17

Advanced IPC

17.1 Introduction

In the previous two chapters, we discussed various forms of IPC, including pipes and sockets. In this chapter, we look at an advanced form of IPC—the UNIX domain socket mechanism—and see what we can do with it. With this form of IPC, we can pass open file descriptors between processes running on the same computer system, server processes can associate names with their file descriptors, and client processes running on the same system can use these names to rendezvous with the servers. We'll also see how the operating system provides a unique IPC channel per client.

17.2 UNIX Domain Sockets

UNIX domain sockets are used to communicate with processes running on the same machine. Although Internet domain sockets can be used for this same purpose, UNIX domain sockets are more efficient. UNIX domain sockets only copy data; they have no protocol processing to perform, no network headers to add or remove, no checksums to calculate, no sequence numbers to generate, and no acknowledgements to send.

UNIX domain sockets provide both stream and datagram interfaces. The UNIX domain datagram service is reliable, however. Messages are neither lost nor delivered out of order. UNIX domain sockets are like a cross between sockets and pipes. You can use the network-oriented socket interfaces with them, or you can use the `socketpair` function to create a pair of unnamed, connected, UNIX domain sockets.

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

int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

Returns: 0 if OK, −1 on error

Although the interface is sufficiently general to allow `socketpair` to be used with other domains, operating systems typically provide support only for the UNIX domain.

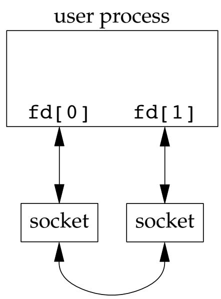

Figure 17.1 A socket pair

A pair of connected UNIX domain sockets acts like a full-duplex pipe: both ends are open for reading and writing (see Figure 17.1). We'll refer to these as "fd-pipes" to distinguish them from normal, half-duplex pipes.

Example — `fd_pipe` Function

Figure 17.2 shows the `fd_pipe` function, which uses the `socketpair` function to create a pair of connected UNIX domain stream sockets.

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

/*
 * Returns a full-duplex pipe (a UNIX domain socket) with
 * the two file descriptors returned in fd[0] and fd[1].
 */
int
fd_pipe(int fd[2])
{
    return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}
```

Figure 17.2 Creating a full-duplex pipe

Some BSD-based systems use UNIX domain sockets to implement pipes. But when `pipe` is called, the write end of the first descriptor and the read end of the second descriptor are both closed. To get a full-duplex pipe, we must call `socketpair` directly. □

Example — Polling XSI Message Queues with the Help of UNIX Domain Sockets

In Section 15.6.4, we said one of the problems with using XSI message queues is that we can't use `poll` or `select` with them, because they aren't associated with file descriptors. However, sockets *are* associated with file descriptors, and we can use them to notify us when messages arrive. We'll use one thread per message queue. Each thread will block in a call to `msgrcv`. When a message arrives, the thread will write it down one end of a UNIX domain socket. Our application will use the other end of the socket to receive the message when `poll` indicates data can be read from the socket.

The program in Figure 17.3 illustrates this technique. The `main` function creates the message queues and UNIX domain sockets and starts one thread to service each queue. Then it uses an infinite loop to poll one end of the sockets. When a socket is readable, it reads from the socket and writes the message on the standard output.

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ      3       /* number of queues */
#define MAXMSZ  512     /* maximum message size */
#define KEY     0x123   /* key for first message queue */

struct threadinfo {
    int qid;
    int fd;
};

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

void *
helper(void *arg)
{
    int                 n;
    struct mymesg       m;
    struct threadinfo   *tip = arg;

    for(;;) {
        memset(&m, 0, sizeof(m));
        if ((n = msgrcv(tip->qid, &m, MAXMSZ, 0, MSG_NOERROR)) < 0)
            err_sys("msgrcv error");
        if (write(tip->fd, m.mtext, n) < 0)
            err_sys("write error");
    }
}

int
main()
```

```
{
    int                 i, n, err;
    int                 fd[2];
    int                 qid[NQ];
    struct pollfd       pfd[NQ];
    struct threadinfo   ti[NQ];
    pthread_t           tid[NQ];
    char                buf[MAXMSZ];

    for (i = 0; i < NQ; i++) {
        if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
            err_sys("msgget error");

        printf("queue ID %d is %d\n", i, qid[i]);

        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("socketpair error");
        pfd[i].fd = fd[0];
        pfd[i].events = POLLIN;
        ti[i].qid = qid[i];
        ti[i].fd = fd[1];
        if ((err = pthread_create(&tid[i], NULL, helper, &ti[i])) != 0)
            err_exit(err, "pthread_create error");
    }

    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
            err_sys("poll error");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, buf, sizeof(buf))) < 0)
                    err_sys("read error");
                buf[n] = 0;
                printf("queue id %d, message %s\n", qid[i], buf);
            }
        }
    }

    exit(0);
}
```

Figure 17.3 Poll for XSI messages using UNIX domain sockets

Note that we use datagram (SOCK_DGRAM) sockets instead of stream sockets. This allows us to retain message boundaries so when we read from the socket, we read only one message at a time.

This technique allows us to use either poll or select (indirectly) with message queues. As long as the costs of one thread per queue and copying each message two extra times (once to write it to the socket and once to read it from the socket) are acceptable, this technique will make it easier to use XSI message queues.

Well use the program shown in Figure 17.4 to send messages to our test program from Figure 17.3.

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

#define MAXMSZ 512

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

int
main(int argc, char *argv[])
{
    key_t key;
    long qid;
    size_t nbytes;
    struct mymesg m;

    if (argc != 3) {
        fprintf(stderr, "usage: sendmsg KEY message\n");
        exit(1);
    }
    key = strtol(argv[1], NULL, 0);
    if ((qid = msgget(key, 0)) < 0)
        err_sys("can't open queue key %s", argv[1]);
    memset(&m, 0, sizeof(m));
    strncpy(m.mtext, argv[2], MAXMSZ-1);
    nbytes = strlen(m.mtext);
    m.mtype = 1;
    if (msgsnd(qid, &m, nbytes, 0) < 0)
        err_sys("can't send message");
    exit(0);
}
```

Figure 17.4 Post a message to an XSI message queue

This program takes two arguments: the key associated with the queue and a string to be sent as the body of the message. When we send messages to the server, it prints them as shown below.

