response. If the server offered multiple services, we could use this request message to indicate the service we want, but since the server does only one thing, the content of the 1-byte message doesn't matter.

If the server isn't running, the client will block indefinitely in the call to recvfrom. With the connection-oriented example, the connect call will fail if the server isn't running. To avoid blocking indefinitely, we set an alarm clock before calling recvfrom. □

Example—Connectionless Server

The program in Figure 16.18 is the datagram version of the uptime server.

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

#define BUFLEN      128
#define MAXADDRLEN  256

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, struct sockaddr *  socklen_t, int);

void
serve(int sockfd)
{
    int         n;
    socklen_t   alen;
    FILE        *fp;
    char        buf[BUFLEN];
    char        abuf[MAXADDRLEN];

    for (;;) {
        alen = MAXADDRLEN;
        if ((n = recvfrom(sockfd, buf, BUFLEN, 0,
          (struct sockaddr *)abuf, &alen)) < 0) {
            syslog(LOG_ERR, "ruptimed: recvfrom error: %s",
              strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "error: %s\n", strerror(errno));
            sendto(sockfd, buf, strlen(buf), 0,
              (struct sockaddr *)abuf, alen);
        } else {
            if (fgets(buf, BUFLEN, fp) != NULL)
                sendto(sockfd, buf, strlen(buf), 0,
                    (struct sockaddr *)abuf, alen);
                pclose(fp);
            }
        }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
#ifdef _SC_HOST_NAME_MAX
    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)  /* best guess */
#endif
        n = HOST_NAME_MAX;
    host = malloc(n);
    if (host == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_DGRAM, aip->ai_addr,
          aip->ai_addrlen, 0)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.18 Server providing system uptime over datagrams

The server blocks in `recvfrom` for a request for service. When a request arrives, we save the requester's address and use `popen` to run the `uptime` command. We send the output back to the client using the `sendto` function, with the destination address set to the requester's address. □

16.6 Socket Options

The socket mechanism provides two socket-option interfaces for us to control the behavior of sockets. One interface is used to set an option, and another interface allows us to query the state of an option. We can get and set three kinds of options:

1. Generic options that work with all socket types
2. Options that are managed at the socket level, but depend on the underlying protocols for support
3. Protocol-specific options unique to each individual protocol

The Single UNIX Specification defines only the socket-layer options (the first two option types in the preceding list).

We can set a socket option with the `setsockopt` function.

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

int setsockopt(int sockfd, int level, int option, const void *val,
               socklen_t len);

                                        Returns: 0 if OK, -1 on error
```

The *level* argument identifies the protocol to which the option applies. If the option is a generic socket-level option, then *level* is set to `SOL_SOCKET`. Otherwise, *level* is set to the number of the protocol that controls the option. Examples are `IPPROTO_TCP` for TCP options and `IPPROTO_IP` for IP options. Figure 16.19 summarizes the generic socket-level options defined by the Single UNIX Specification.

The *val* argument points to a data structure or an integer, depending on the option. Some options are on/off switches. If the integer is nonzero, then the option is enabled. If the integer is zero, then the option is disabled. The *len* argument specifies the size of the object to which *val* points.

We can find out the current value of an option with the `getsockopt` function.

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

int getsockopt(int sockfd, int level, int option, void *restrict val,
               socklen_t *restrict lenp);

                                        Returns: 0 if OK, -1 on error
```

Note that the *lenp* argument is a pointer to an integer. Before calling `getsockopt`, we set the integer to the size of the buffer where the option is to be copied. If the actual size

Option	Type of *val* argument	Description
SO_ACCEPTCONN	int	Return whether a socket is enabled for listening (getsockopt only).
SO_BROADCAST	int	Broadcast datagrams if *val is nonzero.
SO_DEBUG	int	Debugging in network drivers enabled if *val is nonzero.
SO_DONTROUTE	int	Bypass normal routing if *val is nonzero.
SO_ERROR	int	Return and clear pending socket error (getsockopt only).
SO_KEEPALIVE	int	Periodic keep-alive messages enabled if *val is nonzero.
SO_LINGER	struct linger	Delay time when unsent messages exist and socket is closed.
SO_OOBINLINE	int	Out-of-band data placed inline with normal data if *val is nonzero.
SO_RCVBUF	int	The size in bytes of the receive buffer.
SO_RCVLOWAT	int	The minimum amount of data in bytes to return on a receive call.
SO_RCVTIMEO	struct timeval	The timeout value for a socket receive call.
SO_REUSEADDR	int	Reuse addresses in bind if *val is nonzero.
SO_SNDBUF	int	The size in bytes of the send buffer.
SO_SNDLOWAT	int	The minimum amount of data in bytes to transmit in a send call.
SO_SNDTIMEO	struct timeval	The timeout value for a socket send call.
SO_TYPE	int	Identify the socket type (getsockopt only).

Figure 16.19 Socket options

of the option is greater than this size, the option is silently truncated. If the actual size of the option is less than or equal to this size, then the integer is updated with the actual size on return.

Example

The function in Figure 16.10 fails to operate properly when the server terminates and we try to restart it immediately. Normally, the implementation of TCP will prevent us from binding the same address until a timeout expires, which is usually on the order of several minutes. Luckily, the SO_REUSEADDR socket option allows us to bypass this restriction, as illustrated in Figure 16.20.

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

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd, err;
    int reuse = 1;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &reuse,
      sizeof(int)) < 0) {
        err = errno;
        goto errout;
    }
    if (bind(fd, addr, alen) < 0) {
        err = errno;
        goto errout;
    }
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
        if (listen(fd, qlen) < 0) {
            err = errno;
            goto errout;
        }
    }
    return(fd);

errout:
    close(fd);
    errno = err;
    return(-1);
}
```

Figure 16.20 Initialize a socket endpoint for use by a server with address reuse

To enable the SO_REUSEADDR option, we set an integer to a nonzero value and pass the address of the integer as the *val* argument to setsockopt. We set the *len* argument to the size of an integer to indicate the size of the object to which *val* points. □

16.7 Out-of-Band Data

Out-of-band data is an optional feature supported by some communication protocols, allowing higher-priority delivery of data than normal. Out-of-band data is sent ahead of any data that is already queued for transmission. TCP supports out-of-band data, but UDP doesn't. The socket interface to out-of-band data is heavily influenced by TCP's implementation of out-of-band data.

TCP refers to out-of-band data as "urgent" data. TCP supports only a single byte of urgent data, but allows urgent data to be delivered out of band from the normal data delivery mechanisms. To generate urgent data, we specify the MSG_OOB flag to any of the three send functions. If we send more than one byte with the MSG_OOB flag, the last byte will be treated as the urgent-data byte.

When urgent data is received, we are sent the SIGURG signal if we have arranged for signal generation by the socket. In Sections 3.14 and 14.6.2, we saw that we could use the F_SETOWN command to fcntl to set the ownership of a socket. If the third argument to fcntl is positive, it specifies a process ID. If it is a negative value other than −1, it represents the process group ID. Thus, we can arrange that our process receive signals from a socket by calling

```
fcntl(sockfd, F_SETOWN, pid);
```

The F_GETOWN command can be used to retrieve the current socket ownership. As with the F_SETOWN command, a negative value represents a process group ID, and a positive value represents a process ID. Thus, the call

```
owner = fcntl(sockfd, F_GETOWN, 0);
```

will return with owner equal to the ID of the process configured to receive signals from the socket if owner is positive and with the absolute value of owner equal to the ID of the process group configured to receive signals from the socket if owner is negative.

TCP supports the notion of an *urgent mark*: the point in the normal data stream where the urgent data would go. We can choose to receive the urgent data inline with the normal data if we use the SO_OOBINLINE socket option. To help us identify when we have reached the urgent mark, we can use the sockatmark function.

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

int sockatmark(int sockfd);

                         Returns: 1 if at mark, 0 if not at mark, −1 on error
```

When the next byte to be read is where the urgent mark is located, sockatmark will return 1.

When out-of-band data is present in a socket's read queue, the select function (Section 14.5.1) will return the file descriptor as having an exception condition pending. We can choose to receive the urgent data inline with the normal data, or we can use the MSG_OOB flag with one of the recv functions to receive the urgent data ahead of any other queue data. TCP queues only one byte of urgent data. If another urgent byte arrives before we receive the current one, the existing one is discarded.

16.8 Nonblocking and Asynchronous I/O

Normally, the recv functions will block when no data is immediately available. Similarly, the send functions will block when there is not enough room in the socket's output queue to send the message. This behavior changes when the socket is in nonblocking mode. In this case, these functions will fail instead of blocking, setting errno to either EWOULDBLOCK or EAGAIN. When this happens, we can use either poll or select to determine when we can receive or transmit data.

The real-time extensions in the Single UNIX Specification include support for a generic asynchronous I/O mechanism. The socket mechanism has its own way of handling asynchronous I/O, but this isn't standardized in the Single UNIX Specification. Some texts refer to the classic socket-based asynchronous I/O mechanism as "signal-based I/O" to distinguish it from the asynchronous I/O mechanism in the real-time extensions.

With socket-based asynchronous I/O, we can arrange to be sent the `SIGIO` signal when we can read data from a socket or when space becomes available in a socket's write queue. Enabling asynchronous I/O is a two-step process.

1. Establish socket ownership so signals can be delivered to the proper processes.
2. Inform the socket that we want it to signal us when I/O operations won't block.

We can accomplish the first step in three ways.

1. Use the `F_SETOWN` command with `fcntl`.
2. Use the `FIOSETOWN` command with `ioctl`.
3. Use the `SIOCSPGRP` command with `ioctl`.

To accomplish the second step, we have two choices.

1. Use the `F_SETFL` command with `fcntl` and enable the `O_ASYNC` file flag.
2. Use the `FIOASYNC` command with `ioctl`.

We have several options, but they are not universally supported. Figure 16.21 summarizes the support for these options provided by the platforms discussed in this text. We show • where support is provided and † where support depends on the particular domain. For example, on Linux, the UNIX domain sockets don't support `FIOSETOWN` or `SIOCSPGRP`.

Mechanism	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`fcntl(fd, F_SETOWN, pid)`	•	•	•	•	•
`ioctl(fd, FIOSETOWN, pid)`		•	†	•	•
`ioctl(fd, SIOCSPGRP, pid)`		•	†	•	•
`fcntl(fd, F_SETFL, flags|O_ASYNC)`		•	•	•	
`ioctl(fd, FIOASYNC, &n);`		•	•	•	•

Figure 16.21 Socket asynchronous I/O management commands

16.9 Summary

In this chapter, we looked at the IPC mechanisms that allow processes to communicate with other processes on different machines as well as within the same machine. We discussed how socket endpoints are named and how we can discover the addresses to use when contacting servers.

We presented examples of clients and servers that use connectionless (i.e., datagram-based) sockets and connection-oriented sockets. We briefly discussed asynchronous and nonblocking socket I/O and the interfaces used to manage socket options.

In the next chapter, we will look at some advanced IPC topics, including how we can use sockets to pass file descriptors between processes running on the same machine.

Exercises

16.1 Write a program to determine your system's byte ordering.

16.2 Write a program to print out which `stat` structure members are supported for sockets on at least two different platforms, and describe how the results differ.

16.3 The program in Figure 16.15 provides service on only a single endpoint. Modify the program to support service on multiple endpoints (each with a different address) at the same time.

16.4 Write a client program and a server program to return the number of processes currently running on a specified host computer.

16.5 In the program in Figure 16.16, the server waits for the child to execute the `uptime` command and exit before accepting the next connect request. Redesign the server so that the time to service one request doesn't delay the processing of incoming connect requests.

16.6 Write two library routines: one to enable asynchronous I/O on a socket and one to disable asynchronous I/O on a socket. Use Figure 16.21 to make sure that the functions work on all platforms with as many socket types as possible.

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 two advanced forms of IPC—STREAMS-based pipes and UNIX domain sockets—and what we can do with them. With these forms of IPC, we can pass open file descriptors between processes, servers can associate names with their file descriptors, and clients can use these names to rendezvous with the servers. We'll also see how the operating system provides a unique IPC channel per client. Many of the ideas that form the basis for the techniques described in this chapter come from the paper by Presotto and Ritchie [1990].

17.2 STREAMS-Based Pipes

A STREAMS-based pipe ("STREAMS pipe," for short) is a bidirectional (full-duplex) pipe. To obtain bidirectional data flow between a parent and a child, only a single STREAMS pipe is required.

Recall from Section 15.1 that STREAMS pipes are supported by Solaris and are available in an optional add-on package with Linux.

Figure 17.1 shows the two ways to view a STREAMS pipe. The only difference between this picture and Figure 15.2 is that the arrows have heads on both ends; since the STREAMS pipe is full duplex, data can flow in both directions.

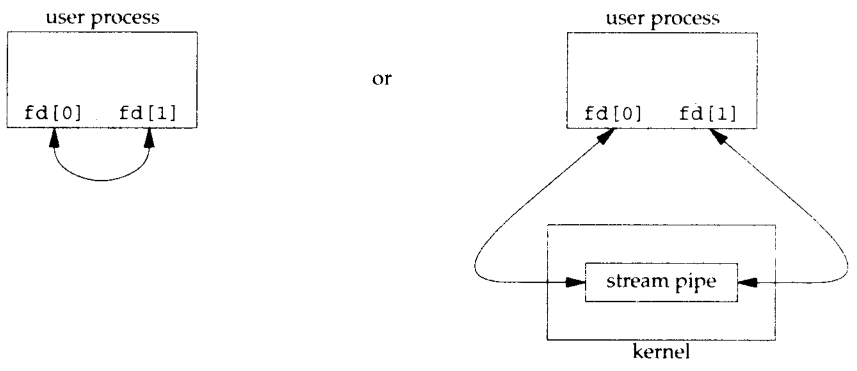

Figure 17.1 Two ways to view a STREAMS pipe

If we look inside a STREAMS pipe (Figure 17.2), we see that it is simply two stream heads, with each write queue (WQ) pointing at the other's read queue (RQ). Data written to one end of the pipe is placed in messages on the other's read queue.

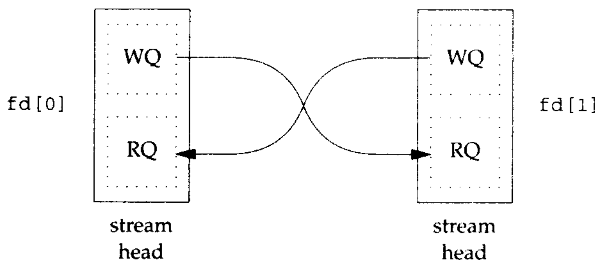

Figure 17.2 Inside a STREAMS pipe

Since a STREAMS pipe is a stream, we can push a STREAMS module onto either end of the pipe to process data written to the pipe (Figure 17.3). But if we push a module on one end, we can't pop it off the other end. If we want to remove it, we need to remove it from the same end on which it was pushed.

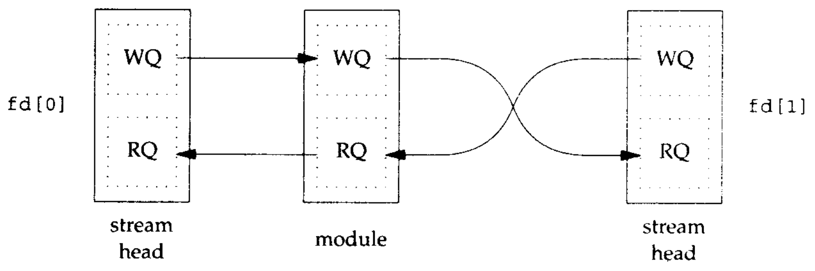

Figure 17.3 Inside a STREAMS pipe with a module

Assuming that we don't do anything fancy, such as pushing modules, a STREAMS pipe behaves just like a non-STREAMS pipe, except that it supports most of the STREAMS `ioctl` commands described in `streamio`(7). In Section 17.2.2, we'll see an example of pushing a module on a STREAMS pipe to provide unique connections when we give the pipe a name in the file system.

