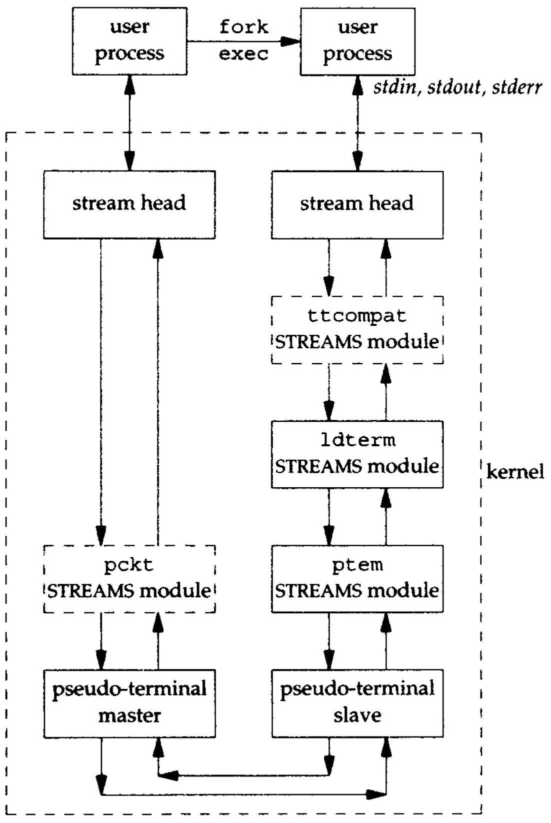

Figure 19.2 Arrangement of pseudo terminals under Solaris

Note that the three STREAMS modules above the slave are the same as the output from the program shown in Figure 14.18 for a network login. In Section 19.3.1, we show how to build this arrangement of STREAMS modules.

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.

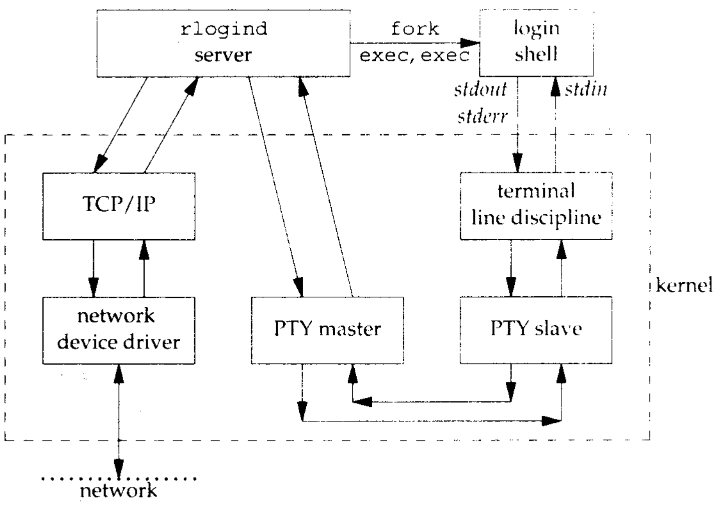

Figure 19.3 Arrangement of processes for `rlogind` server

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.

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 box. This implies that the process must be using some form of I/O multiplexing (Section 14.5), such as `select` or `poll`, or must be divided into two processes or threads.

`script` Program

The `script`(1) program that is supplied with most UNIX systems makes a copy in a file of everything that is input and output during a terminal session. The program 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.

While writing the first edition of this book, Rich Stevens used the `script` program to capture the output of the example programs. This avoided typographical errors that could have occurred if he had copied the program output by hand. The drawback to using `script`, however, is having to deal with control characters that are present in the script file.

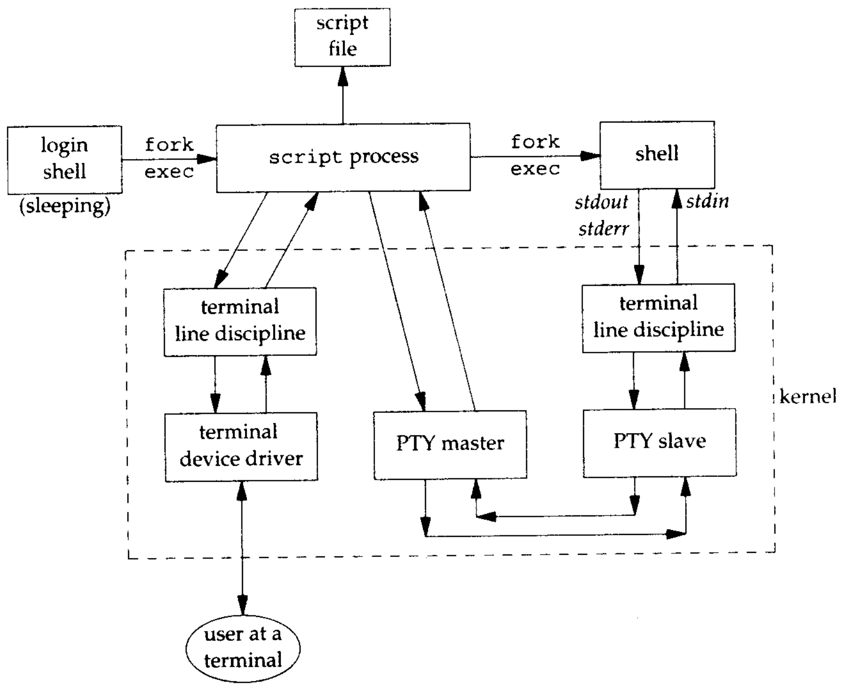

Figure 19.4 The `script` program

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.

`expect` Program

Pseudo terminals can be used to drive interactive programs in noninteractive modes. Numerous programs are hardwired to require a terminal to run. One example is the `passwd`(1) command, which requires that the user enter a password in response to a prompt.

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, 1994] 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 Figure 15.19, 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 15.16 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, to trick the coprocess into thinking that it is being driven from a terminal instead of from another process.

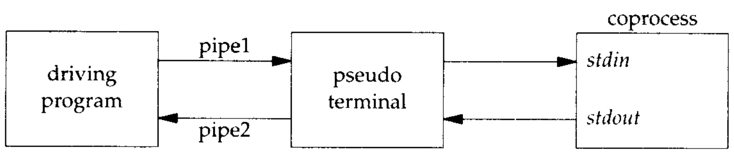

Figure 19.5 Driving a coprocess using a pseudo terminal

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.

The parent can obtain a pseudo terminal between itself and the coprocess in two ways. (The parent in this case could be either the program in Figure 15.18, which used two pipes to communicate with the coprocess, or the program in Figure 17.4, which used a single STREAMS 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 8,192 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.

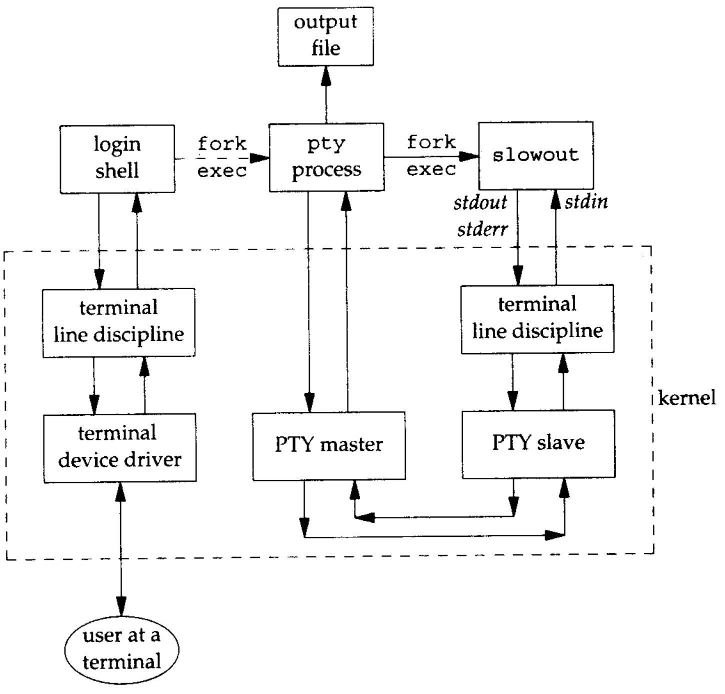

Figure 19.6 Running a slow output program using a pseudo terminal

19.3 Opening Pseudo-Terminal Devices

The way we open a pseudo-terminal device differs among platforms. The Single UNIX Specification includes several functions as XSI extensions in an attempt to unify the methods. These extensions are based on the functions originally provided to manage STREAMS-based pseudo terminals in System V Release 4.

The `posix_openpt` function is provided as a portable way to open an available pseudo-terminal master device.

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

Returns: file descriptor of next available PTY master if OK, −1 on error

The *oflag* argument is a bitmask that specifies how the master device is to be opened, similar to the same argument used with open(2). Not all open flags are supported, however. With `posix_openpt`, we can specify `O_RDWR` to open the master device for reading and writing, and we can specify `O_NOCTTY` to prevent the master device from becoming a controlling terminal for the caller. All other open flags result in unspecified behavior.

Before a slave pseudo-terminal device can be used, its permissions need to be set so that it is accessible to applications. The `grantpt` function does just this. It sets the user ID of the slave's device node to be the caller's real user ID and sets the node's group ID to an unspecified value, usually some group that has access to terminal devices. The permissions are set to allow read and write access to individual owners and write access to group owners (0620).

```
#include <stdlib.h>

int grantpt(int filedes);

int unlockpt(int filedes);

                              Both return: 0 on success, -1 on error
```

To change permission on the slave device node, `grantpt` might need to `fork` and `exec` a set-user-ID program (`/usr/lib/pt_chmod` on Solaris, for example). Thus, the behavior is unspecified if the caller is catching `SIGCHLD`.

The `unlockpt` function is used to grant access to the slave pseudo-terminal device, thereby allowing applications to open the device. By preventing others from opening the slave device, applications setting up the devices have an opportunity to initialize the slave and master devices properly before they can be used.

Note that in both `grantpt` and `unlockpt`, the file descriptor argument is the file descriptor associated with the master pseudo-terminal device.

The `ptsname` function is used to find the pathname of the slave pseudo-terminal device, given the file descriptor of the master. This allows applications to identify the slave independent of any particular conventions that might be followed by a given platform. Note that the name returned might be stored in static memory, so it can be overwritten on successive calls.

```
#include <stdlib.h>

char *ptsname(int filedes);

            Returns: pointer to name of PTY slave if OK, NULL on error
```

Figure 19.7 summarizes the pseudo-terminal functions in the Single UNIX Specification and indicates which functions are supported by the platforms discussed in this text.

On FreeBSD, `unlockpt` does nothing; the `O_NOCTTY` flag is defined only for compatibility with applications that call `posix_openpt`. FreeBSD does not allocate a controlling terminal as a side effect of opening a terminal device, so the `O_NOCTTY` flag has no effect.

Function	Description	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`grantpt`	Change permissions of slave PTY device.	•	•	•		•
`posix_openpt`	Open a master PTY device.	•	•			
`ptsname`	Return name of slave PTY device.	•	•	•		•
`unlockpt`	Allow slave PTY device to be opened.	•	•	•		•

Figure 19.7 XSI pseudo-terminal functions

Even though the Single UNIX Specification has tried to improve portability in this area, implementations are still catching up, as illustrated by Figure 19.7. Thus, 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.

```
#include "apue.h"

int ptym_open(char *pts_name, int pts_namesz);

                         Returns: file descriptor of PTY master if OK, −1 on error

int ptys_open(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.

The `ptym_open` function 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 the slave; if the call succeeds, the name of the corresponding slave is returned through *pts_name*. This name is then passed to `ptys_open`, which opens the slave device. The length of the buffer in bytes is passed in *pts_namesz* so that the `ptym_open` function doesn't copy a string that is longer than the buffer.

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 STREAMS-Based Pseudo Terminals

The details of the STREAMS implementation of pseudo terminals under Solaris are covered in Appendix C of Sun Microsystems [2002]. The next available PTY master device is accessed through a STREAMS *clone device*. A clone device is a special device that returns an unused device when it is opened. (STREAMS clone opens are discussed in detail in Rago [1993].)

The STREAMS-based PTY master clone device is `/dev/ptmx`. When we open it, the clone open routine automatically determines the first unused PTY master device and opens that unused device. (We'll see in the next section that, under BSD-based systems, we have to find the first unused PTY master ourselves.)

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <stropts.h>

int
ptym_open(char *pts_name, int pts_namesz)
{
    char    *ptr;
    int     fdm;

    /*
     * Return the name of the master device so that on failure
     * the caller can print an error message.  Null terminate
     * to handle case where strlen("/dev/ptmx") > pts_namesz.
     */
    strncpy(pts_name, "/dev/ptmx", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    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);
    }

    /*
     * Return name of slave.  Null terminate to handle
     * case where strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);            /* return fd of master */
}

int
ptys_open(char *pts_name)
{
    int     fds, setup;

    /*
     * The following open should allocate a controlling terminal.
     */
    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);

    /*
     * Check if stream is already set up by autopush facility.
     */
    if ((setup = ioctl(fds, I_FIND, "ldterm")) < 0) {
        close(fds);
        return(-6);
    }
    if (setup == 0) {
        if (ioctl(fds, I_PUSH, "ptem") < 0) {
            close(fds);
            return(-7);
        }
        if (ioctl(fds, I_PUSH, "ldterm") < 0) {
            close(fds);
            return(-8);
        }
        if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
            close(fds);
            return(-9);
        }
    }
    return(fds);
}
```

Figure 19.8 STREAMS-based pseudo-terminal open functions

We first open the clone device /dev/ptmx to get a 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. On Solaris, it changes the ownership of the slave to the real user ID, changes the group ownership to the group tty, and 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 grantpt executes the program /usr/lib/pt_chmod, which 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. Solaris follows the historical System V behavior: if the caller is a session leader that does not already have a controlling terminal, this call to 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 might need to push three STREAMS modules onto the slave's stream. Together, the pseudo terminal emulation module (ptem) and the terminal line discipline module (ldterm) act like a real terminal. The ttcompat module 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 the program shown in Figure 14.18), we push it onto the slave's stream.

The reason that we might not need to push these three modules is that they might be there already. The STREAMS system supports a facility known as *autopush*, which allows an administrator to configure a list of modules to be pushed onto a stream whenever a particular device is opened (see Rago [1993] for more details). We use the I_FIND ioctl command to see whether ldterm is already on the stream. If so, we assume that the stream has been configured by the autopush mechanism and avoid pushing the modules a second time.

The result of calling ptym_open and ptys_open is two file descriptors open in the calling process: one for the master and one for the slave.

19.3.2 BSD-Based Pseudo Terminals

Under BSD-based systems and Linux-based systems, we provide our own versions of the XSI functions, which we can optionally include in our library, depending on which functions (if any) are provided by the underlying platform.

In our version of posix_openpt, we have to determine the first available PTY master device. 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; 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*. On Linux, if the name of the PTY master is /dev/pty/m*XX*, then the name of the corresponding PTY slave is /dev/pty/s*XX*.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <grp.h>

#ifndef _HAS_OPENPT
int
posix_openpt(int oflag)
{
    int     fdm;
    char    *ptr1, *ptr2;
    char    ptm_name[16];

    strcpy(ptm_name, "/dev/ptyXY");
    /* array index:   0123456789   (for references in following code) */
    for (ptr1 = "pqrstuvwxyzPQRST"; *ptr1 != 0; ptr1++) {
        ptm_name[8] = *ptr1;
        for (ptr2 = "0123456789abcdef"; *ptr2 != 0; ptr2++) {
            ptm_name[9] = *ptr2;

            /*
             * Try to open the master.
             */
            if ((fdm = open(ptm_name, oflag)) < 0) {
                if (errno == ENOENT)    /* different from EIO */
                    return(-1);         /* out of pty devices */
                else
                    continue;           /* try next pty device */
            }
            return(fdm);        /* got it, return fd of master */
            }
        }
        errno = EAGAIN;
        return(-1);     /* out of pty devices */
}
#endif

#ifndef _HAS_PTSNAME
char *
ptsname(int fdm)
{
        static char pts_name[16];
        char        *ptm_name;

        ptm_name = ttyname(fdm);
        if (ptm_name == NULL)
            return(NULL);
        strncpy(pts_name, ptm_name, sizeof(pts_name));
        pts_name[sizeof(pts_name) - 1] = '\0';
        if (strncmp(pts_name, "/dev/pty/", 9) == 0)
            pts_name[9] = 's';  /* change /dev/pty/mXX to /dev/pty/sXX */
        else
            pts_name[5] = 't';  /* change "pty" to "tty" */
        return(pts_name);
}
#endif

#ifndef _HAS_GRANTPT
int
grantpt(int fdm)
{
        struct group    *grptr;
        int             gid;
        char            *pts_name;

        pts_name = ptsname(fdm);
        if ((grptr = getgrnam("tty")) != NULL)
            gid = grptr->gr_gid;
        else
            gid = -1;       /* group tty is not in the group file */

        /*
         * The following two calls won't work unless we're the superuser.
         */
        if (chown(pts_name, getuid(), gid) < 0)
            return(-1);
        return(chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP));
}
#endif

#ifndef _HAS_UNLOCKPT
int
unlockpt(int fdm)
```

```
{
    return(0); /* nothing to do */
}
#endif

int
ptym_open(char *pts_name, int pts_namesz)
{
    char    *ptr;
    int     fdm;

    /*
     * Return the name of the master device so that on failure
     * the caller can print an error message.  Null terminate
     * to handle case where string length > pts_namesz.
     */
    strncpy(pts_name, "/dev/ptyXX", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    if ((fdm = posix_openpt(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);
    }

    /*
     * Return name of slave.  Null terminate to handle
     * case where strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);            /* return fd of master */
}

int
ptys_open(char *pts_name)
{
    int fds;

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);
    return(fds);
}
```

Figure 19.9 Pseudo-terminal open functions for BSD and Linux

In our version of grantpt, 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 be placed into a set-user-ID root executable, similar to the way Solaris implements it.

The function ptys_open in Figure 19.9 simply opens the slave device. No other initialization is necessary. The open of the slave PTY under BSD-based systems does not have the side effect of allocating the device as the controlling terminal. In Section 19.4, we'll see how to allocate the controlling terminal under BSD-based systems.

Our version of posix_openpt 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).

19.3.3 Linux-Based Pseudo Terminals

Linux supports the BSD method for accessing pseudo terminals, so the same functions shown in Figure 19.9 will also work on Linux. However, Linux also supports a clone-style interface to pseudo terminals using /dev/ptmx (but this is not a STREAMS device). The clone interface requires extra steps to identify and unlock a slave device. The functions we can use to access these pseudo terminals on Linux are shown in Figure 19.10.

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

#ifndef _HAS_OPENPT
int
posix_openpt(int oflag)
{
    int     fdm;

    fdm = open("/dev/ptmx", oflag);
    return(fdm);
}
#endif

#ifndef _HAS_PTSNAME
char *
ptsname(int fdm)
{
    int         sminor;
    static char pts_name[16];

    if (ioctl(fdm, TIOCGPTN, &sminor) < 0)
        return(NULL);
    snprintf(pts_name, sizeof(pts_name), "/dev/pts/%d", sminor);
    return(pts_name);
```

```
}
#endif

#ifndef _HAS_GRANTPT
int
grantpt(int fdm)
{
    char            *pts_name;

    pts_name = ptsname(fdm);
    return(chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP));
}
#endif

#ifndef _HAS_UNLOCKPT
int
unlockpt(int fdm)
{
    int lock = 0;

    return(ioctl(fdm, TIOCSPTLCK, &lock));
}
#endif

int
ptym_open(char *pts_name, int pts_namesz)
{
    char    *ptr;
    int     fdm;

    /*
     * Return the name of the master device so that on failure
     * the caller can print an error message.  Null terminate
     * to handle case where string length > pts_namesz.
     */
    strncpy(pts_name, "/dev/ptmx", pts_namesz);
    pts_name[pts_namesz - 1] = '\0';

    fdm = posix_openpt(O_RDWR);
    if (fdm < 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);
    }
    /*
     * Return name of slave.  Null terminate to handle case
     * where strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);            /* return fd of master */
}

int
ptys_open(char *pts_name)
{
    int fds;

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-5);
    return(fds);
}
```

Figure 19.10 Pseudo-terminal open functions for Linux

On Linux, the PTY slave device is already owned by group `tty`, so all we need to do in `grantpt` is ensure that the permissions are correct.