```
$ ./pollmsg &                              run the server in the background
[1] 12814
$ queue ID 0 is 196608
queue ID 1 is 196609
queue ID 2 is 196610
$ ./sendmsg 0x123 "hello, world"           send a message to the first queue
queue id 196608, message hello, world
$ ./sendmsg 0x124 "just a test"            send a message to the second queue
queue id 196609, message just a test
$ ./sendmsg 0x125 "bye"                    send a message to the third queue
queue id 196610, message bye
```

□

17.2.1 Naming UNIX Domain Sockets

Although the `socketpair` function creates sockets that are connected to each other, the individual sockets don't have names. This means that they can't be addressed by unrelated processes.

In Section 16.3.4, we learned how to bind an address to an Internet domain socket. Just as with Internet domain sockets, UNIX domain sockets can be named and used to advertise services. The address format used with UNIX domain sockets differs from that used with Internet domain sockets, however.

Recall from Section 16.3 that socket address formats differ from one implementation to the next. An address for a UNIX domain socket is represented by a `sockaddr_un` structure. On Linux 3.2.0 and Solaris 10, the `sockaddr_un` structure is defined in the header `<sys/un.h>` as

```
struct sockaddr_un {
     sa_family_t  sun_family;     /* AF_UNIX */
     char         sun_path[108];  /* pathname */
};
```

On FreeBSD 8.0 and Mac OS X 10.6.8, however, the `sockaddr_un` structure is defined as

```
struct sockaddr_un {
     unsigned char  sun_len;        /* sockaddr length */
     sa_family_t    sun_family;     /* AF_UNIX */
     char           sun_path[104];  /* pathname */
};
```

The `sun_path` member of the `sockaddr_un` structure contains a pathname. When we bind an address to a UNIX domain socket, the system creates a file of type `S_IFSOCK` with the same name.

This file exists only as a means of advertising the socket name to clients. The file can't be opened or otherwise used for communication by applications.

If the file already exists when we try to bind the same address, the `bind` request will fail. When we close the socket, this file is not automatically removed, so we need to make sure that we unlink it before our application exits.

Example

The program in Figure 17.5 shows an example of binding an address to a UNIX domain socket.

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int
main(void)
{
    int fd, size;
```

```
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket failed");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("bind failed");
    printf("UNIX domain socket bound\n");
    exit(0);
}
```

Figure 17.5 Binding an address to a UNIX domain socket

When we run this program, the `bind` request succeeds. If we run the program a second time, however, we get an error, because the file already exists. The program won't succeed again until we remove the file.

```
$ ./a.out                                          run the program
UNIX domain socket bound
$ ls -l foo.socket                                 look at the socket file
srwxr-xr-x  1 sar       0 May 18 00:44 foo.socket
$ ./a.out                                          try to run the program again
bind failed: Address already in use
$ rm foo.socket                                    remove the socket file
$ ./a.out                                          run the program a third time
UNIX domain socket bound                           now it succeeds
```

The way we determine the size of the address to bind is to calculate the offset of the `sun_path` member in the `sockaddr_un` structure and add to it the length of the pathname, not including the terminating null byte. Since implementations vary in which members precede `sun_path` in the `sockaddr_un` structure, we use the `offsetof` macro from `<stddef.h>` (included by `apue.h`) to calculate the offset of the `sun_path` member from the start of the structure. If you look in `<stddef.h>`, you'll see a definition similar to the following:

```
#define offsetof(TYPE, MEMBER)  ((int)&((TYPE *)0)->MEMBER)
```

The expression evaluates to an integer, which is the starting address of the member, assuming that the structure begins at address 0. □

17.3 Unique Connections

A server can arrange for unique UNIX domain connections to clients using the standard `bind`, `listen`, and `accept` functions. Clients use `connect` to contact the server; after the connect request is accepted by the server, a unique connection exists between the client and the server. This style of operation is the same that we illustrated with Internet domain sockets in Figures 16.16 and 16.17.

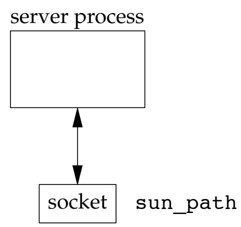

Figure 17.6 Client and server sockets before a `connect`

Figure 17.6 shows a client process and a server process before a connection exists between the two. The server has bound its socket to a `sockaddr_un` address and is listening for connection requests. Figure 17.7 shows the unique connection between the client and server after the server has accepted the client's connection request.

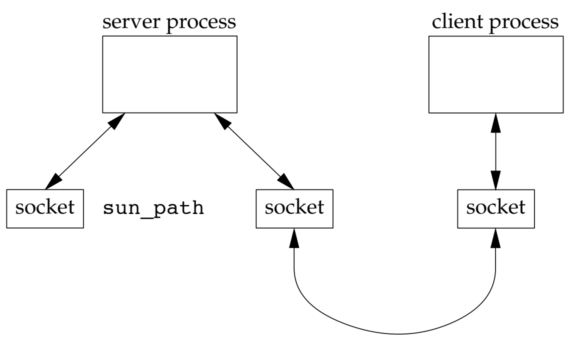

Figure 17.7 Client and server sockets after a `connect`

We will now develop three functions that can be used to create unique connections between unrelated processes running on the same machine. These functions mimic the connection-oriented socket functions discussed in Section 16.4. We use UNIX domain sockets for the underlying communication mechanism here.

```
#include "apue.h"

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

Returns: file descriptor to listen on if OK, negative value on error

```
int serv_accept(int listenfd, uid_t *uidptr);
```

Returns: new file descriptor if OK, negative value on error

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

Returns: file descriptor if OK, negative value on error

The `serv_listen` function (Figure 17.8) can be used by a server to announce its willingness to listen for client connect requests on a well-known name (some pathname in the file system). Clients will use this name when they want to connect to the server. The return value is the server's UNIX domain socket used to receive client connection requests.

The `serv_accept` function (Figure 17.9) is used by a server to wait for a client's connect request to arrive. When one arrives, the system automatically creates a new UNIX domain socket, connects it to the client's socket, and returns the new socket to the server. Additionally, the effective user ID of the client is stored in the memory to which *uidptr* points.

A client calls `cli_conn` (Figure 17.10) to connect to a server. The *name* argument specified by the client must be the same name that was advertised by the server's call to `serv_listen`. On return, the client gets a file descriptor connected to the server.

Figure 17.8 shows the `serv_listen` function.

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define QLEN    10

/*
 * Create a server endpoint of a connection.
 * Returns fd if all OK, <0 on error.
 */
int
serv_listen(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-2);

    unlink(name);   /* in case it already exists */

    /* fill in socket address structure */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -3;
```

```
        goto errout;
    }

    if (listen(fd, QLEN) < 0) { /* tell kernel we're a server */
        rval = -4;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
```

Figure 17.8 The `serv_listen` function

First, we create a single UNIX domain socket by calling `socket`. We then fill in a `sockaddr_un` structure with the well-known pathname to be assigned to the socket. This structure is the argument to `bind`. Note that we don't need to set the `sun_len` field present on some platforms, because the operating system sets this for us, deriving it from the address length we pass to the `bind` function.