Example

Let's redo the coprocess example, Figure 15.18, with a single STREAMS pipe. Figure 17.4 shows the new `main` function. The `add2` coprocess is the same (Figure 15.17). We call a new function, `s_pipe`, to create a single STREAMS pipe. (We show versions of this function for both STREAMS pipes and UNIX domain sockets shortly.)

```
#include "apue.h"

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

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

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

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

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                               /* parent */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;     /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }
        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                    /* child */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO &&
          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("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
    }
    exit(0);
}

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

Figure 17.4 Program to drive the add2 filter, using a STREAMS pipe

The parent uses only fd[0], and the child uses only fd[1]. Since each end of the STREAMS pipe is full duplex, the parent reads and writes fd[0], and the child duplicates fd[1] to both standard input and standard output. Figure 17.5 shows the resulting descriptors. Note that this example also works with full-duplex pipes that are not based on STREAMS, because it doesn't make use of any STREAMS features other than the full-duplex nature of STREAMS-based pipes.

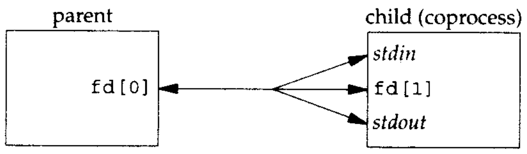

Figure 17.5 Arrangement of descriptors for coprocess

Rago [1993] covers STREAMS-based pipes in more detail. Recall from Figure 15.1 that FreeBSD supports full-duplex pipes, but these pipes are not based on the STREAMS mechanism.

□

We define the function s_pipe to be similar to the standard pipe function. Both functions take the same argument, but the descriptors returned by s_pipe are open for reading and writing.

Example—STREAMS-Based s_pipe Function

Figure 17.6 shows the STREAMS-based version of the s_pipe function. This version simply calls the standard pipe function, which creates a full-duplex pipe.

```
#include "apue.h"

/*
 * Returns a STREAMS-based pipe, with the two file descriptors
 * returned in fd[0] and fd[1].
 */
int
s_pipe(int fd[2])
{
    return(pipe(fd));
}
```

Figure 17.6 STREAMS version of the s_pipe function

□

17.2.1 Naming STREAMS Pipes

Normally, pipes can be used only between related processes: child processes inheriting pipes from their parent processes. In Section 15.5, we saw that unrelated processes can communicate using FIFOs, but this provides only a one-way communication path. The STREAMS mechanism provides a way for processes to give a pipe a name in the file system. This bypasses the problem of dealing with unidirectional FIFOs.

We can use the fattach function to give a STREAMS pipe a name in the file system.

```
#include <stropts.h>

int fattach(int filedes, const char *path);

                                    Returns: 0 if OK, -1 on error
```

The *path* argument must refer to an existing file, and the calling process must either own the file and have write permissions to it or be running with superuser privileges.

Once a STREAMS pipe is attached to the file system namespace, the underlying file is inaccessible. Any process that opens the name will gain access to the pipe, not the underlying file. Any processes that had the underlying file open before fattach was called, however, can continue to access the underlying file. Indeed, these processes generally will be unaware that the name now refers to a different file.

Figure 17.7 shows a pipe attached to the pathname /tmp/pipe. Only one end of the pipe is attached to a name in the file system. The other end is used to communicate with processes that open the attached filename. Even though it can attach any kind of STREAMS file descriptor to a name in the file system, the fattach function is most commonly used to give a name to a STREAMS pipe.

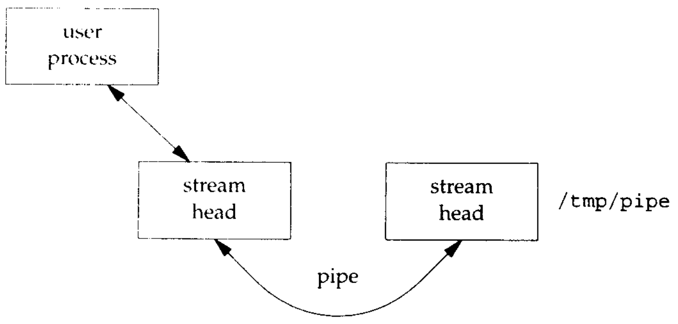

Figure 17.7 A pipe mounted on a name in the file system

A process can call `fdetach` to undo the association between a STREAMS file and the name in the file system.

```
#include <stropts.h>

int fdetach(const char *path);

                                   Returns: 0 if OK, -1 on error
```

After `fdetach` is called, any processes that had accessed the STREAMS pipe by opening the *path* will still continue to access the stream, but subsequent opens of the *path* will access the original file residing in the file system.

17.2.2 Unique Connections

Although we can attach one end of a STREAMS pipe to the file system namespace, we still have problems if multiple processes want to communicate with a server using the named STREAMS pipe. Data from one client will be interleaved with data from the other clients writing to the pipe. Even if we guarantee that the clients write less than `PIPE_BUF` bytes so that the writes are atomic, we have no way to write back to an individual client and guarantee that the intended client will read the message. With multiple clients reading from the same pipe, we cannot control which one will be scheduled and actually read what we send.

The `connld` STREAMS module solves this problem. Before attaching a STREAMS pipe to a name in the file system, a server process can push the `connld` module on the end of the pipe that is to be attached. This results in the configuration shown in Figure 17.8.

In Figure 17.8, the server process has attached one end of its pipe to the path `/tmp/pipe`. We show a dotted line to indicate a client process in the middle of opening the attached STREAMS pipe. Once the open completes, we have the configuration shown in Figure 17.9.

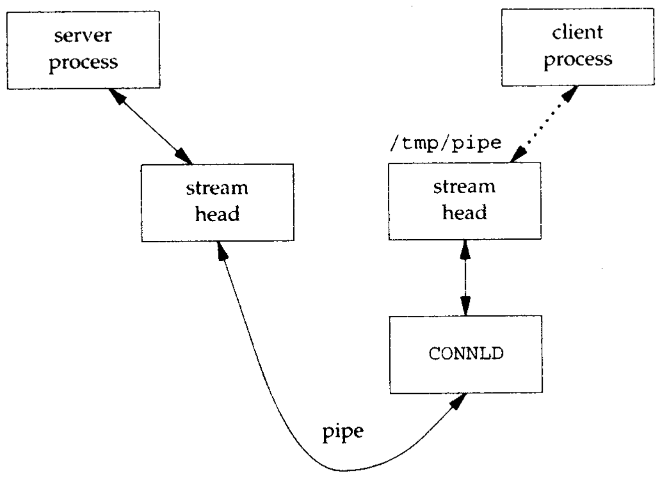

Figure 17.8 Setting up `connld` for unique connections

The client process never receives an open file descriptor for the end of the pipe that it opened. Instead, the operating system creates a new pipe and returns one end to the client process as the result of opening `/tmp/pipe`. The system sends the other end of the new pipe to the server process by passing its file descriptor over the existing (attached) pipe, resulting in a unique connection between the client process and the server process. We'll see the mechanics of passing file descriptors using STREAMS pipes in Section 17.4.1.

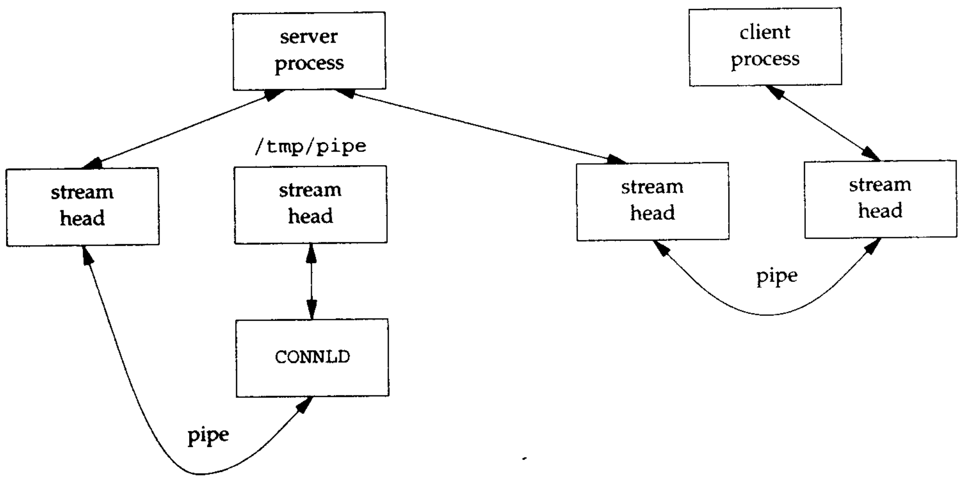

Figure 17.9 Using `connld` to make unique connections

The `fattach` function is built on top of the `mount` system call. This facility is known as *mounted streams*. Mounted streams and the `connld` module were developed by Presotto and Ritchie [1990] for the Research UNIX system. These mechanisms were then picked up by SVR4.

We will now develop three functions that can be used to create unique connections between unrelated processes. These functions mimic the connection-oriented socket functions discussed in Section 16.4. We use STREAMS pipes for the underlying communication mechanism here, but we'll see alternate implementations of these functions that use UNIX domain sockets in Section 17.3.

```
#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 17.10) 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 end of the STREAMS pipe.

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

/* pipe permissions: user rw, group rw, others rw */
#define FIFO_MODE  (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

/*
 * Establish an endpoint to listen for connect requests.
 * Returns fd if all OK, <0 on error
 */
int
serv_listen(const char *name)
{
    int     tempfd;
    int     fd[2];

    /*
     * Create a file: mount point for fattach().
     */
    unlink(name);
    if ((tempfd = creat(name, FIFO_MODE)) < 0)
        return(-1);
    if (close(tempfd) < 0)
        return(-2);
    if (pipe(fd) < 0)
        return(-3);
    /*
     * Push connld & fattach() on fd[1].
     */
    if (ioctl(fd[1], I_PUSH, "connld") < 0) {
        close(fd[0]);
        close(fd[1]);
        return(-4);
    }
    if (fattach(fd[1], name) < 0) {
        close(fd[0]);
        close(fd[1]);
        return(-5);
    }
    close(fd[1]);   /* fattach holds this end open */

    return(fd[0]);  /* fd[0] is where client connections arrive */
}
```

Figure 17.10 The serv_listen function using STREAMS pipes

The serv_accept function (Figure 17.11) is used by a server to wait for a client's connect request to arrive. When one arrives, the system automatically creates a new STREAMS pipe, and the function returns one end to the server. Additionally, the effective user ID of the client is stored in the memory to which *uidptr* points.

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

/*
 * Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID.
 * Returns new fd if all OK, <0 on error.
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    struct strrecvfd    recvfd;

    if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
        return(-1);     /* could be EINTR if signal caught */
    if (uidptr != NULL)
        *uidptr = recvfd.uid;   /* effective uid of caller */
    return(recvfd.fd);  /* return the new descriptor */
}
```

Figure 17.11 The serv_accept function using STREAMS pipes

A client calls cli_conn (Figure 17.12) 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.

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

/*
 * 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;

    /* open the mounted stream */
    if ((fd = open(name, O_RDWR)) < 0)
        return(-1);
    if (isastream(fd) == 0) {
        close(fd);
        return(-2);
    }
    return(fd);
}
```

Figure 17.12 The `cli_conn` function using STREAMS pipes

We double-check that the returned descriptor refers to a STREAMS device, in case the server has not been started but the pathname still exists in the file system. In Section 17.6, we'll see how these three functions are used.

17.3 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 arbitrary domains, operating systems typically provide support only for the UNIX domain.

Example—s_pipe Function Using UNIX Domain Sockets

Figure 17.13 shows the socket-based version of the s_pipe function previously shown in Figure 17.6. The function creates a pair of connected UNIX domain stream sockets.

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

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

Figure 17.13 Socket version of the s_pipe function

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. □

17.3.1 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 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 2.4.22 and Solaris 9, the sockaddr_un structure is defined in the header <sys/un.h> as follows:

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

On FreeBSD 5.2.1 and Mac OS X 10.3, however, the sockaddr_un structure is defined as

```
struct sockaddr_un {
    unsigned char  sun_len;         /* length including null */
    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 17.14 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 17.14 Binding an address to a UNIX domain socket

When we run this program, the bind request succeeds, but if we run the program a second time, 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
srwxrwxr-x  1 sar       0 Aug 22 12:43 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 determine the offset of the `sun_path` member in the `sockaddr_un` structure and add to this the length of the pathname, not including the terminating null byte. Since implementations vary in what 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. □

17.3.2 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 16.14 and 16.15.

Figure 17.15 shows the UNIX domain socket version of 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;

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);
    unlink(name);   /* in case it already exists */

    /* fill in socket address structure */
    memset(&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 = -2;
        goto errout;
    }

    if (listen(fd, QLEN) < 0) { /* tell kernel we're a server */
        rval = -3;
        goto errout;
    }
    return(fd);

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

Figure 17.15 The `serv_listen` function for UNIX domain sockets

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 using 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 17.16).

```
#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, len, err, rval;
    time_t              staletime;
    struct sockaddr_un  un;
    struct stat         statbuf;

    len = sizeof(un);
    if ( (clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0)
        return(-1);     /* 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 */
    un.sun_path[len] = 0;           /* null terminate */

    if (stat(un.sun_path, &statbuf) < 0) {
        rval = -2;
        goto errout;
    }
#ifdef  S_ISSOCK    /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0) {
        rval = -3;      /* not a socket */
        goto errout;
    }
#endif
    if ( (statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
         (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
          rval = -4;    /* is not rwx------ */
          goto errout;
    }

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
        statbuf.st_ctime < staletime ||
        statbuf.st_mtime < staletime) {
          rval = -5;    /* i-node is too old */
          goto errout;
    }

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

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

Figure 17.16 The serv_accept function for UNIX domain sockets

The server blocks in the call to accept, waiting for a client to call cli_conn. When accept returns, its return value is a brand new descriptor that is connected to the client. (This is somewhat similar to what the connld module does with the STREAMS subsystem.) Additionally, the pathname that the client assigned to its socket (the name that contained the client's process ID) is also returned by accept, through the second argument (the pointer to the sockaddr_un structure). We null terminate this pathname and call stat. This lets us verify that the pathname is indeed a socket and that the permissions allow only user-read, user-write, and user-execute. We also verify that the three times associated with the socket are no older than 30 seconds. (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 accept as the I_RECVFD ioctl command does.)

The client initiates the connection to the server by calling the cli_conn function (Figure 17.17).

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

#define CLI_PATH    "/var/tmp/"      /* +5 for pid = 14 chars */
#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;

    /* 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%05d", CLI_PATH, 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;
            goto errout;
        }

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

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

Figure 17.17 The cli_conn function for UNIX domain sockets

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. 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

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing client–server applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, negotiating locks for the file, etc.) 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 3.7, 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 17.18 shows the desired arrangement.

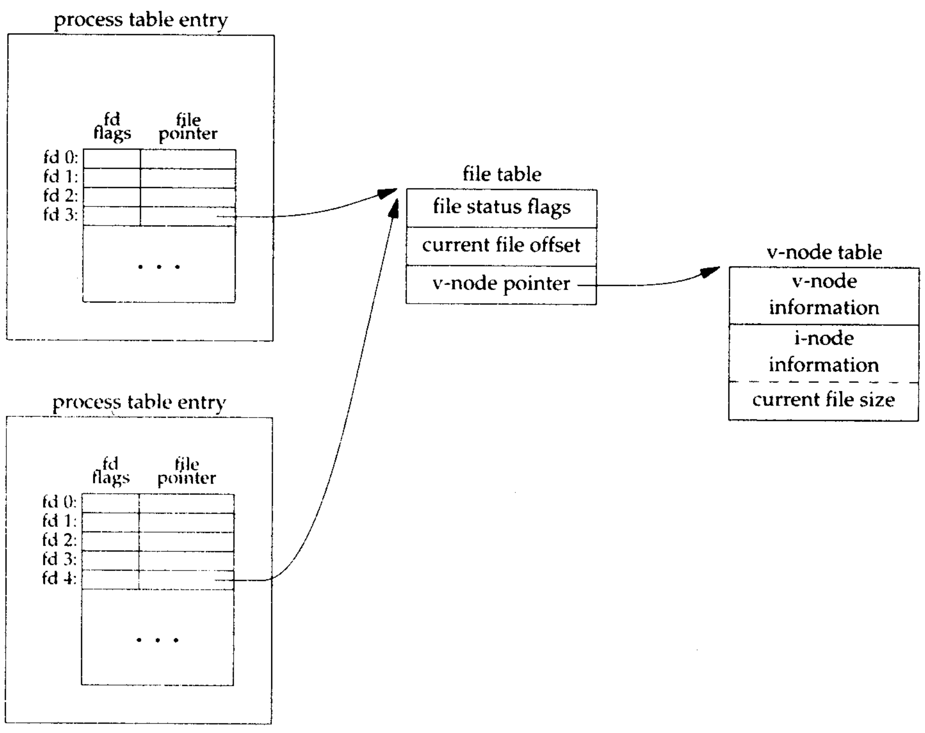

Figure 17.18 Passing an open file from the top process to the bottom process

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 for both STREAMS and sockets.