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 the slave with a call to `fork`, establishing the child as a session leader with a controlling terminal.

```
#include "apue.h"
#include <termios.h>
#include <sys/ioctl.h>  /* find struct winsize on BSD systems */

pid_t pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
               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 non-null, 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 non-null, the system uses the referenced structure to initialize the terminal line discipline of the slave. If this pointer is null, the system sets the slave's `termios` structure to an implementation-defined initial state. Similarly, if the *slave_winsize* pointer is non-null, the referenced structure initializes the slave's window size. If this pointer is null, the `winsize` structure is normally initialized to 0.

Figure 19.11 shows the code for this function. It works on all four platforms described in this text, 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, 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 loses any association it might have had with its previous controlling terminal. Under Linux and Solaris, the slave becomes the controlling terminal of this new session when ptys_open is called. Under FreeBSD and Mac OS X, we have to call ioctl with an argument of TIOCSCTTY to allocate the controlling terminal. (Linux also supports the TIOCSCTTY ioctl command.) 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 execs 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 the process ID of the child. In the next section, we use the pty_fork function in the pty program.

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

pid_t
pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
         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, sizeof(pts_name))) < 0)
        err_sys("can't open master pty: %s, error %d", pts_name, fdm);

    if (slave_name != NULL) {
        /*
         * Return name of slave.  Null terminate to handle case
         * where strlen(pts_name) > slave_namesz.
         */
        strncpy(slave_name, pts_name, slave_namesz);
        slave_name[slave_namesz - 1] = '\0';
    }

    if ((pid = fork()) < 0) {
        return(-1);
    } else if (pid == 0) {        /* child */
        if (setsid() < 0)
            err_sys("setsid error");

        /*
         * System V acquires controlling terminal on open().
         */
        if ((fds = ptys_open(pts_name)) < 0)
            err_sys("can't open slave pty");
        close(fdm);      /* all done with master in child */

#if defined(TIOCSCTTY)
        /*
         * TIOCSCTTY is the BSD way to acquire a controlling terminal.
         */
        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 != STDIN_FILENO && fds != STDOUT_FILENO &&
          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 */
    }
}
```

Figure 19.11 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 (Figure 19.12) contains the main function. It calls the pty_fork function from the previous section.

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>  /* for struct winsize */
#endif

#ifdef LINUX
#define OPTSTR "+d:einv"
#else
#define OPTSTR "d:einv"
#endif

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;
    char            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, OPTSTR)) != 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, sizeof(slave_name),
          &orig_termios, &size);
    } else {
        pid = pty_fork(&fdm, slave_name, sizeof(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);

    /*
     * Also turn off NL to CR/NL mapping on output.
     */
    stermios.c_oflag &= ~(ONLCR);

    if (tcsetattr(fd, TCSANOW, &stermios) < 0)
        err_sys("tcsetattr error");
}
```

Figure 19.12 The `main` function for the `pty` program

In the next section, we'll look at the various command-line options when we examine different uses of the `pty` program. The `getopt` function helps us parse command-line arguments in a consistent manner. We'll discuss `getopt` in more detail in Chapter 21.

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. In this case, the parent also sets an exit handler to reset the terminal state when `exit` is called. We describe the `do_driver` function in the next section.

The parent then calls the function `loop` (Figure 19.13), which copies everything received from the standard input to the PTY master and everything from the PTY master to standard output. For variety, we have coded it in two processes this time, although a single process using `select`, `poll`, or multiple threads would also work.

```
#include "apue.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    buf[BUFFSIZE];

    if ((child = fork()) < 0) {
        err_sys("fork error");
    } else if (child == 0) {    /* child copies stdin to ptym */
        for ( ; ; ) {
            if ((nread = read(STDIN_FILENO, buf, BUFFSIZE)) < 0)
                err_sys("read error from stdin");
            else if (nread == 0)
                break;      /* EOF on stdin means we're done */
            if (writen(ptym, buf, nread) != nread)
                err_sys("writen error to master pty");
        }

        /*
         * We always terminate when we encounter an EOF on stdin,
         * but we notify the parent only 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, buf, BUFFSIZE)) <= 0)
            break;      /* signal caught, error, or EOF */
        if (writen(STDOUT_FILENO, buf, 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);
    /*
     * Parent returns to caller.
     */
}

/*
 * The child sends us SIGTERM when it gets EOF on the pty slave or
 * when read() fails.  We probably interrupted the read() of ptym.
 */
static void
sig_term(int signo)
{
    sigcaught = 1;      /* just set flag and return */
}
```

Figure 19.13 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 command-line options.

If our shell is the Korn shell, 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 Figure 18.16, we can run the pty program as follows:

```
$ who
sar  :0      Oct  5 18:07
sar  pts/0   Oct  5 18:07
sar  pts/1   Oct  5 18:07
sar  pts/2   Oct  5 18:07
sar  pts/3   Oct  5 18:07
sar  pts/4   Oct  5 18:07        pts/4 is the highest PTY currently in use
$ pty ttyname                    run program in Figure 18.16 from PTY
fd 0: /dev/pts/5                 pts/5 is the next available PTY
fd 1: /dev/pts/5
fd 2: /dev/pts/5
```

utmp File

In Section 6.8, we described the utmp file that records all users currently logged in to a UNIX system. The question is whether a user running a program on a pseudo terminal is considered logged in. 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 (which is considered to be a security hole), random programs that use pseudo terminals won't be able to write to this file.

Job Control Interaction

If we run a job-control shell under pty, it works normally. For example,

```
pty ksh
```

runs the Korn shell under pty. We can run programs under this new shell and use job control just as we do with 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 the job-control suspend character. At that point, the job-control character is echoed as ^Z and is ignored. Under earlier BSD-based systems, 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 19.14 shows the arrangement when pty cat is running.

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 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.

Historically, implementations have handled this condition differently. POSIX.1 says only that the SIGTSTP signal can't be delivered to the process.. Systems derived from 4.3BSD delivered SIGKILL instead, which the process can't even catch. In 4.4BSD, this behavior was changed to conform to POSIX.1. Instead of sending SIGKILL, the 4.4BSD kernel silently discards the SIGTSTP signal if it has the default disposition and is to be delivered to a process in an orphaned process group. Most current implementations follow this behavior.

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

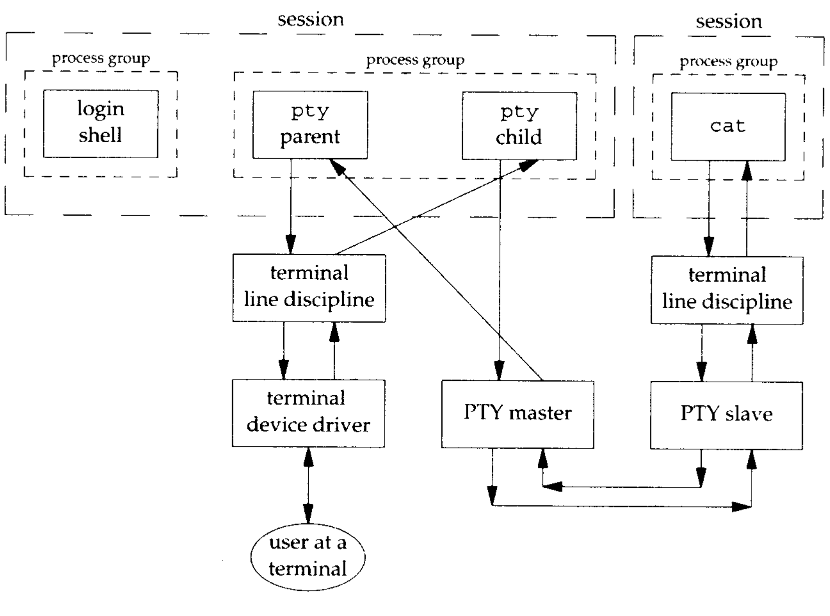

Figure 19.14 Process groups and sessions for `pty cat`

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). The reason is that 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 &
```

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 Figure 19.13 to exit 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 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 19.15 details these relationships.

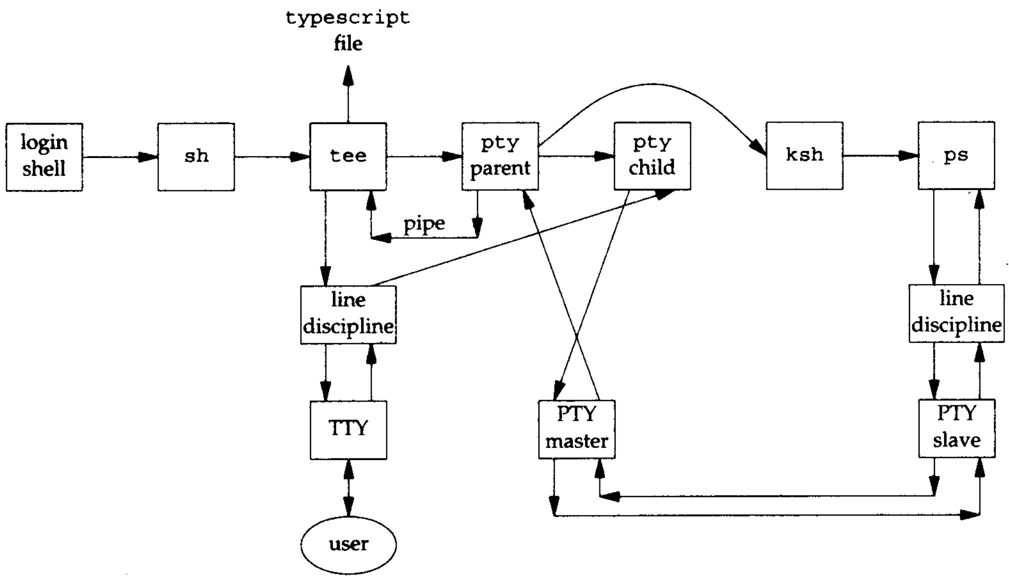

Figure 19.15 Arrangement of processes for script shell script

In this example, we assume that the SHELL variable is the Korn shell (probably /bin/ksh). As we mentioned earlier, script copies only 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 Figure 15.8, the coprocess couldn't use the standard I/O functions, because standard input and standard output do not refer to a terminal, so the standard I/O functions treat them as fully buffered. 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 19.16 shows the arrangement of processes when we run the coprocess with a pseudo terminal as its input and output. It is an expansion of Figure 19.5, showing all the process connections and data flow. The box labeled "driving program" is the program from Figure 15.8, with the `execl` changed as described previously.

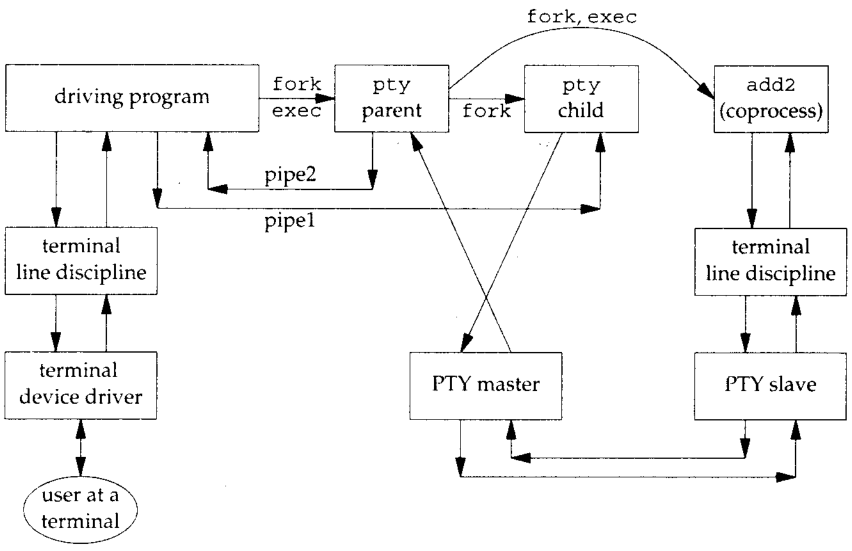

Figure 19.16 Running a coprocess with a pseudo terminal as its input and output

This example shows the need for the `-e` (no echo) option for the `pty` program. The `pty` program is not running interactively, because its standard input is not connected to a terminal. In Figure 19.12, 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 14.8 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 the program from Figure 15.8 returning less than a line. The solution was to not use the program shown in Figure 15.8, but to use the version of this program from Exercise 15.5 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 `read`s as required to obtain a complete line. The `while` loop in Figure 15.8 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, never looking at what it sends or what it gets back.

As an example, we can run the `telnet` command under `pty` talking directly to the remote host:

```
pty telnet 192.168.1.3
```

Doing this provides no benefit over just typing `telnet 192.168.1.3`, but we would like to run the `telnet` program from a script, perhaps to check some condition on the remote host. If the file `telnet.cmd` contains the four lines

```
sar
passwd
uptime
exit
```

the first line is the user name we use to log in to the remote host, the second line is the password, the third line is a command we'd like to run, and the fourth line terminates the session. But if we run this script as

```
pty -i < telnet.cmd telnet 192.168.1.3
```

it doesn't do what we want. What happens is that the contents of the file `telnet.cmd` are sent to the remote host before it has a chance to prompt us for an account name and password. When it turns off echoing to read the password, `login` uses the `tcsetattr` option, which discards any data already queued. Thus, the data we send is thrown away.

When we run the `telnet` program interactively, we wait for the remote host to prompt for a password before we type it, 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 the program in Figure 15.8, as we showed earlier, doesn't help, because the program in Figure 15.8 assumes that each line 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, the program in Figure 15.8

always sent a line to the coprocess before reading from it. This won't work when we want to read from the coprocess 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 ten 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 19.16, but in the current scenario, pty does the fork and exec of the driver process. Also, instead of two half-duplex pipes, we'll use a single bidirectional pipe between pty and the driver process.

Figure 19.17 shows the source for the do_driver function, which is called by the main function of pty (Figure 19.12) when the -d option is specified.

```
#include "apue.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");
        if (pipe[0] != STDIN_FILENO && pipe[0] != STDOUT_FILENO)
            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");
    if (pipe[1] != STDIN_FILENO && pipe[1] != STDOUT_FILENO)
        close(pipe[1]);

    /*
     * Parent returns, but with stdin and stdout connected
     * to the driver.
     */
}
```

Figure 19.17 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 it has its standard input and standard output connected to pty, the driver process 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 fewer than 50 lines of code.

19.7 Advanced Features

Pseudo terminals have some additional capabilities that we briefly mention here. These capabilities are further documented in Sun Microsystems [2002] and the BSD pty(4) manual page.

Packet Mode

Packet mode lets the PTY master learn of state changes in the PTY slave. On Solaris, this mode is enabled by pushing the STREAMS module pckt onto the PTY master side. We showed this optional module in Figure 19.2. On FreeBSD, Linux, and Mac OS X, this mode is enabled with the TIOCPKT ioctl command.

The details of packet mode differ between Solaris and the other platforms. Under Solaris, 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 nondata STREAMS messages. With the other platforms, 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 FreeBSD 5.2.1, Mac OS X 10.3, and Solaris 9 use the same command to enable and disable this feature, under Solaris the third argument to `ioctl` is an integer, whereas with FreeBSD and Mac OS X, it is a pointer to an integer. (Linux 2.4.22 doesn't support this command.)

When it sets this mode, the PTY master 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 Solaris 9, this is done with an `ioctl` of `TIOCSIGNAL`, with the third argument set to the signal number. With FreeBSD 5.2.1 and Mac OS X 10.3, the `ioctl` is `TIOCSIG`, and the third argument is a pointer to the integer signal number. (Linux 2.4.22 doesn't support this `ioctl` command either.)

19.8 Summary

We started this chapter with an overview of how to use pseudo terminals and a look at some use cases. We continued by examining the code required to open a pseudo terminal under the four platforms discussed in this text. 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 function `grantpt` from Figure 19.9 to invoke a set-user-ID program to change the ownership and protection of the PTY slave device on a BSD system (similar to what the Solaris version of the `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 (Figure 19.13) 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 19.14, identify which process groups are foreground and which are background, and identify the session leaders.

19.7 In Figure 19.14, in what order do the processes terminate when we type the end-of-file character? Verify this with process accounting, if possible.

19.8 The `script(1)` program normally adds to the beginning of the output file a line with the starting time, and to the end of the output file another line 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, the program should print that it did; 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 whether the slave process also terminates; if so, how does it terminate?

20

A Database Library

20.1 Introduction

During the early 1980s, the UNIX System was considered a hostile environment for running multiuser database systems. (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 byte-range locking. Many of these deficiencies were remedied, however. By the late 1980s, the UNIX System had evolved to provide a suitable environment for running reliable, multiuser database systems. Since then, numerous commercial firms have offered these types of database systems.

In this chapter, we develop a simple, multiuser database 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 UNIX System interface a database library requires and how that interface relates to the topics we've already covered (such as record—byte-range—locking, in Section 14.3).

20.2 History

One popular library of database functions in the UNIX System is 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 BSD releases, and was also provided in SVR4's BSD-compatibility library [AT&T 1990c]. The BSD developers extended the dbm library and called it ndbm. The ndbm library was included in BSD as well as in SVR4. The ndbm functions are standardized in the XSI extensions of the Single UNIX Specification.

Seltzer and Yigit [1991] provide a detailed history of the dynamic hashing algorithm used by the dbm library and other implementations of this library, including gdbm, the GNU version of the dbm library. Unfortunately, a basic limitation of all these implementations is that none allows concurrent updating of the database by multiple processes. These implementations provide no type of concurrency controls (such as record locking).

4.4BSD provided a new db(3) library that supports three forms of access: (a) record oriented, (b) hashing, and (c) a B-tree. Again, no form of concurrency was provided (as was plainly stated in the BUGS section of the db(3) manual page).

> Sleepycat Software (http://www.sleepycat.com) provides versions of the db library that do support concurrent access, locking, and transactions.

Most commercial database libraries do provide the concurrency controls required for multiple processes to update a database simultaneously. These systems typically use advisory locking, as we described in Section 14.3, but they often implement their own locking primitives to avoid the overhead of a system call to acquire an uncontested lock. These commercial systems usually implement their database using B+ trees [Comer 1979] or some dynamic hashing technique, such as linear hashing [Litwin 1980] or extendible hashing [Fagin et al. 1979].

Figure 20.1 summarizes the database libraries commonly found in the four operating systems described in this book. Note that on Linux, the gdbm library provides support for both dbm and ndbm functions.