Finally, we call `listen` (Section 16.4) to tell the kernel that the process will be acting as a server awaiting connections from clients. When a connect request from a client arrives, the server calls the `serv_accept` function (Figure 17.9).

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>
#include <errno.h>

#define STALE   30  /* client's name can't be older than this (sec) */

/*
 * Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID from the pathname
 * that it must bind before calling us.
 * Returns new fd if all OK, <0 on error
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    int                 clifd, err, rval;
    socklen_t           len;
    time_t              staletime;
    struct sockaddr_un  un;
    struct stat         statbuf;
    char                *name;

    /* allocate enough space for longest name plus terminating null */
```

```
    if ((name = malloc(sizeof(un.sun_path + 1))) == NULL)
        return(-1);
    len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0) {
        free(name);
        return(-2);     /* often errno=EINTR, if signal caught */
    }

    /* obtain the client's uid from its calling address */
    len -= offsetof(struct sockaddr_un, sun_path); /* len of pathname */
    memcpy(name, un.sun_path, len);
    name[len] = 0;          /* null terminate */
    if (stat(name, &statbuf) < 0) {
        rval = -3;
        goto errout;
    }

#ifdef  S_ISSOCK    /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0) {
        rval = -4;      /* not a socket */
        goto errout;
    }
#endif

    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
          rval = -5;    /* is not rwx------ */
          goto errout;
    }

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
        statbuf.st_ctime < staletime ||
        statbuf.st_mtime < staletime) {
          rval = -6;    /* i-node is too old */
          goto errout;
    }

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid;   /* return uid of caller */
    unlink(name);       /* we're done with pathname now */
    free(name);
    return(clifd);

errout:
    err = errno;
    close(clifd);
    free(name);
    errno = err;
    return(rval);
}
```

Figure 17.9 The serv_accept function

The server blocks in the call to `accept`, waiting for a client to call `cli_conn`. When `accept` returns, its return value is a brand-new descriptor that is connected to the client. Additionally, the pathname that the client assigned to its socket (the name that contained the client's process ID) is returned by `accept`, through the second argument (the pointer to the `sockaddr_un` structure). We copy this pathname and ensure that it is null terminated (if the pathname takes up all available space in the `sun_path` member of the `sockaddr_un` structure, there won't be room for the terminating null byte). Then we call `stat` to verify that the pathname is indeed a socket and that the permissions allow only user-read, user-write, and user-execute. We also verify that the three times associated with the socket are no older than 30 seconds. (Recall from Section 6.10 that the `time` function returns the current time and date in seconds past the Epoch.)

If all these checks are OK, we assume that the identity of the client (its effective user ID) is the owner of the socket. Although this check isn't perfect, it's the best we can do with current systems. (It would be better if the kernel returned the effective user ID to us through a parameter to `accept`.)

The client initiates the connection to the server by calling the `cli_conn` function (Figure 17.10).

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define CLI_PATH    "/var/tmp/"
#define CLI_PERM    S_IRWXU         /* rwx for user only */

/*
 * Create a client endpoint and connect to a server.
 * Returns fd if all OK, <0 on error.
 */
int
cli_conn(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un, sun;
    int                 do_unlink = 0;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /* fill socket address structure with our address */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    sprintf(un.sun_path, "%s%05ld", CLI_PATH, (long)getpid());
```

```
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);

    unlink(un.sun_path);        /* in case it already exists */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }
    if (chmod(un.sun_path, CLI_PERM) < 0) {
        rval = -3;
        do_unlink = 1;
        goto errout;
    }

    /* fill socket address structure with server's address */
    memset(&sun, 0, sizeof(sun));
    sun.sun_family = AF_UNIX;
    strcpy(sun.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);
    if (connect(fd, (struct sockaddr *)&sun, len) < 0) {
        rval = -4;
        do_unlink = 1;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    if (do_unlink)
        unlink(un.sun_path);
    errno = err;
    return(rval);
}
```

Figure 17.10 The `cli_conn` function

We call `socket` to create the client's end of a UNIX domain socket. We then fill in a `sockaddr_un` structure with a client-specific name.

We don't let the system choose a default address for us, because the server would be unable to distinguish one client from another (if we don't explicitly bind a name to a UNIX domain socket, the kernel implicitly binds an address to it on our behalf and no file is created in the file system to represent the socket). Instead, we bind our own address—a step we usually don't take when developing a client program that uses sockets.

The last five characters of the pathname we bind are made from the process ID of the client. We call `unlink`, just in case the pathname already exists. We then call `bind` to assign a name to the client's socket. This creates a socket file in the file system with the same name as the bound pathname. We call `chmod` to turn off all permissions other than user-read, user-write, and user-execute. In `serv_accept`, the server checks these permissions and the user ID of the socket to verify the client's identity.

We then have to fill in another `sockaddr_un` structure, this time with the well-known pathname of the server. Finally, we call the `connect` function to initiate the connection with the server.

17.4 Passing File Descriptors

Passing an open file descriptor between processes is a powerful technique. It can lead to different ways of designing client–server applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, and negotiating locks for the file) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

We must be more specific about what we mean by ''passing an open file descriptor'' from one process to another. Recall Figure 3.8, which showed two processes that have opened the same file. Although they share the same v-node, each process has its own file table entry.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Figure 17.11 shows the desired arrangement.

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.) Having two processes share an open file table is exactly what happens after a `fork` (recall Figure 8.2).

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

We define the following three functions that we use in this chapter to send and receive file descriptors. Later in this section, we'll show the code for these three functions.

