h>
                        Returns: 0 if OK, –1 on error

char    *strerror(int errnum);
                        <string.h>                          p. 14
                        Returns: pointer to message string

size_t    strftime(char *buf, size_t maxsize, const char *format, const struct tm *tmptr);
                        <time.h>                            p. 157
                        Returns: #characters stored in array if room, else 0

int       symlink(const char *actualpath, const char *sympath);
                        <unistd.h>                          p. 102
                        Returns: 0 if OK, –1 on error

void      sync(void);
                        <unistd.h>                          p. 116
```

```
long      sysconf(int name);
                  <unistd.h>                                  p. 35
                  name: _SC_ARG_MAX, _SC_CHILD_MAX, _SC_CLK_TCK,
                        _SC_NGROUPS_MAX, _SC_OPEN_MAX, _SC_PASS_MAX,
                        _SC_STREAM_MAX, _SC_TZNAME_MAX, _SC_JOB_CONTROL,
                        _SC_SAVED_IDS, _SC_VERSION, _SC_XOPEN_VERSION
                  Returns: corresponding value if OK, –1 on error

void      syslog(int priority, char *format, ...);
                  <syslog.h>                                  p. 422

int       system(const char *cmdstring);
                  <stdlib.h>                                  p. 222
                  Returns: termination status of shell

int       tcdrain(int filedes);
                  <termios.h>                                 p. 344
                  Returns: 0 if OK, –1 on error

int       tcflow(int filedes, int action);
                  <termios.h>                                 p. 344
                  action: TCOOFF, TCOON, TCIOFF, TCION
                  Returns: 0 if OK, –1 on error

int       tcflush(int filedes, int queue);
                  <termios.h>                                 p. 344
                  queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
                  Returns: 0 if OK, –1 on error

int       tcgetattr(int filedes, struct termios *termptr);
                  <termios.h>                                 p. 336
                  Returns: 0 if OK, –1 on error

pid_t     tcgetpgrp(int filedes);
                  <sys/types.h>                               p. 248
                  <unistd.h>
                  Returns: process group ID of foreground process group if OK, –1 on error

int       tcsendbreak(int filedes, int duration);
                  <termios.h>                                 p. 344
                  Returns: 0 if OK, –1 on error

int       tcsetattr(int filedes, int opt, const struct termios *termptr);
                  <termios.h>                                 p. 336
                  opt: TCSANOW, TCSADRAIN, TCSAFLUSH
                  Returns: 0 if OK, –1 on error

int       tcsetpgrp(int filedes, pid_t pgrpid);
                  <sys/types.h>                               p. 248
                  <unistd.h>
                  Returns: 0 if OK, –1 on error

char      *tempnam(const char *directory, const char *prefix);
                  <stdio.h>                                   p. 141
                  Returns: pointer to unique pathname
```

```
time_t   time(time_t *calptr);
                    <time.h>                                    p. 155
                    Returns: value of time if OK, –1 on error

clock_t  times(struct tms *buf);
                    <sys/times.h>                               p. 232
                    Returns: elapsed wall clock time in clock ticks if OK, –1 on error

FILE     *tmpfile(void);
                    <stdio.h>                                   p. 140
                    Returns: file pointer if OK, NULL on error

char     *tmpnam(char *ptr);
                    <stdio.h>                                   p. 140
                    Returns: pointer to unique pathname

int      truncate(const char *pathname, off_t length);
                    <sys/types.h>                               p. 92
                    <unistd.h>
                    Returns: 0 if OK, –1 on error

char     *ttyname(int filedes);
                    <unistd.h>                                  p. 346
                    Returns: pointer to pathname of terminal, NULL on error

mode_t   umask(mode_t cmask);
                    <sys/types.h>                               p. 84
                    <sys/stat.h>
                    Returns: previous file mode creation mask

int      uname(struct utsname *name);
                    <sys/utsname.h>                             p. 154
                    Returns: nonnegative value if OK, –1 on error

int      ungetc(int c, FILE *fp);
                    <stdio.h>                                   p. 129
                    Returns: c if OK, EOF on error

int      unlink(const char *pathname);
                    <unistd.h>                                  p. 96
                    Returns: 0 if OK, –1 on error

void     unsetenv(const char *name);
                    <stdlib.h>                                  p. 173

int      utime(const char *pathname, const struct utimbuf *times);
                    <sys/types.h>                               p. 103
                    <utime.h>
                    Returns: 0 if OK, –1 on error

int      vfprintf(FILE *fp, const char *format, va_list arg);
                    <stdarg.h>                                  p. 137
                    <stdio.h>
                    Returns: #characters output if OK, negative value if output error
```

```
int      vprintf(const char *format, va_list arg);
                 <stdarg.h>                          p. 137
                 <stdio.h>
                 Returns: #characters output if OK, negative value if output error

int      vsprintf(char *buf, const char *format, va_list arg);
                 <stdarg.h>                          p. 137
                 <stdio.h>
                 Returns: #characters stored in array

pid_t    wait(int *statloc);
                 <sys/types.h>                       p. 197
                 <sys/wait.h>
                 Returns: process ID if OK, 0, or -1 on error

pid_t    wait3(int *statloc, int options, struct rusage *rusage);
                 <sys/types.h>                       p. 203
                 <sys/wait.h>
                 <sys/time.h>
                 <sys/resource.h>
                 options: 0, WNOHANG, WUNTRACED
                 Returns: process ID if OK, 0, or -1 on error

pid_t    wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
                 <sys/types.h>                       p. 203
                 <sys/wait.h>
                 <sys/time.h>
                 <sys/resource.h>
                 options: 0, WNOHANG, WUNTRACED
                 Returns: process ID if OK, 0, or -1 on error

pid_t    waitpid(pid_t pid, int *statloc, int options);
                 <sys/types.h>                       p. 197
                 <sys/wait.h>
                 options: 0, WNOHANG, WUNTRACED
                 Returns: process ID if OK, 0, or -1 on error

ssize_t  write(int filedes, const void *buff, size_t nbytes);
                 <unistd.h>                          p. 55
                 Returns: #bytes written if OK, -1 on error

ssize_t  writev(int filedes, const struct iovec iov[], int iovcnt);
                 <sys/types.h>                       p. 404
                 <sys/uio.h>
                 Returns: #bytes written if OK, -1 on error
```

# Appendix B

# Miscellaneous Source Code

## B.1 Our Header File

Most programs in the text include the header `ourhdr.h`, shown in Program B.1. It defines constants (such as `MAXLINE`) and prototypes for our own functions.

Since most programs need to include the following headers: `<stdio.h>`, `<stdlib.h>` (for the `exit` function prototype), and `<unistd.h>` (for all the standard Unix function prototypes), our header automatically includes these system headers, along with `<string.h>`. This also reduces the size of all the program listings in the text.

```
/* Our own header, to be included *after* all standard system headers */

#ifndef __ourhdr_h
#define __ourhdr_h

#include    <sys/types.h>   /* required for some of our prototypes */
#include    <stdio.h>       /* for convenience */
#include    <stdlib.h>      /* for convenience */
#include    <string.h>      /* for convenience */
#include    <unistd.h>      /* for convenience */

#define MAXLINE 4096            /* max line length */

#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
                    /* default file access permissions for new files */
#define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
                    /* default permissions for new directories */

typedef void    Sigfunc(int);   /* for signal handlers */
```

```
                    /* 4.3BSD Reno <signal.h> doesn't define SIG_ERR */
#if defined(SIG_IGN) && !defined(SIG_ERR)
#define SIG_ERR ((Sigfunc *)-1)
#endif

#define min(a,b)    ((a) < (b) ? (a) : (b))
#define max(a,b)    ((a) > (b) ? (a) : (b))

                    /* prototypes for our own functions */
char    *path_alloc(int *);             /* Program 2.2 */
int      open_max(void);                /* Program 2.3 */
void     clr_fl(int, int);              /* Program 3.5 */
void     set_fl(int, int);              /* Program 3.5 */
void     pr_exit(int);                  /* Program 8.3 */
void     pr_mask(const char *);         /* Program 10.10 */
Sigfunc *signal_intr(int, Sigfunc *);/* Program 10.13 */

int      tty_cbreak(int);               /* Program 11.10 */
int      tty_raw(int);                  /* Program 11.10 */
int      tty_reset(int);                /* Program 11.10 */
void     tty_atexit(void);              /* Program 11.10 */
#ifdef  ECHO    /* only if <termios.h> has been included */
struct termios  *tty_termios(void); /* Program 11.10 */
#endif

void     sleep_us(unsigned int);        /* Exercise 12.6 */
ssize_t  readn(int, void *, size_t);/* Program 12.13 */
ssize_t  writen(int, const void *, size_t);/* Program 12.12 */
int      daemon_init(void);             /* Program 13.1 */

int      s_pipe(int *);                 /* Programs 15.2 and 15.3 */
int      recv_fd(int, ssize_t (*func)(int, const void *, size_t));
                                        /* Programs 15.6 and 15.8 */
int      send_fd(int, int);             /* Programs 15.5 and 15.7 */
int      send_err(int, int, const char *);/* Program 15.4 */
int      serv_listen(const char *); /* Programs 15.19 and 15.22 */
int      serv_accept(int, uid_t *); /* Programs 15.20 and 15.24 */
int      cli_conn(const char *);    /* Programs 15.21 and 15.23 */
int      buf_args(char *, int (*func)(int, char **));
                                        /* Program 15.17 */

int      ptym_open(char *);             /* Programs 19.1 and 19.2 */
int      ptys_open(int, char *);        /* Programs 19.1 and 19.2 */
#ifdef  TIOCGWINSZ
pid_t    pty_fork(int *, char *, const struct termios *,
                  const struct winsize *);  /* Program 19.3 */
#endif

int     lock_reg(int, int, int, off_t, int, off_t);
                                        /* Program 12.2 */
#define read_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
#define readw_lock(fd, offset, whence, len) \
```