Library	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
dbm			gdbm		•
ndbm	XSI	•	gdbm	•	•
db		•	•	•	•

Figure 20.1 Support for database libraries on various platforms

20.3 The Library

The library we develop in this chapter will be similar to the ndbm library, but we'll add the concurrency control mechanisms to allow multiple processes to update the same database at the same time. We 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 handle (an opaque pointer) representing the database. We'll pass this handle to the remaining database functions.

```
#include "apue_db.h"

DBHANDLE db_open(const char *pathname, int oflag, ... /* int mode */);
```

Returns: database handle if OK, NULL on error

```
void db_close(DBHANDLE 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.). The *mode* argument is used as the third argument to open (the file access permissions) if the database files are created.

When we're done with a database, we call db_close. It closes the index file and the 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 employee records with the same employee ID, for example.)

```
#include "apue_db.h"

int db_store(DBHANDLE db, const char *key, const char *data,
             int flag);
```

Returns: 0 if OK, nonzero on error (see following)

The *key* and *data* arguments 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.

The *flag* argument can be DB_INSERT (to insert a new record), DB_REPLACE (to replace an existing record), or DB_STORE (to either insert or replace, whichever is appropriate). These three constants are defined in the apue_db.h header. If we specify either DB_INSERT or DB_STORE and the record does not exist, a new record is inserted. If we specify either DB_REPLACE or DB_STORE and the record already exists, the existing record is replaced with the new record. If we specify DB_REPLACE and the record doesn't exist, we set errno to ENOENT and return −1 without adding the new record. If we specify DB_INSERT and the record already exists, no record is inserted. In this case, the return value is 1 to distinguish this from a normal error return (−1).

We can fetch any record from the database by specifying its *key*.

```
#include "apue_db.h"

char *db_fetch(DBHANDLE 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 "apue_db.h"

int db_delete(DBHANDLE 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 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 in a loop to read each sequential record.

```
#include "apue_db.h"

void db_rewind(DBHANDLE db);

char *db_nextrec(DBHANDLE db, char *key);

                        Returns: pointer to data if OK, NULL on end of file
```

If *key* is a non-null pointer, db_nextrec returns the key by copying it to the memory 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.

20.4 Implementation Overview

Database access libraries often 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 one with a suffix of .dat.

We store the key and the index as null-terminated character strings; they cannot contain arbitrary binary data. Some database systems store numerical data in a binary format (1, 2, or 4 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 a network has two systems that use different formats for storing binary integers, we need to handle this if we 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 a small cost for portability.

With db_store, only one record for each key is allowed. Some database systems allow a key to have multiple records and then provide a way to access all the records associated with a given key. Additionally, we have only a single index file, meaning that each data record can have only a single key (we don't support secondary keys).

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, all index files must 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 20.2 shows a general picture of the database implementation.

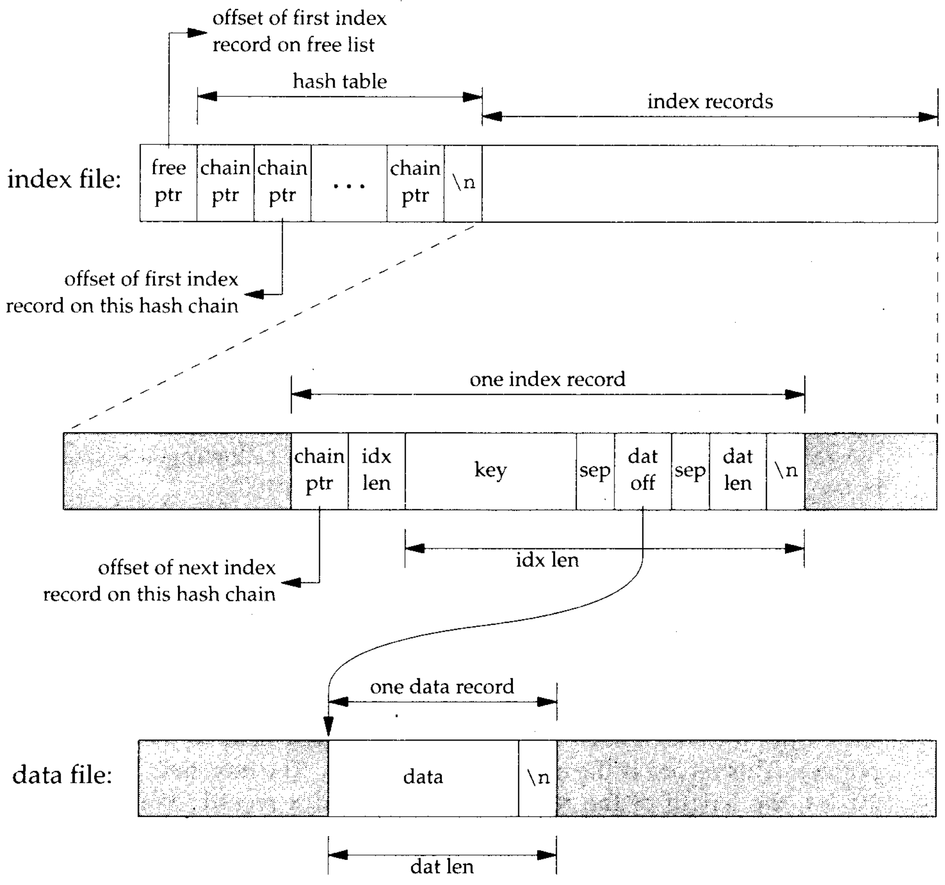

Figure 20.2 Arrangement of index file and data file

The index file consists of three portions: the free-list pointer, the hash table, and the index records. In Figure 20.2, all the fields called *ptr* are simply 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. The program in Figure 20.3 creates a new database and writes three records to it. Since we store all the fields in the database as ASCII characters, we can look at the actual index file and data file using any of the standard UNIX System tools:

```
$ ls -l db4.*
-rw-r--r--  1 sar        28 Oct 19 21:33 db4.dat
-rw-r--r--  1 sar        72 Oct 19 21:33 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
```

To keep this example small, we have set the size of each *ptr* field to four ASCII characters; 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 20.9, 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 more than 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*) and 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 System 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 6 bytes. (As with the index file, we automatically append a newline to

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>

int
main(void)
{
    DBHANDLE    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);
}
```

Figure 20.3 Create a database and write three records to it

each data record, so we can use the normal UNIX System tools with the file. This newline at the end is not returned to the caller 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.

Note 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 Figure 20.3. Since the O_TRUNC flag was specified for db_open, the index file and the 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 hashing. Dynamic hashing has the advantage that any data record can be located with only two disk accesses (see Litwin [1980] or Fagin et al. [1979] for details). B-trees have the advantage of traversing the database in (sorted) key order (something that we can't do with the db_nextrec function using a hash table.)

20.5 Centralized or Decentralized?

Given multiple processes accessing the same database, we can implement the functions in two ways:

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. Given adequate locking routines, the decentralized implementation is usually faster, because IPC is avoided. Figure 20.4 depicts the operation of the centralized approach.

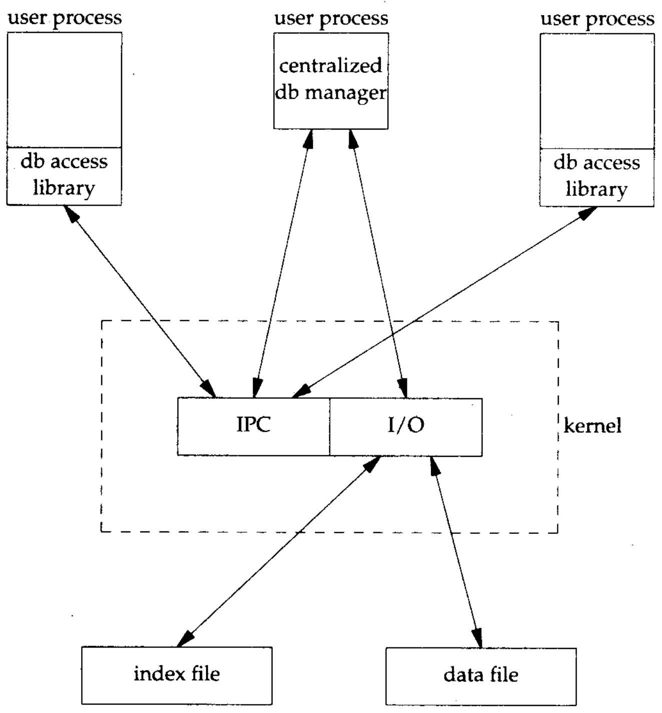

Figure 20.4 Centralized approach for database access

We purposely show the IPC going through the kernel, as most forms of message passing under the UNIX System operate this way. (Shared memory, as described in Section 15.9, 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 more difficult to do. We are usually at the mercy of the kernel's disk I/O scheduling policy and locking policy; that is, if three processes are waiting for a lock to become available, which process gets the lock next?

Another advantage of the centralized approach is that recovery is easier than with the decentralized approach. All the state information is in one place in the centralized approach, so if the database processes are killed, we have only one place to look to identify the outstanding transactions we need to resolve to restore the database to a consistent state.

The decentralized approach is shown in Figure 20.5. This is the design that we'll implement in this chapter.

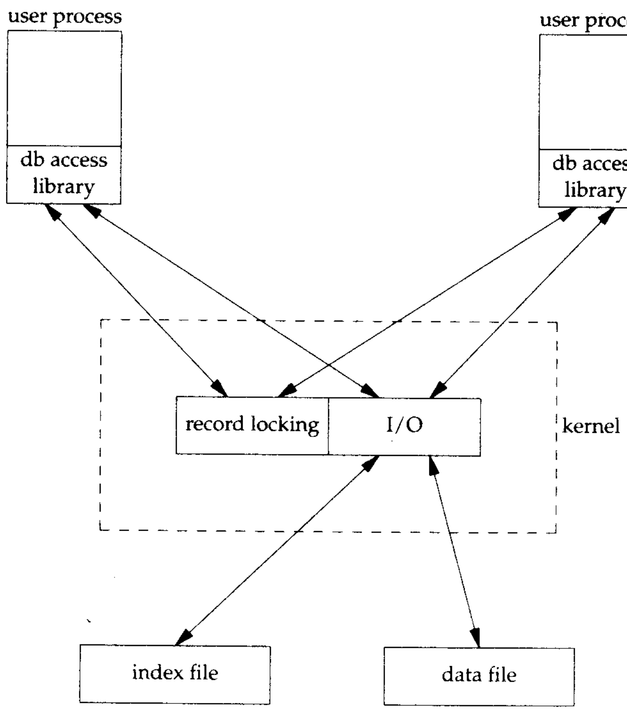

Figure 20.5 Decentralized approach for database access

The user processes that call the functions in the database library to perform I/O are considered cooperating processes, since they use byte-range locking to provide concurrent access.

20.6 Concurrency

We purposely chose a two-file implementation (an index file and a data file) because that is a common implementation technique. It requires us to handle the locking interactions of both files. But there are numerous ways to handle the locking of these two files.

Coarse-Grained 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-grained 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 System byte-range locking semantics to allow any number of readers at one time, but only one writer at a time. (Recall Figure 14.3.) 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-grained 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-Grained Locking

We enhance coarse-grained locking to allow more concurrency and call this *fine-grained locking*. We first require a reader or a 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 needing to access the free list (either `db_delete` or `db_store`) must obtain a write lock on the free list. Finally, whenever it appends a new record to the end of either the index file or the data file, `db_store` has to obtain a write lock on that portion of the file.

We expect fine-grained locking to provide more concurrency than coarse-grained locking. In Section 20.9, we'll show some actual measurements. In Section 20.8, we show the source code to our implementation of fine-grained locking and discuss the details of implementing locking. (Coarse-grained locking is merely 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. Although it is possible to use byte-range locking with the standard I/O library, careful handling of buffering is required. 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 16 of Date [2004] for additional details on concurrency.

20.7 Building the Library

The database library consists of two files: a public C header file and a C source file. We can build a static library using the commands

```
gcc -I../include -Wall -c db.c
ar rsv libapue_db.a db.o
```

Applications that want to link with libapue_db.a will also need to link with libapue.a, since we use some of our common functions in the database library.

If, on the other hand, we want to build a dynamic shared library version of the database library, we can use the following commands:

```
gcc -I../include -Wall -fPIC -c db.c
gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 \
    -L../lib -lapue -lc db.o
```

The resulting shared library, libapue_db.so.1, needs to be placed in a common directory where the dynamic linker/loader can find it. Alternatively, we can place it in a private directory and modify our LD_LIBRARY_PATH environment variable to include the private directory in the search path of the dynamic linker/loader.

The steps used to build shared libraries vary among platforms. Here, we have shown how to do it on a Linux system with the GNU C compiler.

20.8 Source Code

We start with the apue_db.h header shown first. This header is included by the library source code and all applications that call the library.

For the remainder of this text, we depart from the style of the previous examples in several ways. First, because the source code example is longer than usual, we number the lines. This makes it easier to link the discussion with the corresponding source code. Second, we place the description of the source code immediately below the source code on the same page.

This style was inspired by John Lions in his book documenting the UNIX Version 6 operating system source code [Lions 1977, 1996]. It simplifies the task of studying large amounts of source code.

Note that we do not bother to number blank lines. Although this departs from the normal behavior of such tools as pr(1), we have nothing interesting to say about blank lines.

```
 1  #ifndef _APUE_DB_H
 2  #define _APUE_DB_H

 3  typedef    void *  DBHANDLE;

 4  DBHANDLE  db_open(const char *, int, ...);
 5  void      db_close(DBHANDLE);
 6  char     *db_fetch(DBHANDLE, const char *);
 7  int       db_store(DBHANDLE, const char *, const char *, int);
 8  int       db_delete(DBHANDLE, const char *);
 9  void      db_rewind(DBHANDLE);
10  char     *db_nextrec(DBHANDLE, char *);

11  /*
12   * Flags for db_store().
13   */
14  #define DB_INSERT     1     /* insert new record only */
15  #define DB_REPLACE    2     /* replace existing record */
16  #define DB_STORE      3     /* replace or insert */

17  /*
18   * Implementation limits.
19   */
20  #define IDXLEN_MIN    6     /* key, sep, start, sep, length, \n */
21  #define IDXLEN_MAX 1024     /* arbitrary */
22  #define DATLEN_MIN    2     /* data byte, newline */
23  #define DATLEN_MAX 1024     /* arbitrary */

24  #endif /* _APUE_DB_H */
```

[1–3] We use the `_APUE_DB_H` symbol to ensure that the contents of the header file are included only once. The `DBHANDLE` type represents an active reference to the database and is used to isolate applications from the implementation details of the database. Compare this technique with the way the standard I/O library exposes the `FILE` structure to applications.

[4–10] Next, we declare the prototypes for the database library's public functions. Since this header is included by applications that want to use the library, we don't declare the prototypes for the library's private functions here.

[11–24] The legal flags that can be passed to the `db_store` function are defined next, followed by fundamental limits of the implementation. These limits can be changed, if desired, to support bigger databases.

The minimum index record length is specified by `IDXLEN_MIN`. This represents a 1-byte key, a 1-byte separator, a 1-byte starting offset, another 1-byte separator, a 1-byte length, and a terminating newline character. (Recall the format of an index record from Figure 20.2.) An index record will usually be larger than `IDXLEN_MIN` bytes, but this is the bare minimum size.

The next file is db.c, the C source file for the library. For simplicity, we include all functions in a single file. This has the advantage that we can hide private functions by declaring them static.

```
 1   #include "apue.h"
 2   #include "apue_db.h"
 3   #include <fcntl.h>      /* open & db_open flags */
 4   #include <stdarg.h>
 5   #include <errno.h>
 6   #include <sys/uio.h>    /* struct iovec */

 7   /*
 8    * Internal index file constants.
 9    * These are used to construct records in the
10    * index file and data file.
11    */
12   #define IDXLEN_SZ     4     /* index record length (ASCII chars) */
13   #define SEP         ':'     /* separator char in index record */
14   #define SPACE       ' '     /* space character */
15   #define NEWLINE     '\n'    /* newline character */

16   /*
17    * The following definitions are for hash chains and free
18    * list chain in the index file.
19    */
20   #define PTR_SZ        6     /* size of ptr field in hash chain */
21   #define PTR_MAX  999999     /* max file offset = 10**PTR_SZ - 1 */
22   #define NHASH_DEF   137     /* default hash table size */
23   #define FREE_OFF      0     /* free list offset in index file */
24   #define HASH_OFF PTR_SZ     /* hash table offset in index file */

25   typedef unsigned long  DBHASH; /* hash values */
26   typedef unsigned long  COUNT;  /* unsigned counter */
```

[1–6] We include apue.h because we use some of the functions from our private library. In turn, apue.h includes several standard header files, including <stdio.h> and <unistd.h>. We include <stdarg.h> because the db_open function uses the variable-argument functions declared by <stdarg.h>.

[7–26] The size of an index record is specified by IDXLEN_SZ. We use some characters, such as colon and newline, as delimiters in the database. We use the space character as "white out" when we delete a record.

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.

```
27  /*
28   * Library's private representation of the database.
29   */
30  typedef struct {
31    int    idxfd;  /* fd for index file */
32    int    datfd;  /* fd for data file */
33    char  *idxbuf; /* malloc'ed buffer for index record */
34    char  *datbuf; /* malloc'ed buffer for data record*/
35    char  *name;   /* name db was opened under */
36    off_t  idxoff; /* offset in index file of index record */
37                   /* key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
38    size_t idxlen; /* length of index record */
39                   /* excludes IDXLEN_SZ bytes at front of record */
40                   /* includes newline at end of index record */
41    off_t  datoff; /* offset in data file of data record */
42    size_t datlen; /* length of data record */
43                   /* includes newline at end */
44    off_t  ptrval; /* contents of chain ptr in index record */
45    off_t  ptroff; /* chain ptr offset pointing to this idx record */
46    off_t  chainoff; /* offset of hash chain for this index record */
47    off_t  hashoff;  /* offset in index file of hash table */
48    DBHASH nhash;    /* current hash table size */
49    COUNT  cnt_delok;    /* delete OK */
50    COUNT  cnt_delerr;   /* delete error */
51    COUNT  cnt_fetchok;  /* fetch OK */
52    COUNT  cnt_fetcherr; /* fetch error */
53    COUNT  cnt_nextrec;  /* nextrec */
54    COUNT  cnt_stor1;    /* store: DB_INSERT, no empty, appended */
55    COUNT  cnt_stor2;    /* store: DB_INSERT, found empty, reused */
56    COUNT  cnt_stor3;    /* store: DB_REPLACE, diff len, appended */
57    COUNT  cnt_stor4;    /* store: DB_REPLACE, same len, overwrote */
58    COUNT  cnt_storerr;  /* store error */
59  } DB;
```

[27–48] The DB structure is where we keep all the information for each open database. The DBHANDLE value that is returned by db_open and used by all the other functions is really just a pointer to one of these structures, but we hide that from the callers.

Since we store pointers and lengths as ASCII in the database, we convert these to numeric values and save them in the DB structure. We also save the hash table size even though it is fixed, just in case we decide to enhance the library to allow callers to specify the size when the database is created (see Exercise 20.7).

[49–59] The last ten fields in the DB structure count both successful and unsuccessful operations. If we want to analyze the performance of our database, we can write a function to return these statistics, but for now, we only maintain the counters.