```
#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
```

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 STREAMS pipe or UNIX domain socket represented by *fd*.

We'll use the term *s-pipe* to refer to a bidirectional communication channel that could be implemented as either a STREAMS pipe or a UNIX domain stream socket.

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 s-pipe until it encounters a null byte. Any characters read up to this point are passed to the caller's *userfunc*. The next byte read by `recv_fd` is the status byte. If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

The function `send_err` calls the `send_fd` function after writing the error message to the s-pipe. This is shown in Figure 17.19.

```
#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 17.19 The send_err function

In the next two sections, we'll look at the implementation of the send_fd and recv_fd functions.

17.4.1 Passing File Descriptors over STREAMS-Based Pipes

With STREAMS pipes, file descriptors are exchanged using two ioctl commands: I_SENDFD and I_RECVFD. To send a descriptor, we set the third argument for ioctl to the actual descriptor. This is shown in Figure 17.20.

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

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

    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (fd_to_send < 0) {
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        buf[1] = 0;     /* zero status means OK */
    }

    if (write(fd, buf, 2) != 2)
        return(-1);
    if (fd_to_send >= 0)
        if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
    return(0);
}
```

Figure 17.20 The send_fd function for STREAMS pipes

When we receive a descriptor, the third argument for ioctl is a pointer to a strrecvfd structure:

```
struct strrecvfd {
    int    fd;        /* new descriptor */
    uid_t  uid;       /* effective user ID of sender */
    gid_t  gid;       /* effective group ID of sender */
    char   fill[8];
};
```

The recv_fd function reads the STREAMS pipe until the first byte of the 2-byte protocol (the null byte) is received. When we issue the I_RECVFD ioctl command, the next message on the stream head's read queue must be a descriptor from an I_SENDFD call, or we get an error. This function is shown in Figure 17.21.

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

/*
 * Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes).  We have a
 * 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int                 newfd, nread, flag, status;
    char                *ptr;
    char                buf[MAXLINE];
    struct strbuf       dat;
    struct strrecvfd    recvfd;

    status = -1;
    for ( ; ; ) {
        dat.buf = buf;
        dat.maxlen = MAXLINE;
        flag = 0;
        if (getmsg(fd, NULL, &dat, &flag) < 0)
            err_sys("getmsg error");
        nread = dat.len;
        if (nread == 0) {
            err_ret("connection closed by server");
```

```
                return(-1);
            }
            /*
             * See if this is the final data with null & status.
             * Null must be next to last byte of buffer, status
             * byte is last byte.  Zero status means there must
             * be a file descriptor to receive.
             */
            for (ptr = buf; ptr < &buf[nread]; ) {
                if (*ptr++ == 0) {
                    if (ptr != &buf[nread-1])
                        err_dump("message format error");
                    status = *ptr & 0xFF;   /* prevent sign extension */
                    if (status == 0) {
                        if (ioctl(fd, I_RECVFD, &recvfd) < 0)
                            return(-1);
                        newfd = recvfd.fd;  /* new descriptor */
                    } else {
                        newfd = -status;
                    }
                    nread -= 2;
                }
            }
            if (nread > 0)
                if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                    return(-1);

            if (status >= 0)    /* final data has arrived */
                return(newfd);  /* descriptor, or -status */
        }
    }
```

Figure 17.21 The recv_fd function for STREAMS pipes

17.4.2 Passing File Descriptors over UNIX Domain Sockets

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 14.7). The msg_flags field contains flags describing the message received, as summarized in Figure 16.13.

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 17.22 is the send_fd function for UNIX domain sockets.

```
#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 17.22 The send_fd function for UNIX domain sockets

In the sendmsg call, we send both the protocol data (the null and the status byte) and the descriptor.

To receive a descriptor (Figure 17.23), 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. This is shown in Figure 17.23.

```
#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_sys("recvmsg error");
        } 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 17.23 The recv_fd function for UNIX domain sockets

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 receive a descriptor.

When it comes to passing file descriptors, one difference between UNIX domain sockets and STREAMS pipes is that we get the identity of the sending process with STREAMS pipes. Some versions of UNIX domain sockets provide similar functionality, but their interfaces differ.

> FreeBSD 5.2.1 and Linux 2.4.22 provide support for sending credentials over UNIX domain sockets, but they do it differently. Mac OS X 10.3 is derived in part from FreeBSD, but has credential passing disabled. Solaris 9 doesn't support sending credentials over UNIX domain sockets.

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 it in for us to prevent an application from pretending to have a different identity.

On Linux, credentials are transmitted as a ucred structure:

```
struct ucred {
    uint32_t  pid;   /* sender's process ID */
    uint32_t  uid;   /* sender's user ID */
    uint32_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 17.24 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 17.24 Sending credentials over UNIX domain sockets

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

The function in Figure 17.25 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 CREDOPT         LOCAL_PEERCRED
```

```
#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;
    int                 newfd, nr, status;
    char                *ptr;
    char                buf[MAXLINE];
    struct iovec        iov[1];
    struct msghdr       msg;
    const int           on = 1;

    status = -1;
    newfd = -1;
    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt failed");
        return(-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_sys("recvmsg error");
            } 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 17.25 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. But instead of sending the contents of the file back to the calling process, the server sends back an open file descriptor. This lets the server work with any type of file (such as a device or a socket) and not simply regular files. It also means that a minimum of information is exchanged using IPC: the filename and open mode from the client to the server, and the returned descriptor from the server to the client. The contents of the file are not exchanged using IPC.

There are several advantages in designing the server to be a separate executable program (either one that is 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 s-pipe (either a STREAMS-based pipe or a UNIX domain socket pair) and then calls `fork` and `exec` to invoke the server. The client sends requests across the s-pipe, and the server sends back responses across the s-pipe.

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 s-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 17.26), 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 17.26 The `open.h` header

The main function (Figure 17.27) 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], 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 17.27 The client main function, version 1

The function csopen (Figure 17.28) does the fork and exec of the server, after creating the s-pipe.

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

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      fd[2] = { -1, -1 };

    if (fd[0] < 0) {     /* fork/exec our open server first time */
        if (s_pipe(fd) < 0)
            err_sys("s_pipe error");
        if ( (pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {       /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO &&
              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_sys("writev error");

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

Figure 17.28 The csopen function, version 1

The child closes one end of the pipe, and the parent closes the other. For the server that it executes, the child also duplicates its end of the 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 17.28. First, we have the `opend.h` header (Figure 17.29), 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    request(char *, int, int);
```

Figure 17.29 The opend.h header, version 1

The main function (Figure 17.30) reads the requests from the client on the s-pipe (its standard input) and calls the function 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 */
        request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

Figure 17.30 The server main function, version 1

The function request in Figure 17.31 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 s-pipe (its standard output). If an error is encountered, send_err is called to send back an error message, using the client–server protocol that we described earlier.

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

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

    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request not null terminated: %*.*s\n",
          nread, nread, buf);
        send_err(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) {
        sprintf(errmsg, "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 17.31 The request function, version 1

The client's request is a null-terminated string of white-space-separated arguments. The function buf_args in Figure 17.32 breaks this string into a standard argv-style argument list and calls a user function to process the arguments. We'll use the buf_args function later in this chapter. 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 17.32 The buf_args function

The server's function that is called by buf_args is cli_args (Figure 17.33). 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
 * 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 17.33 The cli_args function

This completes the open server that is invoked by a fork and exec from the client. A single s-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 exec are avoided. We still use an s-pipe 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 17.2.2. This server also demonstrates how a single server can handle multiple clients, using both the select and poll functions from Section 14.5.

The client is similar to the client from Section 17.5. Indeed, the file main.c is identical (Figure 17.27). We add the following line to the open.h header (Figure 17.26):

```
#define CS_OPEN "/home/sar/opend"   /* server's well-known name */
```

The file open.c does change from Figure 17.28, since we now call cli_conn instead of doing the fork and exec. This is shown in Figure 17.34.

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

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    int             len;
    char            buf[10];
    struct iovec    iov[3];
    static int      csfd = -1;

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

    sprintf(buf, " %d", oflag);      /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";  /* 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_sys("writev error");

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

Figure 17.34 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 17.35) includes the standard headers and declares the global variables and the function prototypes.

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

#define CS_OPEN "/home/sar/opend"   /* 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     request(char *, int, int, uid_t);
```

Figure 17.35 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 17.36 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 17.36 Functions to manipulate `client` array

The first time `client_add` is called, it calls `client_alloc`, which 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.

The main function (Figure 17.37) 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 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 17.37 The server main function, version 2

The function loop is the server's infinite loop. We'll show two versions of this function. Figure 17.38 shows one version that uses select; Figure 17.39 shows another version that uses poll.

```
#include    "opend.h"
#include    <sys/time.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 */
                    request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.38 The loop function using select

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

1. The descriptor `listenfd` can be ready for reading, which means that a new client has called `cli_conn`. To handle this, we call `serv_accept` and then update the `client` array and associated bookkeeping information for the new client. (We keep track of the highest descriptor number for the first argument to `select`. We also keep track of the highest index in 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 `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 17.39.

```
#include    "opend.h"
#include    <poll.h>
#if !defined(BSD) && !defined(MACOS)
#include    <stropts.h>
#endif

void
loop(void)
{
    int             i, maxi, listenfd, clifd, nread;
    char            buf[MAXLINE];
    uid_t           uid;
    struct pollfd   *pollfd;

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

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

    for ( ; ; ) {
        if (poll(pollfd, maxi + 1, -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);
            i = client_add(clifd, uid);
            pollfd[i].fd = clifd;
            pollfd[i].events = POLLIN;
            if (i > maxi)
                maxi = i;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i <= maxi; i++) {
            if ((clifd = client[i].fd) < 0)
                continue;
            if (pollfd[i].revents & POLLHUP) {
                goto hungup;
            } else if (pollfd[i].revents & POLLIN) {
                /* read argument buffer from client */
                if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", clifd);
                } else if (nread == 0) {
hungup:
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, clifd);
                    client_del(clifd);  /* client has closed conn */
                    pollfd[i].fd = -1;
                    close(clifd);
                } else {        /* process client's request */
                    request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.39 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. (Recall the open_max function from Figure 2.16.)

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

For an existing client, we have to handle two different events from poll: a client termination is indicated by POLLHUP, and a new request from an existing client is indicated by POLLIN. Recall from Exercise 15.7 that the hang-up message can arrive at the stream head while there is still data to be read from the stream. With a pipe, we want to read all the data before processing the hangup. But with this server, when we receive the hangup from the client, we can close the connection (the stream) to the client, effectively throwing away any data still on the stream. There is no reason to process any requests still on the stream, since we can't send any responses back.

As with the `select` version of this function, new requests from a client are handled by calling the `request` function (Figure 17.40). This function is similar to the earlier version (Figure 17.31). It calls the same function, `buf_args` (Figure 17.32), that calls `cli_args` (Figure 17.33), but since it runs from a daemon process, it logs error messages instead of printing them on the standard error stream.

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

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

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

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

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

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

Figure 17.40 The `request` function, version 2

This completes the second version of the open server, using 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. We've seen how to do this using both STREAMS pipes and UNIX domain sockets. 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 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. The final version also used the client–server connection functions introduced in Section 17.2.2 and the I/O multiplexing functions from Section 14.5.

Exercises

17.1 Recode Figure 17.4 to use the standard I/O library instead of `read` and `write` on the STREAMS pipe.

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 opens an existing file and passes the open descriptor to the parent. The child then positions the file using `lseek` and notifies the parent. The parent reads the file's current offset and prints it for verification. If the file was passed from the child to the parent as we described, 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 17.29 and 17.30, we differentiated between declaring and defining the global variables. What is the difference?

17.4 Recode the `buf_args` function (Figure 17.32), 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 17.38 and Figure 17.39. Implement your optimizations.

18

Terminal I/O

18.1 Introduction

The handling of terminal I/O is a messy area, regardless of the operating system. The UNIX System is no exception. The manual page for terminal I/O is usually one of the longest in most editions of the programmer's manuals.

With the UNIX System, a schism formed in the late 1970s when System III developed a different set of terminal routines from those of Version 7. The System III style of terminal I/O continued through System V, and the Version 7 style became the standard for the BSD-derived systems. As with signals, this difference between the two worlds has been conquered by POSIX.1. In this chapter, we look at all the POSIX.1 terminal functions and some of the platform-specific additions.

Part of the complexity of the terminal I/O system occurs because people use terminal I/O for so many different things: terminals, hardwired lines between computers, modems, printers, and so on.

18.2 Overview

Terminal I/O has two modes:

1. Canonical mode input processing. In this mode, terminal input is processed as lines. The terminal driver returns at most one line per read request.
2. Noncanonical mode input processing. The input characters are not assembled into lines.

If we don't do anything special, canonical mode is the default. For example, if the shell redirects standard input to the terminal and we use `read` and `write` to copy standard input to standard output, the terminal is in canonical mode, and each `read` returns at most one line. Programs that manipulate the entire screen, such as the `vi` editor, use noncanonical mode, since the commands may be single characters and are not terminated by newlines. Also, this editor doesn't want processing by the system of the special characters, since they may overlap with the editor commands. For example, the Control-D character is often the end-of-file character for the terminal, but it's also a `vi` command to scroll down one-half screen.

The Version 7 and older BSD-style terminal drivers supported three modes for terminal input: (a) cooked mode (the input is collected into lines, and the special characters are processed), (b) raw mode (the input is not assembled into lines, and there is no processing of special characters), and (c) cbreak mode (the input is not assembled into lines, but some of the special characters are processed). Figure 18.20 shows a POSIX.1 function that places a terminal in cbreak or raw mode.

POSIX.1 defines 11 special input characters, 9 of which we can change. We've been using some of these throughout the text: the end-of-file character (usually Control-D) and the suspend character (usually Control-Z), for example. Section 18.3 describes each of these characters.

We can think of a terminal device as being controlled by a terminal driver, usually within the kernel. Each terminal device has an input queue and an output queue, shown in Figure 18.1.

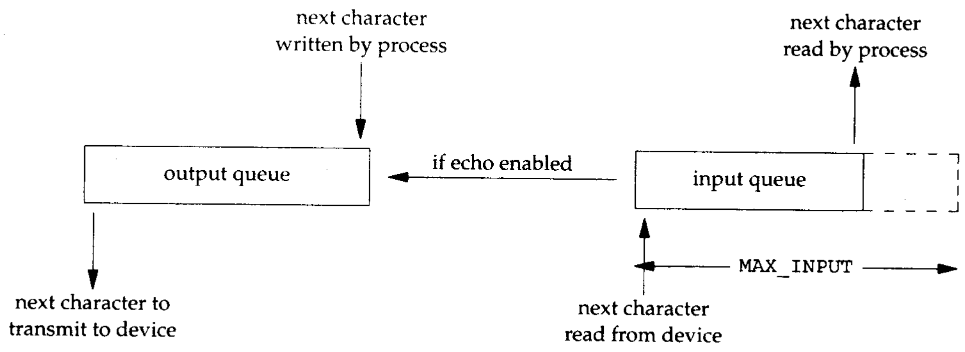
next character written by process
next character read by process
output queue
if echo enabled
input queue
MAX_INPUT
next character to transmit to device
next character read from device

Figure 18.1 Logical picture of input and output queues for a terminal device

There are several points to consider from this picture.

- If echoing is enabled, there is an implied link between the input queue and the output queue.
- The size of the input queue, `MAX_INPUT` (see Figure 2.11), is finite. When the input queue for a particular device fills, the system behavior is implementation dependent. Most UNIX systems echo the bell character when this happens.

- There is another input limit, `MAX_CANON`, that we don't show here. This limit is the maximum number of bytes in a canonical input line.
- Although the size of the output queue is finite, no constants defining that size are accessible to the program, because when the output queue starts to fill up, the kernel simply puts the writing process to sleep until room is available.
- We'll see how the `tcflush` flush function allows us to flush either the input queue or the output queue. Similarly, when we describe the `tcsetattr` function, we'll see how we can tell the system to change the attributes of a terminal device only after the output queue is empty. (We want to do this, for example, if we're changing the output attributes.) We can also tell the system to discard everything in the input queue when changing the terminal attributes. (We want to do this if we're changing the input attributes or changing between canonical and noncanonical modes, so that previously entered characters aren't interpreted in the wrong mode.)

Most UNIX systems implement all the canonical processing in a module called the *terminal line discipline*. We can think of this module as a box that sits between the kernel's generic read and write functions and the actual device driver (see Figure 18.2).

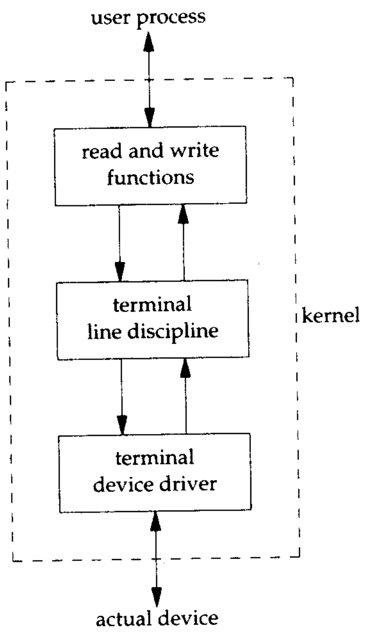

Figure 18.2 Terminal line discipline

Note the similarity of this picture and the diagram of a stream shown in Figure 14.14. We'll return to this picture in Chapter 19, when we discuss pseudo terminals.

All the terminal device characteristics that we can examine and change are contained in a termios structure. This structure is defined in the header <termios.h>, which we use throughout this chapter:

```
struct termios {
  tcflag_t  c_iflag;    /* input flags */
  tcflag_t  c_oflag;    /* output flags */
  tcflag_t  c_cflag;    /* control flags */
  tcflag_t  c_lflag;    /* local flags */
  cc_t      c_cc[NCCS]; /* control characters */
};
```

Roughly speaking, the input flags control the input of characters by the terminal device driver (strip eighth bit on input, enable input parity checking, etc.), the output flags control the driver output (perform output processing, map newline to CR/LF, etc.), the control flags affect the RS-232 serial lines (ignore modem status lines, one or two stop bits per character, etc.), and the local flags affect the interface between the driver and the user (echo on or off, visually erase characters, enable terminal-generated signals, job control stop signal for background output, etc.).

The type tcflag_t is big enough to hold each of the flag values and is often defined as an unsigned int or an unsigned long. The c_cc array contains all the special characters that we can change. NCCS is the number of elements in this array and is typically between 15 and 20 (since most implementations of the UNIX System support more than the 11 POSIX-defined special characters). The cc_t type is large enough to hold each special character and is typically an unsigned char.

Versions of System V that predated the POSIX standard had a header named <termio.h> and a structure named termio. POSIX.1 added an s to the names, to differentiate them from their predecessors.

Figures 18.3 through 18.6 list all the terminal flags that we can change to affect the characteristics of a terminal device. Note that even though the Single UNIX Specification defines a common subset that all platforms start from, all the implementations have their own additions. Most of these additions come from the historical differences between the systems. We'll discuss each of these flag values in detail in Section 18.5.

Given all the options available, how do we examine and change these characteristics of a terminal device? Figure 18.7 summarizes the various functions defined by the Single UNIX Specification that operate on terminal devices. (All the functions listed are part of the base POSIX specification, except for tcgetsid, which is an XSI extension. We described tcgetpgrp, tcgetsid, and tcsetpgrp in Section 9.7.)

Note that the Single UNIX Specification doesn't use the classic ioctl on terminal devices. Instead, it uses the 13 functions shown in Figure 18.7. The reason is that the ioctl function for terminal devices uses a different data type for its final argument, which depends on the action being performed. This makes type checking of the arguments impossible.

Although only 13 functions operate on terminal devices, the first two functions in Figure 18.7 (`tcgetattr` and `tcsetattr`) manipulate almost 70 different flags (see Figures 18.3 through 18.6). The handling of terminal devices is complicated by the large number of options available for terminal devices and trying to determine which options are required for a particular device (be it a terminal, modem, printer, or whatever).

Flag	Description	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`CBAUDEXT`	extended baud rate					•
`CCAR_OFLOW`	DCD flow control of output		•		•	
`CCTS_OFLOW`	CTS flow control of output		•		•	•
`CDSR_OFLOW`	DSR flow control of output		•		•	
`CDTR_IFLOW`	DTR flow control of input		•		•	
`CIBAUDEXT`	extended input baud rate					•
`CIGNORE`	ignore control flags		•		•	
`CLOCAL`	ignore modem status lines	•	•	•	•	•
`CREAD`	enable receiver	•	•	•	•	•
`CRTSCTS`	enable hardware flow control		•	•	•	•
`CRTS_IFLOW`	RTS flow control of input		•		•	•
`CRTSXOFF`	enable input hardware flow control					•
`CSIZE`	character size mask	•	•	•	•	•
`CSTOPB`	send two stop bits, else one	•	•	•	•	•
`HUPCL`	hang up on last close	•	•	•	•	•
`MDMBUF`	same as `CCAR_OFLOW`		•		•	
`PARENB`	parity enable	•	•	•	•	•
`PAREXT`	mark or space parity					•
`PARODD`	odd parity, else even	•	•	•	•	•

Figure 18.3 `c_cflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`BRKINT`	generate `SIGINT` on BREAK	•	•	•	•	•
`ICRNL`	map CR to NL on input	•	•	•	•	•
`IGNBRK`	ignore BREAK condition	•	•	•	•	•
`IGNCR`	ignore CR	•	•	•	•	•
`IGNPAR`	ignore characters with parity errors	•	•	•	•	•
`IMAXBEL`	ring bell on input queue full		•	•	•	•
`INLCR`	map NL to CR on input	•	•	•	•	•
`INPCK`	enable input parity checking	•	•	•	•	•
`ISTRIP`	strip eighth bit off input characters	•	•	•	•	•
`IUCLC`	map uppercase to lowercase on input			•		•
`IXANY`	enable any characters to restart output	XSI	•	•	•	•
`IXOFF`	enable start/stop input flow control	•	•	•	•	•
`IXON`	enable start/stop output flow control	•	•	•	•	•
`PARMRK`	mark parity errors	•	•	•	•	•

Figure 18.4 `c_iflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
ALTWERASE	use alternate WERASE algorithm		•		•	
ECHO	enable echo	•	•	•	•	•
ECHOCTL	echo control chars as ^(Char)		•	•	•	•
ECHOE	visually erase chars	•	•	•	•	•
ECHOK	echo kill	•	•	•	•	•
ECHOKE	visual erase for kill		•	•	•	•
ECHONL	echo NL	•	•	•	•	•
ECHOPRT	visual erase mode for hard copy		•	•	•	•
EXTPROC	external character processing		•		•	
FLUSHO	output being flushed		•	•	•	•
ICANON	canonical input	•	•	•	•	•
IEXTEN	enable extended input char processing	•	•	•	•	•
ISIG	enable terminal-generated signals	•	•	•	•	•
NOFLSH	disable flush after interrupt or quit	•	•	•	•	•
NOKERNINFO	no kernel output from STATUS		•		•	
PENDIN	retype pending input		•	•	•	•
TOSTOP	send SIGTTOU for background output	•	•	•	•	•
XCASE	canonical upper/lower presentation			•		•

Figure 18.5 c_lflag terminal flags

Flag	Description	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
BSDLY	backspace delay mask	XSI		•		•
CMSPAR	mark or space parity			•		
CRDLY	CR delay mask	XSI		•		•
FFDLY	form feed delay mask	XSI		•		•
NLDLY	NL delay mask	XSI		•		•
OCRNL	map CR to NL on output	XSI	•	•		•
OFDEL	fill is DEL, else NUL	XSI		•		•
OFILL	use fill character for delay	XSI		•		•
OLCUC	map lowercase to uppercase on output			•		•
ONLCR	map NL to CR-NL	XSI	•	•	•	•
ONLRET	NL performs CR function	XSI	•	•		•
ONOCR	no CR output at column 0	XSI	•	•		•
ONOEOT	discard EOTs (^D) on output		•		•	
OPOST	perform output processing	•	•	•	•	•
OXTABS	expand tabs to spaces		•		•	
TABDLY	horizontal tab delay mask	XSI		•		•
VTDLY	vertical tab delay mask	XSI		•		•

Figure 18.6 c_oflag terminal flags

Function	Description
`tcgetattr`	fetch attributes (`termios` structure)
`tcsetattr`	set attributes (`termios` structure)
`cfgetispeed`	get input speed
`cfgetospeed`	get output speed
`cfsetispeed`	set input speed
`cfsetospeed`	set output speed
`tcdrain`	wait for all output to be transmitted
`tcflow`	suspend transmit or receive
`tcflush`	flush pending input and/or output
`tcsendbreak`	send BREAK character
`tcgetpgrp`	get foreground process group ID
`tcsetpgrp`	set foreground process group ID
`tcgetsid`	get process group ID of session leader for controlling TTY (XSI extension)

Figure 18.7 Summary of terminal I/O functions

The relationships among the 13 functions shown in Figure 18.7 are shown in Figure 18.8.

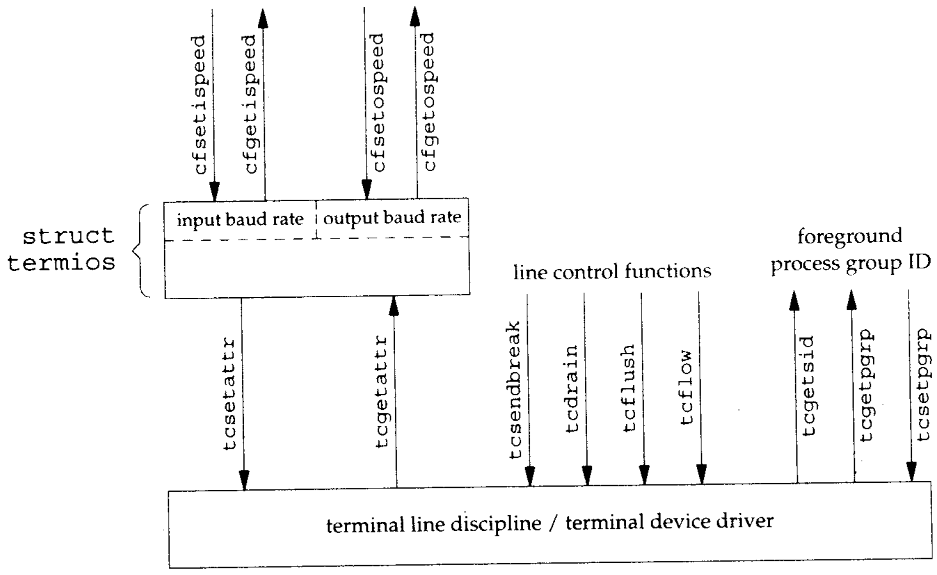

Figure 18.8 Relationships among the terminal-related functions

POSIX.1 doesn't specify where in the `termios` structure the baud rate information is stored; that is an implementation detail. Some systems, such as Linux and Solaris, store this information in the `c_cflag` field. BSD-derived systems, such as FreeBSD and Mac OS X, have two separate fields in the structure: one for the input speed and one for the output speed.

18.3 Special Input Characters

POSIX.1 defines 11 characters that are handled specially on input. Implementations define additional special characters. Figure 18.9 summarizes these special characters.

Character	Description	c_cc subscript	Enabled by field	Enabled by flag	Typical value	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
CR	carriage return	(can't change)	`c_lflag`	`ICANON`	`\r`	•	•	•	•	•
DISCARD	discard output	`VDISCARD`	`c_lflag`	`IEXTEN`	^O		•	•	•	•
DSUSP	delayed suspend (`SIGTSTP`)	`VDSUSP`	`c_lflag`	`ISIG`	^Y		•		•	•
EOF	end of file	`VEOF`	`c_lflag`	`ICANON`	^D	•	•	•	•	•
EOL	end of line	`VEOL`	`c_lflag`	`ICANON`		•	•	•	•	•
EOL2	alternate end of line	`VEOL2`	`c_lflag`	`ICANON`			•	•	•	•
ERASE	backspace one character	`VERASE`	`c_lflag`	`ICANON`	^H, ^?	•	•	•	•	•
ERASE2	alternate backspace character	`VERASE2`	`c_lflag`	`ICANON`	^H, ^?		•			
INTR	interrupt signal (`SIGINT`)	`VINTR`	`c_lflag`	`ISIG`	^?, ^C	•	•	•	•	•
KILL	erase line	`VKILL`	`c_lflag`	`ICANON`	^U	•	•	•	•	•
LNEXT	literal next	`VLNEXT`	`c_lflag`	`IEXTEN`	^V		•	•	•	•
NL	line feed (newline)	(can't change)	`c_lflag`	`ICANON`	`\n`	•	•	•	•	•
QUIT	quit signal (`SIGQUIT`)	`VQUIT`	`c_lflag`	`ISIG`	^\	•	•	•	•	•
REPRINT	reprint all input	`VREPRINT`	`c_lflag`	`ICANON`	^R		•	•	•	•
START	resume output	`VSTART`	`c_iflag`	`IXON/IXOFF`	^Q	•	•	•	•	•
STATUS	status request	`VSTATUS`	`c_lflag`	`ICANON`	^T		•		•	
STOP	stop output	`VSTOP`	`c_iflag`	`IXON/IXOFF`	^S	•	•	•	•	•
SUSP	suspend signal (`SIGTSTP`)	`VSUSP`	`c_lflag`	`ISIG`	^Z	•	•	•	•	•
WERASE	backspace one word	`VWERASE`	`c_lflag`	`ICANON`	^W		•	•	•	•

Figure 18.9 Summary of special terminal input characters

Of the 11 POSIX.1 special characters, we can change 9 of them to almost any value that we like. The exceptions are the newline and carriage return characters (`\n` and `\r`, respectively) and perhaps the STOP and START characters (depends on the implementation). To do this, we modify the appropriate entry in the `c_cc` array of the `termios` structure. The elements in this array are referred to by name, with each name beginning with a `V` (the third column in Figure 18.9).

POSIX.1 allows us to disable these characters. If we set the value of an entry in the `c_cc` array to the value of `_POSIX_VDISABLE`, then we disable the corresponding special character.

In older versions of the Single UNIX Specification, support for `_POSIX_VDISABLE` was optional. It is now required.

All four platforms discussed in this text support this feature. Linux 2.4.22 and Solaris 9 define `_POSIX_VDISABLE` as 0; FreeBSD 5.2.1 and Mac OS X 10.3 define it as `0xff`.

Some earlier UNIX systems disabled a feature if the corresponding special input character was 0.

Example

Before describing all the special characters in detail, let's look at a small program that changes them. The program in Figure 18.10 disables the interrupt character and sets the end-of-file character to Control-B.

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

int
main(void)
{
    struct termios  term;
    long            vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("standard input is not a terminal device");

    if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
        err_sys("tcgetattr error");

    term.c_cc[VINTR] = vdisable;     /* disable INTR character */
    term.c_cc[VEOF]  = 2;            /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Figure 18.10 Disable interrupt character and change end-of-file character

Note the following in this program.

- We modify the terminal characters only if standard input is a terminal device. We call `isatty` (Section 18.9) to check this.
- We fetch the `_POSIX_VDISABLE` value using `fpathconf`.
- The function `tcgetattr` (Section 18.4) fetches a `termios` structure from the kernel. After we've modified this structure, we call `tcsetattr` to set the attributes. The only attributes that change are the ones we specifically modified.
- Disabling the interrupt key is different from ignoring the interrupt signal. The program in Figure 18.10 simply disables the special character that causes the terminal driver to generate `SIGINT`. We can still use the `kill` function to send the signal to the process. □

We now describe each of the special characters in more detail. We call these the special input characters, but two of the characters, STOP and START (Control-S and Control-Q), are also handled specially when output. Note that when recognized by the terminal driver and processed specially, most of these special characters are then discarded: they are not returned to the process in a read operation. The exceptions to this are the newline characters (NL, EOL, EOL2) and the carriage return (CR).

CR	The carriage return character. We cannot change this character. This character is recognized on input in canonical mode. When both `ICANON` (canonical mode) and `ICRNL` (map CR to NL) are set and `IGNCR` (ignore CR) is not set, the CR character is translated to NL and has the same effect as a NL character. This character is returned to the reading process (perhaps after being translated to a NL).
DISCARD	The discard character. This character, recognized on input in extended mode (`IEXTEN`), causes subsequent output to be discarded until another DISCARD character is entered or the discard condition is cleared (see the `FLUSHO` option). This character is discarded when processed (i.e., it is not passed to the process).
DSUSP	The delayed-suspend job-control character. This character is recognized on input in extended mode (`IEXTEN`) if job control is supported and if the `ISIG` flag is set. Like the SUSP character, this delayed-suspend character generates the `SIGTSTP` signal that is sent to all processes in the foreground process group (refer to Figure 9.7). But the delayed-suspend character generates a signal only when a process reads from the controlling terminal, not when the character is typed. This character is discarded when processed (i.e., it is not passed to the process).
EOF	The end-of-file character. This character is recognized on input in canonical mode (`ICANON`). When we type this character, all bytes waiting to be read are immediately passed to the reading process. If no bytes are waiting to be read, a count of 0 is returned. Entering an EOF character at the beginning of the line is the normal way to indicate an end of file to a program. This character is discarded when processed in canonical mode (i.e., it is not passed to the process).
EOL	The additional line delimiter character, like NL. This character is recognized on input in canonical mode (`ICANON`) and is returned to the reading process; however, this character is not normally used.
EOL2	Another line delimiter character, like NL. This character is treated identically to the EOL character.
ERASE	The erase character (backspace). This character is recognized on input in canonical mode (`ICANON`) and erases the previous character in the line, not erasing beyond the beginning of the line. This character is discarded when processed in canonical mode (i.e., it is not passed to the process).

ERASE2	The alternate erase character (backspace). This character is treated exactly like the erase character (ERASE).
INTR	The interrupt character. This character is recognized on input if the `ISIG` flag is set and generates the `SIGINT` signal that is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).
KILL	The kill character. (The name "kill" is overused; recall the `kill` function used to send a signal to a process. This character should be called the line-erase character; it has nothing to do with signals.) It is recognized on input in canonical mode (`ICANON`). It erases the entire line and is discarded when processed (i.e., it is not passed to the process).
LNEXT	The literal-next character. This character is recognized on input in extended mode (`IEXTEN`) and causes any special meaning of the next character to be ignored. This works for all special characters listed in this section. We can use this character to type any character to a program. The LNEXT character is discarded when processed, but the next character entered is passed to the process.
NL	The newline character, which is also called the line delimiter. We cannot change this character. This character is recognized on input in canonical mode (`ICANON`). This character is returned to the reading process.
QUIT	The quit character. This character is recognized on input if the `ISIG` flag is set. The quit character generates the `SIGQUIT` signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process). Recall from Figure 10.1 that the difference between INTR and QUIT is that the QUIT character not only terminates the process by default, but also generates a `core` file.
REPRINT	The reprint character. This character is recognized on input in extended, canonical mode (both `IEXTEN` and `ICANON` flags set) and causes all unread input to be output (reechoed). This character is discarded when processed (i.e., it is not passed to the process).
START	The start character. This character is recognized on input if the `IXON` flag is set and is automatically generated as output if the `IXOFF` flag is set. A received START character with `IXON` set causes stopped output (from a previously entered STOP character) to restart. In this case, the START character is discarded when processed (i.e., it is not passed to the process). When `IXOFF` is set, the terminal driver automatically generates a START character to resume input that it had previously stopped, when the new input will not overflow the input buffer.

STATUS	The BSD status-request character. This character is recognized on input in extended, canonical mode (both IEXTEN and ICANON flags set) and generates the SIGINFO signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). Additionally, if the NOKERNINFO flag is not set, status information on the foreground process group is also displayed on the terminal. This character is discarded when processed (i.e., it is not passed to the process).
STOP	The stop character. This character is recognized on input if the IXON flag is set and is automatically generated as output if the IXOFF flag is set. A received STOP character with IXON set stops the output. In this case, the STOP character is discarded when processed (i.e., it is not passed to the process). The stopped output is restarted when a START character is entered. When IXOFF is set, the terminal driver automatically generates a STOP character to prevent the input buffer from overflowing.
SUSP	The suspend job-control character. This character is recognized on input if job control is supported and if the ISIG flag is set. The suspend character generates the SIGTSTP signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).
WERASE	The word-erase character. This character is recognized on input in extended, canonical mode (both IEXTEN and ICANON flags set) and causes the previous word to be erased. First, it skips backward over any white space (spaces or tabs), then backward over the previous token, leaving the cursor positioned where the first character of the previous token was located. Normally, the previous token ends when a white space character is encountered. We can change this, however, by setting the ALTWERASE flag. This flag causes the previous token to end when the first nonalphanumeric character is encountered. The word-erase character is discarded when processed (i.e., it is not passed to the process).

Another "character" that we need to define for terminal devices is the BREAK character. BREAK is not really a character, but rather a condition that occurs during asynchronous serial data transmission. A BREAK condition is signaled to the device driver in various ways, depending on the serial interface.

Most old serial terminals have a key labeled BREAK that generates the BREAK condition, which is why most people think of BREAK as a character. Some newer terminal keyboards don't have a BREAK key. On PCs, the break key might be mapped for other purpose. For example, the Windows command interpreter can be interrupted by typing Control-BREAK.

For asynchronous serial data transmission, a BREAK is a sequence of zero-valued bits that continues for longer than the time required to send one byte. The entire sequence of zero-valued bits is considered a single BREAK. In Section 18.8, we'll see how to send a BREAK with the tcsendbreak function.

18.4 Getting and Setting Terminal Attributes

To get and set a `termios` structure, we call two functions: `tcgetattr` and `tcsetattr`. This is how we examine and modify the various option flags and special characters to make the terminal operate the way we want it to.

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);

int tcsetattr(int filedes, int opt, const struct termios *termptr);

                                    Both return: 0 if OK, -1 on error
```