```
            lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
#define write_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
#define writew_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
#define un_lock(fd, offset, whence, len) \
            lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)

pid_t   lock_test(int, int, off_t, int, off_t);
                                    /* Program 12.3 */

#define is_readlock(fd, offset, whence, len) \
            lock_test(fd, F_RDLCK, offset, whence, len)
#define is_writelock(fd, offset, whence, len) \
            lock_test(fd, F_WRLCK, offset, whence, len)

void    err_dump(const char *, ...);    /* Appendix B */
void    err_msg(const char *, ...);
void    err_quit(const char *, ...);
void    err_ret(const char *, ...);
void    err_sys(const char *, ...);

void    log_msg(const char *, ...);     /* Appendix B */
void    log_open(const char *, int, int);
void    log_quit(const char *, ...);
void    log_ret(const char *, ...);
void    log_sys(const char *, ...);

void    TELL_WAIT(void);        /* parent/child from Section 8.8 */
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif  /* __ourhdr_h */
```

**Program B.1** Our header ourhdr.h.

The reason we include our header after all the normal system headers is to fix up any system differences (such as the missing SIG_ERR from 4.3BSD Reno) and to define some of our prototypes, needed only if certain headers have been included. Some ANSI C compilers complain if they encounter references to structures in prototypes, when the structure has not been defined.

## B.2 Standard Error Routines

We have two sets of error functions that are used in most of the examples throughout the text to handle error conditions. One set begins with err_ and outputs an error message to standard error. The other set begins with log_ and is for daemon processes (Chapter 13) that probably have no controlling terminal.

The reason for our own error functions is to let us write our error handling with a single line of C code, as in

```
if (error condition)
        err_dump (printf format with any number of arguments) ;
```

instead of

```
if (error condition) {
        char  buff[200];
        sprintf(buff, printf format with any number of arguments) ;
        perror(buff);
        abort();
}
```

Our error functions use the variable-length argument list facility from ANSI C. See Section 7.3 of Kernighan and Ritchie [1988] for additional details. Be aware that this ANSI C facility differs from the `varargs` facility provided by earlier systems (such as SVR3 and 4.3BSD). The names of the macros are the same, but the arguments to some of the macros have changed.

Figure B.1 details the differences between the various error functions.

| Function | `strerror(errno)` ? | Terminate ? |
|---|---|---|
| `err_ret` | yes | `return;` |
| `err_sys` | yes | `exit(1);` |
| `err_dump` | yes | `abort();` |
| `err_msg` | no | `return;` |
| `err_quit` | no | `exit(1);` |
| `log_ret` | yes | `return;` |
| `log_sys` | yes | `exit(2);` |
| `log_msg` | no | `return;` |
| `log_quit` | no | `exit(2);` |

**Figure B.1** Our standard error functions.

Program B.2 shows the error functions that output to standard error.

```
#include    <errno.h>       /* for definition of errno */
#include    <stdarg.h>      /* ANSI C header file */
#include    "ourhdr.h"

static void err_doit(int, const char *, va_list);

char    *pname = NULL;      /* caller can set this from argv[0] */

/* Nonfatal error related to a system call.
 * Print a message and return. */

void
err_ret(const char *fmt, ...)
{
    va_list     ap;
```

```
    va_start(ap, fmt);
    err_doit(1, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error related to a system call.
 * Print a message and terminate. */

void
err_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, fmt, ap);
    va_end(ap);
    exit(1);
}

/* Fatal error related to a system call.
 * Print a message, dump core, and terminate. */

void
err_dump(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, fmt, ap);
    va_end(ap);
    abort();        /* dump core and terminate */
    exit(1);        /* shouldn't get here */
}

/* Nonfatal error unrelated to a system call.
 * Print a message and return. */

void
err_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error unrelated to a system call.
 * Print a message and terminate. */

void
err_quit(const char *fmt, ...)
```

```
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, fmt, ap);
    va_end(ap);
    exit(1);
}

/* Print a message and return to caller.
 * Caller specifies "errnoflag". */

static void
err_doit(int errnoflag, const char *fmt, va_list ap)
{
    int     errno_save;
    char    buf[MAXLINE];

    errno_save = errno;     /* value caller might want printed */
    vsprintf(buf, fmt, ap);
    if (errnoflag)
        sprintf(buf+strlen(buf), ": %s", strerror(errno_save));
    strcat(buf, "\n");
    fflush(stdout);     /* in case stdout and stderr are the same */
    fputs(buf, stderr);
    fflush(NULL);       /* flushes all stdio output streams */
    return;
}
```

**Program B.2** Error functions that output to standard error.

Program B.3 shows the `log_XXX` error functions. These require the caller to define the variable `debug` and set it nonzero if the process is not running as a daemon. In this case the error messages are sent to standard error. If the `debug` flag is 0, the `syslog` facility (Section 13.4.2) is used.

```
/* Error routines for programs that can run as a daemon. */

#include    <errno.h>       /* for definition of errno */
#include    <stdarg.h>      /* ANSI C header file */
#include    <syslog.h>
#include    "ourhdr.h"

static void log_doit(int, int, const char *, va_list ap);

extern int  debug;      /* caller must define and set this:
                           nonzero if interactive, zero if daemon */

/* Initialize syslog(), if running as daemon. */

void
log_open(const char *ident, int option, int facility)
{
```

```
    if (debug == 0)
        openlog(ident, option, facility);
}

/* Nonfatal error related to a system call.
 * Print a message with the system's errno value and return. */

void
log_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error related to a system call.
 * Print a message and terminate. */

void
log_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/* Nonfatal error unrelated to a system call.
 * Print a message and return. */

void
log_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error unrelated to a system call.
 * Print a message and terminate. */

void
log_quit(const char *fmt, ...)
{
    va_list     ap;
```

```
    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/* Print a message and return to caller.
 * Caller specifies "errnoflag" and "priority". */

static void
log_doit(int errnoflag, int priority, const char *fmt, va_list ap)
{
    int     errno_save;
    char    buf[MAXLINE];

    errno_save = errno;     /* value caller might want printed */
    vsprintf(buf, fmt, ap);
    if (errnoflag)
        sprintf(buf+strlen(buf), ": %s", strerror(errno_save));
    strcat(buf, "\n");
    if (debug) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else
        syslog(priority, buf);
    return;
}
```

**Program B.3** Error functions for daemons.

# Appendix C

# Solutions to Selected Exercises

## Chapter 1

**1.1** For this exercise we use the following two arguments for the `ls`(1) command: `-i` prints the i-node number of the file or directory (we say more about i-nodes in Section 4.14), and `-d` which outputs information about a directory, instead of information on all the files in the directory.

Execute the following

```
$ ls -ldi /etc/. /etc/..          -i says print i-node number
 3077 drwxr-sr-x  7 bin    2048 Aug  5 20:12 /etc/./
    2 drwxr-xr-x 13 root    512 Aug  5 20:11 /etc/../
$ ls -ldi /. /..                    both . and .. have i-node number 2
    2 drwxr-xr-x 13 root    512 Aug  5 20:11 /./
    2 drwxr-xr-x 13 root    512 Aug  5 20:11 /../
```

**1.2** Unix is a multiprogramming or multitasking system. Other processes were running at the time this program was run.

**1.3** Since the *ptr* argument to `perror` is a pointer, `perror` could modify the string that *ptr* points to. The qualifier `const`, however, says that `perror` does not modify what the pointer points to. The error number argument to `strerror`, however, is an integer, and since C passes all arguments by value, the `strerror` function couldn't modify this value even if it wanted to. (If the handling of function arguments in C is not clear, you should review Section 5.2 of Kernighan and Ritchie [1988].)

1.4 It is possible for the calls to `fflush`, `fprintf`, and `vprintf` to modify `errno`. If they did modify its value and we didn't save it, the error message finally printed would be incorrect.

This specific problem has shown up in many historical programs that didn't save `errno` as we have done. The classic error message often printed was "Not a typewriter." In Section 5.4 we'll see that the standard I/O library changes the buffering of some standard I/O streams, based on whether the stream refers to a terminal device or not. The function `isatty` (Section 11.9) is usually called to determine if the stream refers to a terminal device. If the stream doesn't refer to a terminal device, `errno` can be set to `ENOTTY`, causing this error. Program C.1 shows this feature.