```
60  /*
61   * Internal functions.
62   */
63  static DB       *_db_alloc(int);
64  static void      _db_dodelete(DB *);
65  static int       _db_find_and_lock(DB *, const char *, int);
66  static int       _db_findfree(DB *, int, int);
67  static void      _db_free(DB *);
68  static DBHASH    _db_hash(DB *, const char *);
69  static char     *_db_readdat(DB *);
70  static off_t     _db_readidx(DB *, off_t);
71  static off_t     _db_readptr(DB *, off_t);
72  static void      _db_writedat(DB *, const char *, off_t, int);
73  static void      _db_writeidx(DB *, const char *, off_t, int, off_t);
74  static void      _db_writeptr(DB *, off_t, off_t);

75  /*
76   * Open or create a database.  Same arguments as open(2).
77   */
78  DBHANDLE
79  db_open(const char *pathname, int oflag, ...)
80  {
81      DB          *db;
82      int         len, mode;
83      size_t      i;
84      char        asciiptr[PTR_SZ + 1],
85                  hash[(NHASH_DEF + 1) * PTR_SZ + 2];
86                      /* +2 for newline and null */
87      struct stat statbuff;

88      /*
89       * Allocate a DB structure, and the buffers it needs.
90       */
91      len = strlen(pathname);
92      if ((db = _db_alloc(len)) == NULL)
93          err_dump("db_open: _db_alloc error for DB");
```

[60–74] We have chosen to name all the user-callable (public) functions starting with db_ and all the internal (private) functions starting with _db_. The public functions were declared in the library's header file, apue_db.h. We declare the internal functions as static so they are visible only to functions residing in the same file (the file containing the library implementation).

[75–93] The db_open function has the same arguments as open(2). If the caller wants to create the database files, the optional third argument specifies the file permissions. The db_open function opens the index file and the data file, initializing the index file, if necessary. The function starts by calling _db_alloc to allocate and initialize a DB structure.

```
 94     db->nhash   = NHASH_DEF;/* hash table size */
 95     db->hashoff = HASH_OFF; /* offset in index file of hash table */
 96     strcpy(db->name, pathname);
 97     strcat(db->name, ".idx");

 98     if (oflag & O_CREAT) {
 99         va_list ap;

100         va_start(ap, oflag);
101         mode = va_arg(ap, int);
102         va_end(ap);

103         /*
104          * Open index file and data file.
105          */
106         db->idxfd = open(db->name, oflag, mode);
107         strcpy(db->name + len, ".dat");
108         db->datfd = open(db->name, oflag, mode);
109     } else {
110         /*
111          * Open index file and data file.
112          */
113         db->idxfd = open(db->name, oflag);
114         strcpy(db->name + len, ".dat");
115         db->datfd = open(db->name, oflag);
116     }

117     if (db->idxfd < 0 || db->datfd < 0) {
118         _db_free(db);
119         return(NULL);
120     }
```

[94–97] We continue to initialize the `DB` structure. The pathname passed in by the caller specifies the prefix of the database filenames. We append the suffix `.idx` to create the name for the database index file.

[98–108] If the caller wants to create the database files, we use the variable argument functions from `<stdarg.h>` to find the optional third argument. Then we use open to create and open the index file and data file. Note that the filename of the data file starts with the same prefix as the index file but has `.dat` as a suffix instead.

[109–116] If the caller doesn't specify the `O_CREAT` flag, then we're opening existing database files. In this case, we simply call open with two arguments.

[117–120] If we hit an error opening or creating either database file, we call `_db_free` to clean up the `DB` structure and then return `NULL` to the caller. If one open succeeded and one failed, `_db_free` will take care of closing the open file descriptor, as we shall see shortly.

```
121     if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
122         /*
123          * If the database was created, we have to initialize
124          * it.  Write lock the entire file so that we can stat
125          * it, check its size, and initialize it, atomically.
126          */
127         if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
128             err_dump("db_open: writew_lock error");

129         if (fstat(db->idxfd, &statbuff) < 0)
130             err_sys("db_open: fstat error");

131         if (statbuff.st_size == 0) {
132             /*
133              * We have to build a list of (NHASH_DEF + 1) chain
134              * ptrs with a value of 0.  The +1 is for the free
135              * list pointer that precedes the hash table.
136              */
137             sprintf(asciiptr, "%*d", PTR_SZ, 0);
```

[121–130] We encounter locking if the database is being created. Consider two processes trying to create the same database at about the same time. Assume that the first process calls `fstat` and is blocked by the kernel after `fstat` returns. The second process calls `db_open`, finds that the length of the index file is 0, and initializes the free list and hash chain. The second process then 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 macros `readw_lock`, `writew_lock`, and `un_lock` from Section 14.3.

[131–137] If the size of the index file is 0, we have just created it, so we need to initialize the free list and hash chain pointers it contains. Note that we use the format string `%*d` to convert a database pointer from an integer to an ASCII string. (We'll use this type of format again in `_db_writeidx` and `_db_writeptr`.) This format tells `sprintf` to take the `PTR_SZ` argument and use it as the minimum field width for the next argument, which is 0 in this instance (here we are initializing the pointers to 0, since we are creating a new database). This has the effect of forcing the string created to be at least `PTR_SZ` characters (padded on the left with spaces). In `_db_writeidx` and `_db_writeptr`, we will pass a pointer value instead of zero, but we will first verify that the pointer value isn't greater than `PTR_MAX`, to guarantee that every pointer string we write to the database occupies exactly `PTR_SZ` (6) characters.

```
138                 hash[0] = 0;
139                 for (i = 0; i < NHASH_DEF + 1; i++)
140                     strcat(hash, asciiptr);
141                 strcat(hash, "\n");
142                 i = strlen(hash);
143                 if (write(db->idxfd, hash, i) != i)
144                     err_dump("db_open: index file init write error");
145             }
146             if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
147                 err_dump("db_open: un_lock error");
148         }
149         db_rewind(db);
150         return(db);
151     }

152     /*
153      * Allocate & initialize a DB structure and its buffers.
154      */
155     static DB *
156     _db_alloc(int namelen)
157     {
158         DB      *db;

159         /*
160          * Use calloc, to initialize the structure to zero.
161          */
162         if ((db = calloc(1, sizeof(DB))) == NULL)
163             err_dump("_db_alloc: calloc error for DB");
164         db->idxfd = db->datfd = -1;             /* descriptors */

165         /*
166          * Allocate room for the name.
167          * +5 for ".idx" or ".dat" plus null at end.
168          */
169         if ((db->name = malloc(namelen + 5)) == NULL)
170             err_dump("_db_alloc: malloc error for name");
```

[138–151] We continue to initialize the newly created database. We build the hash table and write it to the index file. Then we unlock the index file, reset the database file pointers, and return a pointer to the DB structure as the opaque handle for the caller to use with the other database functions.

[152–164] The `_db_alloc` function is called by db_open to allocate storage for the DB structure, an index buffer, and a data buffer. We use `calloc` to allocate memory to hold the DB structure and ensure that it is initialized to all zeros. Since this has the side effect of setting the database file descriptors to zero, we need to reset them to −1 to indicate that they are not yet valid.

[165–170] We allocate space to hold the name of the database file. We use this buffer to create both filenames by changing the suffix to refer to either the index file or the data file, as we saw in db_open.

```
171     /*
172      * Allocate an index buffer and a data buffer.
173      * +2 for newline and null at end.
174      */
175     if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
176         err_dump("_db_alloc: malloc error for index buffer");
177     if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
178         err_dump("_db_alloc: malloc error for data buffer");
179     return(db);
180 }

181 /*
182  * Relinquish access to the database.
183  */
184 void
185 db_close(DBHANDLE h)
186 {
187     _db_free((DB *)h);  /* closes fds, free buffers & struct */
188 }

189 /*
190  * Free up a DB structure, and all the malloc'ed buffers it
191  * may point to.  Also close the file descriptors if still open.
192  */
193 static void
194 _db_free(DB *db)
195 {
196     if (db->idxfd >= 0)
197         close(db->idxfd);
198     if (db->datfd >= 0)
199         close(db->datfd);
```

[171–180] We allocate space for buffers for the index and data files. The buffer sizes are defined in apue_db.h. 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. Finally, we return a pointer to the DB structure that we allocated.

[181–188] The db_close function is a wrapper that casts a database handle to a DB structure pointer, passing it to _db_free to release any resources and free the DB structure.

[189–199] The _db_free function is called by db_open if an error occurs while opening the index file or data file and is also called by db_close when an application is done using the database. If the file descriptor for the database index file is valid, we close it. The same is done with the file descriptor for the data file. (Recall that when we allocate a new DB structure in _db_alloc, we initialize each file descriptor to −1. If we are unable to open one of the database files, the corresponding file descriptor will still be set to −1, and we will avoid trying to close it.)

```
200      if (db->idxbuf != NULL)
201          free(db->idxbuf);
202      if (db->datbuf != NULL)
203          free(db->datbuf);
204      if (db->name != NULL)
205          free(db->name);
206      free(db);
207  }

208  /*
209   * Fetch a record.  Return a pointer to the null-terminated data.
210   */
211  char *
212  db_fetch(DBHANDLE h, const char *key)
213  {
214      DB      *db = h;
215      char    *ptr;

216      if (_db_find_and_lock(db, key, 0) < 0) {
217          ptr = NULL;             /* error, record not found */
218          db->cnt_fetcherr++;
219      } else {
220          ptr = _db_readdat(db);  /* return pointer to data */
221          db->cnt_fetchok++;
222      }

223      /*
224       * Unlock the hash chain that _db_find_and_lock locked.
225       */
226      if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
227          err_dump("db_fetch: un_lock error");
228      return(ptr);
229  }
```

[200–207] Next, we free any dynamically-allocated buffers. We can safely pass a null pointer to `free`, so we don't need to check the value of each buffer pointer beforehand, but we do so anyway because we consider it better style to free only those objects that we allocated. (Not all deallocator functions are as forgiving as `free`.) Finally, we free the memory backing the `DB` structure.

[208–218] The `db_fetch` function is used to read a record given its key. We first try to find the record by calling `_db_find_and_lock`. If the record can't be found, we set the return value (`ptr`) to `NULL` and increment the count of unsuccessful record searches. Because `_db_find_and_lock` returns with the database index file locked, we can't return until we unlock it.

[219–229] If the record is found, we call `_db_readdat` to read the corresponding data record and increment the count of the successful record searches. Before returning, we unlock the index file by calling `un_lock`. Then we return a pointer to the record found (or `NULL` if the record wasn't found).

```
230  /*
231   * Find the specified record.  Called by db_delete, db_fetch,
232   * and db_store.  Returns with the hash chain locked.
233   */
234  static int
235  _db_find_and_lock(DB *db, const char *key, int writelock)
236  {
237      off_t   offset, nextoffset;

238      /*
239       * Calculate the hash value for this key, then calculate the
240       * byte offset of corresponding chain ptr in hash table.
241       * This is where our search starts.  First we calculate the
242       * offset in the hash table for this key.
243       */
244      db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
245      db->ptroff = db->chainoff;

246      /*
247       * We lock the hash chain here.  The caller must unlock it
248       * when done.  Note we lock and unlock only the first byte.
249       */
250      if (writelock) {
251          if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
252              err_dump("_db_find_and_lock: writew_lock error");
253      } else {
254          if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
255              err_dump("_db_find_and_lock: readw_lock error");
256      }

257      /*
258       * Get the offset in the index file of first record
259       * on the hash chain (can be 0).
260       */
261      offset = _db_readptr(db, db->ptroff);
```

[230–237] The `_db_find_and_lock` function is used internally by the library to find a record given its key. We set the `writelock` parameter to a nonzero value if we want to acquire a write lock on the index file while we search for the record. If we set `writelock` to zero, we read-lock the index file while we search it.

[238–256] We prepare to traverse a hash chain in `_db_find_and_lock`. We convert the key into a hash value, which we use to calculate the starting address of the hash chain in the file (`chainoff`). We wait for the lock to be granted before going through the hash chain. Note that we lock only the first byte in the start of the hash chain. This increases concurrency by allowing multiple processes to search different hash chains at the same time.

[257–261] We call `_db_readptr` to read the first pointer in the hash chain. If this returns zero, the hash chain is empty.

```
262     while (offset != 0) {
263         nextoffset = _db_readidx(db, offset);
264         if (strcmp(db->idxbuf, key) == 0)
265             break;        /* found a match */
266         db->ptroff = offset; /* offset of this (unequal) record */
267         offset = nextoffset; /* next one to compare */
268     }
269     /*
270      * offset == 0 on error (record not found).
271      */
272     return(offset == 0 ? -1 : 0);
273 }

274 /*
275  * Calculate the hash value for a key.
276  */
277 static DBHASH
278 _db_hash(DB *db, const char *key)
279 {
280     DBHASH      hval = 0;
281     char        c;
282     int         i;

283     for (i = 1; (c = *key++) != 0; i++)
284         hval += c * i;      /* ascii char times its 1-based index */
285     return(hval % db->nhash);
286 }
```

[262–268] In the `while` loop, we go through each index record on the hash chain, comparing keys. We call `_db_readidx` to read each index record. It populates the `idxbuf` field with the key of the current record. If `_db_readidx` returns zero, we've reached the last entry in the chain.

[269–273] If `offset` is zero after the loop, we've reached the end of a hash chain without finding a matching key, so we return −1. Otherwise, we found a match (and exited the loop with the `break` statement), so we return success (0). In this case, the `ptroff` field contains the address of the previous index record, `datoff` contains the address of the data record, and `datlen` contains the size of the data record. As we make our way through the hash chain, we save 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 to delete the current record.

[274–286] `_db_hash` calculates the hash value for a given key. It multiplies each ASCII character times its 1-based index and divides the result by the number of hash table entries. The remainder from the division is the hash value for this key. Recall that the number of hash table entries is 137, which is a prime number. According to Knuth [1998], prime hashes generally provide good distribution characteristics.

```
287   /*
288    * Read a chain ptr field from anywhere in the index file:
289    * the free list pointer, a hash table chain ptr, or an
290    * index record chain ptr.
291    */
292   static off_t
293   _db_readptr(DB *db, off_t offset)
294   {
295     char    asciiptr[PTR_SZ + 1];

296     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
297         err_dump("_db_readptr: lseek error to ptr field");
298     if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
299         err_dump("_db_readptr: read error of ptr field");
300     asciiptr[PTR_SZ] = 0;        /* null terminate */
301     return(atol(asciiptr));
302   }

303   /*
304    * Read the next index record.  We start at the specified offset
305    * in the index file.  We read the index record into db->idxbuf
306    * and replace the separators with null bytes.  If all is OK we
307    * set db->datoff and db->datlen to the offset and length of the
308    * corresponding data record in the data file.
309    */
310   static off_t
311   _db_readidx(DB *db, off_t offset)
312   {
313     ssize_t           i;
314     char             *ptr1, *ptr2;
315     char             asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
316     struct iovec     iov[2];
```

[287–302] `_db_readptr` reads any one of three different chain pointers: (a) the pointer at the beginning of the index file that points to the first index record on the free list, (b) the pointers in the hash table that point to the first index record on each hash chain, and (c) 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). We convert the pointer from ASCII to a long integer before returning it. No locking is done by this function; that is up to the caller.

[303–316] The `_db_readidx` function is used to read the record at the specified offset from the index file. On success, the function will return the offset of the next record in the list. In this case, the function will populate several fields in the `DB` structure: `idxoff` contains the offset of the current record in the index file, `ptrval` contains the offset of the next index entry in the list, `idxlen` contains the length of the current index record, `idxbuf` contains the actual index record, `datoff` contains the offset of the record in the data file, and `datlen` contains the length of the data record.

```
317      /*
318       * Position index file and record the offset.  db_nextrec
319       * calls us with offset==0, meaning read from current offset.
320       * We still need to call lseek to record the current offset.
321       */
322      if ((db->idxoff = lseek(db->idxfd, offset,
323        offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
324          err_dump("_db_readidx: lseek error");

325      /*
326       * Read the ascii chain ptr and the ascii length at
327       * the front of the index record.  This tells us the
328       * remaining size of the index record.
329       */
330      iov[0].iov_base = asciiptr;
331      iov[0].iov_len  = PTR_SZ;
332      iov[1].iov_base = asciilen;
333      iov[1].iov_len  = IDXLEN_SZ;
334      if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
335          if (i == 0 && offset == 0)
336              return(-1);      /* EOF for db_nextrec */
337          err_dump("_db_readidx: readv error of index record");
338      }

339      /*
340       * This is our return value; always >= 0.
341       */
342      asciiptr[PTR_SZ] = 0;          /* null terminate */
343      db->ptrval = atol(asciiptr); /* offset of next key in chain */

344      asciilen[IDXLEN_SZ] = 0;      /* null terminate */
345      if ((db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
346        db->idxlen > IDXLEN_MAX)
347          err_dump("_db_readidx: invalid length");
```

[317–324] We start by seeking to the index file offset provided by the caller. We record the offset in the DB structure, so even if the caller wants to read the record at the current file offset (by setting `offset` to 0), we still need to call `lseek` to determine the current offset. Since an index record will never be stored at offset 0 in the index file, we can safely overload the value of 0 to mean "read from the current offset."

[325–338] 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.

[339–347] We convert the offset of the next record to an integer and store it in the `ptrval` field (this will be used as the return value for this function). Then we convert the length of the index record into an integer and save it in the `idxlen` field.

```
348     /*
349      * Now read the actual index record.  We read it into the key
350      * buffer that we malloced when we opened the database.
351      */
352     if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
353         err_dump("_db_readidx: read error of index record");
354     if (db->idxbuf[db->idxlen-1] != NEWLINE)    /* sanity check */
355         err_dump("_db_readidx: missing newline");
356     db->idxbuf[db->idxlen-1] = 0;     /* replace newline with null */

357     /*
358      * Find the separators in the index record.
359      */
360     if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
361         err_dump("_db_readidx: missing first separator");
362     *ptr1++ = 0;                  /* replace SEP with null */

363     if ((ptr2 = strchr(ptr1, SEP)) == NULL)
364         err_dump("_db_readidx: missing second separator");
365     *ptr2++ = 0;                  /* replace SEP with null */

366     if (strchr(ptr2, SEP) != NULL)
367         err_dump("_db_readidx: too many separators");

368     /*
369      * Get the starting offset and length of the data record.
370      */
371     if ((db->datoff = atol(ptr1)) < 0)
372         err_dump("_db_readidx: starting offset < 0");
373     if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
374         err_dump("_db_readidx: invalid length");
375     return(db->ptrval);      /* return offset of next key in chain */
376 }
```

[348–356] We read the variable-length index record into the `idxbuf` field in the `DB` structure. The record should be terminated with a newline, which we replace with a null byte. If the index file is corrupt, we terminate and drop core by calling `err_dump`.

[357–367] We separate the index record into three fields: the key, the offset of the corresponding data record, and the length of the data record. The `strchr` function finds the first occurrence of the specified character in the given string. Here we look for the character that separates fields in the record (`SEP`, which we define to be a colon).

[368–376] We convert the data record offset and length into integers and store them in the `DB` structure. Then we return the offset of the next record in the hash chain. Note that we do not read the data record. That is left to the caller. In `db_fetch`, for example, we don't read the data record until `_db_find_and_lock` has read the index record that matches the key that we're looking for.