Both functions take a pointer to a `termios` structure and either return the current terminal attributes or set the terminal's attributes. Since these two functions operate only on terminal devices, `errno` is set to `ENOTTY` and −1 is returned if *filedes* does not refer to a terminal device.

The argument *opt* for `tcsetattr` lets us specify when we want the new terminal attributes to take effect. This argument is specified as one of the following constants.

`TCSANOW`	The change occurs immediately.
`TCSADRAIN`	The change occurs after all output has been transmitted. This option should be used if we are changing the output parameters.
`TCSAFLUSH`	The change occurs after all output has been transmitted. Furthermore, when the change takes place, all input data that has not been read is discarded (flushed).

The return status of `tcsetattr` confuses the programming. This function returns OK if it was able to perform *any* of the requested actions, even if it couldn't perform all the requested actions. If the function returns OK, it is our responsibility to see whether all the requested actions were performed. This means that after we call `tcsetattr` to set the desired attributes, we need to call `tcgetattr` and compare the actual terminal's attributes to the desired attributes to detect any differences.

18.5 Terminal Option Flags

In this section, we list all the various terminal option flags, expanding the descriptions of all the options from Figures 18.3 through 18.6. This list is alphabetical and indicates in which of the four terminal flag fields the option appears. (The field a given option is controlled by is usually not apparent from the option name alone.) We also note whether each option is defined by the Single UNIX Specification and list the platforms that support it.

All the flags listed specify one or more bits that we turn on or clear, unless we call the flag a *mask*. A mask defines multiple bits grouped together from which a set of values is defined. We have a defined name for the mask and a name for each value. For example, to set the character size, we first zero the bits using the character-size mask CSIZE, and then set one of the values CS5, CS6, CS7, or CS8.

The six delay values supported by Linux and Solaris are also masks: BSDLY, CRDLY, FFDLY, NLDLY, TABDLY, and VTDLY. Refer to the termio(7I) manual page on Solaris for the length of each delay value. In all cases, a delay mask of 0 means no delay. If a delay is specified, the OFILL and OFDEL flags determine whether the driver does an actual delay or whether fill characters are transmitted instead.

Example

Figure 18.11 demonstrates the use of these masks to extract a value and to set a value.

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

int
main(void)
{
    struct termios  term;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 bits/byte\n");
        break;
    case CS6:
        printf("6 bits/byte\n");
        break;
    case CS7:
        printf("7 bits/byte\n");
        break;
    case CS8:
        printf("8 bits/byte\n");
        break;
    default:
        printf("unknown bits/byte\n");
    }

    term.c_cflag &= ~CSIZE;     /* zero out the bits */
    term.c_cflag |= CS8;        /* set 8 bits/byte */
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Figure 18.11 Example of tcgetattr and tcsetattr

□

We now describe each of the flags.

`ALTWERASE`	(`c_lflag`, FreeBSD, Mac OS X) If set, an alternate word-erase algorithm is used when the WERASE character is entered. Instead of moving backward until the previous white space character, this flag causes the WERASE character to move backward until the first nonalphanumeric character is encountered.
`BRKINT`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If this flag is set and `IGNBRK` is not set, the input and output queues are flushed when a BREAK is received, and a `SIGINT` signal is generated. This signal is generated for the foreground process group if the terminal device is a controlling terminal. If neither `IGNBRK` nor `BRKINT` is set, then a BREAK is read as a single character `\0`, unless `PARMRK` is set, in which case the BREAK is read as the 3-byte sequence `\377`, `\0`, `\0`.
`BSDLY`	(`c_oflag`, XSI, Linux, Solaris) Backspace delay mask. The values for the mask are `BS0` or `BS1`.
`CBAUDEXT`	(`c_cflag`, Solaris) Extended baud rates. Used to enable baud rates greater than B38400. (We discuss baud rates in Section 18.7.)
`CCAR_OFLOW`	(`c_cflag`, FreeBSD, Mac OS X) Enable hardware flow control of the output using the RS-232 modem carrier signal (DCD, known as Data-Carrier-Detect). This is the same as the old `MDMBUF` flag.
`CCTS_OFLOW`	(`c_cflag`, FreeBSD, Mac OS X, Solaris) Enable hardware flow control of the output using the Clear-To-Send (CTS) RS-232 signal.
`CDSR_OFLOW`	(`c_cflag`, FreeBSD, Mac OS X) Flow control the output according to the Data-Set-Ready (DSR) RS-232 signal.
`CDTR_IFLOW`	(`c_cflag`, FreeBSD, Mac OS X) Flow control the input according to the Data-Terminal-Ready (DTR) RS-232 signal.
`CIBAUDEXT`	(`c_cflag`, Solaris) Extended input baud rates. Used to enable input baud rates greater than B38400. (We discuss baud rates in Section 18.7.)
`CIGNORE`	(`c_cflag`, FreeBSD, Mac OS X) Ignore control flags.
`CLOCAL`	(`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the modem status lines are ignored. This usually means that the device is directly attached. When this flag is not set, an open of a terminal device usually blocks until the modem answers a call and establishes a connection, for example.
`CMSPAR`	(`c_oflag`, Linux) Select mark or space parity. If `PARODD` is set, the parity bit is always 1 (mark parity). Otherwise, the parity bit is always 0 (space parity).
`CRDLY`	(`c_oflag`, XSI, Linux, Solaris) Carriage return delay mask. The values for the mask are `CR0`, `CR1`, `CR2`, or `CR3`.

`CREAD` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the receiver is enabled, and characters can be received.

`CRTSCTS` (`c_cflag`, FreeBSD, Linux, Mac OS X, Solaris) Behavior depends on platform. For Solaris, enables outbound hardware flow control if set. On the other three platforms, enables both inbound and outbound hardware flow control (equivalent to `CCTS_OFLOW|CRTS_IFLOW`).

`CRTS_IFLOW` (`c_cflag`, FreeBSD, Mac OS X, Solaris) Request-To-Send (RTS) flow control of input.

`CRTSXOFF` (`c_cflag`, Solaris) If set, inbound hardware flow control is enabled. The state of the Request-To-Send RS-232 signal controls the flow control.

`CSIZE` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) This field is a mask that specifies the number of bits per byte for both transmission and reception. This size does not include the parity bit, if any. The values for the field defined by this mask are `CS5`, `CS6`, `CS7`, and `CS8`, for 5, 6, 7, and 8 bits per byte, respectively.