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);

int send_err(int fd, int status, const char *errmsg);
```

Both return: 0 if OK, −1 on error

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

Returns: file descriptor if OK, negative value on error

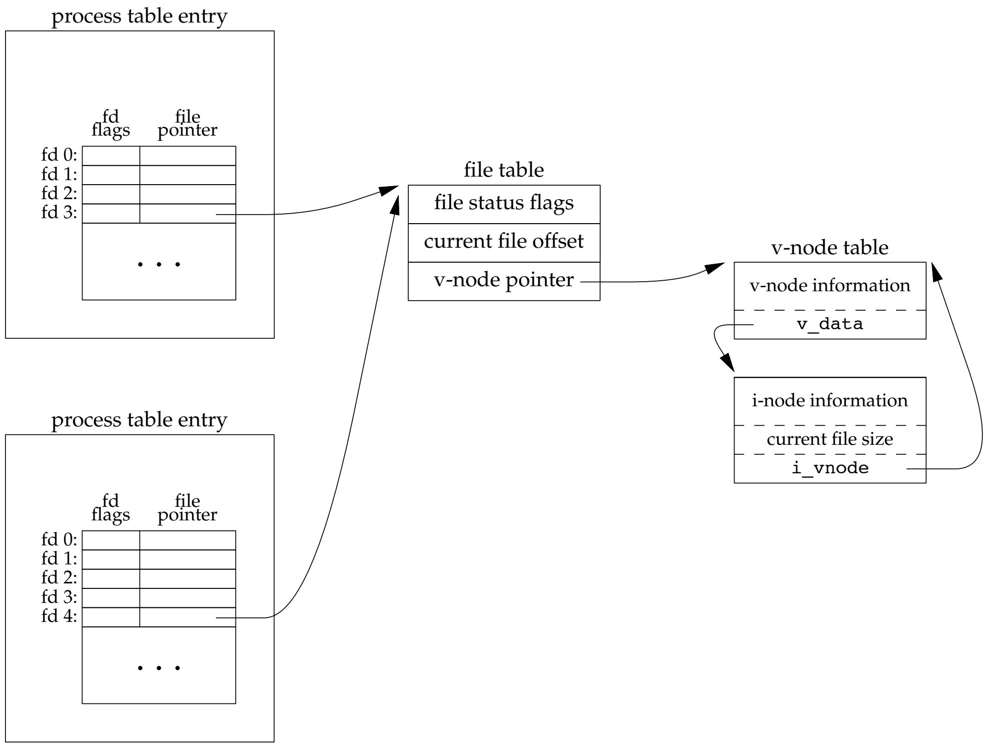

Figure 17.11 Passing an open file from the top process to the bottom process

A process (normally a server) that wants to pass a descriptor to another process calls either `send_fd` or `send_err`. The process waiting to receive the descriptor (the client) calls `recv_fd`.

The `send_fd` function sends the descriptor *fd_to_send* across using the UNIX domain socket represented by *fd*. The `send_err` function sends the *errmsg* using *fd*, followed by the *status* byte. The value of *status* must be in the range −1 through −255.

Clients call `recv_fd` to receive a descriptor. If all is OK (the sender called `send_fd`), the non-negative descriptor is returned as the value of the function. Otherwise, the value returned is the *status* that was sent by `send_err` (a negative value in the range −1 through −255). Additionally, if an error message was sent by the server, the client's *userfunc* is called to process the message. The first argument to *userfunc* is the constant `STDERR_FILENO`, followed by a pointer to the error message and its length. The return value from *userfunc* is the number of bytes written or a negative number on error. Often, the client specifies the normal `write` function as the *userfunc*.

We implement our own protocol that is used by these three functions. To send a descriptor, `send_fd` sends two bytes of 0, followed by the actual descriptor. To send an error, `send_err` sends the *errmsg*, followed by a byte of 0, followed by the absolute value of the *status* byte (1 through 255). The `recv_fd` function reads everything on the socket until it encounters a null byte. Any characters read up to this point are passed to the caller's *userfunc*. The next byte read by `recv_fd` is the status byte. If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

The function `send_err` calls the `send_fd` function after writing the error message to the socket. This is shown in Figure 17.12.

```
#include "apue.h"

/*
 * Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead.  We send the error back
 * using the send_fd()/recv_fd() protocol.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int     n;

    if ((n = strlen(msg)) > 0)
        if (writen(fd, msg, n) != n)    /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;   /* must be negative */

    if (send_fd(fd, errcode) < 0)
        return(-1);

    return(0);
}
```

Figure 17.12 The `send_err` function

To exchange file descriptors using UNIX domain sockets, we call the `sendmsg`(2) and `recvmsg`(2) functions (Section 16.5). Both functions take a pointer to a `msghdr` structure that contains all the information on what to send or receive. The structure on your system might look similar to the following:

```
struct msghdr {
    void         *msg_name;        /* optional address */
    socklen_t     msg_namelen;     /* address size in bytes */
    struct iovec *msg_iov;         /* array of I/O buffers */
    int           msg_iovlen;      /* number of elements in array */
    void         *msg_control;     /* ancillary data */
    socklen_t     msg_controllen;  /* number of ancillary bytes */
    int           msg_flags;       /* flags for received message */
};
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram. The next two elements allow us to specify an array of buffers (scatter read or gather write), as we described for the `readv` and `writev` functions (Section 14.6). The `msg_flags` field contains flags describing the message received, as summarized in Figure 16.15.

Two elements deal with the passing or receiving of control information. The `msg_control` field points to a `cmsghdr` (control message header) structure, and the `msg_controllen` field contains the number of bytes of control information.

```
struct cmsghdr {
    socklen_t  cmsg_len;    /* data byte count, including header */
    int        cmsg_level;  /* originating protocol */
    int        cmsg_type;   /* protocol-specific type */
    /* followed by the actual control message data */
};
```

To send a file descriptor, we set `cmsg_len` to the size of the `cmsghdr` structure, plus the size of an integer (the descriptor). The `cmsg_level` field is set to `SOL_SOCKET`, and `cmsg_type` is set to `SCM_RIGHTS`, to indicate that we are passing access rights. (`SCM` stands for *socket-level control message*.) Access rights can be passed only across a UNIX domain socket. The descriptor is stored right after the `cmsg_type` field, using the macro `CMSG_DATA` to obtain the pointer to this integer.

Three macros are used to access the control data, and one macro is used to help calculate the value to be used for `cmsg_len`.

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

unsigned char *CMSG_DATA(struct cmsghdr *cp);
```

Returns: pointer to data associated with `cmsghdr` structure

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

Returns: pointer to first `cmsghdr` structure associated with the `msghdr` structure, or `NULL` if none exists

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);
```

Returns: pointer to next `cmsghdr` structure associated with the `msghdr` structure given the current `cmsghdr` structure, or `NULL` if we're at the last one

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

Returns: size to allocate for data object *nbytes* large

The Single UNIX Specification defines the first three macros, but omits `CMSG_LEN`.

The `CMSG_LEN` macro returns the number of bytes needed to store a data object of size *nbytes*, after adding the size of the `cmsghdr` structure, adjusting for any alignment constraints required by the processor architecture, and rounding up.

The program in Figure 17.13 is the `send_fd` function, which passes a file descriptor over a UNIX domain socket. In the `sendmsg` call, we send both the protocol data (the null and the status byte) and the descriptor.