```
377  /*
378   * Read the current data record into the data buffer.
379   * Return a pointer to the null-terminated data buffer.
380   */
381  static char *
382  _db_readdat(DB *db)
383  {
384      if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
385          err_dump("_db_readdat: lseek error");
386      if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
387          err_dump("_db_readdat: read error");
388      if (db->datbuf[db->datlen-1] != NEWLINE)    /* sanity check */
389          err_dump("_db_readdat: missing newline");
390      db->datbuf[db->datlen-1] = 0; /* replace newline with null */
391      return(db->datbuf);     /* return pointer to data record */
392  }

393  /*
394   * Delete the specified record.
395   */
396  int
397  db_delete(DBHANDLE h, const char *key)
398  {
399      DB      *db = h;
400      int     rc = 0;         /* assume record will be found */

401      if (_db_find_and_lock(db, key, 1) == 0) {
402          _db_dodelete(db);
403          db->cnt_delok++;
404      } else {
405          rc = -1;            /* not found */
406          db->cnt_delerr++;
407      }
408      if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
409          err_dump("db_delete: un_lock error");
410      return(rc);
411  }
```

[377–392] The `_db_readdat` function populates the `datbuf` field in the `DB` structure with the contents of the data record, expecting that the `datoff` and `datlen` fields have been properly initialized already.

[393–411] The `db_delete` function is used to delete a record given its key. We use `_db_find_and_lock` to determine whether the record exists in the database. If it does, we call `_db_dodelete` to do the work needed to delete the record. The third argument to `_db_find_and_lock` controls whether the chain is read-locked or write-locked. Here we are requesting a write lock, since we will potentially change the list. Since `_db_find_and_lock` returns with the lock still held, we need to unlock it, regardless of whether the record was found.

```
412   /*
413    * Delete the current record specified by the DB structure.
414    * This function is called by db_delete and db_store, after
415    * the record has been located by _db_find_and_lock.
416    */
417   static void
418   _db_dodelete(DB *db)
419   {
420     int     i;
421     char    *ptr;
422     off_t   freeptr, saveptr;

423     /*
424      * Set data buffer and key to all blanks.
425      */
426     for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
427         *ptr++ = SPACE;
428     *ptr = 0;   /* null terminate for _db_writedat */
429     ptr = db->idxbuf;
430     while (*ptr)
431         *ptr++ = SPACE;

432     /*
433      * We have to lock the free list.
434      */
435     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
436         err_dump("_db_dodelete: writew_lock error");

437     /*
438      * Write the data record with all blanks.
439      */
440     _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

[412–431] The `_db_dodelete` function does all the work necessary to delete a record from the database. (This function is also called by `db_store`.) 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.

[432–440] We call `writew_lock` to write-lock 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.

We write the all-blank data record by calling `_db_writedat`. Note that there is no need for `_db_writedat` to lock the data file in this case. 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.

```
441     /*
442      * Read the free list pointer.  Its value becomes the
443      * chain ptr field of the deleted index record.  This means
444      * the deleted record becomes the head of the free list.
445      */
446     freeptr = _db_readptr(db, FREE_OFF);
447     /*
448      * Save the contents of index record chain ptr,
449      * before it's rewritten by _db_writeidx.
450      */
451     saveptr = db->ptrval;
452     /*
453      * Rewrite the index record.  This also rewrites the length
454      * of the index record, the data offset, and the data length,
455      * none of which has changed, but that's OK.
456      */
457     _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);
458     /*
459      * Write the new free list pointer.
460      */
461     _db_writeptr(db, FREE_OFF, db->idxoff);
462     /*
463      * Rewrite the chain ptr that pointed to this record being
464      * deleted.  Recall that _db_find_and_lock sets db->ptroff to
465      * point to this chain ptr.  We set this chain ptr to the
466      * contents of the deleted record's chain ptr, saveptr.
467      */
468     _db_writeptr(db, db->ptroff, saveptr);
469     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
470         err_dump("_db_dodelete: un_lock error");
471 }
```

[441–461] We read the free-list pointer and then update the index record so that its next record pointer is set to the first record on the free list. (If the free list was empty, this new chain pointer is 0.) We have already cleared the key. Then we update the free-list pointer with the offset of the index record we are deleting. This means that the free list is handled on a last-in, first-out basis; that is, deleted records are added to the front of the free list (although we remove entries from the free list on a first-fit basis).

We don't have a separate free list for each file. When we add a deleted index record to the free list, the index record still points to the deleted data record. There are better ways to do this, in exchange for added complexity.

[462–471] We update the previous record in the hash chain to point to the record after the one we are deleting, thus removing the deleted record from the hash chain. Finally, we unlock the free list.

```
472   /*
473    * Write a data record.  Called by _db_dodelete (to write
474    * the record with blanks) and db_store.
475    */
476   static void
477   _db_writedat(DB *db, const char *data, off_t offset, int whence)
478   {
479     struct iovec    iov[2];
480     static char     newline = NEWLINE;

481     /*
482      * If we're appending, we have to lock before doing the lseek
483      * and write to make the two an atomic operation.  If we're
484      * overwriting an existing record, we don't have to lock.
485      */
486     if (whence == SEEK_END) /* we're appending, lock entire file */
487         if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
488             err_dump("_db_writedat: writew_lock error");

489     if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
490         err_dump("_db_writedat: lseek error");
491     db->datlen = strlen(data) + 1;  /* datlen includes newline */

492     iov[0].iov_base = (char *) data;
493     iov[0].iov_len  = db->datlen - 1;
494     iov[1].iov_base = &newline;
495     iov[1].iov_len  = 1;
496     if (writev(db->datfd, &iov[0], 2) != db->datlen)
497         err_dump("_db_writedat: writev error of data record");

498     if (whence == SEEK_END)
499         if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
500             err_dump("_db_writedat: un_lock error");
501   }
```

[472–491] We call _db_writedat to write a data record. When we delete a record, we use _db_writedat to overwrite the record with blanks; _db_writedat doesn't need to lock the data file, because db_delete has write-locked the hash chain for this record. Thus, no other process could be reading or writing this particular data record. When we cover db_store later in this section, we'll encounter the case in which _db_writedat is appending to the data file and has to lock it.

We seek to the location where we want to write the data record. The amount to write is the record size plus 1 byte for the terminating newline we add.

[492–501] We set up the iovec array and call 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, so we write the newline from a separate buffer. If we are appending a record to the file, we release the lock we acquired earlier.

```
502   /*
503    * Write an index record.  _db_writedat is called before
504    * this function to set the datoff and datlen fields in the
505    * DB structure, which we need to write the index record.
506    */
507   static void
508   _db_writeidx(DB *db, const char *key,
509                off_t offset, int whence, off_t ptrval)
510   {
511       struct iovec    iov[2];
512       char            asciiptrlen[PTR_SZ + IDXLEN_SZ +1];
513       int             len;
514       char            *fmt;

515       if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
516           err_quit("_db_writeidx: invalid ptr: %d", ptrval);
517       if (sizeof(off_t) == sizeof(long long))
518           fmt = "%s%c%lld%c%d\n";
519       else
520           fmt = "%s%c%ld%c%d\n";
521       sprintf(db->idxbuf, fmt, key, SEP, db->datoff, SEP, db->datlen);
522       if ((len = strlen(db->idxbuf)) < IDXLEN_MIN || len > IDXLEN_MAX)
523           err_dump("_db_writeidx: invalid length");
524       sprintf(asciiptrlen, "%*ld%*d", PTR_SZ, ptrval, IDXLEN_SZ, len);

525       /*
526        * If we're appending, we have to lock before doing the lseek
527        * and write to make the two an atomic operation.  If we're
528        * overwriting an existing record, we don't have to lock.
529        */
530       if (whence == SEEK_END)      /* we're appending */
531           if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
532             SEEK_SET, 0) < 0)
533               err_dump("_db_writeidx: writew_lock error");
```

[502–524] The `_db_writeidx` function is called to write an index record. After validating the next pointer in the chain, we create the index record and store the second half of it in `idxbuf`. We need the size of this portion of the index record to create the first half of the index record, which we store in the local variable `asciiptrlen`.

Note that we select the format string passed to `sprintf` based on the size of the `off_t` data type. Even a 32-bit system can provide 64-bit file offsets, so we can't make any assumptions about the size of the `off_t` data type.

[525–533] 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. In this case, the caller has write-locked the hash chain, so no additional locking is required.

```
534    /*
535     * Position the index file and record the offset.
536     */
537    if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
538        err_dump("_db_writeidx: lseek error");

539    iov[0].iov_base = asciiptrlen;
540    iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
541    iov[1].iov_base = db->idxbuf;
542    iov[1].iov_len  = len;
543    if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
544        err_dump("_db_writeidx: writev error of index record");

545    if (whence == SEEK_END)
546        if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
547          SEEK_SET, 0) < 0)
548            err_dump("_db_writeidx: un_lock error");
549  }

550  /*
551   * Write a chain ptr field somewhere in the index file:
552   * the free list, the hash table, or in an index record.
553   */
554  static void
555  _db_writeptr(DB *db, off_t offset, off_t ptrval)
556  {
557    char    asciiptr[PTR_SZ + 1];

558    if (ptrval < 0 || ptrval > PTR_MAX)
559        err_quit("_db_writeptr: invalid ptr: %d", ptrval);
560    sprintf(asciiptr, "%*ld", PTR_SZ, ptrval);

561    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
562        err_dump("_db_writeptr: lseek error to ptr field");
563    if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
564        err_dump("_db_writeptr: write error of ptr field");
565  }
```

[534–549] We seek to the location where we want to write the index record and save this offset in the `idxoff` field of the `DB` structure. Since we built the index record in two separate buffers, we use `writev` to store it in the index file. If we were appending to the file, we release the lock we acquired before seeking. This makes the seek and the write an atomic operation from the perspective of concurrently running processes appending new records to the same database.

[550–565] `_db_writeptr` is used to write a chain pointer to the index file. We validate that the chain pointer is within bounds, then convert it to an ASCII string. We seek to the specified offset in the index file and write the pointer.

```
566  /*
567   * Store a record in the database.  Return 0 if OK, 1 if record
568   * exists and DB_INSERT specified, -1 on error.
569   */
570  int
571  db_store(DBHANDLE h, const char *key, const char *data, int flag)
572  {
573      DB      *db = h;
574      int     rc, keylen, datlen;
575      off_t   ptrval;

576      if (flag != DB_INSERT && flag != DB_REPLACE &&
577        flag != DB_STORE) {
578          errno = EINVAL;
579          return(-1);
580      }
581      keylen = strlen(key);
582      datlen = strlen(data) + 1;      /* +1 for newline at end */
583      if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
584          err_dump("db_store: invalid data length");

585      /*
586       * _db_find_and_lock calculates which hash table this new record
587       * goes into (db->chainoff), regardless of whether it already
588       * exists or not. The following calls to _db_writeptr change the
589       * hash table entry for this chain to point to the new record.
590       * The new record is added to the front of the hash chain.
591       */
592      if (_db_find_and_lock(db, key, 1) < 0) { /* record not found */
593          if (flag == DB_REPLACE) {
594              rc = -1;
595              db->cnt_storerr++;
596              errno = ENOENT;     /* error, record does not exist */
597              goto doreturn;
598          }
```

[566–584] We use `db_store` to add a record to the database. We first validate the flag value we are passed. Then we make sure that the length of the data record is valid. If it isn't, we drop core and exit. This is OK for an example, but if we were building a production-quality library, we'd return an error status instead, which would give the application a chance to recover.

[585–598] We call `_db_find_and_lock` to see if the record already exists. It is OK if the record doesn't exist and either `DB_INSERT` or `DB_STORE` is specified, or if the record already exists and either `DB_REPLACE` or `DB_STORE` is specified. If we're replacing an existing record, that implies that the keys are identical but that the data records probably differ. Note that the final argument to `_db_find_and_lock` specifies that the hash chain must be write-locked, as we will probably be modifying this hash chain.

```
599         /*
600          * _db_find_and_lock locked the hash chain for us; read
601          * the chain ptr to the first index record on hash chain.
602          */
603         ptrval = _db_readptr(db, db->chainoff);

604         if (_db_findfree(db, keylen, datlen) < 0) {
605             /*
606              * Can't find an empty record big enough. Append the
607              * new record to the ends of the index and data files.
608              */
609             _db_writedat(db, data, 0, SEEK_END);
610             _db_writeidx(db, key, 0, SEEK_END, ptrval);

611             /*
612              * db->idxoff was set by _db_writeidx.  The new
613              * record goes to the front of the hash chain.
614              */
615             _db_writeptr(db, db->chainoff, db->idxoff);
616             db->cnt_stor1++;
617         } else {
618             /*
619              * Reuse an empty record. _db_findfree removed it from
620              * the free list and set both db->datoff and db->idxoff.
621              * Reused record goes to the front of the hash chain.
622              */
623             _db_writedat(db, data, db->datoff, SEEK_SET);
624             _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
625             _db_writeptr(db, db->chainoff, db->idxoff);
626             db->cnt_stor2++;
627         }
```

[599–603] After we call _db_find_and_lock, the code divides into four cases. In the first two, no record was found, so we are adding a new record. We read the offset of the first entry on the hash list.

[604–616] Case 1: we call _db_findfree to search the free list for a deleted record with the same size key and same size data. If no such record is found, we have to append the new record to the ends of the index and data files. We call _db_writedat to write the data part, _db_writeidx to write the index part, and _db_writeptr to place the new record on the front of the hash chain. We increment a count (cnt_stor1) of the number of times we executed this case to allow us to characterize the behavior of the database.

[617–627] Case 2: _db_findfree found an empty record with the correct sizes and removed it from the free list (we'll see the implementation of _db_findfree shortly). We write the data and index portions of the new record and add the record to the front of the hash chain as we did in case 1. The cnt_stor2 field counts how many times we've executed this case.

```
628     } else {                            /* record found */
629         if (flag == DB_INSERT) {
630             rc = 1;     /* error, record already in db */
631             db->cnt_storerr++;
632             goto doreturn;
633         }

634         /*
635          * We are replacing an existing record.  We know the new
636          * key equals the existing key, but we need to check if
637          * the data records are the same size.
638          */
639         if (datlen != db->datlen) {
640             _db_dodelete(db);   /* delete the existing record */

641             /*
642              * Reread the chain ptr in the hash table
643              * (it may change with the deletion).
644              */
645             ptrval = _db_readptr(db, db->chainoff);

646             /*
647              * Append new index and data records to end of files.
648              */
649             _db_writedat(db, data, 0, SEEK_END);
650             _db_writeidx(db, key, 0, SEEK_END, ptrval);

651             /*
652              * New record goes to the front of the hash chain.
653              */
654             _db_writeptr(db, db->chainoff, db->idxoff);
655             db->cnt_stor3++;
656         } else {
```

[628–633] Now we reach the two cases in which a record with the same key already exists in the database. If the caller isn't replacing the record, we set the return code to indicate that a record exists, increment the count of the number of store errors, and jump to the end of the function, where we handle the common return logic.

[634–656] Case 3: an existing record is being replaced, and the length of the new data record differs from the length of the existing one. We call `_db_dodelete` to delete the existing record. Recall that this places the deleted record at the head of the free list. Then we append the new record to the ends of the data and index files by calling `_db_writedat` and `_db_writeidx`. (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`. The `cnt_stor3` counter in the `DB` structure records the number of times we've executed this case.

```
657               /*
658                * Same size data, just replace data record.
659                */
660               _db_writedat(db, data, db->datoff, SEEK_SET);
661               db->cnt_stor4++;
662           }
663       }
664       rc = 0;     /* OK */

665   doreturn: /* unlock hash chain locked by _db_find_and_lock */
666       if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
667           err_dump("db_store: un_lock error");
668       return(rc);
669   }

670   /*
671    * Try to find a free index record and accompanying data record
672    * of the correct sizes.  We're only called by db_store.
673    */
674   static int
675   _db_findfree(DB *db, int keylen, int datlen)
676   {
677       int     rc;
678       off_t   offset, nextoffset, saveoffset;

679       /*
680        * Lock the free list.
681        */
682       if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
683           err_dump("_db_findfree: writew_lock error");

684       /*
685        * Read the free list pointer.
686        */
687       saveoffset = FREE_OFF;
688       offset = _db_readptr(db, saveoffset);
```

[657–663] Case 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 simply rewrite the data record and increment the counter (cnt_stor4) for this case.

[664–669] In the normal case, we set the return code to indicate success and fall through to the common return logic. We unlock the hash chain that was locked as a result of calling _db_find_and_lock and return to the caller.

[670–688] The _db_findfree function tries to find a free index record and associated data record of the specified sizes. We need to write-lock the free list to avoid interfering with any other processes using the free list. After locking the free list, we get the pointer address at the head of the list.

```
689     while (offset != 0) {
690         nextoffset = _db_readidx(db, offset);
691         if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
692             break;      /* found a match */
693         saveoffset = offset;
694         offset = nextoffset;
695     }

696     if (offset == 0) {
697         rc = -1;    /* no match found */
698     } else {
699         /*
700          * Found a free record with matching sizes.
701          * The index record was read in by _db_readidx above,
702          * which sets db->ptrval.  Also, saveoffset points to
703          * the chain ptr that pointed to this empty record on
704          * the free list.  We set this chain ptr to db->ptrval,
705          * which removes the empty record from the free list.
706          */
707         _db_writeptr(db, saveoffset, db->ptrval);
708         rc = 0;

709         /*
710          * Notice also that _db_readidx set both db->idxoff
711          * and db->datoff.  This is used by the caller, db_store,
712          * to write the new index record and data record.
713          */
714     }

715     /*
716      * Unlock the free list.
717      */
718     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
719         err_dump("_db_findfree: un_lock error");
720     return(rc);
721 }
```

[689–695] The `while` loop in `_db_findfree` goes through the free list, looking for a record with matching key and data sizes. 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.

[696–714] If we can't find an available record of the requested key and data sizes, we set the return code to indicate failure. Otherwise, we write the previous record's chain pointer to point to the next chain pointer value of the record we have found. This removes the record from the free list.

[715–721] Once we are done with the free list, we release the write lock. Then we return the status to the caller.

```
722   /*
723    * Rewind the index file for db_nextrec.
724    * Automatically called by db_open.
725    * Must be called before first db_nextrec.
726    */
727   void
728   db_rewind(DBHANDLE h)
729   {
730     DB      *db = h;
731     off_t   offset;

732     offset = (db->nhash + 1) * PTR_SZ;  /* +1 for free list ptr */

733     /*
734      * We're just setting the file offset for this process
735      * to the start of the index records; no need to lock.
736      * +1 below for newline at end of hash table.
737      */
738     if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
739         err_dump("db_rewind: lseek error");
740   }

741   /*
742    * Return the next sequential record.
743    * We just step our way through the index file, ignoring deleted
744    * records.  db_rewind must be called before this function is
745    * called the first time.
746    */
747   char *
748   db_nextrec(DBHANDLE h, char *key)
749   {
750     DB      *db = h;
751     char    c;
752     char    *ptr;
```

[722–740] The db_rewind function is used to reset the database to "the beginning;" we set the file offset for the index file to point to the first record in the index file (immediately following the hash table). (Recall the structure of the index file from Figure 20.2.)

[741–752] The db_nextrec function returns the next record in the database. The return value is a pointer to the data buffer. If the caller provides a non-null value for the key parameter, the corresponding key is copied to this address. The caller is responsible for allocating a buffer big enough to store the key. A buffer whose size is IDXLEN_MAX bytes is large enough to hold any key.