```
#include    <stdio.h>

/*
 * The following prints errno=25 (ENOTTY) under 4.3BSD and SVR2,
 * when stdout is redirected to a file.
 * Under SVR4 and 4.3+BSD it works OK.
 */

int
main()
{
    int         fd;
    extern int  errno;

    if ( (fd = open("/no/such/file", 0)) < 0) {
        printf("open error: ");
        printf("errno = %d\n", errno);
    }
    exit(0);
}
```

**Program C.1** Show `errno` interaction with `printf`.

Running this program we have

```
$ grep BSD /etc/motd
4.3 BSD UNIX #29: Thu Mar 29 11:14:13 MST 1990
$ a.out
open error: errno = 2     works correctly because stdout is a terminal device
$ a.out > temp.foo
$ cat temp.foo
open error: errno = 25    wrong
```

1.5 During the year 2038. (Actually, a more important date is January 1, 2000, when many computer programs across the world could break.)

1.6 Approximately 248 days.

## Chapter 2

**2.1** The following technique is used by 4.3+BSD. The primitive data types that can appear in multiple headers are defined with an uppercase name in the header <machine/ansi.h>. For example,

```
#ifndef _ANSI_H_
#define _ANSI_H_

#define _CLOCK_T_   unsigned long
#define _SIZE_T_    unsigned int
...

#endif  /* _ANSI_H_ */
```

In each of the six headers that can define the size_t primitive system data type, we have the sequence

```
#ifdef  _SIZE_T_
typedef _SIZE_T_  size_t;
#undef  _SIZE_T_
#endif
```

This way the actual typedef is only executed once.

## Chapter 3

**3.1** All disk I/O goes through the kernel's block buffers (also called the kernel's buffer cache). The exception to this is I/O on a raw disk device, which we aren't considering. Chapter 3 of Bach [1986] describes the operation of this buffer cache. Since the data that we read or write is buffered by the kernel, the term "unbuffered I/O" refers to the fact that there is no automatic buffering in the user process with these two functions. Each read or write invokes a single system call.

**3.3** Each call to open gives us a new file table entry. But since both opens reference the same file, both file table entries point to the same v-node table entry. The call to dup references the existing file table entry. We show this in Figure C.1. An F_SETFD on fd1 affects only the file descriptor flags for fd1. But an F_SETFL on fd1 affects the file table entry that both fd1 and fd2 point to.

**3.4** If fd is 1, then the dup2(fd, 1) returns 1 without closing descriptor 1. (Remember our discussion of this in Section 3.12.) After the three calls to dup2 all three descriptors point to the same file table entry. Nothing needs to be closed.

If fd is 3, however, after the three calls to dup2 there are four descriptors pointing to the same file table entry. In this case we need to close descriptor 3.

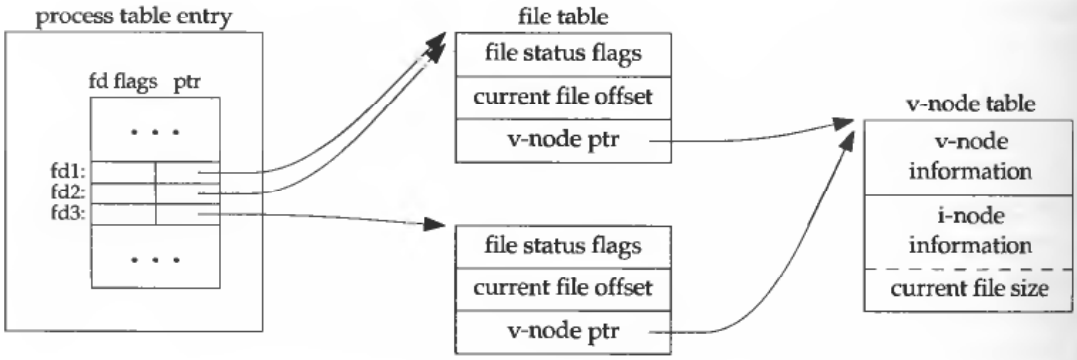


**Figure C.1** Result of dup and open.

3.5 Since the shells process their command line from left to right, the command

```
a.out > outfile 2>&1
```

first sets standard output to `outfile` and then dups standard output onto descriptor 2 (standard error). The result is that standard output and standard error are set to the same file. Descriptors 1 and 2 both point to the same file table entry. With

```
a.out 2>&1 > outfile
```

however, the dup is executed first, causing descriptor 2 to be the terminal (assuming the command is run interactively). Then standard output is redirected to the file `outfile`. The result is that descriptor 1 points to the file table entry for `outfile` and descriptor 2 points to the file table entry for the terminal.

3.6 You can still `lseek` and `read` anywhere in the file, but a `write` automatically resets the file offset to the end of file before the data is actually written. This makes it impossible to `write` anywhere other than at the end of file.

## Chapter 4

4.1 If `stat` is called, it always tries to follow a symbolic link (Figure 4.10), so the program will never print a file type of "symbolic link." For the example shown in the text, where `/bin` is a symbolic link to `/usr/bin`, `stat` reports that `/bin` is a directory, not a symbolic link. If the symbolic link points to a nonexistent file, `stat` returns an error.

4.2 The following lines can be added to `ourhdr.h`.

```
#if defined(S_IFLNK) && !defined(S_ISLNK)
#define S_ISLNK(mode)    (((mode) & S_IFMT) == S_IFLNK)
#endif
```

This is an example of how our own header can mask certain system differences.

**4.3** All permissions are turned off.

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
----------  1 stevens     29 Jan 14 06:39 temp.foo
```

**4.4** The following shows what happens when user-read permission is turned off.

```
$ date > foo
$ chmod u-r foo                    turn off user-read permission
$ ls -l foo                        verify the file's permissions
--w-rw-r--  1 stevens     29 Jul 31 09:00 foo
$ cat foo                          and try to read it
cat: foo: Permission denied
```

**4.5** If we try to create a file that already exists, using either `open` or `creat`, the file's access permission bits are not changed. We can verify this by running Program 4.3.

```
$ rm foo bar                       delete the files in case they already exist
$ date > foo                       create them with some data
$ date > bar
$ chmod a-r foo bar                turn off all read permissions
$ ls -l foo bar                    verify their permissions
--w--w----  1 stevens     29 Jul 31 10:47 bar
--w--w----  1 stevens     29 Jul 31 10:47 foo
$ a.out                            run Program 4.3
$ ls -l foo bar                    check permissions and sizes
--w--w----  1 stevens      0 Jul 31 10:47 bar
--w--w----  1 stevens      0 Jul 31 10:47 foo
```

Notice that the permissions didn't change but the files were truncated.

**4.6** The size of a directory should never be 0 since there should always be entries for dot and dot-dot. The size of a symbolic link is the number of characters in the pathname contained in the symbolic link, and this pathname must always contain at least one character.

**4.8** The kernel has a default setting for the file access permission bits when it creates a new `core` file. In this example it was `rw-r--r--`. This default value may or may not be modified by the `umask` value. The shell also has a default setting for the file access permission bits when it creates a new file for redirection. In this example it was `rw-rw-rw-` and this value is always modified by our current `umask`. In this example our `umask` was 02.

**4.9** We can't use `du` because it requires either the name of the file, as in

```
du tempfile
```

or a directory name, as in

```
du .
```

But when the `unlink` function returns, the directory entry for `tempfile` is gone. The `du .` command just shown would not account for the space still taken by `tempfile`. We have to use the `df` command in this example, to see the actual amount of free space on the filesystem.

**4.10** If the link being removed is not the last link to the file, the file is not removed. In this case the changed-status time of the file is updated. But if the link being removed is the last link to the file, it makes no sense to update this time, because all the information about the file (the i-node) is removed with the file.

**4.11** We recursively call our function `dopath` after opening a directory with `opendir`. Assuming that `opendir` uses a single file descriptor this means that each time we descend one level we use another descriptor. (We assume the descriptor isn't closed until we're finished with a directory and call `closedir`.) This limits the depth of the filesystem tree that we can traverse to the maximum number of open descriptors for the process. Notice that the SVR4 function `ftw` allows the caller to specify the number of descriptors to use, implying that this implementation can close and reuse descriptors.

**4.13** The `chroot` function is used by the Internet File Transfer Program (FTP) to aid in security. Users without accounts on a system (termed "anonymous FTP") are placed in a separate directory and a `chroot` is done to that directory. This prevents the user from accessing any file outside this new root directory.

`chroot` can also be used to build a copy of a filesystem hierarchy at a new location and then modify this new copy without changing the original filesystem. This could be used, for example, to test the installation of new software packages.

`chroot` can be executed only by the superuser, and once you change the root of a process, it (and all its descendants) can never get back to the original root.

**4.14** First call `stat` to fetch the three times for the file, then call `utime` to set the desired value. The value that we don't want to change in the call to `utime` should be the corresponding value from `stat`.

**4.15** `finger`(1) calls `stat` on the mailbox. The last-modification time is the time that mail was last received, and the last-access time is when the mail was last read.

**4.16** Both `cpio` and `tar` store only the modification time (`st_mtime`) on the archive. The access time isn't stored because its value corresponds to the time the archive was created, since the file has to be read to be archived. The `-a` option to `cpio` has it reset the access time of each input file after the file has been read. This way the creation of the archive doesn't change the access time. (Resetting the access time, however, does modify the changed-status time.) The changed-status time isn't stored on the archive because we can't set this value on extraction even if it was archived. (The `utime` function can change only the access time and the modification time.)

When the archive is read back (extracted), `tar`, by default, restores the modification time to the value on the archive. The `m` option to `tar` tells it to not restore the modification time from the archive—instead the modification time is set to the

time of extraction. In all cases with `tar`, the access time after extraction will be the time of extraction.

On the other hand, `cpio` sets the access time and the modification time to the time of extraction. By default it doesn't try to set the modification time to the value on the archive. The `-m` option to `cpio` has it set both the access time and the modification time to the value that was archived.