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

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;

    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send;      /* the fd to pass */
        buf[1] = 0;     /* zero status means OK */
    }

    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Figure 17.13 Sending a file descriptor over a UNIX domain socket

To receive a descriptor (Figure 17.14), we allocate enough room for a `cmsghdr` structure and a descriptor, set `msg_control` to point to the allocated area, and call `recvmsg`. We use the `CMSG_LEN` macro to calculate the amount of space needed.

We read from the socket until we read the null byte that precedes the final status byte. Everything up to this null byte is an error message from the sender.

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int             newfd, nr, status;
    char            *ptr;
    char            buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;

    status = -1;
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
```

```
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr);
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

Figure 17.14 Receiving a file descriptor over a UNIX domain socket

Note that we are always prepared to receive a descriptor (we set `msg_control` and `msg_controllen` before each call to `recvmsg`), but only if `msg_controllen` is nonzero on return did we actually receive a descriptor.

Recall the hoops we needed to jump through to determine the identity of the caller in the `serv_accept` function (Figure 17.9). It would have been better for the kernel to pass us the credentials of the caller on return from the call to `accept`. Some UNIX domain socket implementations provide similar functionality when exchanging messages, but their interfaces differ.

> FreeBSD 8.0 and Linux 3.2.0 provide support for sending credentials over UNIX domain sockets, but they do it differently. Mac OS X 10.6.8 is derived in part from FreeBSD, but has credential passing disabled. Solaris 10 doesn't support sending credentials over UNIX domain sockets. However, it supports the ability to obtain the credentials of a process passing a file descriptor over a STREAMS pipe, although we do not discuss the details here.

With FreeBSD, credentials are transmitted as a `cmsgcred` structure:

```
#define CMGROUP_MAX 16

struct cmsgcred {
    pid_t  cmcred_pid;                    /* sender's process ID */
    uid_t  cmcred_uid;                    /* sender's real UID */
    uid_t  cmcred_euid;                   /* sender's effective UID */
    gid_t  cmcred_gid;                    /* sender's real GID */
    short  cmcred_ngroups;                /* number of groups */
    gid_t  cmcred_groups[CMGROUP_MAX];    /* groups */
};
```

When we transmit credentials, we need to reserve space only for the `cmsgcred` structure. The kernel will fill in this structure for us to prevent an application from pretending to have a different identity.

On Linux, credentials are transmitted as a `ucred` structure:

```
struct ucred {
    pid_t  pid;  /* sender's process ID */
    uid_t  uid;  /* sender's user ID */
    gid_t  gid;  /* sender's group ID */
};
```

Unlike FreeBSD, Linux requires that we initialize this structure before transmission. The kernel will ensure that applications either use values that correspond to the caller or have the appropriate privilege to use other values.

Figure 17.15 shows the `send_fd` function updated to include the credentials of the sending process.

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

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
#define CREDSTRUCT      ucred
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct CREDSTRUCT   *credp;
    struct cmsghdr      *cmp;
    struct iovec        iov[1];
    struct msghdr       msg;
    char                buf[2]; /* send_fd/recv_ufd 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
```

```
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    msg.msg_flags = 0;
    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        cmp = cmptr;
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_RIGHTS;
        cmp->cmsg_len    = RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send;    /* the fd to pass */
        cmp = CMSG_NXTHDR(&msg, cmp);
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_CREDTYPE;
        cmp->cmsg_len    = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS)
        credp->uid = geteuid();
        credp->gid = getegid();
        credp->pid = getpid();
#endif
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Figure 17.15 Sending credentials over UNIX domain sockets

Note that we need to initialize the credentials structure only on Linux.

The function in Figure 17.16 is a modified version of `recv_fd`, called `recv_ufd`, that returns the user ID of the sender through a reference parameter.

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */
#include <sys/un.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define CR_UID          cmcred_uid
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
```

```
#define CREDSTRUCT      ucred
#define CR_UID          uid
#define CREDOPT         SO_PASSCRED
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_ufd(int fd, uid_t *uidptr,
         ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr      *cmp;
    struct CREDSTRUCT   *credp;
    char                *ptr;
    char                buf[MAXLINE];
    struct iovec        iov[1];
    struct msghdr       msg;
    int                 nr;
    int                 newfd = -1;
    int                 status = -1;
#if defined(CREDOPT)
    const int           on = 1;

    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt error");
        return(-1);
    }
#endif
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
```

```
            err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");

                    /* process the control data */
                    for (cmp = CMSG_FIRSTHDR(&msg);
                      cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                        if (cmp->cmsg_level != SOL_SOCKET)
                            continue;
                        switch (cmp->cmsg_type) {
                        case SCM_RIGHTS:
                            newfd = *(int *)CMSG_DATA(cmp);
                            break;
                        case SCM_CREDTYPE:
                            credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                            *uidptr = credp->CR_UID;
                        }
                    }
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

Figure 17.16 Receiving credentials over UNIX domain sockets

On FreeBSD, we specify `SCM_CREDS` to transmit credentials; on Linux, we use `SCM_CREDENTIALS`.

17.5 An Open Server, Version 1

Using file descriptor passing, we now develop an open server—a program that is executed by a process to open one or more files. Instead of sending the contents of the file back to the calling process, however, this server sends back an open file descriptor. As a result, the open server can work with any type of file (such as a device or a socket) and not simply regular files. The client and server exchange a minimum amount of information using IPC: the filename and open mode sent by the client, and the descriptor returned by the server. The contents of the file are not exchanged using IPC.

There are several advantages in designing the server to be a separate executable program (either one that is executed by the client, as we develop in this section, or a daemon server, which we develop in the next section).

- The server can easily be contacted by any client, similar to the client calling a library function. We are not hard-coding a particular service into the application, but designing a general facility that others can reuse.
- If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor). Shared libraries can simplify this updating (Section 7.7).
- The server can be a set-user-ID program, providing it with additional permissions that the client does not have. Note that a library function (or shared library function) can't provide this capability.

The client process creates an fd-pipe and then calls `fork` and `exec` to invoke the server. The client sends requests across the fd-pipe using one end, and the server sends back responses over the fd-pipe using the other end.

We define the following application protocol between the client and the server.

1. The client sends a request of the form "`open` <*pathname*> <*openmode*>\0" across the fd-pipe to the server. The <*openmode*> is the numeric value, in ASCII decimal, of the second argument to the `open` function. This request string is terminated by a null byte.
2. The server sends back an open descriptor or an error by calling either `send_fd` or `send_err`.

This is an example of a process sending an open descriptor to its parent. In Section 17.6, we'll modify this example to use a single daemon server, where the server sends a descriptor to a completely unrelated process.

We first have the header, `open.h` (Figure 17.17), which includes the standard headers and defines the function prototypes.