`CSTOPB` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, two stop bits are used; otherwise, one stop bit is used.

`ECHO` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, input characters are echoed back to the terminal device. Input characters can be echoed in either canonical or noncanonical mode.

`ECHOCTL` (`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ECHO` is set, ASCII control characters (those characters in the range 0 through octal 37, inclusive) other than the ASCII TAB, the ASCII NL, and the START and STOP characters are echoed as ^*X*, where *X* is the character formed by adding octal 100 to the control character. This means that the ASCII Control-A character (octal 1) is echoed as `^A`. Also, the ASCII DELETE character (octal 177) is echoed as `^?`. If this flag is not set, the ASCII control characters are echoed as themselves. As with the `ECHO` flag, this flag affects the echoing of control characters in both canonical and noncanonical modes.

Be aware that some systems echo the EOF character differently, since its typical value is Control-D. (Control-D is the ASCII EOT character, which can cause some terminals to hang up.) Check your manual.

`ECHOE` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ICANON` is set, the ERASE character erases the last character in the current line from the display. This is usually done in the terminal driver by writing the three-character sequence backspace, space, backspace.

If the WERASE character is supported, `ECHOE` causes the previous word to be erased using one or more of the same three-character sequence.

If the `ECHOPRT` flag is supported, the actions described here for `ECHOE` assume that the `ECHOPRT` flag is not set.

`ECHOK`	(`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ICANON` is set, the KILL character erases the current line from the display or outputs the NL character (to emphasize that the entire line was erased). If the `ECHOKE` flag is supported, this description of `ECHOK` assumes that `ECHOKE` is not set.
`ECHOKE`	(`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ICANON` is set, the KILL character is echoed by erasing each character on the line. The way in which each character is erased is selected by the `ECHOE` and `ECHOPRT` flags.
`ECHONL`	(`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ICANON` is set, the NL character is echoed, even if `ECHO` is not set.
`ECHOPRT`	(`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) If set and if both `ICANON` and `ECHO` are set, then the ERASE character (and WERASE character, if supported) cause all the characters being erased to be printed as they are erased. This is often useful on a hard-copy terminal to see exactly which characters are being deleted.
`EXTPROC`	(`c_lflag`, FreeBSD, Mac OS X) If set, canonical character processing is performed external to the operating system. This can be the case if the serial communication peripheral card can offload the host processor by doing some of the line discipline processing. This can also be the case when using pseudo terminals (Chapter 19).
`FFDLY`	(`c_oflag`, XSI, Linux, Solaris) Form feed delay mask. The values for the mask are `FF0` or `FF1`.
`FLUSHO`	(`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) If set, output is being flushed. This flag is set when we type the DISCARD character; the flag is cleared when we type another DISCARD character. We can also set or clear this condition by setting or clearing this terminal flag.
`HUPCL`	(`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the modem control lines are lowered (i.e., the modem connection is broken) when the last process closes the device.
`ICANON`	(`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, canonical mode is in effect (Section 18.10). This enables the following characters: EOF, EOL, EOL2, ERASE, KILL, REPRINT, STATUS, and WERASE. The input characters are assembled into lines. If canonical mode is not enabled, read requests are satisfied directly from the input queue. A read does not return until at least MIN bytes have been received or the timeout value TIME has expired between bytes. Refer to Section 18.11 for additional details.

`ICRNL`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if `IGNCR` is not set, a received CR character is translated into a NL character.
`IEXTEN`	(`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the extended, implementation-defined special characters are recognized and processed.
`IGNBRK`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, a BREAK condition on input is ignored. See `BRKINT` for a way to have a BREAK condition either generate a `SIGINT` signal or be read as data.
`IGNCR`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, a received CR character is ignored. If this flag is not set, it is possible to translate the received CR into a NL character if the `ICRNL` flag is set.
`IGNPAR`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, an input byte with a framing error (other than a BREAK) or an input byte with a parity error is ignored.
`IMAXBEL`	(`c_iflag`, FreeBSD, Linux, Mac OS X, Solaris) Ring bell when input queue is full.
`INLCR`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, a received NL character is translated into a CR character.
`INPCK`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, input parity checking is enabled. If `INPCK` is not set, input parity checking is disabled.
	Parity "generation and detection" and "input parity checking" are two different things. The generation and detection of parity bits is controlled by the `PARENB` flag. Setting this flag usually causes the device driver for the serial interface to generate parity for outgoing characters and to verify the parity of incoming characters. The flag `PARODD` determines whether the parity should be odd or even. If an input character arrives with the wrong parity, then the state of the `INPCK` flag is checked. If this flag is set, then the `IGNPAR` flag is checked (to see whether the input byte with the parity error should be ignored); if the byte should not be ignored, then the `PARMRK` flag is checked to see what characters should be passed to the reading process.
`ISIG`	(`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the input characters are compared against the special characters that cause the terminal-generated signals to be generated (INTR, QUIT, SUSP, and DSUSP); if equal, the corresponding signal is generated.
`ISTRIP`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, valid input bytes are stripped to 7 bits. When this flag is not set, all 8 bits are processed.

`IUCLC`	(`c_iflag`, Linux, Solaris) Map uppercase to lowercase on input.
`IXANY`	(`c_iflag`, XSI, FreeBSD, Linux, Mac OS X, Solaris) Enable any characters to restart output.
`IXOFF`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, start–stop input control is enabled. When it notices that the input queue is getting full, the terminal driver outputs a STOP character. This character should be recognized by the device that is sending the data and cause the device to stop. Later, when the characters on the input queue have been processed, the terminal driver will output a START character. This should cause the device to resume sending data.
`IXON`	(`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, start–stop output control is enabled. When the terminal driver receives a STOP character, output stops. While the output is stopped, the next START character resumes the output. If this flag is not set, the START and STOP characters are read by the process as normal characters.
`MDMBUF`	(`c_cflag`, FreeBSD, Mac OS X) Flow control the output according to the modem carrier flag. This is the old name for the `CCAR_OFLOW` flag.
`NLDLY`	(`c_oflag`, XSI, Linux, Solaris) Newline delay mask. The values for the mask are `NL0` or `NL1`.
`NOFLSH`	(`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) By default, when the terminal driver generates the `SIGINT` and `SIGQUIT` signals, both the input and output queues are flushed. Also, when it generates the `SIGSUSP` signal, the input queue is flushed. If the `NOFLSH` flag is set, this normal flushing of the queues does not occur when these signals are generated.
`NOKERNINFO`	(`c_lflag`, FreeBSD, Mac OS X) When set, this flag prevents the STATUS character from printing information on the foreground process group. Regardless of this flag, however, the STATUS character still causes the `SIGINFO` signal to be sent to the foreground process group.
`OCRNL`	(`c_oflag`, XSI, FreeBSD, Linux, Solaris) If set, map CR to NL on output.
`OFDEL`	(`c_oflag`, XSI, Linux, Solaris) If set, the output fill character is ASCII DEL; otherwise, it's ASCII NUL. See the `OFILL` flag.
`OFILL`	(`c_oflag`, XSI, Linux, Solaris) If set, fill characters (either ASCII DEL or ASCII NUL; see the `OFDEL` flag) are transmitted for a delay, instead of using a timed delay. See the six delay masks: `BSDLY`, `CRDLY`, `FFDLY`, `NLDLY`, `TABDLY`, and `VTDLY`.
`OLCUC`	(`c_oflag`, Linux, Solaris) If set, map lowercase characters to uppercase characters on output.

ONLCR (`c_oflag`, XSI, FreeBSD, Linux, Mac OS X, Solaris) If set, map NL to CR-NL on output.

ONLRET (`c_oflag`, XSI, FreeBSD, Linux, Solaris) If set, the NL character is assumed to perform the carriage return function on output.

ONOCR (`c_oflag`, XSI, FreeBSD, Linux, Solaris) If set, a CR is not output at column 0.

ONOEOT (`c_oflag`, FreeBSD, Mac OS X) If set, EOT (^D) characters are discarded on output. This may be necessary on some terminals that interpret the Control-D as a hangup.

OPOST (`c_oflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, implementation-defined output processing takes place. Refer to Figure 18.6 for the various implementation-defined flags for the `c_oflag` word.

OXTABS (`c_oflag`, FreeBSD, Mac OS X) If set, tabs are expanded to spaces on output. This produces the same effect as setting the horizontal tab delay (TABDLY) to XTABS or TAB3.

PARENB (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, parity generation is enabled for outgoing characters, and parity checking is performed on incoming characters. The parity is odd if PARODD is set; otherwise, it is even parity. See also the discussion of the INPCK, IGNPAR, and PARMRK flags.

PAREXT (`c_cflag`, Solaris) Select mark or space parity. If PARODD is set, the parity bit is always 1 (mark parity). Otherwise, the parity bit is always 0 (space parity).

PARMRK (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set and if IGNPAR is not set, a byte with a framing error (other than a BREAK) or a byte with a parity error is read by the process as the three-character sequence `\377`, `\0`, *X*, where *X* is the byte received in error. If ISTRIP is not set, a valid `\377` is passed to the process as `\377`, `\377`. If neither IGNPAR nor PARMRK is set, a byte with a framing error (other than a BREAK) or with a parity error is read as a single character `\0`.

PARODD (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the parity for outgoing and incoming characters is odd parity. Otherwise, the parity is even parity. Note that the PARENB flag controls the generation and detection of parity.

The PARODD flag also controls whether mark or space parity is used when either the CMSPAR or PAREXT flag is set.

PENDIN (`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) If set, any input that has not been read is reprinted by the system when the next character is input. This action is similar to what happens when we type the REPRINT character.

TABDLY	(c_oflag, XSI, Linux, Solaris) Horizontal tab delay mask. The values for the mask are TAB0, TAB1, TAB2, or TAB3.
	The value XTABS is equal to TAB3. This value causes the system to expand tabs into spaces. The system assumes a tab stop every eight spaces, and we can't change this assumption.
TOSTOP	(c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if the implementation supports job control, the SIGTTOU signal is sent to the process group of a background process that tries to write to its controlling terminal. By default, this signal stops all the processes in the process group. This signal is not generated by the terminal driver if the background process that is writing to the controlling terminal is either ignoring or blocking the signal.
VTDLY	(c_oflag, XSI, Linux, Solaris) Vertical tab delay mask. The values for the mask are VT0 or VT1.
XCASE	(c_lflag, Linux, Solaris) If set and if ICANON is also set, the terminal is assumed to be uppercase only, and all input is converted to lowercase. To input an uppercase character, precede it with a backslash. Similarly, an uppercase character is output by the system by being preceded by a backslash. (This option flag is obsolete today, since most, if not all, uppercase-only terminals have disappeared.)

18.6 stty Command

All the options described in the previous section can be examined and changed from within a program, with the tcgetattr and tcsetattr functions (Section 18.4) or from the command line (or a shell script), with the stty(1) command. This command is simply an interface to the first six functions that we listed in Figure 18.7. If we execute this command with its -a option, it displays all the terminal options:

```
$ stty -a
speed 9600 baud; 25 rows; 80 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -ocrnl -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts
        -dsrflow -dtrflow -mdmbuf
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; erase2 = ^?; intr = ^C; kill = ^U;
        lnext = ^V; min = 1; quit = ^; reprint = ^R; start = ^Q;
        status = ^T; stop = ^S; susp = ^Z; time = 0; werase = ^W;
```

Option names preceded by a hyphen are disabled. The last four lines display the current settings for each of the terminal special characters (Section 18.3). The first line displays the number of rows and columns for the current terminal window; we discuss this in Section 18.12.

> The `stty` command uses its standard input to get and set the terminal option flags. Although some older implementations used standard output, POSIX.1 requires that the standard input be used. All four implementations discussed in this text provide versions of `stty` that operate on standard input. This means that we can type
>
> ```
> stty -a </dev/tty1a
> ```
>
> if we are interested in discovering the settings on the terminal named `tty1a`.

18.7 Baud Rate Functions

The term *baud rate* is a historical term that should be referred to today as "bits per second." Although most terminal devices use the same baud rate for both input and output, the capability exists to set the two to different values, if the hardware allows this.

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);

speed_t cfgetospeed(const struct termios *termptr);

                                    Both return: baud rate value

int cfsetispeed(struct termios *termptr, speed_t speed);

int cfsetospeed(struct termios *termptr, speed_t speed);

                                    Both return: 0 if OK, -1 on error
```

The return value from the two `cfget` functions and the *speed* argument to the two `cfset` functions are one of the following constants: `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, or `B38400`. The constant `B0` means "hang up." When `B0` is specified as the output baud rate when `tcsetattr` is called, the modem control lines are no longer asserted.

> Most systems define additional baud rate values, such as `B57600` and `B115200`.

To use these functions, we must realize that the input and output baud rates are stored in the device's `termios` structure, as shown in Figure 18.8. Before calling either of the `cfget` functions, we first have to obtain the device's `termios` structure using `tcgetattr`. Similarly, after calling either of the two `cfset` functions, all we've done is set the baud rate in a `termios` structure. For this change to affect the device, we have to call `tcsetattr`. If there is an error in either of the baud rates that we set, we may not find out about the error until we call `tcsetattr`.

The four baud rate functions exist to insulate applications from differences in the way that implementations represent baud rates in the `termios` structure. BSD-derived platforms tend to store baud rates as numeric values equal to the rates (i.e., 9,600 baud is stored as the value 9,600), whereas Linux and System V–derived platforms tend to encode the baud rate in a bitmask. The speed values we get from the `cfget` functions and pass to the `cfset` functions are untranslated from their representation as they are stored in the `termios` structure.

18.8 Line Control Functions

The following four functions provide line control capability for terminal devices. All four require that *filedes* refer to a terminal device; otherwise, an error is returned with `errno` set to `ENOTTY`.

```
#include <termios.h>

int tcdrain(int filedes);

int tcflow(int filedes, int action);

int tcflush(int filedes, int queue);

int tcsendbreak(int filedes, int duration);

                              All four return: 0 if OK, -1 on error
```

The `tcdrain` function waits for all output to be transmitted. The `tcflow` function gives us control over both input and output flow control. The *action* argument must be one of the following four values:

`TCOOFF`	Output is suspended.
`TCOON`	Output that was previously suspended is restarted.
`TCIOFF`	The system transmits a STOP character, which should cause the terminal device to stop sending data.
`TCION`	The system transmits a START character, which should cause the terminal device to resume sending data.

The `tcflush` function lets us flush (throw away) either the input buffer (data that has been received by the terminal driver, which we have not read) or the output buffer (data that we have written, which has not yet been transmitted). The *queue* argument must be one of the following three constants:

`TCIFLUSH`	The input queue is flushed.
`TCOFLUSH`	The output queue is flushed.
`TCIOFLUSH`	Both the input and the output queues are flushed.

The `tcsendbreak` function transmits a continuous stream of zero bits for a specified duration. If the *duration* argument is 0, the transmission lasts between 0.25 seconds and 0.5 seconds. POSIX.1 specifies that if *duration* is nonzero, the transmission time is implementation dependent.

18.9 Terminal Identification

Historically, the name of the controlling terminal in most versions of the UNIX System has been `/dev/tty`. POSIX.1 provides a runtime function that we can call to determine the name of the controlling terminal.