**4.17** Some versions of `file`(1) call `utime` to reset the file's access time, trying to undo the fact that `read` updates the access time. Doing this, however, updates the changed-status time.

**4.18** The kernel has no inherent limit on the depth of a directory tree. But many commands will fail on pathnames that exceed `PATH_MAX`. Program C.2 creates a directory tree that is 100 levels deep, with each level being a 45-character name. We are able to create this structure and obtain the absolute pathname of the directory at the 100th level using `getcwd`. (We have to call `realloc` numerous times to obtain a buffer that is large enough.) Running this program gives us

```
$ a.out
getcwd failed, size = 1025: Result too large
getcwd failed, size = 1125: Result too large
...                              33 more lines
getcwd failed, size = 4525: Result too large
length = 4613
                                 the 4613-byte pathname is printed here
```

We are not able to archive this directory, however, using either `tar` or `cpio`. Both complain of a filename that is too long. (With `cpio` it is the `find`(1) program that complains.) The command `rm -r` also fails because of the long pathname. (How can you delete the directory tree?)

**4.19** The `/dev` directory has all write permissions turned off to prevent a normal user from removing the filenames in the directory. This means the `unlink` fails.

## Chapter 5

**5.2** `fgets` reads up through and including the next newline *or* until the buffer is full (leaving room, of course, for the terminating null). Also, `fputs` writes everything in the buffer until it hits a null byte—it doesn't care if there is a newline in the buffer or not. So, if `MAXLINE` is too small, both functions still work, they're just called more often than they would be if the buffer were larger.

If either of these functions removed or added the newline (as `gets` and `puts` do) then we would have to assure that our buffer was big enough for the largest line.

**5.3** The function call

```
printf("");
```

returns 0 since no characters are output.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

#define DEPTH   100         /* directory depth */
#define MYHOME  "/home/stevens"
#define NAME    "alonglonglonglonglonglonglonglonglongname"

int
main(void)
{
    int     i, size;
    char    *path;

    if (chdir(MYHOME) < 0)
        err_sys("chdir error");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("mkdir failed, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("chdir failed, i = %d", i);
    }
    if (creat("afile", FILE_MODE) < 0)
        err_sys("creat error");

    /*
     * The deep directory is created, with a file at the leaf.
     * Now let's try and obtain its pathname.
     */

    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL)
            break;
        else {
            err_ret("getcwd failed, size = %d", size);
            size += 100;
            if ( (path = realloc(path, size)) == NULL)
                err_sys("realloc error");
        }
    }
    printf("length = %d\n%s\n", strlen(path), path);

    exit(0);
}
```

**Program C.2** Create a deep directory tree.

**5.4** This is a common error with the standard I/O library. The return value from `getc` (and hence `getchar`) is an integer, not a character. Since `EOF` is often defined to be –1, if the system uses signed characters, the code normally works. But if the system uses unsigned characters, after the `EOF` returned by `getchar` is stored as an unsigned character, it no longer equals –1, so the loop never terminates.

**5.5** A 5-character prefix, a 4-character per-process unique identifier, and a 5-character per-system unique identifier (the process ID) equals 14 characters, the traditional Unix limit on a filename.

**5.6** Call `fsync` after each call to `fflush`. The argument to `fsync` is obtained with the `fileno` function. Calling `fsync` without calling `fflush` might do nothing, if all the data were still be in memory buffers.

**5.7** Standard input and standard output are both line buffered when the programs are run interactively. When `fgets` is called, standard output is automatically flushed.

## Chapter 6

**6.1** Under SVR4 the functions to access the shadow password file are documented in the `getspent(3)` manual page. To compare an encrypted password we can't use the value returned in the `pw_passwd` field by the functions described in Section 6.2, since that field is not the encrypted password. Instead we need to find the user's entry in the shadow file and use the field in the shadow file that contains the encrypted password.

With 4.3+BSD the shadowing of the password file is done automatically. When the `passwd` structure is returned by either `getpwnam` or `getpwuid`, the field `pw_passwd` is filled in with the encrypted password only if the caller's effective user ID is 0.

**6.2** Under SVR4 Program C.3 prints the encrypted password. Unless this program is run with superuser permissions, the call to `getspnam` fails with an error of `EACCES`. Under 4.3+BSD Program C.4 prints the encrypted password, if the program is run with superuser permissions. Otherwise the value returned in `pw_passwd` is an asterisk.

**6.4** Program C.5 prints the date in a format similar to `date`. Running this program gives us

```
$ echo $TZ                          author's default
MST7
$ a.out
Wed Jan 15 06:48:57 MST 1992
$ TZ=EST5EDT a.out                  U.S. East Coast
Wed Jan 15 08:49:06 EST 1992
$ TZ=JST-9 a.out                    Japan
Wed Jan 15 22:49:12 JST 1992
```

```
#include    <sys/types.h>
#include    <shadow.h>
#include    "ourhdr.h"

int
main(void)      /* SVR4 version */
{
    struct spwd *ptr;

    if ( (ptr = getspnam("stevens")) == NULL)
        err_sys("getspnam error");

    printf("sp_pwdp = %s\n",
            ptr->sp_pwdp == NULL || ptr->sp_pwdp[0] == 0 ?
            "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

**Program C.3** Print encrypted password under SVR4

```
#include    <sys/types.h>
#include    <pwd.h>
#include    "ourhdr.h"

int
main(void)      /* 4.3+BSD version */
{
    struct passwd   *ptr;

    if ( (ptr = getpwnam("stevens")) == NULL)
        err_sys("getpwnam error");

    printf("pw_passwd = %s\n",
            ptr->pw_passwd == NULL || ptr->pw_passwd[0] == 0 ?
            "(null)" : ptr->pw_passwd);
    exit(0);
}
```

**Program C.4** Print encrypted password under 4.3+BSD

## Chapter 7

**7.1** It appears that the return value from `printf` (the number of characters output) becomes the return value of `main`. Not all systems exhibit this property.

**7.2** When the program is run interactively, standard output is usually line buffered, so the actual output occurs when each newline is output. If standard output were

```
#include    <time.h>
#include    "ourhdr.h"

int
main(void)
{
    time_t      caltime;
    struct tm   *tm;
    char        line[MAXLINE];

    if ( (caltime = time(NULL)) == -1)
        err_sys("time error");
    if ( (tm = localtime(&caltime)) == NULL)
        err_sys("localtime error");

    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("strftime error");
    fputs(line, stdout);

    exit(0);
}
```

**Program C.5** Print the time and date in a format similar to `date`(1).

directed to a file, however, it would probably be fully buffered, and the actual output wouldn't occur until the standard I/O cleanup is performed.

**7.3** On most Unix systems there is no way to do this. Copies of `argc` and `argv` are not kept in global variables like `environ`.

**7.4** This provides a way to terminate the process when it tries to dereference a null pointer, a common C programming error.

**7.5** The definitions are:

```
typedef void    Exitfunc(void);

int atexit(Exitfunc *func);
```

**7.6** `calloc` initializes the memory that it allocates to all zero bits. ANSI C does not guarantee that this is the same as either a floating point 0 or a null pointer.

**7.7** The heap and stack aren't allocated until a program is executed by one of the `exec` functions (described in Section 8.9).

**7.8** The executable file (`a.out`) contains symbol table information that can be helpful in debugging a `core` file. To remove this information the `strip`(1) command is used. Executing this command on the two `a.out` files reduces their size to 98304 and 16384.

**7.9** When shared libraries are not used, a large portion of the executable file is occupied by the standard I/O library.

7.10 The code is incorrect since it references the automatic integer `val` through a pointer after the automatic variable is no longer in existence. Automatic variables declared after the left brace that starts a compound statement disappear after the matching right brace.

## Chapter 8

8.1 Replace the call to `printf` with the lines

```
i = printf("pid = %d, glob = %d, var = %d\n",
                                    getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

You need to define the variables `i` and `buf` also.

This assumes the standard I/O stream `stdout` is closed when the child calls `exit`, not the file descriptor `STDOUT_FILENO`. Some versions of the standard I/O library close the file descriptor associated with standard output, which would cause the `write` to standard output to also fail. In this case, dup standard output to another descriptor and use this new descriptor for the `write`.

8.2 Consider Program C.6. When `vfork` is called, the parent's stack pointer points to the stack frame for the `f1` function that calls `vfork`. Figure C.2 shows this.

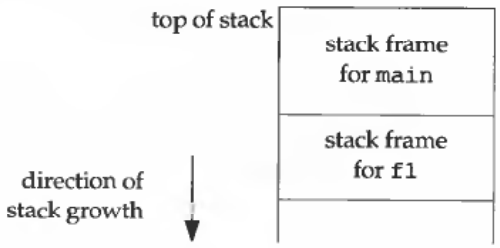


**Figure C.2** Stack frames when `vfork` is called.

`vfork` causes the child to execute first and the child returns from `f1`. The child then calls `f2` and its stack frame overwrites the previous stack frame for `f1`. The child then zeroes out the automatic variable `buf`, setting 1000 bytes of the stack frame to 0. The child returns from `f2`, and then calls `_exit`, but the contents of the stack beneath the stack frame for `main` have been changed. The parent then resumes after the call to `vfork` and does a return from `f1`. The return information is often stored in the stack frame, and that information has probably been modified by the child. What happens with this example, after the parent resumes, depends on many implementation features of your Unix system (where in the stack frame the return information is stored, what information in the stack frame

```
#include    <sys/types.h>
#include    "ourhdr.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t   pid;

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    /* child and parent both return */
}

static void
f2(void)
{
    char    buf[1000];      /* automatic variables */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

**Program C.6** Incorrect use of `vfork`.

is wiped out when the automatic variables are modified, and so on). The normal result is a `core` file, but your results may differ.