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

#define CL_OPEN "open"          /* client's request for server */

int     csopen(char *, int);
```

Figure 17.17 The `open.h` header

The main function (Figure 17.18) is a loop that reads a pathname from standard input and copies the file to standard output. The function calls csopen to contact the open server and return an open descriptor.

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

#define BUFFSIZE    8192

int
main(int argc, char *argv[])
{
    int     n, fd;
    char    buf[BUFFSIZE];
    char    line[MAXLINE];

    /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* replace newline with null */

        /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue;   /* csopen() prints error from server */

        /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }

    exit(0);
}
```

Figure 17.18 The client main function, version 1

The function csopen (Figure 17.19) does the fork and exec of the server, after creating the fd-pipe.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
```

```
    char            buf[10];
    struct iovec    iov[3];
    static int      fd[2] = { -1, -1 };

    if (fd[0] < 0) {    /* fork/exec our open server first time */
        if (fd_pipe(fd) < 0) {
            err_ret("fd_pipe error");
            return(-1);
        }
        if ((pid = fork()) < 0) {
            err_ret("fork error");
            return(-1);
        } else if (pid == 0) {      /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO &&
              dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            if (fd[1] != STDOUT_FILENO &&
              dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            if (execl("./opend", "opend", (char *)0) < 0)
                err_sys("execl error");
        }
        close(fd[1]);               /* parent */
    }
    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";      /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len) {
        err_ret("writev error");
        return(-1);
    }

    /* read descriptor, returned errors handled by write() */
    return(recv_fd(fd[0], write));
}
```

Figure 17.19 The csopen function, version 1

The child closes one end of the fd-pipe, and the parent closes the other. For the server that it executes, the child also duplicates its end of the fd-pipe onto its standard input and standard output. (Another option would have been to pass the ASCII representation of the descriptor `fd[1]` as an argument to the server.)

The parent sends to the server the request containing the pathname and open mode. Finally, the parent calls `recv_fd` to return either the descriptor or an error. If an error is returned by the server, `write` is called to output the message to standard error.

Now let's look at the open server. It is the program `opend` that is executed by the client in Figure 17.19. First, we have the `opend.h` header (Figure 17.20), which includes the standard headers and declares the global variables and function prototypes.

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

#define CL_OPEN "open"          /* client's request for server */

extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open() flag: O_xxx ... */
extern char *pathname;  /* of file to open() for client */

int      cli_args(int, char **);
void     handle_request(char *, int, int);
```

Figure 17.20 The `opend.h` header, version 1

The `main` function (Figure 17.21) reads the requests from the client on the fd-pipe (its standard input) and calls the function `handle_request`.

```
#include    "opend.h"

char     errmsg[MAXLINE];
int      oflag;
char    *pathname;

int
main(void)
{
    int     nread;
    char    buf[MAXLINE];

    for ( ; ; ) {   /* read arg buffer from client, process request */
        if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break;      /* client has closed the stream pipe */
        handle_request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

Figure 17.21 The server `main` function, version 1

The function `handle_request` in Figure 17.22 does all the work. It calls the function `buf_args` to break up the client's request into a standard `argv`-style argument list and calls the function `cli_args` to process the client's arguments. If all is OK, `open` is called to open the file, and then `send_fd` sends the descriptor back to the client across the fd-pipe (its standard output). If an error is encountered, `send_err` is called to send back an error message, using the client–server protocol that we described earlier.

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

void
handle_request(char *buf, int nread, int fd)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
          "request not null terminated: %*.*s\n", nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }
    if (buf_args(buf, cli_args) < 0) {  /* parse args & set options */
        send_err(fd, -1, errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can't open %s: %s\n", pathname,
          strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }
    if (send_fd(fd, newfd) < 0)     /* send the descriptor */
        err_sys("send_fd error");
    close(newfd);       /* we're done with descriptor */
}
```

Figure 17.22 The `handle_request` function, version 1

The client's request is a null-terminated string of white-space-separated arguments. The function `buf_args` in Figure 17.23 breaks this string into a standard `argv`-style argument list and calls a user function to process the arguments. We use the ISO C function `strtok` to tokenize the string into separate arguments.

```
#include "apue.h"

#define MAXARGC     50  /* max number of arguments in buf */
#define WHITE   " \t\n" /* white space for tokenizing arguments */

/*
 * buf[] contains white-space-separated arguments.  We convert it to an
 * argv-style array of pointers, and call the user's function (optfunc)
 * to process the array.  We return -1 if there's a problem parsing buf,
 * else we return whatever optfunc() returns.  Note that user's buf[]
 * array is modified (nulls placed after each token).
 */
int
buf_args(char *buf, int (*optfunc)(int, char **))
{
```

```
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL)     /* an argv[0] is required */
        return(-1);
    argv[argc = 0] = buf;
    while ((ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)    /* -1 for room for NULL at end */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    /*
     * Since argv[] pointers point into the user's buf[],
     * user's function can just copy the pointers, even
     * though argv[] array will disappear on return.
     */
    return((*optfunc)(argc, argv));
}
```

Figure 17.23 The buf_args function

The server's function that is called by `buf_args` is `cli_args` (Figure 17.24). It verifies that the client sent the right number of arguments and stores the pathname and open mode in global variables.

```
#include    "opend.h"

/*
 * This function is called by buf_args(), which is called by
 * handle_request().  buf_args() has broken up the client's
 * buffer into an argv[]-style array, which we now process.
 */
int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }
    pathname = argv[1];     /* save ptr to pathname to open */
    oflag = atoi(argv[2]);
    return(0);
}
```

Figure 17.24 The cli_args function

This completes the open server that is invoked by a `fork` and `exec` from the client. A single fd-pipe is created before the `fork` and is used to communicate between the client and the server. With this arrangement, we have one server per client.

17.6 An Open Server, Version 2

In the previous section, we developed an open server that was invoked by a `fork` and `exec` by the client, demonstrating how we can pass file descriptors from a child to a parent. In this section, we develop an open server as a daemon process. One server handles all clients. We expect this design to be more efficient, since a `fork` and an `exec` are avoided. We use a UNIX domain socket connection between the client and the server and demonstrate passing file descriptors between unrelated processes. We'll use the three functions `serv_listen`, `serv_accept`, and `cli_conn` introduced in Section 17.3. This server also demonstrates how a single server can handle multiple clients, using both the `select` and `poll` functions from Section 14.4.