```
#include <stdio.h>

char *ctermid(char *ptr);
```

Returns: pointer to name of controlling terminal
on success, pointer to empty string on error

If *ptr* is non-null, it is assumed to point to an array of at least `L_ctermid` bytes, and the name of the controlling terminal of the process is stored in the array. The constant `L_ctermid` is defined in `<stdio.h>`. If *ptr* is a null pointer, the function allocates room for the array (usually as a static variable). Again, the name of the controlling terminal of the process is stored in the array.

In both cases, the starting address of the array is returned as the value of the function. Since most UNIX systems use `/dev/tty` as the name of the controlling terminal, this function is intended to aid portability to other operating systems.

All four platforms described in this text return the string `/dev/tty` when we call `ctermid`.

Example—`ctermid` Function

Figure 18.12 shows an implementation of the POSIX.1 `ctermid` function.

```
#include    <stdio.h>
#include    <string.h>

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty"));    /* strcpy() returns str */
}
```

Figure 18.12 Implementation of POSIX.1 `ctermid` function

Note that we can't protect against overrunning the caller's buffer, because we have no way to determine its size. □

Two functions that are more interesting for a UNIX system are `isatty`, which returns true if a file descriptor refers to a terminal device, and `ttyname`, which returns the pathname of the terminal device that is open on a file descriptor.

```
#include <unistd.h>

int isatty(int filedes);

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

char *ttyname(int filedes);

                    Returns: pointer to pathname of terminal, NULL on error
```

Example—`isatty` Function

The `isatty` function is trivial to implement, as we show in Figure 18.13. We simply try one of the terminal-specific functions (that doesn't change anything if it succeeds) and look at the return value.

```
#include    <termios.h>

int
isatty(int fd)
{
    struct termios  ts;

    return(tcgetattr(fd, &ts) != -1); /* true if no error (is a tty) */
}
```

Figure 18.13 Implementation of POSIX.1 `isatty` function

We test our `isatty` function with the program in Figure 18.14.

```
#include "apue.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "not a tty");
    exit(0);
}
```

Figure 18.14 Test the `isatty` function

When we run the program from Figure 18.14, we get the following output:

```
$ ./a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ ./a.out </etc/passwd 2>/dev/null
fd 0: not a tty
fd 1: tty
fd 2: not a tty
```

□

Example—`ttyname` Function

The `ttyname` function (Figure 18.15) is longer, as we have to search all the device entries, looking for a match.

```
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    <string.h>
#include    <termios.h>
#include    <unistd.h>
#include    <stdlib.h>

struct devdir {
    struct devdir   *d_next;
    char            *d_name;
};

static struct devdir    *head;
static struct devdir    *tail;
static char             pathname[_POSIX_PATH_MAX + 1];

static void
add(char *dirname)
{
    struct devdir   *ddp;
    int             len;

    len = strlen(dirname);

    /*
     * Skip ., .., and /dev/fd.
     */
    if ((dirname[len-1] == '.') && (dirname[len-2] == '/' ||
      (dirname[len-2] == '.' && dirname[len-3] == '/')))
        return;
    if (strcmp(dirname, "/dev/fd") == 0)
        return;
    ddp = malloc(sizeof(struct devdir));
    if (ddp == NULL)
```

```
            return;

        ddp->d_name = strdup(dirname);
        if (ddp->d_name == NULL) {
            free(ddp);
            return;
        }
        ddp->d_next = NULL;
        if (tail == NULL) {
            head = ddp;
            tail = ddp;
        } else {
            tail->d_next = ddp;
            tail = ddp;
        }
    }

    static void
    cleanup(void)
    {
        struct devdir   *ddp, *nddp;

        ddp = head;
        while (ddp != NULL) {
            nddp = ddp->d_next;
            free(ddp->d_name);
            free(ddp);
            ddp = nddp;
        }
        head = NULL;
        tail = NULL;
    }

    static char *
    searchdir(char *dirname, struct stat *fdstatp)
    {
        struct stat     devstat;
        DIR             *dp;
        int             devlen;
        struct dirent   *dirp;

        strcpy(pathname, dirname);
        if ((dp = opendir(dirname)) == NULL)
            return(NULL);
        strcat(pathname, "/");
        devlen = strlen(pathname);
        while ((dirp = readdir(dp)) != NULL) {
            strncpy(pathname + devlen, dirp->d_name,
              _POSIX_PATH_MAX - devlen);

            /*
             * Skip aliases.
             */
        if (strcmp(pathname, "/dev/stdin") == 0 ||
          strcmp(pathname, "/dev/stdout") == 0 ||
          strcmp(pathname, "/dev/stderr") == 0)
            continue;
        if (stat(pathname, &devstat) < 0)
            continue;
        if (S_ISDIR(devstat.st_mode)) {
            add(pathname);
            continue;
        }
        if (devstat.st_ino == fdstatp->st_ino &&
          devstat.st_dev == fdstatp->st_dev) {  /* found a match */
            closedir(dp);
            return(pathname);
        }
    }

    closedir(dp);
    return(NULL);
}

char *
ttyname(int fd)
{
    struct stat     fdstat;
    struct devdir   *ddp;
    char            *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    rval = searchdir("/dev", &fdstat);
    if (rval == NULL) {
        for (ddp = head; ddp != NULL; ddp = ddp->d_next)
            if ((rval = searchdir(ddp->d_name, &fdstat)) != NULL)
                break;
    }

    cleanup();
    return(rval);
}
```

Figure 18.15 Implementation of POSIX.1 `ttyname` function

The technique is to read the `/dev` directory, looking for an entry with the same device number and i-node number. Recall from Section 4.23 that each file system has a unique device number (the `st_dev` field in the `stat` structure, from Section 4.2), and each directory entry in that file system has a unique i-node number (the `st_ino` field in the stat structure). We assume in this function that when we hit a matching device number and matching i-node number, we've located the desired directory entry. We could also verify that the two entries have matching st_rdev fields (the major and minor device numbers for the terminal device) and that the directory entry is also a character special file. But since we've already verified that the file descriptor argument is both a terminal device and a character special file, and since a matching device number and i-node number is unique on a UNIX system, there is no need for the additional comparisons.

The name of our terminal might reside in a subdirectory in /dev. Thus, we might need to search the entire file system tree under /dev. We skip several directories that might produce incorrect or odd-looking results: /dev/., /dev/.., and /dev/fd. We also skip the aliases /dev/stdin, /dev/stdout, and /dev/stderr, since they are symbolic links to files in /dev/fd.

We can test this implementation with the program shown in Figure 18.16.

```
#include "apue.h"

int
main(void)
{
    char *name;

    if (isatty(0)) {
        name = ttyname(0);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 0: %s\n", name);
    if (isatty(1)) {
        name = ttyname(1);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 1: %s\n", name);
    if (isatty(2)) {
        name = ttyname(2);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 2: %s\n", name);
    exit(0);
}
```

Figure 18.16 Test the ttyname function

Running the program from Figure 18.16 gives us

```
$ ./a.out < /dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttyp3
fd 2: not a tty
```

□

18.10 Canonical Mode

Canonical mode is simple: we issue a read, and the terminal driver returns when a line has been entered. Several conditions cause the read to return.

- The read returns when the requested number of bytes have been read. We don't have to read a complete line. If we read a partial line, no information is lost; the next read starts where the previous read stopped.
- The read returns when a line delimiter is encountered. Recall from Section 18.3 that the following characters are interpreted as end of line in canonical mode: NL, EOL, EOL2, and EOF. Also, recall from Section 18.5 that if `ICRNL` is set and if `IGNCR` is not set, then the CR character also terminates a line, since it acts just like the NL character.

  Realize that of these five line delimiters, one (EOF) is discarded by the terminal driver when it's processed. The other four are returned to the caller as the last character of the line.
- The read also returns if a signal is caught and if the function is not automatically restarted (Section 10.5).

Example—`getpass` Function

We now show the function `getpass`, which reads a password of some type from the user at a terminal. This function is called by the `login`(1) and `crypt`(1) programs. To read the password, the function must turn off echoing, but it can leave the terminal in canonical mode, as whatever we type as the password forms a complete line. Figure 18.17 shows a typical implementation on a UNIX system.

There are several points to consider in this example.

- Instead of hardwiring `/dev/tty` into the program, we call the function `ctermid` to open the controlling terminal.
- We read and write only to the controlling terminal and return an error if we can't open this device for reading and writing. There are other conventions to use. The BSD version of `getpass` reads from standard input and writes to standard error if the controlling terminal can't be opened for reading and writing. The System V version always writes to standard error but reads only from the controlling terminal.

```
#include    <signal.h>
#include    <stdio.h>
#include    <termios.h>

#define MAX_PASS_LEN    8       /* max #chars for user to enter */

char *
getpass(const char *prompt)
{
    static char     buf[MAX_PASS_LEN + 1];  /* null byte at end */
    char            *ptr;
    sigset_t        sig, osig;
    struct termios  ts, ots;
    FILE            *fp;
    int             c;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig);
    sigaddset(&sig, SIGINT);        /* block SIGINT */
    sigaddset(&sig, SIGTSTP);       /* block SIGTSTP */
    sigprocmask(SIG_BLOCK, &sig, &osig);    /* and save mask */

    tcgetattr(fileno(fp), &ts);     /* save tty state */
    ots = ts;                       /* structure copy */
    ts.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &ts);
    fputs(prompt, fp);

    ptr = buf;
    while ((c = getc(fp)) != EOF && c != '\n')
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    *ptr = 0;               /* null terminate */
    putc('\n', fp);         /* we echo a newline */

    tcsetattr(fileno(fp), TCSAFLUSH, &ots); /* restore TTY state */
    sigprocmask(SIG_SETMASK, &osig, NULL);  /* restore mask */
    fclose(fp);             /* done with /dev/tty */
    return(buf);
}
```

Figure 18.17 Implementation of getpass function

- We block the two signals SIGINT and SIGTSTP. If we didn't do this, entering the INTR character would abort the program and leave the terminal with echoing disabled. Similarly, entering the SUSP character would stop the program and return to the shell with echoing disabled. We choose to block the signals while we have echoing disabled. If they are generated while we're reading the password, they are held until we return. There are other ways to handle these signals. Some versions just ignore SIGINT (saving its previous action) while in getpass, resetting the action for this signal to its previous value before returning. This means that any occurrence of the signal while it's ignored is lost. Other versions catch SIGINT (saving its previous action) and if the signal is caught, send themselves the signal with the kill function after resetting the terminal state and signal action. None of the versions of getpass catch, ignore, or block SIGQUIT, so entering the QUIT character aborts the program and probably leaves the terminal with echoing disabled.

- Be aware that some shells, notably the Korn shell, turn echoing back on whenever they read interactive input. These shells are the ones that provide command-line editing and therefore manipulate the state of the terminal every time we enter an interactive command. So, if we invoke this program under one of these shells and abort it with the QUIT character, it may reenable echoing for us. Other shells that don't provide this form of command-line editing, such as the Bourne shell, will abort the program and leave the terminal in no-echo mode. If we do this to our terminal, the stty command can reenable echoing.
- We use standard I/O to read and write the controlling terminal. We specifically set the stream to be unbuffered; otherwise, there might be some interactions between the writing and reading of the stream (we would need some calls to fflush). We could have also used unbuffered I/O (Chapter 3), but we would have to simulate the getc function using read.
- We store only up to eight characters as the password. Any additional characters that are entered are ignored.

The program in Figure 18.18 calls getpass and prints what we enter to let us verify that the ERASE and KILL characters work (as they should in canonical mode).

```
#include "apue.h"

char    *getpass(const char *);

int
main(void)
{
    char    *ptr;

    if ((ptr = getpass("Enter password:")) == NULL)
        err_sys("getpass error");
    printf("password: %s\n", ptr);

    /* now use password (probably encrypt it) ... */

    while (*ptr != 0)
        *ptr++ = 0;     /* zero it out when we're done with it */
    exit(0);
}
```

Figure 18.18 Call the getpass function

Whenever a program that calls getpass is done with the cleartext password, the program should zero it out in memory, just to be safe. If the program were to generate a core file that others might be able to read or if some other process were somehow able to read our memory, they might be able to read the cleartext password. (By "cleartext," we mean the password that we type at the prompt that is printed by getpass. Most UNIX system programs then modify this cleartext password into an "encrypted" password. The field pw_passwd in the password file, for example, contains the encrypted password, not the cleartext password.) □

18.11 Noncanonical Mode

Noncanonical mode is specified by turning off the ICANON flag in the c_lflag field of the termios structure. In noncanonical mode, the input data is not assembled into lines. The following special characters (Section 18.3) are not processed: ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS, and WERASE.

As we said, canonical mode is easy: the system returns up to one line at a time. But with noncanonical mode, how does the system know when to return data to us? If it returned one byte at a time, overhead would be excessive. (Recall Figure 3.5, which showed the overhead in reading one byte at a time. Each time we doubled the amount of data returned, we halved the system call overhead.) The system can't always return multiple bytes at a time, since sometimes we don't know how much data to read until we start reading it.

The solution is to tell the system to return when either a specified amount of data has been read or after a given amount of time has passed. This technique uses two variables in the c_cc array in the termios structure: MIN and TIME. These two elements of the array are indexed by the names VMIN and VTIME.

MIN specifies the minimum number of bytes before a read returns. TIME specifies the number of tenths of a second to wait for data to arrive. There are four cases.

Case A: MIN > 0, TIME > 0

TIME specifies an interbyte timer that is started only when the first byte is received. If MIN bytes are received before the timer expires, read returns MIN bytes. If the timer expires before MIN bytes are received, read returns the bytes received. (At least one byte is returned if the timer expires, because the timer is not started until the first byte is received.) In this case, the caller blocks until the first byte is received. If data is already available when read is called, it is as if the data had been received immediately after the read.

Case B: MIN > 0, TIME == 0

The read does not return until MIN bytes have been received. This can cause a read to block indefinitely.

Case C: MIN == 0, TIME > 0

TIME specifies a read timer that is started when `read` is called. (Compare this to case A, in which a nonzero TIME represented an interbyte timer that was not started until the first byte was received.) The `read` returns when a single byte is received or when the timer expires. If the timer expires, `read` returns 0.

Case D: MIN == 0, TIME == 0

If some data is available, `read` returns up to the number of bytes requested. If no data is available, `read` returns 0 immediately.

Realize in all these cases that MIN is only a minimum. If the program requests more than MIN bytes of data, it's possible to receive up to the requested amount. This also applies to cases C and D, in which MIN is 0.

Figure 18.19 summarizes the four cases for noncanonical input. In this figure, *nbytes* is the third argument to `read` (the maximum number of bytes to return).

	MIN > 0	MIN == 0
TIME > 0	**A**: `read` returns [MIN, *nbytes*] before timer expires; `read` returns [1, MIN) if timer expires. (TIME = interbyte timer. Caller can block indefinitely.)	**C**: `read` returns [1, *nbytes*] before timer expires; `read` returns 0 if timer expires. (TIME = `read` timer.)
TIME == 0	**B**: `read` returns [MIN, *nbytes*] when available. (Caller can block indefinitely.)	**D**: `read` returns [0, *nbytes*] immediately.

Figure 18.19 Four cases for noncanonical input

Be aware that POSIX.1 allows the subscripts `VMIN` and `VTIME` to have the same values as `VEOF` and `VEOL`, respectively. Indeed, Solaris does this for backward compatibility with older versions of System V. This creates a portability problem, however. In going from noncanonical to canonical mode, we must now restore `VEOF` and `VEOL` also. If `VMIN` equals `VEOF` and we don't restore their values, when we set `VMIN` to its typical value of 1, the end-of-file character becomes Control-A. The easiest way around this problem is to save the entire `termios` structure when going into noncanonical mode and restore it when going back to canonical mode.

Example

The program in Figure 18.20 defines the `tty_cbreak` and `tty_raw` functions that set the terminal in *cbreak mode* and *raw mode*. (The terms *cbreak* and *raw* come from the Version 7 terminal driver.) We can reset the terminal to its original state (the state before either of these functions was called) by calling the function `tty_reset`.

If we've called `tty_cbreak`, we need to call `tty_reset` before calling `tty_raw`. The same goes for calling `tty_cbreak` after calling `tty_raw`. This improves the chances that the terminal will be left in a usable state if we encounter any errors.

Two additional functions are also provided: `tty_atexit` can be established as an exit handler to ensure that the terminal mode is reset by `exit`, and `tty_termios` returns a pointer to the original canonical mode `termios` structure.

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

static struct termios       save_termios;
static int                  ttysavefd = -1;
static enum { RESET, RAW, CBREAK } ttystate = RESET;

int
tty_cbreak(int fd)  /* put terminal into a cbreak mode */
{
    int             err;
    struct termios  buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* structure copy */

    /*
     * Echo off, canonical mode off.
     */
    buf.c_lflag &= ~(ECHO | ICANON);

    /*
     * Case B: 1 byte at a time, no timer.
     */
    buf.c_cc[VMIN] = 1;
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);

    /*
     * Verify that the changes stuck.  tcsetattr can return 0 on
     * partial success.
     */
    if (tcgetattr(fd, &buf) < 0) {
        err = errno;
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = err;
        return(-1);
    }
    if ((buf.c_lflag & (ECHO | ICANON)) || buf.c_cc[VMIN] != 1 ||
      buf.c_cc[VTIME] != 0) {
        /*
         * Only some of the changes were made.  Restore the
         * original settings.
         */
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = EINVAL;
        return(-1);
    }

    ttystate = CBREAK;
    ttysavefd = fd;
    return(0);
}

int
tty_raw(int fd)     /* put terminal into a raw mode */
{
    int             err;
    struct termios  buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* structure copy */

    /*
     * Echo off, canonical mode off, extended input
     * processing off, signal chars off.
     */
    buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    /*
     * No SIGINT on BREAK, CR-to-NL off, input parity
     * check off, don't strip 8th bit on input, output
     * flow control off.
     */
    buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    /*
     * Clear size bits, parity checking off.
     */
    buf.c_cflag &= ~(CSIZE | PARENB);

    /*
     * Set 8 bits/char.
     */
    buf.c_cflag |= CS8;

    /*
```

```
         * Output processing off.
         */
        buf.c_oflag &= ~(OPOST);

        /*
         * Case B: 1 byte at a time, no timer.
         */
        buf.c_cc[VMIN] = 1;
        buf.c_cc[VTIME] = 0;
        if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
            return(-1);

        /*
         * Verify that the changes stuck.  tcsetattr can return 0 on
         * partial success.
         */
        if (tcgetattr(fd, &buf) < 0) {
            err = errno;
            tcsetattr(fd, TCSAFLUSH, &save_termios);
            errno = err;
            return(-1);
        }
        if ((buf.c_lflag & (ECHO | ICANON | IEXTEN | ISIG)) ||
          (buf.c_iflag & (BRKINT | ICRNL | INPCK | ISTRIP | IXON)) ||
          (buf.c_cflag & (CSIZE | PARENB | CS8)) != CS8 ||
          (buf.c_oflag & OPOST) || buf.c_cc[VMIN] != 1 ||
          buf.c_cc[VTIME] != 0) {
            /*
             * Only some of the changes were made.  Restore the
             * original settings.
             */
            tcsetattr(fd, TCSAFLUSH, &save_termios);
            errno = EINVAL;
            return(-1);
        }

        ttystate = RAW;
        ttysavefd = fd;
        return(0);
    }

    int
    tty_reset(int fd)       /* restore terminal's mode */
    {
        if (ttystate == RESET)
            return(0);
        if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
            return(-1);
        ttystate = RESET;
        return(0);
    }
```

```
void
tty_atexit(void)        /* can be set up by atexit(tty_atexit) */
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}

struct termios *
tty_termios(void)       /* let caller see original tty state */
{
    return(&save_termios);
}
```