**8.3** In Program 8.7 we have the parent output first. When the parent is done the child writes its output, but we let the parent terminate. Whether the parent terminates or whether the child finishes its output first depends on the kernel's scheduling of the two processes (another race condition). When the parent terminates, the shell starts up the next program and this next program can interfere with the output from the previous child.

We can prevent this from happening by not letting the parent terminate until the child has also finished its output. Replace the code following the `fork` with the following:

```
    else if (pid == 0) {
        WAIT_PARENT();          /* parent goes first */
        charatatime("output from child\n");
        TELL_PARENT(getppid()); /* tell parent we're done */
    } else {
        charatatime("output from parent\n");
        TELL_CHILD(pid);        /* tell child we're done */
        WAIT_CHILD();           /* wait for child to finish */
    }
```

We won't see this happen if we let the child go first, since the shell doesn't start the next program until the parent terminates.

**8.4** The same value (`/home/stevens/bin/testinterp`) is printed for `argv[2]`. The reason is that `execlp` ends up calling `execve` with the same *pathname* as when we call `execl` directly. Recall Figure 8.6.

**8.5** A function is not provided to return the saved set-user-ID. Instead, we must save the effective user ID when the process is started.

**8.6** Program C.7 creates a zombie.

```
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)      /* child */
        exit(0);

    /* parent */
    sleep(4);

    system("ps");

    exit(0);
}
```

**Program C.7** Create a zombie and look at it's status with ps.

Zombies are usually designated by ps(1) with a status of "Z".

```
$ a.out
  PID TT STAT  TIME COMMAND
 5940 p3 S     0:00 a.out
 5941 p3 Z     0:00 <defunct>        the zombie
 5942 p3 S     0:00 sh -c ps
 5943 p3 R     0:00 ps
```

## Chapter 9

**9.1** `init` is the process that learns when a terminal user logs out, because `init` is the parent of the login shell and receives the `SIGCHLD` signal when the login shell terminates.

For a network login, however, `init` is not involved. Instead the login entries in the `utmp` and `wtmp` files, and their corresponding logout entries are usually written by the process that handles the login and detects the logout (`telnetd` in our example).

## Chapter 10

**10.1** The program terminates the first time we send it a signal. This is because the `pause` function returns whenever a signal is caught.

**10.2** Program C.8 implements the `raise` function.

```
#include    <sys/types.h>
#include    <signal.h>
#include    <unistd.h>

int
raise(int signo)
{
    return( kill(getpid(), signo) );
}
```

**Program C.8** Implementation of `raise` function.

**10.3** Figure C.3 shows the stack frames.

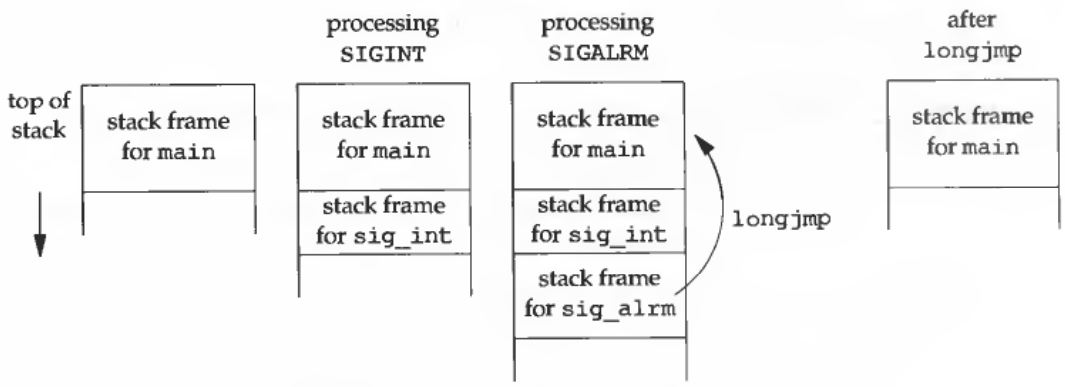


**Figure C.3** Stack frames before and after `longjmp`.

The `longjmp` from `sig_alrm` back to `main` effectively aborts the call to `sig_int`.

**10.4** We again have a race condition, this time between the first call to `alarm` and the call to `setjmp`. If the process is blocked by the kernel between these two function calls, the alarm will go off, the signal handler is called, and `longjmp` is called. But since `setjmp` was never called, the buffer `env_alrm` is not set. The operation of `longjmp` is undefined if its jump buffer has not been initialized by `setjmp`.

**10.5** See "Implementing Software Timers" by Don Libes (*C Users Journal*, Vol. 8, no. 11, Nov. 1990) for an example.

**10.7** If we just called `_exit` the termination status of the process would not show that it was terminated by the `SIGABRT` signal.

**10.8** If the signal was sent by a process owned by some other user, the process has to be set-user-ID to either root or to the owner of the receiving process or the `kill` won't work. Therefore, the real user ID provides more information to the receiver of the signal.

**10.10** On one system used by the author the value for the number of seconds increased by one about every 60–90 minutes. This skew is because each call to `sleep` schedules an event for a time in the future, but we're not awakened exactly when that event occurs (because of CPU scheduling). Plus there is a finite amount of time required for our process to start running and call `sleep` again.

A program such as the BSD `cron` has to fetch the current time every minute. It also has to set its first sleep period so that it wakes up at the beginning of the next minute. (Convert the current time to the local time and look at the `tm_sec` value.) Every minute, it sets the next sleep period so that it'll wake up at the next minute. Most of the calls will probably be `sleep(60)`, with an occasional `sleep(59)` to resynchronize with the next minute. But if at some point the process takes a long time executing commands or if the system gets heavily loaded and scheduling delays hold up the process, the sleep value can be much less than 60.

**10.11** Under SVR4 the signal handler for `SIGXFSZ` is never called. But `write` returns a count of 24 as soon as the file's size reaches 1024 bytes.

Under 4.3+BSD the signal handler is called after the file's size has reached 1500 bytes. The `write` returns –1 with `errno` set to `EFBIG` ("File too big").

SunOS 4.1.2 is similar to SVR4, but the signal handler is called.

In summary, it appears that System V returns a short count (without any error) as soon as the file reaches the soft limit, while BSD returns an error (without writing any data) when it determines the limit has been passed.

**10.12** The results depend on the implementation of the standard I/O library—how the `fwrite` function handles an interrupted `write`.

## Chapter 11

**11.1** Note that you have to terminate the `reset` command with a linefeed character, not a return, since the terminal is in noncanonical mode.

**11.2** It builds a table for each of the 128 characters and sets the high-order bit (the parity bit) according to the user's specification. It then uses eight-bit I/O, handling the parity generation itself.

**11.3** Under SVR4 execute `stty -a` with standard input redirected to the terminal running `vi`. This shows that `vi` sets MIN to 1 and TIME to 1. The `read`s wait for at least one character to be typed, but after that character is entered, `read` waits only one-tenth of a second for additional characters before returning.

**11.4** Under SVR4 the extended general terminal interface is used. This is documented in the `termiox`(7) manual page in AT&T [1991]. Under 4.3+BSD the flags `CCTS_OFLOW` and `CRTS_IFLOW` in the `c_cflag` field are used (Figure 11.3).

## Chapter 12

**12.1** The program works fine (it doesn't get the `ENOLCK` error). The first time through the loop we call `writew_lock`, `write`, and `un_lock`. The call to `un_lock` releases the lock from the current end of file through any future end of file, as before, leaving just the first byte locked. We then go through the loop again, but this time the call to `writew_lock` causes this new lock that we've specified to be merged with the existing lock on the first byte. Figure C.4 shows the state of the file after the second time through the loop.

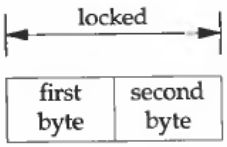


**Figure C.4** State of record lock after second time through loop.

Each time through the loop we extend this single lock by an additional byte. Since the kernel merges each lock with the existing lock, only a single lock is maintained by the kernel, and it never runs out of lock structures.

**12.2** Both SVR4 and 4.3+BSD define the `fd_set` data type to be a structure that contains a single member: an array of long integers. One bit in this array corresponds to each descriptor. The four `FD_` macros then manipulate this array of longs, turning specific bits on and off and testing specific bits.

One reason that the data type is defined to be a structure containing an array and not just an array is to allow variables of type `fd_set` to be assigned to one another with the C assignment statement.

**12.3** SVR4 and 4.3+BSD allow us to define the constant `FD_SETSIZE` before including the header `<sys/types.h>`. For example, we can write

```
#define FD_SETSIZE  2048
#include    <sys/types.h>
```

to define the `fd_set` data type to accommodate 2048 descriptors.

**12.4** The following table lists the functions that do similar things.

| | |
|---|---|
| FD_ZERO | sigemptyset |
| FD_SET | sigaddset |
| FD_CLR | sigdelset |
| FD_ISSET | sigismember |

There is not an `FD_xxx` function that corresponds to `sigfillset`. With signal sets the pointer to the set is always the first argument and the signal number is the second argument. With descriptor sets the descriptor number is the first argument and the pointer to the set is the next argument.