This version of the client is similar to the client from Section 17.5. Indeed, the file `main.c` is identical (Figure 17.18). We add the following line to the `open.h` header (Figure 17.17):

```
#define CS_OPEN "/tmp/opend.socket"  /* server's well-known name */
```

The file `open.c` does change from Figure 17.19, since we now call `cli_conn` instead of doing the `fork` and `exec`. This is shown in Figure 17.25.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    int             len;
    char            buf[12];
    struct iovec    iov[3];
    static int      csfd = -1;

    if (csfd < 0) {     /* open connection to conn server */
        if ((csfd = cli_conn(CS_OPEN)) < 0) {
            err_ret("cli_conn error");
            return(-1);
        }
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";  /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* null always sent */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
```

```
    if (writev(csfd, &iov[0], 3) != len) {
        err_ret("writev error");
        return(-1);
    }

    /* read back descriptor; returned errors handled by write() */
    return(recv_fd(csfd, write));
}
```

Figure 17.25 The csopen function, version 2

The protocol from the client to the server remains the same.

Next, we'll look at the server. The header `opend.h` (Figure 17.26) includes the standard headers and declares the global variables and the function prototypes.

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

#define CS_OPEN "/tmp/opend.socket" /* well-known name */
#define CL_OPEN "open"              /* client's request for server */

extern int   debug;     /* nonzero if interactive (not daemon) */
extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open flag: O_xxx ... */
extern char *pathname;  /* of file to open for client */

typedef struct {    /* one Client struct per connected client */
  int   fd;         /* fd, or -1 if available */
  uid_t uid;
} Client;

extern Client   *client;        /* ptr to malloc'ed array */
extern int       client_size;   /* # entries in client[] array */

int      cli_args(int, char **);
int      client_add(int, uid_t);
void     client_del(int);
void     loop(void);
void     handle_request(char *, int, int, uid_t);
```

Figure 17.26 The opend.h header, version 2

Since this server handles all clients, it must maintain the state of each client connection. This is done with the `client` array declared in the `opend.h` header. Figure 17.27 defines three functions that manipulate this array.

```
#include    "opend.h"

#define NALLOC  10      /* # client structs to alloc/realloc for */

static void
client_alloc(void)      /* alloc more entries in the client[] array */
```

```
{
    int     i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size+NALLOC)*sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

    /* initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;  /* fd of -1 means entry available */

    client_size += NALLOC;
}

/*
 * Called by loop() when connection request from a new client arrives.
 */
int
client_add(int fd, uid_t uid)
{
    int     i;

    if (client == NULL)     /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) {   /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i);  /* return index in client[] array */
        }
    }

    /* client array full, time to realloc for more */
    client_alloc();
    goto again;     /* and search again (will work this time) */
}

/*
 * Called by loop() when we're done with a client.
 */
void
client_del(int fd)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
```

```
                return;
            }
        }
        log_quit("can't find client entry for fd %d", fd);
    }
```

Figure 17.27 Functions to manipulate `client` array

The first time `client_add` is called, it calls `client_alloc`, which in turn calls `malloc` to allocate space for ten entries in the array. After these ten entries are all in use, a later call to `client_add` causes `realloc` to allocate additional space. By dynamically allocating space this way, we have not limited the size of the `client` array at compile time to some value that we guessed and put into a header. These functions call the `log_` functions (Appendix B) if an error occurs, since we assume that the server is a daemon.

Normally the server will run as a daemon, but we want to provide an option that allows it to be run in the foreground, with diagnostic messages sent to the standard error. This should make the server easier to test and debug, especially if we don't have permission to read the log file where the diagnostic messages are normally written. We'll use a command-line option to control whether the server runs in the foreground or as a daemon in the background.

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 immediately 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 <unistd.h>

int getopt(int argc, char * 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 `getopt` function is normally used 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 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 open server's `main` function (Figure 17.28) defines the global variables, processes the command-line options, and calls the function `loop`. If we invoke the server with the `-d` option, the server runs interactively instead of as a daemon. This option is used when testing the server.

```
#include    "opend.h"
#include    <syslog.h>

int      debug, oflag, client_size, log_to_stderr;
char     errmsg[MAXLINE];
char    *pathname;
```

```
Client  *client = NULL;

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

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ((c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = log_to_stderr = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemonize("opend");

    loop();     /* never returns */
}
```

Figure 17.28 The server `main` function, version 2

The function `loop` is the server's infinite loop. We'll show two versions of this function. Figure 17.29 shows one version that uses `select`; Figure 17.30 shows another version that uses `poll`.

```
#include    "opend.h"
#include    <sys/select.h>

void
loop(void)
{
    int     i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;

    FD_ZERO(&allset);

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    FD_SET(listenfd, &allset);
    maxfd = listenfd;
    maxi = -1;
```

```
    for ( ; ; ) {
        rset = allset;  /* rset gets modified each time around */
        if ((n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
            log_sys("select error");

        if (FD_ISSET(listenfd, &rset)) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            FD_SET(clifd, &allset);
            if (clifd > maxfd)
                maxfd = clifd;  /* max fd for select() */
            if (i > maxi)
                maxi = i;   /* max index in client[] array */
            log_msg("new connection: uid %d, fd %d", uid, clifd);
            continue;
        }

        for (i = 0; i <= maxi; i++) {   /* go through client[] array */
            if ((clifd = client[i].fd) < 0)
                continue;
            if (FD_ISSET(clifd, &rset)) {
                /* read argument buffer from client */
                if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", clifd);
                } else if (nread == 0) {
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, clifd);
                    client_del(clifd);  /* client has closed cxn */
                    FD_CLR(clifd, &allset);
                    close(clifd);
                } else {    /* process client's request */
                    handle_request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.29 The `loop` function using `select`

This function calls `serv_listen` (Figure 17.8) to create the server's endpoint for the client connections. The remainder of the function is a loop that starts with a call to `select`. Two conditions can be true after `select` returns.

1. The descriptor `listenfd` can be ready for reading, which means that a new client has called `cli_conn`. To handle this, we call `serv_accept` (Figure 17.9) and then update the `client` array and associated bookkeeping information for the new client. (We keep track of the highest descriptor number for the first

argument to `select`. We also keep track of the highest index in use in the `client` array.)

2. An existing client's connection can be ready for reading. This means that the client has either terminated or sent a new request. We find out about a client termination by `read` returning 0 (end of file). If `read` returns a value greater than 0, there is a new request to process, which we handle by calling `handle_request`.

We keep track of which descriptors are currently in use in the `allset` descriptor set. As new clients connect to the server, the appropriate bit is turned on in this descriptor set. The appropriate bit is turned off when the client terminates.

We always know when a client terminates, whether the termination is voluntary or not, since all the client's descriptors (including the connection to the server) are automatically closed by the kernel. This differs from the XSI IPC mechanisms.

The `loop` function that uses `poll` is shown in Figure 17.30.