Records are returned sequentially, in the order that they happen to be stored in the database file. Thus, the records are not sorted by key value. Also, because we do not follow the hash chains, we can come across records that have been deleted, but we will not return these to the caller.

```
753      /*
754       * We read lock the free list so that we don't read
755       * a record in the middle of its being deleted.
756       */
757      if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
758          err_dump("db_nextrec: readw_lock error");

759      do {
760          /*
761           * Read next sequential index record.
762           */
763          if (_db_readidx(db, 0) < 0) {
764              ptr = NULL;     /* end of index file, EOF */
765              goto doreturn;
766          }

767          /*
768           * Check if key is all blank (empty record).
769           */
770          ptr = db->idxbuf;
771          while ((c = *ptr++) != 0  &&  c == SPACE)
772              ;   /* skip until null byte or nonblank */
773      } while (c == 0);   /* loop until a nonblank key is found */

774      if (key != NULL)
775          strcpy(key, db->idxbuf);    /* return key */
776      ptr = _db_readdat(db);  /* return pointer to data buffer */
777      db->cnt_nextrec++;

778  doreturn:
779      if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
780          err_dump("db_nextrec: un_lock error");
781      return(ptr);
782  }
```

[753–758] We first need to read-lock the free list so that no other processes can remove a record while we are reading it.

[759–773] We call `_db_readidx` to read the next record. We pass in an offset of 0 to tell `_db_readidx` to continue reading from the current offset. Since we are reading the index file sequentially, we can come across records that have been deleted. We want to return only valid records, so we skip any record whose key is all spaces (recall that `_db_dodelete` clears a key by setting it to all spaces).

[774–782] When we find a valid key, we copy it to the caller's buffer if one was supplied. Then we read the data record and set the return value to point to the internal buffer containing the data record. We increment a statistics counter, unlock the free list, and return the pointer to the data record.

The normal use of db_rewind and db_nextrec 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.

If the database is being modified while db_nextrec is called from a loop, the records returned by db_nextrec are simply 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 used 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, thereby avoiding any interaction with _db_dodelete and _db_findfree.

Before we conclude our study of the db.c source file, we need to describe the locking when new index records or data records are appended to the end of the 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, indicating 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 20.3.)

20.9 Performance

To test the database library and to obtain some timing measurements of the database access patterns of typical applications, 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), forks the number of child processes, and waits for all the children to terminate. Each child performs the following steps.

1. Write *nrec* records to the database.
2. Read the *nrec* records back by key value.
3. Perform the following loop *nrec* × 5 times.
   a. Read a random record.
   b. Every 37 times through the loop, delete a random record.
   c. Every 11 times through the loop, insert a new record and read the record back.
   d. 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.
4. Delete all the records that this child wrote. Every time a record is deleted, ten random records are looked up.

The 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 20.6.

Operation	Count
`db_store`, `DB_INSERT`, no empty record, appended	678
`db_store`, `DB_INSERT`, empty record reused	164
`db_store`, `DB_REPLACE`, different data length, appended	97
`db_store`, `DB_REPLACE`, equal data length	109
`db_store`, record not found	19
`db_fetch`, record found	8,114
`db_fetch`, record not found	732
`db_delete`, record found	842
`db_delete`, record not found	110

Figure 20.6 Typical count of operations performed by each child when *nrec* is 500

We performed about ten 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. 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-grained locking versus fine-grained locking and to compare the three 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 20.8, which we've called fine-grained locking. The second version changed the locking calls to implement coarse-grained locking, as described in Section 20.6. 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-grained locking and coarse-grained 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-grained locking.)

All the timing tests in this section were done on a SPARC system running Solaris 9.

Single-Process Results

Figure 20.7 shows the results when only a single child process ran, with an *nrec* of 500, 1,000, and 2,000.

	No locking			Advisory locking						Mandatory locking		
				Coarse-grained locking			Fine-grained locking			Fine-grained locking		
nrec	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock
500	0.42	0.89	1.31	0.42	1.17	1.59	0.41	1.04	1.45	0.46	1.49	1.95
1,000	1.51	3.89	5.41	1.64	4.13	5.78	1.63	4.12	5.76	1.73	6.34	8.07
2,000	3.91	10.06	13.98	4.09	10.30	14.39	4.03	10.63	14.66	4.47	16.21	20.70

Figure 20.7 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 six columns under "Advisory locking" are almost equal for each row. This makes sense because for a single process, there is no difference between coarse-grained locking and fine-grained locking.

Comparing no locking versus advisory locking, we see that adding the locking calls adds between 2 percent and 31 percent 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 20.7 is that mandatory locking adds between 43 percent and 54 percent to the system CPU time, compared to advisory locking. Since the number of locking calls is the same for advisory fine-grained locking and mandatory fine-grained locking, the additional system call overhead must be in the reads and writes.

The final test 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. Different errors occurred every time the test program was run. This illustrates a classic race condition: 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-grained locking and fine-grained locking. As we said earlier, intuitively, we expect fine-grained locking to provide additional concurrency, since there is less time that portions of the database are locked from other processes. Figure 20.8 shows the results for an *nrec* of 500, varying the number of children from 1 to 12.

	Advisory locking							Mandatory locking			
	Coarse-grained locking			Fine-grained locking			Δ	Fine-grained locking			Δ
#Proc	User	Sys	Clock	User	Sys	Clock	Clock	User	Sys	Clock	Percent
1	0.41	1.00	1.42	0.41	1.05	1.47	0.05	0.47	1.40	1.87	33
2	1.10	2.81	3.92	1.11	2.80	3.92	0.00	1.15	4.06	5.22	45
3	2.17	5.27	7.44	2.19	5.18	7.37	-0.07	2.31	7.67	9.99	48
4	3.36	8.55	11.91	3.26	8.67	11.94	0.03	3.51	12.69	16.20	46
5	4.72	13.08	17.80	4.99	12.64	17.64	-0.16	4.91	19.21	24.14	52
6	6.45	17.96	24.42	6.83	17.29	24.14	-0.28	7.03	26.59	33.66	54
7	8.46	23.12	31.62	8.67	22.96	31.65	0.03	9.25	35.47	44.74	54
8	10.83	29.68	40.55	11.00	29.39	40.41	-0.14	11.67	45.90	57.63	56
9	13.35	36.81	50.23	13.43	36.28	49.76	-0.47	14.45	58.02	72.49	60
10	16.35	45.28	61.66	16.09	44.10	60.23	-1.43	17.43	70.90	88.37	61
11	18.97	54.24	73.24	19.13	51.70	70.87	-2.37	20.62	84.98	105.69	64
12	22.92	63.54	86.51	22.94	61.28	84.29	-2.22	24.41	101.68	126.20	66

Figure 20.8 Comparison of various locking techniques, *nrec* = 500

All times are in seconds and 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-grained locking to advisory fine-grained locking. This is the measurement of how much concurrency we obtain by going from coarse-grained locking to fine-grained locking. On the system used for these tests, coarse-grained locking is roughly the same until we have more than seven processes. Even after seven processes, the decrease in clock time using fine-grained locking isn't that great (less than 3 percent), which makes us wonder whether the additional code required to implement fine-grained locking is worth the effort.

We would like the clock time to decrease from coarse-grained to fine-grained locking, as it eventually does, but we expect the system time to remain higher for fine-grained locking, for any number of processes. The reason we expect this is that with fine-grained locking, we are issuing more `fcntl` calls than with coarse-grained locking. If we total the number of `fcntl` calls in Figure 20.6 for coarse-grained locking and fine-grained locking, we have an average of 21,730 for coarse-grained locking and 25,292 for fine-grained locking. (To get these numbers, realize that each operation in Figure 20.6 requires two calls to `fcntl` for coarse-grained locking and that the first three calls to `db_store` along with record deletion [record found] each require four calls to `fcntl` for fine-grained locking.) We expect this increase of 16 percent in the number of calls to `fcntl` to result in an increased system time for fine-grained locking.

Therefore, the slight decrease in system time for fine-grained locking, when the number of processes exceeds seven, is puzzling.

The reason for the decrease is that with coarse-grained locking, we hold locks for longer periods of time, thus increasing the likelihood that other processes will block on a lock. With fine-grained locking, the locking is done over shorter intervals, so there is less chance that processes will block. If we analyze the system behavior running 12 database processes, we will see that there is three times as much process switching with coarse-grained locking as with fine-grained locking. This means that processes block on locks less often with fine-grained locking.

The final column, labeled "Δ percent," is the percentage increase in the system CPU time from advisory fine-grained locking to mandatory fine-grained locking. These percentages verify what we saw in Figure 20.7, that mandatory locking adds significantly (between 33 percent and 66 percent) 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-grained and mandatory fine-grained locking), we expect the user CPU times to be the same across any row.

The values in the first row of Figure 20.8 are similar to those for an *nrec* of 500 in Figure 20.7. This corresponds to our expectation.

Figure 20.9 is a graph of the data from Figure 20.8 for advisory fine-grained locking. We plot the clock time as the number of processes goes from 1 to 12. We also plot the user CPU time divided by the number of processes and the system CPU time divided by the number of processes.

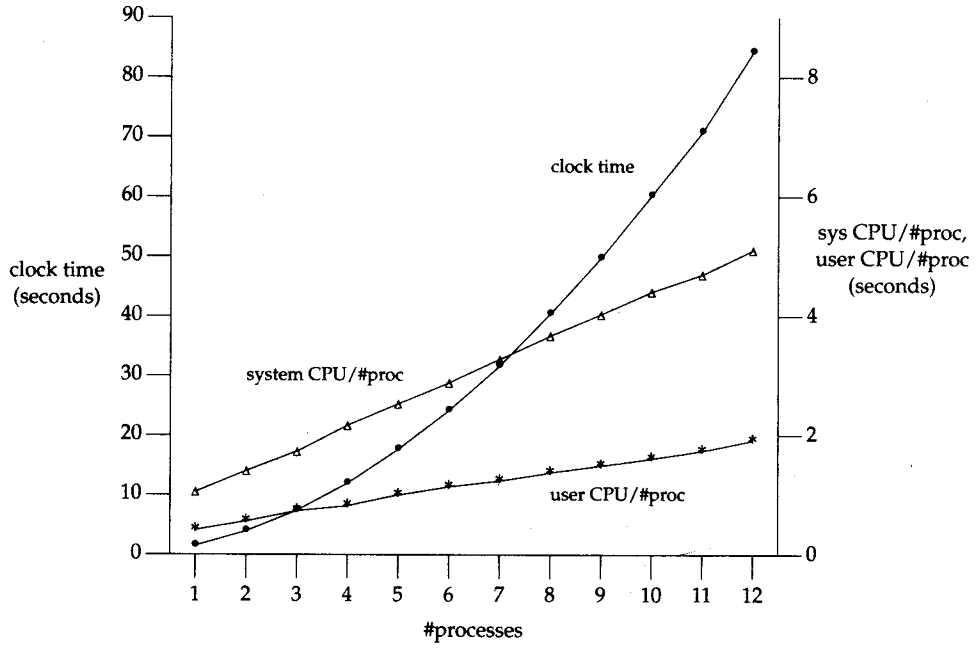

Figure 20.9 Values from Figure 20.8 for advisory fine-grained locking

Note that both CPU times, divided by the number of processes, are linear but that the plot of the clock time is nonlinear. The probable reason is the added amount of CPU time used by the operating system to switch between processes 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 that there are more records in the database. Each hash chain is getting longer, so it takes the `_db_find_and_lock` function longer, on the average, to find a record.

20.10 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 numbers of processes using no locking, advisory locking (fine-grained and coarse-grained), and mandatory locking. We saw that advisory locking adds less than 10 percent to the clock time over no locking and that mandatory locking adds another 33 percent to 66 percent over advisory locking.

Exercises

20.1 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?

20.2 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`.)

20.3 At the end of Section 20.8, 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?

20.4 How would you integrate the `fsync` function into this database library?

20.5 In `db_store`, we write the data record before the index record. What happens if you do it in the opposite order?

20.6 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 `_db_hash` adequate?

20.7 Modify the database functions so that the number of hash chains in the index file can be specified when the database is created.

20.8 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 accessed via NFS. Does the record locking provided by the database library still work?

21

Communicating with a Network Printer

21.1 Introduction

We now develop a program that can communicate with a network printer. These printers are connected to multiple computers via Ethernet and often support PostScript files as well as plaintext files. Applications generally use the Internet Printing Protocol (IPP) to communicate with these printers, although some support alternate communication protocols.

We are about to describe two programs: a print spooler daemon that sends jobs to a printer and a command to submit print jobs to the spooler daemon. Since the print spooler has to do multiple things (communicate with clients submitting jobs, communicate with the printer, read files, scan directories, etc.), this gives us a chance to use many of the functions from earlier chapters. For example, we use threads (Chapters 11 and 12) to simplify the design of the print spooler and sockets (Chapter 16) to communicate between the program used to schedule a file to be printed and the print spooler, and also between the print spooler and the network printer.

21.2 The Internet Printing Protocol

IPP specifies the communication rules for building network-based printing systems. By embedding an IPP server inside a printer with an Ethernet card, the printer can service requests from many computer systems. These computer systems need not be located on the same physical network, however. IPP is built on top of standard Internet protocols, so any computer that can create a TCP/IP connection to the printer can submit a print job.

Specifically, IPP is built on top of HTTP, the Hypertext Transfer Protocol (Section 21.3). HTTP, in turn, is built on top of TCP/IP. The structure of an IPP message is shown in Figure 21.1.

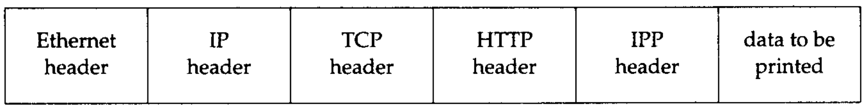

Figure 21.1 Structure of an IPP message

IPP is a request–response protocol. A client sends a request message to a server, and the server answers with a response message. The IPP header contains a field that indicates the requested operation. Operations are defined to submit print jobs, cancel print jobs, get job attributes, get printer attributes, pause and restart the printer, place a job on hold, and release a held job.

Figure 21.2 shows the structure of an IPP message header. The first 2 bytes are the IPP version number. For protocol version 1.1, each byte has a value of 1. For a protocol request, the next 2 bytes contain a value identifying the requested operation. For a protocol response, these 2 bytes contain a status code instead.

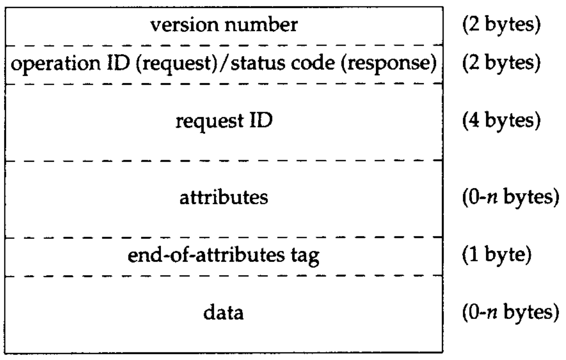

Figure 21.2 Structure of an IPP header

The next 4 bytes contain an integer identifying the request. Optional attributes follow this, terminated by an end-of-attributes tag. Any data that might be associated with the request follows immediately after the end-of-attributes tag.

In the header, integers are stored as signed, two's-complement, binary values in big-endian byte order (i.e., network byte order). Attributes are stored in groups. Each group starts with a single byte identifying the group. Within each group, an attribute is generally represented as a 1-byte tag, followed by a 2-byte name length, followed by the name of the attribute, followed by a 2-byte value length, and finally the value itself. The value can be encoded as a string, a binary integer, or a more complex structure, such as a date/timestamp.

Figure 21.3 shows how the `attributes-charset` attribute would be encoded with a value of `utf-8`.

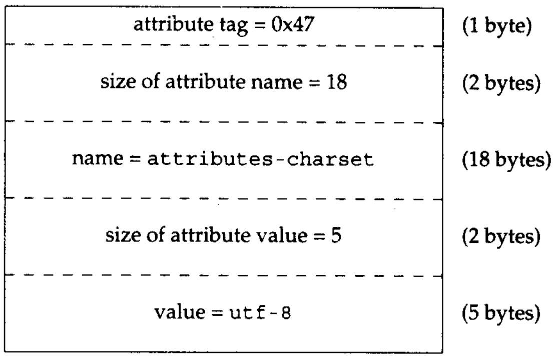
attribute tag = 0x47
(1 byte)
size of attribute name = 18
(2 bytes)
name = attributes-charset
(18 bytes)
size of attribute value = 5
(2 bytes)
value = utf-8
(5 bytes)

Figure 21.3 Sample IPP attribute encoding

Depending on the operation requested, some attributes are required to be provided in the request message, whereas others are optional. For example, Figure 21.4 shows the attributes defined for a print-job request.

Attribute	Status	Description
`attributes-charset`	required	The character set used by attributes of type `text` or `name`
`attributes-natural-language`	required	The natural language used by attributes of type `text` or `name`
`printer-uri`	required	The printer's Universal Resource Identifier
`requesting-user-name`	optional	Name of user submitting job (used for authentication, if enabled)
`job-name`	optional	Name of job used to distinguish between multiple jobs
`ipp-attribute-fidelity`	optional	If true, tells printer to reject job if all attributes can't be met; otherwise, printer does its best to print the job
`document-name`	optional	The name of the document (suitable for printing in a banner, for example)
`document-format`	optional	The format of the document (plaintext, PostScript, etc.)
`document-natural-language`	optional	The natural language of the document
`compression`	optional	The algorithm used to compress the document data
`job-k-octets`	optional	Size of the document in 1,024-octet units
`job-impressions`	optional	Number of impressions (images imposed on a page) submitted in this job
`job-media-sheets`	optional	Number of sheets printed by this job

Figure 21.4 Attributes of print-job request

The IPP header contains a mixture of text and binary data. Attribute names are stored as text, but sizes are stored as binary integers. This complicates the process of building and parsing the header, since we need to worry about such things as network byte order and whether our host processor can address an integer on an arbitrary byte boundary. A better alternative would have been to design the header to contain text only. This simplifies processing at the cost of slightly larger protocol messages.

IPP is specified in a series of documents (Requests For Comments, or RFCs) available at `http://www.pwg.org/ipp`. The main documents are listed in Figure 21.5, although many other documents are available to further specify administrative procedures, job attributes, and the like.

RFC	Title
2567	Design Goals for an Internet Printing Protocol
2568	Rationale for the Structure of the Model and Protocol for the Internet Printing Protocol
2911	Internet Printing Protocol/1.1: Model and Semantics
2910	Internet Printing Protocol/1.1: Encoding and Transport
3196	Internet Printing Protocol/1.1: Implementor's Guide

Figure 21.5 Primary IPP RFCs

21.3 The Hypertext Transfer Protocol

Version 1.1 of HTTP is specified in RFC 2616. HTTP is also a request–response protocol. A request message contains a start line, followed by header lines, a blank line, and an optional entity body. The entity body contains the IPP header and data in this case.

HTTP headers are ASCII, with each line terminated by a carriage return (`\r`) and a line feed (`\n`). The start line consists of a *method* that indicates what operation the client is requesting, a Uniform Resource Locator (URL) that describes the server and protocol, and a string indicating the HTTP version. The only method used by IPP is `POST`, which is used to send data to a server.

The header lines specify attributes, such as the format and length of the entity body. A header line consists of an attribute name followed by a colon, optional white space, and the attribute value, and is terminated by a carriage return and a line feed. For example, to specify that the entity body contains an IPP message, we include the header line

```
Content-Type: application/ipp
```

The start line in an HTTP response message contains a version string followed by a numeric status code and a status message, terminated by a carriage return and a line feed. The remainder of the HTTP response message has the same format as the request message: headers followed by a blank line and an optional entity body.