**12.5** Up to five different types of information are returned by `getmsg`: the data itself, the length of the data, the control information, the length of the control information, and the flags.

**12.6** Program C.9 shows an implementation using `select`. As the BSD `usleep`(3) manual page states, `usleep` utilizes the `setitimer` interval timer and performs eight system calls each time it's called. It correctly interacts with other timers set by the calling process, and it is not interrupted if a signal is caught.

Program C.10 shows an implementation using `poll`.

**12.7** No. What we would like to do is have `TELL_WAIT` create a temporary file and use one byte for the parent's lock and one byte for the child's lock. `WAIT_CHILD` would have the parent wait to obtain a lock on the child's byte, and `TELL_PARENT` would have the child release the lock on the child's byte. The problem, however, is that calling `fork` releases all the locks in the child, so the child can't start off with any locks of its own.

**12.8** A solution using `select` is shown in Program C.11. The same technique can be used with `poll`.

Under SVR4 and SunOS 4.1.1 the values calculated using both `select` and `poll` equal the values from Figure 2.6. Under 4.3+BSD the value calculated using `select` is 3073.

**12.9** Under SVR4, 4.3+BSD, and SunOS 4.1.2 Program 12.14 does update the last-access time for the input file.

## Chapter 13

**13.1** If the process calls `chroot` it will not be able to open `/dev/log`. The solution is for the daemon to call `openlog` with an *option* of `LOG_NDELAY`, before calling

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    <stddef.h>
#include    "ourhdr.h"

void
sleep_us(unsigned int nusecs)
{
    struct timeval  tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

**Program C.9** Implementation of `sleep_us` using `select`.

```
#include    <sys/types.h>
#include    <poll.h>
#include    <stropts.h>
#include    "ourhdr.h"

void
sleep_us(unsigned int nusecs)
{
    struct pollfd   dummy;
    int             timeout;

    if ( (timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

**Program C.10** Implementation of `sleep_us` using `poll`.

`chroot`. This opens the special device file (the Unix domain datagram socket), yielding a descriptor that is still valid, even after a call to `chroot`. This scenario is encountered in daemons such as `tftpd` (the Trivial File Transfer Daemon) that specifically call `chroot` for security reasons, but still need to call `syslog` to log error conditions.

**13.3** Program C.12 shows a solution. The results depend on the implementation and whether we close file descriptors 0, 1, and 2. The reason closing the descriptors affects the outcome is that when the program is started they are connected to the controlling terminal. Closing the three descriptors after calling `daemon_init` means `getlogin` won't have a controlling terminal, so it won't be able to look in the `utmp` file for our login entry.

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    "ourhdr.h"

int
main(void)
{
    int             i, n, fd[2];
    fd_set          writeset;
    struct timeval  tv;

    if (pipe(fd) < 0)
        err_sys("pipe error");
    FD_ZERO(&writeset);

    for (n = 0; ; n++) { /* write 1 byte at a time until pipe is full */
        FD_SET(fd[1], &writeset);
        tv.tv_sec = tv.tv_usec = 0;     /* don't wait at all */
        if ( (i = select(fd[1]+1, NULL, &writeset, NULL, &tv)) < 0)
            err_sys("select error");
        else if (i == 0)
            break;
        if (write(fd[1], "a", 1) != 1)
            err_sys("write error");
    }
    printf("pipe capacity = %d\n", n);
    exit(0);
}
```

**Program C.11** Calculation of pipe capacity using `select`.

```
#include    "ourhdr.h"

int
main(void)
{
    char    *ptr, buff[MAXLINE];

    daemon_init();

    close(0);
    close(1);
    close(2);

    ptr = getlogin();
    sprintf(buff, "login name: %s\n",
                    (ptr == NULL) ? "(empty)" : ptr);
    write(3, buff, strlen(buff));
    exit(0);
}
```

**Program C.12** Call `daemon_init` and then obtain login name.

Under 4.3+BSD, however, the login name is maintained in the process table and copied across a `fork`. This means the process can always get the login name, unless the parent didn't have a login name (such as `init` when the system is bootstrapped).

## Chapter 14

**14.1** If the write end of the pipe is never closed, the reader never sees an end of file. The pager program blocks forever reading from its standard input.

**14.2** The parent terminates right after writing the last line to the pipe. The read end of the pipe is automatically closed when the parent terminates. But the parent is probably running ahead of the child by one pipe buffer, since the child (the pager program) is waiting for us to look at a page of output. If we're running a shell such as the KornShell with interactive command-line editing enabled, the shell probably changes the terminal mode when our parent terminates and the shell prints a prompt. This undoubtably interferes with the pager program, which has also modified the terminal mode. (Most pager programs set the terminal to noncanonical mode when awaiting input to proceed to the next page.)

**14.3** `popen` returns a file pointer, because the shell is executed. But the shell can't execute the nonexistent command so it prints

```
sh: a.out: not found
```

on the standard error and terminates with an exit status of 1. `pclose` returns this exit status of 1.

**14.4** When the parent terminates, look at its termination status with the shell. For the Bourne shell and KornShell the command is `echo $?`. The number printed is 128 plus the signal number.

**14.5** First add the declaration

```
FILE    *fpin, *fpout;
```

Then use `fdopen` to associate the pipe descriptors with a standard I/O stream and set the streams to be line buffered. Do this before the `while` loop that reads from standard input:

```
if ( (fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("fdopen error");
if ( (fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("fdopen error");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
```

The `write` and `read` in the `while` loop are replaced with

```
    if (fputs(line, fpout) == EOF)
        err_sys("fputs error to pipe");
    if (fgets(line, MAXLINE, fpin) == NULL) {
        err_msg("child closed pipe");
        break;
    }
```

**14.6** The `system` function calls `wait` and the first child to terminate is the child generated by `popen`. Since that's not the child that `system` created, it calls `wait` again, and blocks until the `sleep` is done. `system` then returns. When `pclose` calls `wait`, an error is returned since there are no more children to `wait` for. `pclose` returns an error.

**14.7** `select` indicates that the descriptor is readable. When we call `read`, after all the data has been read, it returns 0 to indicate the end of file. But with `poll` (assuming the pipe is a streams device), the `POLLHUP` event is returned, and this event may be returned while there is still data to be read. Once we have read all the data, however, `read` returns 0 to indicate the end of file. After all the data has been read, the `POLLIN` event is not returned, even though we need to issue a `read` to receive the end of file notification (the return of 0).

With an output descriptor that refers to a pipe that has been closed by the reader, `select` indicates that the descriptor is writable. But when we call `write` the `SIGPIPE` signal is generated. If we either ignore this signal or return from its signal handler, `write` returns an error of `EPIPE`. With `poll`, however, if the pipe is a streams device, `poll` returns with an indication of `POLLHUP` for the descriptor.

**14.8** Anything written by the child to standard error appears wherever the parent's standard error would appear. To send standard error back to the parent, include the shell redirection `2>&1` in the *cmdstring*.

**14.9** `popen` `fork`s a child, and the child `exec`s the Bourne shell. The shell in turn calls `fork`, and the child of the shell `exec`s the command string. When *cmdstring* terminates, the shell is waiting for this to happen. The shell then `exit`s, which is what the `waitpid` in `pclose` is waiting for.

**14.10** The trick is to open the FIFO twice—once for reading and once for writing. We never use the descriptor that is opened for writing, but leaving that descriptor open prevents and end of file from being generated when the number of clients goes from 1 to 0. Opening the FIFO twice requires some care, as a nonblocking open is required. We have to do a nonblocking, read-only open first, followed by a blocking open for write-only. (If we tried a nonblocking open for write-only first, it would return an error.) We then turn off nonblocking for the read descriptor. Program C.13 shows the code for this.

**14.11** Randomly reading a message from an active queue would interfere with the client–server protocol, as either a client request or a server's response would be lost. To read the queue, all the process needs to know is the identifier for the queue, and for the queue to allow world-read.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

#define FIFO    "temp.fifo"

int
main(void)
{
    int     fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");

    if ( (fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ( (fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");

    clr_fl(fdread, O_NONBLOCK);