```
#include    "opend.h"
#include    <poll.h>

#define NALLOC  10  /* # pollfd structs to alloc/realloc */

static struct pollfd *
grow_pollfd(struct pollfd *pfd, int *maxfd)
{
    int             i;
    int             oldmax = *maxfd;
    int             newmax = oldmax + NALLOC;

    if ((pfd = realloc(pfd, newmax * sizeof(struct pollfd))) == NULL)
        err_sys("realloc error");
    for (i = oldmax; i < newmax; i++) {
        pfd[i].fd = -1;
        pfd[i].events = POLLIN;
        pfd[i].revents = 0;
    }
    *maxfd = newmax;
    return(pfd);
}

void
loop(void)
{
    int             i, listenfd, clifd, nread;
    char            buf[MAXLINE];
    uid_t           uid;
    struct pollfd   *pollfd;
    int             numfd = 1;
    int             maxfd = NALLOC;

    if ((pollfd = malloc(NALLOC * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");
```

```
    for (i = 0; i < NALLOC; i++) {
        pollfd[i].fd = -1;
        pollfd[i].events = POLLIN;
        pollfd[i].revents = 0;
    }

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0);    /* we use [0] for listenfd */
    pollfd[0].fd = listenfd;

    for ( ; ; ) {
        if (poll(pollfd, numfd, -1) < 0)
            log_sys("poll error");

        if (pollfd[0].revents & POLLIN) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            client_add(clifd, uid);

            /* possibly increase the size of the pollfd array */
            if (numfd == maxfd)
                pollfd = grow_pollfd(pollfd, &maxfd);
            pollfd[numfd].fd = clifd;
            pollfd[numfd].events = POLLIN;
            pollfd[numfd].revents = 0;
            numfd++;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i < numfd; i++) {
            if (pollfd[i].revents & POLLHUP) {
                goto hungup;
            } else if (pollfd[i].revents & POLLIN) {
                /* read argument buffer from client */
                if ((nread = read(pollfd[i].fd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", pollfd[i].fd);
                } else if (nread == 0) {
hungup:
                    /* the client closed the connection */
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, pollfd[i].fd);
                    client_del(pollfd[i].fd);
                    close(pollfd[i].fd);
                    if (i < (numfd-1)) {
                        /* pack the array */
                        pollfd[i].fd = pollfd[numfd-1].fd;
                        pollfd[i].events = pollfd[numfd-1].events;
                        pollfd[i].revents = pollfd[numfd-1].revents;
                        i--;    /* recheck this entry */
```

```
                    }
                    numfd--;
                } else {        /* process client's request */
                    handle_request(buf, nread, pollfd[i].fd,
                      client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.30 The loop function using poll

To allow for as many clients as there are possible open descriptors, we dynamically allocate space for the array of `pollfd` structures using the same strategy as used in the `client_alloc` function for the `client` array (see Figure 17.27).

We use the first entry (index 0) of the `pollfd` array for the `listenfd` descriptor. The arrival of a new client connection is indicated by a `POLLIN` on the `listenfd` descriptor. As before, we call `serv_accept` to accept the connection.

For an existing client, we have to handle two different events from `poll`: a client termination is indicated by `POLLHUP`, and a new request from an existing client is indicated by `POLLIN`. The client can close its end of the connection while there is still data to be read from the server's end of the connection. Even though the endpoint is marked as hung up, the server can read all the data queued on its end. But with this server, when we receive the hangup from the client, we can `close` the connection to the client, effectively throwing away any queued data. There is no reason to process any requests still remaining, since we can't send any responses back.

As with the `select` version of this function, new requests from a client are handled by calling the `handle_request` function (Figure 17.31). This function is similar to the earlier version (Figure 17.22). It calls the same function, `buf_args` (Figure 17.23), that calls `cli_args` (Figure 17.24), but since it runs from a daemon process, it logs error messages instead of printing them on the standard error.

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

void
handle_request(char *buf, int nread, int clifd, uid_t uid)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
          "request from uid %d not null terminated: %*.*s\n",
          uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("request: %s, from uid %d", buf, uid);
```

```
    /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can't open %s: %s\n",
          pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* send the descriptor */
    if (send_fd(clifd, newfd) < 0)
        log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd);       /* we're done with descriptor */
}
```

Figure 17.31 The `request` function, version 2

This completes the second version of the open server, which uses a single daemon to handle all the client requests.

17.7 Summary

The key points in this chapter are the ability to pass file descriptors between processes and the ability of a server to accept unique connections from clients. Although all platforms provide support for UNIX domain sockets (refer back to Figure 15.1), we've seen that there are differences in each implementation, which makes it more difficult for us to develop portable applications.

We used UNIX domain sockets throughout this chapter. We saw how to use them to implement a full-duplex pipe and how they can be used to adapt the I/O multiplexing functions from Section 14.4 to work indirectly with XSI message queues.

We presented two versions of an open server. One version was invoked directly by the client, using `fork` and `exec`. The second was a daemon server that handled all client requests. Both versions used the file descriptor passing and receiving functions.

We also saw how to use the `getopt` function to enforce consistent command-line processing for our programs. The final version of the open server used the `getopt` function, the client–server connection functions introduced in Section 17.3, and the I/O multiplexing functions from Section 14.4.

Exercises

17.1 We chose to use UNIX domain datagram sockets in Figure 17.3, because they retain message boundaries. Describe the changes that would be necessary to use regular pipes instead. How can we avoid copying the messages two extra times?

17.2 Write the following program using the file descriptor passing functions from this chapter and the parent–child synchronization routines from Section 8.9. The program calls `fork`, and the child `open`s an existing file and passes the open descriptor to the parent. The child then positions the file using `lseek` and notifies the parent. The parent reads the file's current offset and prints it for verification. If the file was passed from the child to the parent as we described, they should be sharing the same file table entry, so each time the child changes the file's current offset, that change should also affect the parent's descriptor. Have the child position the file to a different offset and notify the parent again.

17.3 In Figures 17.20 and 17.21, we differentiated between declaring and defining the global variables. What is the difference?

17.4 Recode the `buf_args` function (Figure 17.23), removing the compile-time limit on the size of the `argv` array. Use dynamic memory allocation.

17.5 Describe ways to optimize the function `loop` in Figure 17.29 and Figure 17.30. Implement your optimizations.

17.6 In the `serv_listen` function (Figure 17.8), we unlink the name of the file representing the UNIX domain socket if the file already exists. To avoid unintentionally removing a file that isn't a socket, we could call `stat` first to verify the file type. Explain the two problems with this approach.

17.7 Describe two possible ways to pass more than one file descriptor with a single call to `sendmsg`. Try them out to see if they are supported by your operating system.