The following is a sample HTTP header for a print request for the author's printer:

```
POST /phaser860/ipp HTTP/1.1^M
Content-Length: 21931^M
Content-Type: application/ipp^M
Host: phaser860:ipp^M
^M
```

The `^M` at the end of the each line is the carriage return that precedes the line feed. The line feed doesn't show up as a printable character. Note that the last line of the header is empty, except for the carriage return and line feed.

21.4 Printer Spooling

The programs that we develop in this chapter form the basis of a simple printer spooler. A simple user command sends a file to the printer spooler; the spooler saves it to disk, queues the request, and ultimately sends the file to the printer.

All UNIX Systems provide at least one print spooling system. FreeBSD ships LPD, the BSD print spooling system (see `lpd`(8) and Chapter 13 of Stevens [1990]). Linux and Mac OS X include CUPS, the Common UNIX Printing System (see `cupsd`(8)). Solaris ships with the standard System V printer spooler (see `lp`(1) and `lpsched`(1M)). In this chapter, our interest is not in these spooling systems per se, but in communicating with a network printer. We need to develop a spooling system to solve the problem of multiuser access to a single resource (the printer).

We use a simple command that reads a file and sends it to the printer spooler daemon. The command has one option to force the file to be treated as plaintext (the default assumes that the file is PostScript). We call this command `print`.

In our printer spooler daemon, `printd`, we use multiple threads to divide up the work that the daemon needs to accomplish.

- One thread listens on a socket for new print requests arriving from clients running the `print` command.
- A separate thread is spawned for each client to copy the file to be printed to a spooling area.
- One thread communicates with the printer, sending it queued jobs one at a time.
- One thread handles signals.

Figure 21.6 shows how these components fit together.

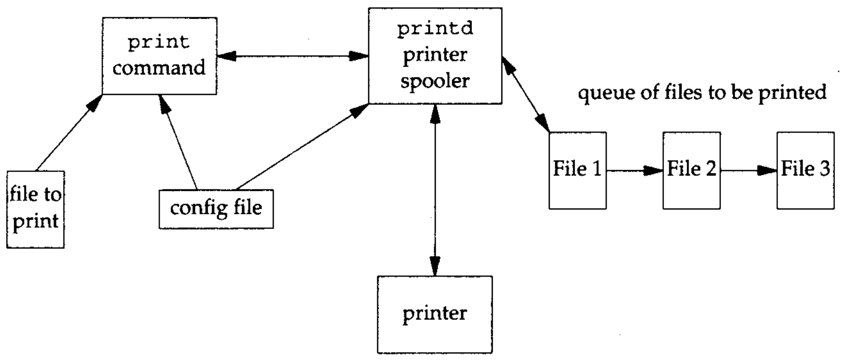

Figure 21.6 Printer spooling components

The print configuration file is `/etc/printer.conf`. It identifies the host name of the server running the printer spooling daemon and the host name of the network printer. The spooling daemon is identified by a line starting with the `printserver` keyword, followed by white space and the host name of the server. The printer is identified by a line starting with the `printer` keyword, followed by white space and the host name of the printer.

A sample printer configuration file might contain the following lines:

```
printserver   blade
printer       phaser860
```

where `blade` is the host name of the computer system running the printer spooling daemon, and `phaser860` is the host name of the network printer.

Security

Programs that run with superuser privileges have the potential to open a computer system up to attack. Such programs usually aren't more vulnerable than any other program, but when compromised can lead to attackers obtaining full access to your system.

The printer spooling daemon in this chapter starts out with superuser privileges in this example to be able to bind a socket to a privileged TCP port number. To make the daemon less vulnerable to attack, we can

- Design the daemon to conform to the principles of least privilege (Section 8.11). After we obtain a socket bound to a privileged port address, we can change the user and group IDs of the daemon to something other that `root` (`lp`, for example). All the files and directories used to store queued print jobs should be owned by this nonprivileged user. This way, the daemon, if compromised, will provide the attacker with access only to the printing subsystem. This is still a concern, but it is far less serious than an attacker getting full access to your system.
- Audit the daemon's source code for all known potential vulnerabilities, such as buffer overruns.
- Log unexpected or suspicious behavior so that an administrator can take note and investigate further.

21.5 Source Code

The source code for this chapter comprises five files, not including some of the common library routines we've used in earlier chapters:

`ipp.h`	Header file containing IPP definitions
`print.h`	Header containing common constants, data structure definitions, and utility routine declarations
`util.c`	Utility routines used by the two programs
`print.c`	The C source file for the command used to print a file
`printd.c`	The C source file for the printer spooling daemon

We will study each file in the order listed.

We start with the ipp.h header file.

```
 1   #ifndef _IPP_H
 2   #define _IPP_H

 3   /*
 4    * Defines parts of the IPP protocol between the scheduler
 5    * and the printer.  Based on RFC2911 and RFC2910.
 6    */

 7   /*
 8    * Status code classes.
 9    */
10   #define STATCLASS_OK(x)      ((x) >= 0x0000 && (x) <= 0x00ff)
11   #define STATCLASS_INFO(x)    ((x) >= 0x0100 && (x) <= 0x01ff)
12   #define STATCLASS_REDIR(x)   ((x) >= 0x0200 && (x) <= 0x02ff)
13   #define STATCLASS_CLIERR(x)  ((x) >= 0x0400 && (x) <= 0x04ff)
14   #define STATCLASS_SRVERR(x)  ((x) >= 0x0500 && (x) <= 0x05ff)

15   /*
16    * Status codes.
17    */
18   #define STAT_OK              0x0000   /* success */
19   #define STAT_OK_ATTRIGN      0x0001   /* OK; some attrs ignored */
20   #define STAT_OK_ATTRCON      0x0002   /* OK; some attrs conflicted */

21   #define STAT_CLI_BADREQ      0x0400   /* invalid client request */
22   #define STAT_CLI_FORBID      0x0401   /* request is forbidden */
23   #define STAT_CLI_NOAUTH      0x0402   /* authentication required */
24   #define STAT_CLI_NOPERM      0x0403   /* client not authorized */
25   #define STAT_CLI_NOTPOS      0x0404   /* request not possible */
26   #define STAT_CLI_TIMOUT      0x0405   /* client too slow */
27   #define STAT_CLI_NOTFND      0x0406   /* no object found for URI */
28   #define STAT_CLI_OBJGONE     0x0407   /* object no longer available */
29   #define STAT_CLI_TOOBIG      0x0408   /* requested entity too big */
30   #define STAT_CLI_TOOLNG      0x0409   /* attribute value too large */
31   #define STAT_CLI_BADFMT      0x040a   /* unsupported doc format */
32   #define STAT_CLI_NOTSUP      0x040b   /* attributes not supported */
33   #define STAT_CLI_NOSCHM      0x040c   /* URI scheme not supported */
34   #define STAT_CLI_NOCHAR      0x040d   /* charset not supported */
35   #define STAT_CLI_ATTRCON     0x040e   /* attributes conflicted */
36   #define STAT_CLI_NOCOMP      0x040f   /* compression not supported */
37   #define STAT_CLI_COMPERR     0x0410   /* data can't be decompressed */
38   #define STAT_CLI_FMTERR      0x0411   /* document format error */
39   #define STAT_CLI_ACCERR      0x0412   /* error accessing data */
```

[1–14] We start the ipp.h header with the standard #ifdef to prevent errors when it is included twice in the same file. Then we define the classes of IPP status codes (see Section 13 in RFC 2911).

[15–39] We define specific status codes based on RFC 2911. We don't use these codes in the program shown here; their use is left as an exercise (See Exercise 21.1).

```
40  #define STAT_SRV_INTERN   0x0500  /* unexpected internal error */
41  #define STAT_SRV_NOTSUP   0x0501  /* operation not supported */
42  #define STAT_SRV_UNAVAIL  0x0502  /* service unavailable */
43  #define STAT_SRV_BADVER   0x0503  /* version not supported */
44  #define STAT_SRV_DEVERR   0x0504  /* device error */
45  #define STAT_SRV_TMPERR   0x0505  /* temporary error */
46  #define STAT_SRV_REJECT   0x0506  /* server not accepting jobs */
47  #define STAT_SRV_TOOBUSY  0x0507  /* server too busy */
48  #define STAT_SRV_CANCEL   0x0508  /* job has been canceled */
49  #define STAT_SRV_NOMULTI  0x0509  /* multi-doc jobs unsupported */

50  /*
51   * Operation IDs
52   */
53  #define OP_PRINT_JOB          0x02
54  #define OP_PRINT_URI          0x03
55  #define OP_VALIDATE_JOB       0x04
56  #define OP_CREATE_JOB         0x05
57  #define OP_SEND_DOC           0x06
58  #define OP_SEND_URI           0x07
59  #define OP_CANCEL_JOB         0x08
60  #define OP_GET_JOB_ATTR       0x09
61  #define OP_GET_JOBS           0x0a
62  #define OP_GET_PRINTER_ATTR   0x0b
63  #define OP_HOLD_JOB           0x0c
64  #define OP_RELEASE_JOB        0x0d
65  #define OP_RESTART_JOB        0x0e
66  #define OP_PAUSE_PRINTER      0x10
67  #define OP_RESUME_PRINTER     0x11
68  #define OP_PURGE_JOBS         0x12

69  /*
70   * Attribute Tags.
71   */
72  #define TAG_OPERATION_ATTR   0x01  /* operation attributes tag */
73  #define TAG_JOB_ATTR         0x02  /* job attributes tag */
74  #define TAG_END_OF_ATTR      0x03  /* end of attributes tag */
75  #define TAG_PRINTER_ATTR     0x04  /* printer attributes tag */
76  #define TAG_UNSUPP_ATTR      0x05  /* unsupported attributes tag */
```

[40–49] We continue to define status codes. The ones in the range `0x500` to `0x5ff` are server error codes. All codes are described in Sections 13.1.1 through 13.1.5 in RFC 2911.

[50–68] We define the various operation IDs next. There is one ID for each task defined by IPP (see Section 4.4.15 in RFC 2911). In our example, we will use only the print-job operation.

[69–76] The attribute tags delimit the attribute groups in the IPP request and response messages. The tag values are defined in Section 3.5.1 of RFC 2910.

```
 77   /*
 78    * Value Tags.
 79    */
 80   #define TAG_UNSUPPORTED      0x10 /* unsupported value */
 81   #define TAG_UNKNOWN          0x12 /* unknown value */
 82   #define TAG_NONE             0x13 /* no value */
 83   #define TAG_INTEGER          0x21 /* integer */
 84   #define TAG_BOOLEAN          0x22 /* boolean */
 85   #define TAG_ENUM             0x23 /* enumeration */
 86   #define TAG_OCTSTR           0x30 /* octetString */
 87   #define TAG_DATETIME         0x31 /* dateTime */
 88   #define TAG_RESOLUTION       0x32 /* resolution */
 89   #define TAG_INTRANGE         0x33 /* rangeOfInteger */
 90   #define TAG_TEXTWLANG        0x35 /* textWithLanguage */
 91   #define TAG_NAMEWLANG        0x36 /* nameWithLanguage */
 92   #define TAG_TEXTWOLANG       0x41 /* textWithoutLanguage */
 93   #define TAG_NAMEWOLANG       0x42 /* nameWithoutLanguage */
 94   #define TAG_KEYWORD          0x44 /* keyword */
 95   #define TAG_URI              0x45 /* URI */
 96   #define TAG_URISCHEME        0x46 /* uriScheme */
 97   #define TAG_CHARSET          0x47 /* charset */
 98   #define TAG_NATULANG         0x48 /* naturalLanguage */
 99   #define TAG_MIMETYPE         0x49 /* mimeMediaType */

100   struct ipp_hdr {
101     int8_t  major_version;  /* always 1 */
102     int8_t  minor_version;  /* always 1 */
103     union {
104         int16_t op; /* operation ID */
105         int16_t st; /* status */
106     } u;
107     int32_t request_id;     /* request ID */
108     char    attr_group[1];  /* start of optional attributes group */
109     /* optional data follows */
110   };

111   #define operation u.op
112   #define status u.st

113   #endif /* _IPP_H */
```

[77–99] The value tags indicate the format of individual attributes and parameters. They are defined in Section 3.5.2 of RFC 2910.

[100–113] We define the structure of an IPP header. Request messages start with the same header as response messages, except that the operation ID in the request is replaced by a status code in the response.

We end the header file with a `#endif` to match the `#ifdef` at the start of the file.

The next file is the `print.h` header.

```
 1  #ifndef _PRINT_H
 2  #define _PRINT_H
 3  /*
 4   * Print server header file.
 5   */
 6  #include <sys/socket.h>
 7  #include <arpa/inet.h>
 8  #if defined(BSD) || defined(MACOS)
 9  #include <netinet/in.h>
10  #endif
11  #include <netdb.h>
12  #include <errno.h>

13  #define CONFIG_FILE    "/etc/printer.conf"
14  #define SPOOLDIR       "/var/spool/printer"
15  #define JOBFILE        "jobno"
16  #define DATADIR        "data"
17  #define REQDIR         "reqs"

18  #define FILENMSZ        64
19  #define FILEPERM        (S_IRUSR|S_IWUSR)
20  #define USERNM_MAX      64
21  #define JOBNM_MAX       256
22  #define MSGLEN_MAX      512

23  #ifndef HOST_NAME_MAX
24  #define HOST_NAME_MAX   256
25  #endif

26  #define IPP_PORT        631
27  #define QLEN            10
28  #define IBUFSZ          512     /* IPP header buffer size */
29  #define HBUFSZ          512     /* HTTP header buffer size */
30  #define IOBUFSZ         8192    /* data buffer size */
```

[1–12] We include all header files that an application might need if it included this header. This makes it easy for applications to include `print.h` without having to track down all the header dependencies.

[13–17] We define the files and directories for the implementation. Copies of the files to be printed will be stored in the directory `/var/spool/printer/data`; control information for each request will be stored in the directory `/var/spool/printer/reqs`. The file containing the next job number is `/var/spool/printer/jobno`.

[18–30] Next, we define limits and constants. `FILEPERM` is the permissions used when creating copies of files submitted to be printed. The permissions are restrictive because we don't want ordinary users to be able to read one another's files while they are waiting to be printed. IPP is defined to use port 631. The `QLEN` is the backlog parameter we pass to `listen` (see Section 16.4 for details).

```
31  #ifndef ETIME
32  #define ETIME ETIMEDOUT
33  #endif

34  extern int getaddrlist(const char *, const char *,
35    struct addrinfo **);
36  extern char *get_printserver(void);
37  extern struct addrinfo *get_printaddr(void);
38  extern ssize_t tread(int, void *, size_t, unsigned int);
39  extern ssize_t treadn(int, void *, size_t, unsigned int);
40  extern int connect_retry(int, const struct sockaddr *, socklen_t);
41  extern int initserver(int, struct sockaddr *, socklen_t, int);

42  /*
43   * Structure describing a print request.
44   */
45  struct printreq {
46     long size;                  /* size in bytes */
47     long flags;                 /* see below */
48     char usernm[USERNM_MAX];    /* user's name */
49     char jobnm[JOBNM_MAX];      /* job's name */
50  };

51  /*
52   * Request flags.
53   */
54  #define PR_TEXT         0x01    /* treat file as plain text */

55  /*
56   * The response from the spooling daemon to the print command.
57   */
58  struct printresp {
59     long retcode;               /* 0=success, !0=error code */
60     long jobid;                 /* job ID */
61     char msg[MSGLEN_MAX];       /* error message */
62  };

63  #endif /* _PRINT_H */
```

[31–33] Some platforms don't define the error `ETIME`, so we define it to an alternate error code that makes sense for these systems.

[34–41] Next, we declare all the public routines contained in `util.c` (we'll look at these next). Note that the `connect_retry` function, from Figure 16.9, and the `initserver` function, from Figure 16.20, are not included in `util.c`.

[42–63] The `printreq` and `printresp` structures define the protocol between the print command and the printer spooling daemon. The `print` command sends the `printreq` structure defining the user name, job name, and file size to the printer spooling daemon. The spooling daemon responds with a `printresp` structure consisting of a return code, a job ID, and an error message if the request failed.

The next file we will look at is `util.c`, the file containing utility routines.

```
 1  #include "apue.h"
 2  #include "print.h"
 3  #include <ctype.h>
 4  #include <sys/select.h>

 5  #define MAXCFGLINE 512
 6  #define MAXKWLEN   16
 7  #define MAXFMTLEN  16

 8  /*
 9   * Get the address list for the given host and service and
10   * return through ailistpp.  Returns 0 on success or an error
11   * code on failure.  Note that we do not set errno if we
12   * encounter an error.
13   *
14   * LOCKING: none.
15   */
16  int
17  getaddrlist(const char *host, const char *service,
18    struct addrinfo **ailistpp)
19  {
20     int             err;
21     struct addrinfo hint;

22     hint.ai_flags = AI_CANONNAME;
23     hint.ai_family = AF_INET;
24     hint.ai_socktype = SOCK_STREAM;
25     hint.ai_protocol = 0;
26     hint.ai_addrlen = 0;
27     hint.ai_canonname = NULL;
28     hint.ai_addr = NULL;
29     hint.ai_next = NULL;
30     err = getaddrinfo(host, service, &hint, ailistpp);
31     return(err);
32  }
```

[1–7] We first define the limits needed by the functions in this file. `MAXCFGLINE` is the maximum size of a line in the printer configuration file, `MAXKWLEN` is the maximum size of a keyword in the configuration file, and `MAXFMTLEN` is the maximum size of the format string we pass to `sscanf`.

[8–32] The first function is `getaddrlist`. It is a wrapper for `getaddrinfo` (Section 16.3.3), since we always call `getaddrinfo` with the same hint structure. Note that we need no mutex locking in this function. The `LOCKING` comment at the beginning of each function is intended only for documenting multithreaded locking. This comment lists the assumptions, if any, that are made regarding the locking, tells which locks the function might acquire or release, and tells which locks must be held to call the function.

```
33  /*
34   * Given a keyword, scan the configuration file for a match
35   * and return the string value corresponding to the keyword.
36   *
37   * LOCKING: none.
38   */
39  static char *
40  scan_configfile(char *keyword)
41  {
42      int             n, match;
43      FILE            *fp;
44      char            keybuf[MAXKWLEN], pattern[MAXFMTLEN];
45      char            line[MAXCFGLINE];
46      static char     valbuf[MAXCFGLINE];

47      if ((fp = fopen(CONFIG_FILE, "r")) == NULL)
48          log_sys("can't open %s", CONFIG_FILE);
49      sprintf(pattern, "%%%ds %%%ds", MAXKWLEN-1, MAXCFGLINE-1);
50      match = 0;
51      while (fgets(line, MAXCFGLINE, fp) != NULL) {
52          n = sscanf(line, pattern, keybuf, valbuf);
53          if (n == 2 && strcmp(keyword, keybuf) == 0) {
54              match = 1;
55              break;
56          }
57      }
58      fclose(fp);
59      if (match != 0)
60          return(valbuf);
61      else
62          return(NULL);
63  }
```

[33–46] The `scan_configfile` function searches through the printer configuration file for the specified keyword.

[47–63] We open the configuration file for reading and build the format string corresponding to the search pattern. The notation `%%%ds` builds a format specifier that limits the string size so we don't overrun the buffers used to store the strings on the stack. We read the file one line at a time and scan for two strings separated by white space; if we find them, we compare the first string with the keyword. If we find a match or we reach the end of the file, the loop ends and we close the file. If the keyword matches, we return a pointer to the buffer containing the string after the keyword; otherwise, we return NULL.