    exit(0);
}
```

**Program C.13** Opening a FIFO for reading and writing, without blocking.

**14.13** We never store actual addresses in a shared memory segment, since it's possible for the server and all the clients to attach the segment at different addresses. Instead, when a linked list is built in a shared memory segment, the list pointers should be stored as offsets to other objects in the shared memory segment. These offsets are formed by subtracting the start of the shared memory segment from the actual address of the object.

**14.14** Figure C.5 shows the relevant events.

## Chapter 15

**15.3** A *declaration* specifies the attributes (such as the data type) of a set of identifiers. If the declaration also causes storage to be allocated, it is called a *definition*.

In the `opend.h` header we declare the three global variables with the `extern` storage class. These declarations do not cause storage to be allocated for the variables. In the `main.c` file we define the three global variables. Sometimes we'll also initialize a global variable when we define it, but typically we let the C default apply.

| Parent i set to | Child i set to | Shared value set to | update returns | Comment |
|---|---|---|---|---|
| | | 0 | | initialized by mmap |
| | 1 | | | child runs first, then is blocked |
| 0 | | | | parent runs |
| | | 1 | | |
| | | | 0 | then parent is blocked |
| | | 2 | | child resumes |
| | | | 1 | |
| | 3 | | | then child is blocked |
| 2 | | | | parent resumes |
| | | 3 | | |
| | | | 2 | then parent is blocked |
| | | 4 | | |
| | | | 3 | |
| | 5 | | | then child is blocked |
| 4 | | | | parent resumes |

**Figure C.5** Alternation between parent and child in Program 14.12.

**15.5** Both `select` and `poll` return the number of ready descriptors as the value of the function. The loop that goes through the `client` array can terminate when the number of ready descriptors have been processed.

## Chapter 16

**16.1** Our conservative locking in `_db_dodelete` is to avoid race conditions with `db_nextrec`. If the call to `_db_writedat` were not protected with a write lock, it would be possible to erase the data record while `db_nextrec` was reading that data record: `db_nextrec` would read an index record, determine it was not all blank, and then read the data record, which could be erased by `_db_dodelete` between the calls to `_db_readidx` and `_db_readdat` in `db_nextrec`.

**16.2** Assume `db_nextrec` calls `_db_readidx`, which reads the key into the index buffer for the process. This process is then stopped by the kernel and another process runs. This other process calls `db_delete`, and the record being read by the other process is deleted. Both its key and data are rewritten in the two files as all blanks. The first process resumes and calls `_db_readdat` (from `db_nextrec`) and reads the all-blank data record. The read lock by `db_nextrec` allows it to do the read of the index record, followed by the read of the data record, as an atomic operation (with regard to other cooperating processes using the same database).

**16.3** With mandatory locking other readers and writers are affected. Other `reads` and `writes` are blocked by the kernel until the locks placed by `_db_writeidx` and `_db_writedat` are removed.

## Chapter 17

**17.1** `psif` has to read the first two bytes of the file and compare them to `%!`. If the file is seekable, it can then rewind the file and `exec` either `lprps` or `textps`. If the file is not seekable, it has to put the two bytes that it read back onto the standard input. One way to do this is to create a pipe and `fork` a child. The parent then sets its standard input to be the pipe and `exec`s either `textps` or `lprps`. The child writes the two bytes that it read to the pipe, followed by the rest of the file to be printed.

## Chapter 18

**18.2** Normally `getopt` is called to process only a single argument list. The global variable `optind` is initialized to 1 in the initialized data segment of the `getopt` function. But in our server we call `getopt` to process multiple argument lists—one argument list per client, so we have to reinitialize `optind` before the first call to `getopt` for each client.

**18.3** We maintain the file offset of the `Systems` file in the `Client` structure. If the file is modified after we've saved this offset, but before it's used the next time, there's a good chance that the saved offset does not reference the line that it previously pointed to. While our server could detect if this file has been modified (how?), we have no way of repositioning the file offset to where it used to point to. Our only recourse if the file is modified is not to try dialing again for any client whose in-progress dial doesn't work.

**18.4** The only time the `client` array can be moved around by `realloc` is when `client_add` is called, which is only after the `select`, not in the loop in which we use `cliptr`.

**18.5** The commands sent to the remote system will be messed up. A check could be added to `take_put_args` to test for this.

**18.6** A common technique is to require the person who modifies any of the files to tell the server, to let the server reread the files. The `SIGHUP` signal is often used for this.

**18.9** You could execute the `stty` command on the remote system and parse its output, but given the wide differences in the output of this command across different Unix systems, this solution would be hard to implement.

## Chapter 19

**19.1** Both servers, `telnetd` and `rlogind`, run with superuser privileges, so their calls to `chown` and `chmod` succeed.

**19.3** Execute

```
pty -n stty -a
```

to prevent the slave's `termios` structure and `winsize` structure from being initialized.

**19.5** Unfortunately the `F_SETFL` command of `fcntl` doesn't allow the read–write status to be changed.

**19.6** There are three process groups: (1) the login shell, (2) the `pty` parent and child, and (3) the `cat` process. The first two process groups constitute a session with the login shell as the session leader. The second session contains just the `cat` process. The first process group (the login shell) is a background process group and the other two are foreground process groups.

**19.7** First `cat` terminates when it receives the end of file from its line discipline. This causes the pty slave to terminate, which causes the pty master to terminate. This in turn generates an end of file for the `pty` parent that's reading from the pty master. The parent sends `SIGTERM` to the child so the child terminates next. (The child doesn't catch this signal.) Finally the parent calls `exit(0)` at the end of the `main` function.

The relevant output from Program 8.17 is

```
cat       e =    270, chars =     274, stat =   0:
pty       e =    262, chars =      40, stat =  15: F      X
pty       e =    288, chars =     188, stat =   0:
```

**19.8** This can be done with the shell's `echo` command and the `date`(1) command, all in a subshell.