Figure 18.20 Set terminal mode to cbreak or raw

Our definition of cbreak mode is the following:

- Noncanonical mode. As we mentioned at the beginning of this section, this mode turns off some input character processing. It does not turn off signal handling, so the user can always type one of the terminal-generated signals. Be aware that the caller should catch these signals, or there's a chance that the signal will terminate the program, and the terminal will be left in cbreak mode.

  As a general rule, whenever we write a program that changes the terminal mode, we should catch most signals. This allows us to reset the terminal mode before terminating.
- Echo off.
- One byte at a time input. To do this, we set MIN to 1 and TIME to 0. This is case B from Figure 18.19. A `read` won't return until at least one byte is available.

We define raw mode as follows:

- Noncanonical mode. We also turn off processing of the signal-generating characters (`ISIG`) and the extended input character processing (`IEXTEN`). Additionally, we disable a BREAK character from generating a signal, by turning off `BRKINT`.
- Echo off.
- We disable the CR-to-NL mapping on input (`ICRNL`), input parity detection (`INPCK`), the stripping of the eighth bit on input (`ISTRIP`), and output flow control (`IXON`).
- Eight-bit characters (`CS8`), and parity checking is disabled (`PARENB`).
- All output processing is disabled (`OPOST`).
- One byte at a time input (MIN = 1, TIME = 0).

The program in Figure 18.21 tests raw and cbreak modes.

```
#include "apue.h"

static void
sig_catch(int signo)
{
    printf("signal caught\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

int
main(void)
{
    int     i;
    char    c;

    if (signal(SIGINT, sig_catch) == SIG_ERR)    /* catch signals */
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("signal(SIGTERM) error");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("Enter raw mode characters, terminate with DELETE\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177)     /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");
    if (tty_cbreak(STDIN_FILENO) < 0)
        err_sys("tty_cbreak error");
    printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        c &= 255;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");

    exit(0);
}
```

Figure 18.21 Test raw and cbreak terminal modes

Running the program in Figure 18.21, we can see what happens with these two terminal modes:

```
$ ./a.out
Enter raw mode characters, terminate with DELETE
                                                     4
                                                      33
                                                        133
                                                           61
                                                             70
                                                               176
                         type DELETE
Enter cbreak mode characters, terminate with SIGINT
1                        type Control-A
10                       type backspace
signal caught            type interrupt key
```

In raw mode, the characters entered were Control-D (04) and the special function key F7. On the terminal being used, this function key generated five characters: *ESC* (033), *[* (0133), *1* (061), *8* (070), and *~* (0176). Note that with the output processing turned off in raw mode (~OPOST), we do not get a carriage return output after each character. Also note that special-character processing is disabled in cbreak mode (so, for example, Control-D, the end-of-file character, and backspace aren't handled specially), whereas the terminal-generated signals are still processed. □

18.12 Terminal Window Size

Most UNIX systems provide a way to keep track of the current terminal window size and to have the kernel notify the foreground process group when the size changes. The kernel maintains a `winsize` structure for every terminal and pseudo terminal:

```
struct winsize {
  unsigned short  ws_row;     /* rows, in characters */
  unsigned short  ws_col;     /* columns, in characters */
  unsigned short  ws_xpixel;  /* horizontal size, pixels (unused) */
  unsigned short  ws_ypixel;  /* vertical size, pixels (unused) */
};
```

The rules for this structure are as follows.

- We can fetch the current value of this structure using an `ioctl` (Section 3.15) of `TIOCGWINSZ`.
- We can store a new value of this structure in the kernel using an `ioctl` of `TIOCSWINSZ`. If this new value differs from the current value stored in the kernel, a `SIGWINCH` signal is sent to the foreground process group. (Note from Figure 10.1 that the default action for this signal is to be ignored.)

- Other than storing the current value of the structure and generating a signal when the value changes, the kernel does nothing else with this structure. Interpreting the structure is entirely up to the application.

The reason for providing this feature is to notify applications (such as the vi editor) when the window size changes. When it receives the signal, the application can fetch the new size and redraw the screen.

Example

Figure 18.22 shows a program that prints the current window size and goes to sleep. Each time the window size changes, SIGWINCH is caught and the new size is printed. We have to terminate this program with a signal.

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

static void
pr_winsize(int fd)
{
    struct winsize  size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
}

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);
    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("signal error");
    pr_winsize(STDIN_FILENO);     /* print initial size */
    for ( ; ; )                   /* and sleep forever */
        pause();
}
```

Figure 18.22 Print window size

Running the program in Figure 18.22 on a windowed terminal gives us

```
$ ./a.out
35 rows, 80 columns          initial size
SIGWINCH received            change window size: signal is caught
40 rows, 123 columns
SIGWINCH received            and again
42 rows, 33 columns
^? $                         type the interrupt key to terminate
```

□

18.13 termcap, terminfo, and curses

termcap stands for "terminal capability," and it refers to the text file /etc/termcap and a set of routines to read this file. The termcap scheme was developed at Berkeley to support the vi editor. The termcap file contains descriptions of various terminals: what features the terminal supports (how many lines and rows, whether the terminal support backspace, etc.) and how to make the terminal perform certain operations (clear the screen, move the cursor to a given location, etc.). Taking this information out of the compiled program and placing it into a text file that can easily be edited allows the vi editor to run on many different terminals.

The routines that support the termcap file were then extracted from the vi editor and placed into a separate curses library. Many features were added to make this library usable for any program that wanted to manipulate the screen.

The termcap scheme was not perfect. As more and more terminals were added to the data file, it took longer to scan the file, looking for a specific terminal. The data file also used two-character names to identify the various terminal attributes. These deficiencies led to development of the terminfo scheme and its associated curses library. The terminal descriptions in terminfo are basically compiled versions of a textual description and can be located faster at runtime. terminfo appeared with SVR2 and has been in all System V releases since then.

Historically, System V–based systems used terminfo, and BSD-derived systems used termcap, but it is now common for systems to provide both. Mac OS X, however, supports only terminfo.

A description of terminfo and the curses library is provided by Goodheart [1991], but this is currently out of print. Strang [1986] describes the Berkeley version of the curses library. Strang, Mui, and O'Reilly [1988] provide a description of termcap and terminfo.

The ncurses library, a free version that is compatible with the SVR4 curses interface, can be found at http://invisible-island.net/ncurses/ncurses.html.

Neither termcap nor terminfo, by itself, addresses the problems we've been looking at in this chapter: changing the terminal's mode, changing one of the terminal special characters, handling the window size, and so on. What they do provide is a way to perform typical operations (clear the screen, move the cursor) on a wide variety of terminals. On the other hand, curses does help with some of the details that we've addressed in this chapter. Functions are provided by curses to set raw mode, set cbreak mode, turn echo on and off, and the like. But the curses library is designed for character-based dumb terminals, which have mostly been replaced by pixel-based graphics terminals today.

18.14 Summary

Terminals have many features and options, most of which we're able to change to suit our needs. In this chapter, we've described numerous functions that change a terminal's operation: special input characters and the option flags. We've looked at all the terminal special characters and the many options that can be set or reset for a terminal device.

There are two modes of terminal input—canonical (line at a time) and noncanonical. We showed examples of both modes and provided functions that map between the POSIX.1 terminal options and the older BSD cbreak and raw modes. We also described how to fetch and change the window size of a terminal.

Exercises

18.1 Write a program that calls tty_raw and terminates (without resetting the terminal mode). If your system provides the reset(1) command (all four systems described in this text do), use it to restore the terminal mode.

18.2 The PARODD flag in the c_cflag field allows us to specify even or odd parity. The BSD tip program, however, also allows the parity bit to be 0 or 1. How does it do this?

18.3 If your system's stty(1) command outputs the MIN and TIME values, do the following exercise. Log in to the system twice and start the vi editor from one login. Use the stty command from your other login to determine what values vi sets MIN and TIME to (since vi sets the terminal to noncanonical mode). (If you are running a windowing system on your terminal, you can do this same test by logging in once and using two separate windows instead.)

19

Pseudo Terminals

19.1 Introduction

In Chapter 9, we saw that terminal logins come in through a terminal device, automatically providing terminal semantics. A terminal line discipline (Figure 18.2) exists between the terminal and the programs that we run, so we can set the terminal's special characters (backspace, line erase, interrupt, etc.) and the like. When a login arrives on a network connection, however, a terminal line discipline is not automatically provided between the incoming network connection and the login shell. Figure 9.5 showed that a *pseudo-terminal* device driver is used to provide terminal semantics.

In addition to network logins, pseudo terminals have other uses that we explore in this chapter. We start with an overview on how to use pseudo terminals, followed by a discussion of specific use cases. We then provide functions to create pseudo terminals on various platforms and then use these functions to write a program that we call `pty`. We'll show various uses of this program: making a transcript of all the character input and output on the terminal (the `script(1)` program) and running coprocesses to avoid the buffering problems we encountered in the program from Figure 15.19.

19.2 Overview

The term *pseudo terminal* implies that it looks like a terminal to an application program, but it's not a real terminal. Figure 19.1 shows the typical arrangement of the processes involved when a pseudo terminal is being used. The key points in this figure are the following.

- Normally, a process opens the pseudo-terminal master and then calls `fork`. The child establishes a new session, opens the corresponding pseudo-terminal slave,

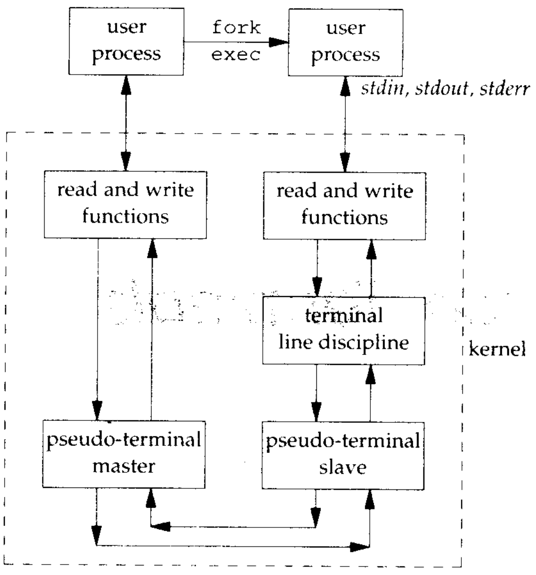

Figure 19.1 Typical arrangement of processes using a pseudo terminal

duplicates the file descriptor to the standard input, standard output, and standard error, and then calls `exec`. The pseudo-terminal slave becomes the controlling terminal for the child process.

- It appears to the user process above the slave that its standard input, standard output, and standard error are a terminal device. The process can issue all the terminal I/O functions from Chapter 18 on these descriptors. But since there is not a real terminal device beneath the slave, functions that don't make sense (change the baud rate, send a break character, set odd parity, etc.) are just ignored.
- Anything written to the master appears as input to the slave and vice versa. Indeed, all the input to the slave comes from the user process above the pseudo-terminal master. This behaves like a bidirectional pipe, but with the terminal line discipline module above the slave, we have additional capabilities over a plain pipe.

Figure 19.1 shows what a pseudo terminal looks like on a FreeBSD, Mac OS X, or Linux system. In Sections 19.3.2 and 19.3.3, we show how to open these devices.

Under Solaris, a pseudo terminal is built using the STREAMS subsystem (Section 14.4). Figure 19.2 details the arrangement of the pseudo-terminal STREAMS modules under Solaris. The two STREAMS modules that are shown as dashed boxes are optional. The `pckt` and `ptem` modules help provide semantics specific to pseudo terminals. The other two modules (`ldterm` and `ttcompat`) provide line discipline processing.