The string returned is stored in a static buffer (`valbuf`), which can be overwritten on successive calls. Thus, `scan_configfile` can't be called by a multithreaded application unless we take care to avoid calling it from multiple threads at the same time.

```
64  /*
65   * Return the host name running the print server or NULL on error.
66   *
67   * LOCKING: none.
68   */
69  char *
70  get_printserver(void)
71  {
72      return(scan_configfile("printserver"));
73  }

74  /*
75   * Return the address of the network printer or NULL on error.
76   *
77   * LOCKING: none.
78   */
79  struct addrinfo *
80  get_printaddr(void)
81  {
82      int             err;
83      char            *p;
84      struct addrinfo *ailist;

85      if ((p = scan_configfile("printer")) != NULL) {
86          if ((err = getaddrlist(p, "ipp", &ailist)) != 0) {
87              log_msg("no address information for %s", p);
88              return(NULL);
89          }
90          return(ailist);
91      }
92      log_msg("no printer address specified");
93      return(NULL);
94  }
```

[64–73] The get_printserver function is simply a wrapper function that calls scan_configfile to find the name of the computer system where the printer spooling daemon is running.

[74–94] We use the get_printaddr function to get the address of the network printer. It is similar to the previous function except that when we find the name of the printer in the configuration file, we use the name to find the corresponding network address.

Both get_printserver and get_printaddr call scan_configfile. If it can't open the printer configuration file, scan_configfile calls log_sys to print an error message and exit. Although get_printserver is meant to be called from a client command and get_printaddr is meant to be called from the daemon, having both call log_sys is OK, because we can arrange for the log functions to print to the standard error instead of to the log file by setting a global variable.

```
 95   /*
 96    * "Timed" read - timout specifies the # of seconds to wait before
 97    * giving up (5th argument to select controls how long to wait for
 98    * data to be readable).  Returns # of bytes read or -1 on error.
 99    *
100    * LOCKING: none.
101    */
102   ssize_t
103   tread(int fd, void *buf, size_t nbytes, unsigned int timout)
104   {
105       int             nfds;
106       fd_set          readfds;
107       struct timeval  tv;

108       tv.tv_sec = timout;
109       tv.tv_usec = 0;
110       FD_ZERO(&readfds);
111       FD_SET(fd, &readfds);
112       nfds = select(fd+1, &readfds, NULL, NULL, &tv);
113       if (nfds <= 0) {
114           if (nfds == 0)
115               errno = ETIME;
116           return(-1);
117       }
118       return(read(fd, buf, nbytes));
119   }
```

[95–107] We provide a function called `tread` to read a specified number of bytes, but block for at most *timout* seconds before giving up. This function is useful when reading from a socket or a pipe. If we don't receive data before the specified time limit, we return −1 with `errno` set to `ETIME`. If data is available within the time limit, we return at most *nbytes* bytes of data, but we can return less than requested if all the data doesn't arrive in time.

We will use `tread` to prevent denial-of-service attacks on the printer spooling daemon. A malicious user might repeatedly try to connect to the daemon without sending it data, just to prevent other users from being able to submit print jobs. By giving up after a reasonable amount of time, we prevent this from happening. The tricky part is selecting a suitable timeout value that is large enough to prevent premature failures when the system is under load and tasks are taking longer to complete. If we choose a value too large, however, we might enable denial-of-service attacks by allowing the daemon to consume too many resources to process the pending requests.

[108–119] We use `select` to wait for the specified file descriptor to be readable. If the time limit expires before data is available to be read, `select` returns 0, so we set `errno` to `ETIME` in this case. If `select` fails or times out, we return −1. Otherwise, we return whatever data is available.

```
120  /*
121   * "Timed" read - timout specifies the number of seconds to wait
122   * per read call before giving up, but read exactly nbytes bytes.
123   * Returns number of bytes read or -1 on error.
124   *
125   * LOCKING: none.
126   */
127  ssize_t
128  treadn(int fd, void *buf, size_t nbytes, unsigned int timout)
129  {
130      size_t  nleft;
131      ssize_t nread;

132      nleft = nbytes;
133      while (nleft > 0) {
134          if ((nread = tread(fd, buf, nleft, timout)) < 0) {
135              if (nleft == nbytes)
136                  return(-1); /* error, return -1 */
137              else
138                  break;      /* error, return amount read so far */
139          } else if (nread == 0) {
140              break;          /* EOF */
141          }
142          nleft -= nread;
143          buf += nread;
144      }
145      return(nbytes - nleft);      /* return >= 0 */
146  }
```

[120–146] We also provide a variation of `tread`, called `treadn`, that reads exactly the number of bytes requested. This is similar to the `readn` function described in Section 14.8, but with the addition of the timeout parameter.

To read exactly *nbytes* bytes, we have to be prepared to make multiple calls to `read`. The difficult part is trying to apply a single timeout value to multiple calls to `read`. We don't want to use an alarm, because signals can be messy to deal with in multithreaded applications. We can't rely on the system updating the `timeval` structure on return from `select` to indicate the amount of time left, because many platforms do not support this (Section 14.5.1). Thus, we compromise and define the timeout value in this case to apply to an individual `read` call. Instead of limiting the total amount of time we wait, it limits the amount of time we'll wait in every iteration of the loop. The maximum time we can wait is bounded by (*nbytes* × *timout*) seconds (worst case, we'll receive only 1 byte at a time).

We use `nleft` to record the number of bytes remaining to be read. If `tread` fails and we have received data in a previous iteration, we break out of the `while` loop and return the number of bytes read; otherwise, we return −1.

The command used to submit a print job is shown next. The C source file is `print.c`.

```
 1   /*
 2    * The client command for printing documents.  Opens the file
 3    * and sends it to the printer spooling daemon.  Usage:
 4    *     print [-t] filename
 5    */
 6   #include "apue.h"
 7   #include "print.h"
 8   #include <fcntl.h>
 9   #include <pwd.h>
10   /*
11    * Needed for logging funtions.
12    */
13   int log_to_stderr = 1;
14   void submit_file(int, int, const char *, size_t, int);
15   int
16   main(int argc, char *argv[])
17   {
18       int             fd, sockfd, err, text, c;
19       struct stat     sbuf;
20       char            *host;
21       struct addrinfo *ailist, *aip;
22       err = 0;
23       text = 0;
24       while ((c = getopt(argc, argv, "t")) != -1) {
25           switch (c) {
26           case 't':
27               text = 1;
28               break;
29           case '?':
30               err = 1;
31               break;
32           }
33       }
```

[1–14] We need to define an integer called `log_to_stderr` to be able to use the log functions in our library. If set to a nonzero value, error messages will be sent to the standard error stream instead of to a log file. Although we don't use any logging functions in `print.c`, we do link `util.o` with `print.o` to build the executable `print` command, and `util.c` contains functions for both user commands and daemons.

[15–33] We support one option, `-t`, to force the file to be printed as text (instead of as a PostScript program, for example). We use the `getopt(3)` function to process the command options.

```
34      if (err || (optind != argc - 1))
35          err_quit("usage: print [-t] filename");
36      if ((fd = open(argv[optind], O_RDONLY)) < 0)
37          err_sys("print: can't open %s", argv[1]);
38      if (fstat(fd, &sbuf) < 0)
39          err_sys("print: can't stat %s", argv[1]);
40      if (!S_ISREG(sbuf.st_mode))
41          err_quit("print: %s must be a regular file\n", argv[1]);

42      /*
43       * Get the hostname of the host acting as the print server.
44       */
45      if ((host = get_printserver()) == NULL)
46          err_quit("print: no print server defined");
47      if ((err = getaddrlist(host, "print", &ailist)) != 0)
48          err_quit("print: getaddrinfo error: %s", gai_strerror(err));

49      for (aip = ailist; aip != NULL; aip = aip->ai_next) {
50          if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
51              err = errno;
52          } else if (connect_retry(sockfd, aip->ai_addr,
53            aip->ai_addrlen) < 0) {
54              err = errno;
```

[34–41] When getopt completes processing the command options, it leaves the variable optind set to the index of the first nonoptional argument. If this is any value other than the index of the last argument, then the wrong number of arguments was specified (we support only one nonoptional argument). Our error processing includes checks to ensure that we can open the file to be printed and that it is a regular file (as opposed to a directory or other type of file).

[42–48] We get the name of the printer spooling daemon by calling the get_printserver function from util.c and then translate the host name into a network address by calling getaddrlist (also from util.c).

Note that we specify the service as "print." As part of installing the printer spooling daemon on a system, we need to make sure that /etc/services (or the equivalent database) has an entry for the printer service. When we select a port number for the daemon, it would be a good idea to select one that is privileged, to prevent malicious users from writing applications that pretend to be a printer spooling daemon but instead steal copies of the files we try to print. This means that the port number should be less than 1,024 (recall Section 16.3.4) and that our daemon will have to run with superuser privileges to allow it to bind to a reserved port.

[49–54] We try to connect to the daemon using one address at a time from the list returned by getaddrinfo. We will try to send the file to the daemon using the first address to which we can connect.

```
55          } else {
56              submit_file(fd, sockfd, argv[1], sbuf.st_size, text);
57              exit(0);
58          }
59      }
60      errno = err;
61      err_ret("print: can't contact %s", host);
62      exit(1);
63  }

64  /*
65   * Send a file to the printer daemon.
66   */
67  void
68  submit_file(int fd, int sockfd, const char *fname, size_t nbytes,
69              int text)
70  {
71      int                 nr, nw, len;
72      struct passwd       *pwd;
73      struct printreq     req;
74      struct printresp    res;
75      char                buf[IOBUFSZ];

76      /*
77       * First build the header.
78       */
79      if ((pwd = getpwuid(geteuid())) == NULL)
80          strcpy(req.usernm, "unknown");
81      else
82          strcpy(req.usernm, pwd->pw_name);
83      req.size = htonl(nbytes);
84      if (text)
85          req.flags = htonl(PR_TEXT);
86      else
87          req.flags = 0;
```

[55–63] If we can make a connection, we call `submit_file` to transmit the file to the printer spooling daemon. If we can't connect to any of the addresses, we print an error message and exit. We use `err_ret` and `exit` instead of making a single call to `err_sys` to avoid a compiler warning, because the last line in `main` wouldn't be a `return` statement or a call to `exit`.

[64–87] `submit_file` sends a print request to the daemon and reads the response. First, we build the `printreq` request header. We use `geteuid` to get the caller's effective user ID and pass this to `getpwuid` to look for the user in the system's password file. We copy the user's name to the request header or use the string unknown if we can't identify the user. We store the size of the file to be printed in the header after converting it to network byte order. Then we do the same with the `PR_TEXT` flag if the file is to be printed as plaintext.

```
 88     if ((len = strlen(fname)) >= JOBNM_MAX) {
 89         /*
 90          * Truncate the filename (+-5 accounts for the leading
 91          * four characters and the terminating null).
 92          */
 93         strcpy(req.jobnm, "... ");
 94         strncat(req.jobnm, &fname[len-JOBNM_MAX+5], JOBNM_MAX-5);
 95     } else {
 96         strcpy(req.jobnm, fname);
 97     }

 98     /*
 99      * Send the header to the server.
100      */
101     nw = writen(sockfd, &req, sizeof(struct printreq));
102     if (nw != sizeof(struct printreq)) {
103         if (nw < 0)
104             err_sys("can't write to print server");
105         else
106             err_quit("short write (%d/%d) to print server",
107               nw, sizeof(struct printreq));
108     }

109     /*
110      * Now send the file.
111      */
112     while ((nr = read(fd, buf, IOBUFSZ)) != 0) {
113         nw = writen(sockfd, buf, nr);
114         if (nw != nr) {
115             if (nw < 0)
116                 err_sys("can't write to print server");
117             else
118                 err_quit("short write (%d/%d) to print server",
119                   nw, nr);
120         }
121     }
```

[88–108] We set the job name to the name of the file being printed. If the name is longer than will fit in the message, we truncate the beginning portion of the name and prepend an ellipsis to indicate that there were more characters than would fit in the field. Then we send the request header to the daemon using `writen`. If the write fails or if we transmit less than we expect, we print an error message and exit.

[109–121] After sending the header to the daemon, we send the file to be printed. We read the file `IOBUFSZ` bytes at a time and use `writen` to send the data to the daemon. As with the header, if the write fails or we write less than we expect, we print an error message and exit.

```
122     /*
123      * Read the response.
124      */
125     if ((nr = readn(sockfd, &res, sizeof(struct printresp))) !=
126       sizeof(struct printresp))
127       err_sys("can't read response from server");
128     if (res.retcode != 0) {
129         printf("rejected: %s\n", res.msg);
130         exit(1);
131     } else {
132         printf("job ID %ld\n", ntohl(res.jobid));
133     }
134     exit(0);
135 }
```

[122–135] After we send the file to be printed to the daemon, we read the daemon's response. If the request failed, the return code (`retcode`) will be nonzero, so we print the textual error message included in the response. If the request succeeded, we print the job ID so that the user knows how to refer to the request in the future. (Writing a command to cancel the print request is left as an exercise; the job ID can be used in the cancellation request to identify the job to be removed from the print queue.)

Note that a successful response from the daemon does not mean that the printer was able to print the file. It merely means that the daemon successfully added the print job to the queue.

Most of what we have seen in `print.c` was discussed in earlier chapters. The only topic that we haven't covered is the `getopt` function, although we saw it earlier in the `pty` program from Chapter 19.

It is important that all commands on a system follow the same conventions, because this makes them easier to use. If someone is familiar with the way command-line options are formed with one command, it would create more chances for mistakes if another command followed different conventions.

This problem is sometimes visible when dealing with white space on the command line. Some commands require that an option be separated from its argument by white space, but other commands require the argument to follow immediate after its option, without any intervening spaces. Without a consistent set of rules to follow, users either have to memorize the syntax of all commands or resort to a trial-and-error process when invoking them.

The Single UNIX Specification includes a set of conventions and guidelines that promote consistent command-line syntax. They include such suggestions as "Restrict each command-line option to a single alphanumeric character" and "All options should be preceded by a – character."

Luckily, the `getopt` function exists to help command developers process command-line options in a consistent manner.

```
#include <fcntl.h>

int getopt(int argc, const * const argv[], const char *options);

extern int optind, opterr, optopt;
extern char *optarg;

                        Returns: the next option character, or
                            −1 when all options have been processed
```

The *argc* and *argv* arguments are the same ones passed to the `main` function of the program. The *options* argument is a string containing the option characters supported by the command. If an option character is followed by a colon, then the option takes an argument. Otherwise, the option exists by itself. For example, if the usage statement for a command was

```
command [-i] [-u username] [-z] filename
```

we would pass `"iu:z"` as the *options* string to `getopt`.

The normal use of `getopt` is in a loop that terminates when `getopt` returns −1. During each iteration of the loop, `getopt` will return the next option processed. It is up to the application to sort out any conflict in options, however; `getopt` simply parses the options and enforces a standard format.

When it encounters an invalid option, `getopt` returns a question mark instead of the character. If an option's argument is missing, `getopt` will also return a question mark, but if the first character in the options string is a colon, `getopt` returns a colon instead. The special pattern `--` will cause `getopt` to stop processing options and return −1. This allows users to provide command arguments that start with a minus sign but aren't options. For example, if you have a file named `-bar`, you can't remove it by typing

```
rm -bar
```

because `rm` will try to interpret `-bar` as options. The way to remove the file is to type

```
rm -- -bar
```

The `getopt` function supports four external variables.

`optarg`	If an option takes an argument, `getopt` sets `optarg` to point to the option's argument string when an option is processed.
`opterr`	If an option error is encountered, `getopt` will print an error message by default. To disable this behavior, applications can set `opterr` to 0.
`optind`	The index in the `argv` array of the next string to be processed. It starts out at 1 and is incremented for each argument processed by `getopt`.
`optopt`	If an error is encountered during options processing, `getopt` will set `optopt` to point to the option string that caused the error.

The last file we will look at is the C source file for the printer spooling daemon.

```
 1  /*
 2   * Print server daemon.
 3   */
 4  #include "apue.h"
 5  #include "print.h"
 6  #include "ipp.h"
 7  #include <fcntl.h>
 8  #include <dirent.h>
 9  #include <ctype.h>
10  #include <pwd.h>
11  #include <pthread.h>
12  #include <strings.h>
13  #include <sys/select.h>
14  #include <sys/uio.h>

15  /*
16   * These are for the HTTP response from the printer.
17   */
18  #define HTTP_INFO(x)    ((x) >= 100 && (x) <= 199)
19  #define HTTP_SUCCESS(x) ((x) >= 200 && (x) <= 299)

20  /*
21   * Describes a print job.
22   */
23  struct job {
24      struct job       *next;      /* next in list */
25      struct job       *prev;      /* previous in list */
26      long              jobid;     /* job ID */
27      struct printreq   req;       /* copy of print request */
28  };

29  /*
30   * Describes a thread processing a client request.
31   */
32  struct worker_thread {
33      struct worker_thread  *next;    /* next in list */
34      struct worker_thread  *prev;    /* previous in list */
35      pthread_t              tid;     /* thread ID */
36      int                    sockfd;  /* socket */
37  };
```

[1–19] The printer spooling daemon includes the IPP header file that we saw earlier, because the daemon needs to communicate with the printer using this protocol. The HTTP_INFO and HTTP_SUCCESS macros define the status of the HTTP request (recall that IPP is built on top of HTTP).

[20–37] The job and worker_thread structures are used by the spooling daemon to keep track of print jobs and threads accepting print requests, respectively.

```
38  /*
39   * Needed for logging.
40   */
41  int                     log_to_stderr = 0;

42  /*
43   * Printer-related stuff.
44   */
45  struct addrinfo         *printer;
46  char                    *printer_name;
47  pthread_mutex_t         configlock = PTHREAD_MUTEX_INITIALIZER;
48  int                     reread;

49  /*
50   * Thread-related stuff.
51   */
52  struct worker_thread    *workers;
53  pthread_mutex_t         workerlock = PTHREAD_MUTEX_INITIALIZER;
54  sigset_t                mask;

55  /*
56   * Job-related stuff.
57   */
58  struct job              *jobhead, *jobtail;
59  int                     jobfd;
```

[38–41] Our logging functions require that we define the `log_to_stderr` variable and set it to 0 to force log messages to be sent to the system log instead of to the standard error. In `print.c`, we defined `log_to_stderr` and set it to 1, even though we don't use the log functions in the user command. We could have avoided this by splitting the utility functions into two separate files: one for the server and one for the client commands.

[42–48] We use the global variable `printer` to hold the network address of the printer. We store the host name of the printer in `printer_name`. The `configlock` mutex protects access to the `reread` variable, which is used to indicate that the daemon needs to reread the configuration file, presumably because an administrator changed the printer or its network address.

[49–54] Next, we define the thread-related variables. We use `workers` as the head of a doubly-linked list of threads that are receiving files from clients. This list is protected by the mutex `workerlock`. The signal mask used by the threads is held in the variable `mask`.

[55–59] For the list of pending jobs, we define `jobhead` to be the start of the list and `jobtail` to be the tail of the list. This list is also doubly linked, but we need to add jobs to the end of the list, so we need to remember a pointer to the list tail. With the list of worker threads, the order doesn't matter, so we can add them to the head of the list and don't need to remember the tail pointer. `jobfd` is the file descriptor for the job file.