```
#!/bin/sh
( echo "Script started on " `date`;
  pty "${SHELL:-/bin/sh}";
  echo "Script done on " `date` ) | tee typescript
```

**19.9** The line discipline above the pty slave has echo enabled so whatever `pty` reads on its standard input and writes to the pty master gets echoed by default. This echoing is done by the line discipline module above the slave even though the program (`ttyname`) never reads the data.

# Bibliography


Adobe Systems Inc. 1985. *PostScript Language Tutorial and Cookbook*. Addison-Wesley, Reading, Mass.

The "blue book."

Adobe Systems Inc. 1986. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Mass.

The "red book." Appendix D of the 1985 version of this book contained detailed information on communication across a serial line with a PostScript printer. This information was removed from the 1986 version.

Adobe Systems Inc. 1988. *PostScript Language Program Design*. Addison-Wesley, Reading, Mass.

The "green book." Chapter 12 contains information on writing a print spooler for a PostScript printer.

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1988. *The AWK Programming Language*. Addison-Wesley, Reading, Mass.

A complete book on the awk programming language. The version of awk described in this book is sometimes called "nawk" (for new awk).

Andrade, J. M., Carges, M. T., and Kovach, K. R. 1989. "Building a Transaction Processing System on UNIX Systems," *Proceedings of the 1989 USENIX Transaction Processing Workshop*, pp. 13–22 (May), Pittsburgh, Pa.

A description of the AT&T Tuxedo Transaction Processing System.

ANSI. 1989. "American National Standard for Information Systems—Programming Language C," X3.159–1989, ANSI (Dec.).

The official standard for the C language and the standard libraries.

This standard can be ordered from Global Engineering Documents at +1 800 854 7179 or +1 714 261 1455.

Arnold, J. Q. 1986. "Shared Libraries on UNIX System V," *Proceedings of the 1986 Summer USENIX Conference*, pp. 395–404, Atlanta, Ga.

Describes the implementation of shared libraries in SVR3.

AT&T. 1989. *System V Interface Definition, Third Edition.* Addison-Wesley, Reading, Mass.

This is a four-volume set that specifies the source code interface and run-time behavior of System V. The third edition corresponds to SVR4. A fifth volume was published in 1991 containing updated versions of commands and functions from volumes 1–4.

AT&T. 1990a. *UNIX Research System Programmer's Manual, Tenth Edition, Volume I.* Saunders College Publishing, Fort Worth, Tex.

The version of the *Unix Programmer's Manual* for the 10th Edition of Research Unix (V10). This volume contains the traditional Unix manual pages (Sections 1–9).

AT&T. 1990b. *UNIX Research System Papers, Tenth Edition, Volume II.* Saunders College Publishing, Fort Worth, Tex.

Volume II for the 10th Edition of Research Unix (V10) contains 40 papers describing various aspects of the system.

AT&T. 1990c. *UNIX System V Release 4 BSD/XENIX Compatability Guide.* Prentice-Hall, Englewood Cliffs, N.J.

Contains manual pages describing the compatibility library.

AT&T. 1990d. *UNIX System V Release 4 Programmer's Guide: STREAMS.* Prentice-Hall, Englewood Cliffs, N.J.

Describes the STREAMS system in SVR4.

AT&T. 1990e. *UNIX System V/386 Release 4 Programmer's Reference Manual.* Prentice-Hall, Englewood Cliffs, N.J.

This is the programmer's reference manual for the SVR4 implementation for the Intel 80386 processor. It contains Sections 1 (commands), 2 (system calls), 3 (subroutines), 4 (file formats), and 5 (miscellaneous facilities).

AT&T. 1991. *UNIX System V/386 Release 4 System Administrator's Reference Manual.* Prentice-Hall, Englewood Cliffs, N.J.

This is the system administrator's reference manual for the SVR4 implementation for the Intel 80386 processor. It contains Sections 1 (commands), 4 (file formats), 5 (miscellaneous facilities), and 7 (special files).

Bach, M. J. 1986. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, N.J.

A book on the details of the design and implementation of the Unix operating system. Although actual Unix source code is not provided in this text (since it is proprietary to AT&T) many of the algorithms and data structures used by the Unix kernel are presented and discussed. This book describes SVR2.

Bolsky, M. I., and Korn, D. G. 1989. *The KornShell Command and Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

Chen, D., Barkley, R. E., and Lee, T. P. 1990. "Insuring Improved VM Performance: Some No-Fault Policies," *Proceedings of the 1990 Winter USENIX Conference,* pp. 11–22, Washington, D.C.

Describes changes made to the virtual memory implementation of SVR4 to improve its performance, especially for `fork` and `exec`.

Comer, D. E. 1979. "The Ubiquitous B-Tree," *ACM Computing Surveys,* vol. 11, no. 2, pp. 121–137 (June).

Date, C. J. 1982. *An Introduction to Database Systems, Volume II.* Addison-Wesley, Reading, Mass.

Fowler, G. S., Korn, D. G., and Vo, K. P. 1989. "An Efficient File Hierarchy Walker," *Proceedings of the 1989 Summer USENIX Conference*, pp. 173–188, Baltimore, Md.

Describes a new library function to traverse a filesystem hierarchy.

Garfinkel, S., and Spafford, G. 1991. *Practical UNIX Security*. O'Reilly & Associates, Sebastopol, Calif.

A detailed book on Unix security.

Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. 1987. "Shared Libraries in SunOS," *Proceedings of the 1987 Summer USENIX Conference*, pp. 131–145, Phoenix, Ariz.

Gingell, R. A., Moran, J. P., and Shannon, W. A. 1987. "Virtual Memory Architecture in SunOS," *Proceedings of the 1987 Summer USENIX Conference*, pp. 81–94, Phoenix, Ariz.

Describes the initial implementation of the `mmap` function and related issues in the virtual memory design.

Goodheart, B. 1991. *UNIX Curses Explained*. Prentice-Hall, Englewood Cliffs, N.J.

A complete reference on terminfo and the `curses` library.

Hume, A. G. 1988. "A Tale of Two Greps," *Softw. Pract. and Exper.*, vol. 18, no. 11, pp. 1063–1072.

IEEE. 1990. "Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]," 1003.1–1990, IEEE (Dec.).

This is the first of the POSIX standards, and it defines the C language systems interface standard, based on the Unix operating system. It is often called POSIX.1.

This standard can be ordered directly from the IEEE: +1 800 678 IEEE, or +1 908 981 1393.

Kernighan, B. W., and Pike, R. 1984. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, N.J.

A general reference for additional details on Unix programming. This book covers numerous Unix commands and utilities, such as `grep`, `sed`, `awk`, and the Bourne shell.

Kernighan, B. W., and Ritchie, D. M. 1988. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.

A book on the ANSI standard version of the C programming language. Appendix B contains a description of the libraries defined by the ANSI standard.

Kleiman, S. R. 1986. "Vnodes: An Architecture for Multiple File System Types in Sun Unix," *Proceedings of the 1986 Summer USENIX Conference*, pp. 238–247, Atlanta, Ga.

A description of the original v-node implementation.

Korn, D. G., and Vo, K. P. 1991. "SFIO: Safe/Fast String/File IO," *Proceedings of the 1991 Summer USENIX Conference*, pp. 235–255, Nashville, Tenn.

A description of an alternative to the standard I/O library.

Krieger, O., Stumm, M., and Unrau, R. 1992. "Exploiting the Advantages of Mapped Files for Stream I/O," *Proceedings of the 1992 Winter USENIX Conference*, pp. 27–42, San Francisco, Calif.

An alternative to the standard I/O library based on mapped files.

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Mass.

An entire book on the 4.3BSD Unix system. This book describes the Tahoe release of 4.3BSD.

Libes, D. 1990. "expect: Curing Those Uncontrollable Fits of Interaction," *Proceedings of the 1990 Summer USENIX Conference*, pp. 183–192, Anaheim, Calif.

A description of the expect program and its implementation.

Libes, D. 1991. "expect: Scripts for Controlling Interactive Processes," *Computing Systems*, vol. 4, no. 2, pp. 99–125 (Spring).

This paper presents numerous expect scripts.

Morris, R., and Thompson, K. 1979. "UNIX Password Security," *Communications ACM*, vol. 22, no. 11, pp. 594–597 (Nov.).

A description of the history of the design of the Unix password scheme.

Nemeth, E., Snyder, G., and Seebass, S. 1989. *UNIX System Administration Handbook*. Prentice-Hall, Englewood Cliffs, N.J.

A book with many details on administering a Unix system.

Olander, D. J., McGrath, G. J., and Israel, R. K. 1986. "A Framework for Networking in System V," *Proceedings of the 1986 Summer USENIX Conference*, pp. 38–45, Atlanta, Ga.

This paper describes the original implementation of service interfaces, streams, and TLI for System V.

Plauger, P. J. 1992. *The Standard C Library*. Prentice-Hall, Englewood Cliffs, N.J.

A complete book on the ANSI C library. It contains a complete C implementation of the library.

Presotto, D. L., and Ritchie, D. M. 1990. "Interprocess Communication in the Ninth Edition UNIX System," *Softw. Pract. and Exper.*, vol. 20, no. S1, pp. S1/3-S1/17 (June).

This paper describes the IPC facilities provided by the Ninth Edition of Unix, developed at the Information Sciences Research Division of AT&T Bell Laboratories. The features are built on the stream input–output system and include full-duplex pipes, the ability to pass file descriptors between processes, and unique client connections to servers. A copy of this paper also appears in AT&T [1990b].

Redman, B. E. 1989. "UUCP UNIX-to-UNIX Copy," in *UNIX Networking*, eds. S. G. Kochan and P. H. Wood, pp. 5–48. Howard W. Sams and Company, Indianapolis, Ind.

This chapter contains additional details on Honey DanBer UUCP. It also contains a detailed history of the UUCP programs.

Ritchie, D. M. 1984. "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910 (Oct.).

The original paper on Streams.

Seltzer, M., and Olson, M. 1992. "LIBTP: Portable, Modular Transactions for UNIX," *Proceedings of the 1992 Winter USENIX Conference*, pp. 9–25, San Francisco, Calif.

A modification of the db(3) library from 4.3+BSD that implements transactions.

Seltzer, M., and Yigit, O. 1991. "A New Hashing Package for UNIX," *Proceedings of the 1991 Winter USENIX Conference*, pp. 173–184, Dallas, Tex.

A description of the dbm(3) library and various implementations of it, and a newer hashing package.

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, N.J.

A detailed book on network programming under Unix.

Stonebraker, M. R. 1981. "Operating System Support for Database Management," *Communications ACM*, vol. 24, no. 7, pp. 412–418 (July).

Strang, J., Mui, L., and O'Reilly, T. 1991. *termcap & terminfo, Third Edition*. O'Reilly & Associates, Sebastopol, Calif.

A book on termcap and terminfo.

Thompson, K. 1978. "UNIX Implementation," *Bell Syst. Technical Journal*, vol. 57, no. 6, pp. 1931–1946 (July-Aug.).

Describes some of the implementation details of Version 7.

Weinberger, P. J. 1982. "Making UNIX Operating Systems Safe for Databases," *Bell Syst. Technical Journal*, vol. 61, no. 9, pp. 2407–2422 (Nov.).

Describes some problems in implementing databases in early Unix systems.

Williams, T. 1989. "Session Management in System V Release 4," *Proceedings of the 1989 Winter USENIX Conference*, pp. 365–375, San Diego, Calif.

Describes the session architecture implemented in SVR4, which is part of POSIX.1. This includes process groups, job control, and controlling terminals. Also describes the security concerns of existing approaches.

X/Open. 1989. *X/Open Portability Guide*. Prentice-Hall, Englewood Cliffs, N.J.

This is a set of seven volumes covering the following areas: commands and utilities (Vol. 1), system interfaces and headers (Vol. 2), supplementary definitions (Vol. 3), programming languages (Vol. 4), data management (Vol. 5), window management (Vol. 6), networking services (Vol. 7).

# Index

The function subentries labeled "definition of" point to where the function prototype appears and, when applicable, to the source code for the function. Functions defined in the text that are used in later examples, such as the `set_fl` function in Program 3.5, are included in this index. The definitions of external functions that are part of the larger examples (Chapters 15–19) are also included in this index, to help in going through these larger examples. Also, significant functions and constants that occur in any of the examples in the text, such as `select` and `poll`, are also included in this index. Trivial functions that occur in almost every example, such as `close` and `exit`, are not referenced when they occur in examples.

# Now That You've Read This Book

## Attend a Course Based on This Material by the Same Author

### Advanced UNIX Programming

### Author: W. Richard Stevens

**Key Benefits:**
- Understand how to use the UNIX system services in C
- Learn the details of UNIX I/O, process control, and signals
- Gain detailed knowledge of interprocess communication in the UNIX Environment
- Use the process control and job control primitives
- Master advanced I/O techniques

**Course Overview:**
A UNIX system programmer knows the system calls for file I/O, process control, job control, signals, terminal I/O, and interprocess communication, and this course develops this knowledge base. The course is intended for applications programmers and system programmers, and is built on the new UNIX standards (POSIX.1) that most major vendors support today.

**Workshops:**
Using workstations, students will write, modify, and debug several C programs to illustrate all the features covered in the class, including:
- Creating Programs that Exploit Advanced I/O Mechanisms
- Using Process Control and Job Control Functions
- Mastering Terminal I/O
- Exploiting Dynamic Memory Allocation

***Also by the Same Author: UNIX System Workshop***

## For more information call Technology Exchange Company 1-800-662-4282 x889

❒ Please send me more information on this course.

❒ Please contact me about attending a public course (available in the USA only).

Location: ❒Boston ❒Chicago ❒Dallas ❒Denver ❒Indianapolis ❒Princeton, NJ ❒San Diego ❒Silicon Valley ❒Washington, DC

❒ Please contact me about on-site training.

Name_____ Title_____

Company_____ Department_____

Street Address_____ Mail Stop_____

City_____ State_____ Zip_____

Phone_____ Fax_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS MAIL PERMIT NO. 11 READING MA

POSTAGE WILL BE PAID BY ADDRESSEE

Technology Exchange Company
Route 128
One Jacob Way
Reading MA 01867-9985