19

Pseudo Terminals

530128

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 (e.g., backspace, line erase, interrupt) 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. Next we provide functions to create pseudo terminals on various platforms, and then we 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.

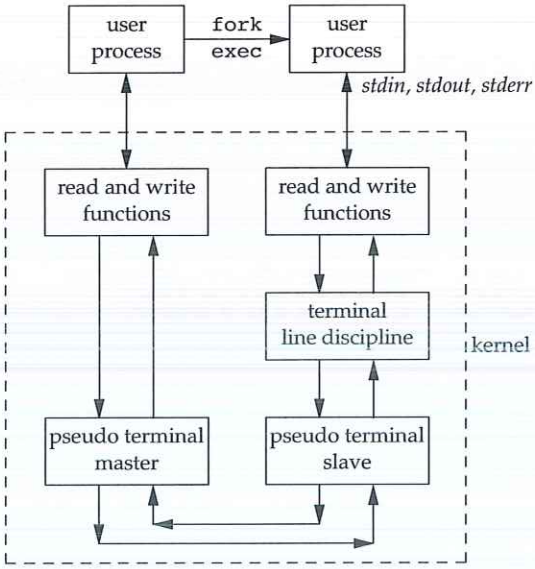

Figure 19.1 Typical arrangement of processes using a pseudo terminal

- Normally, a process opens the pseudo terminal master and then calls `fork`. The child establishes a new session, opens the corresponding pseudo terminal slave, duplicates 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 the slave isn't a real terminal device, functions that don't make sense (e.g., change the baud rate, send a break character, set odd parity) 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 Section 19.3, we show how to open these devices.

Under Solaris, a pseudo terminal is built using the STREAMS subsystem. 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. In Section 19.3, we show how to build this arrangement of STREAMS modules.

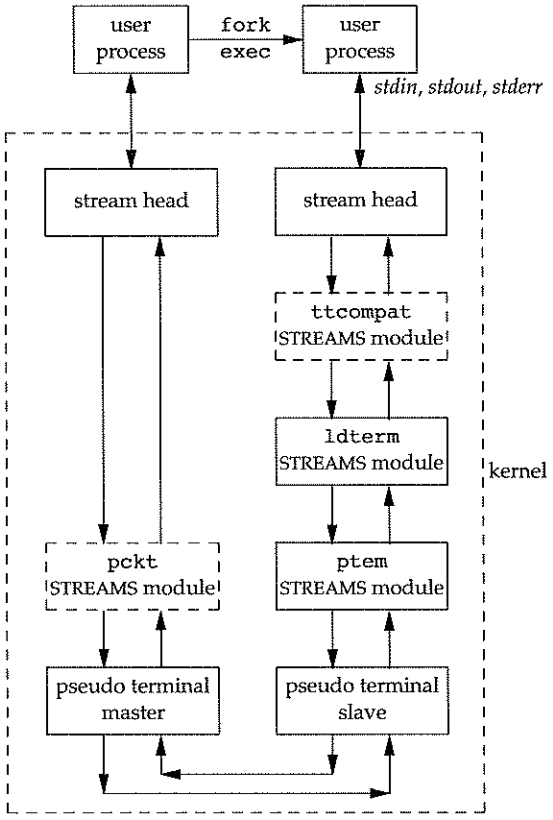

Figure 19.2 Arrangement of pseudo terminals under Solaris

From this point on, we'll simplify the figures by not showing the "read and write functions" from Figure 19.1 or the "stream head" from Figure 19.2. We'll also use the abbreviation PTY for pseudo terminal and lump all the STREAMS modules above the slave PTY in Figure 19.2 into a box called "terminal line discipline," as in Figure 19.1.

We'll now examine some of the typical uses of pseudo terminals.

Network Login Servers

Pseudo terminals are built into servers that provide network logins. The typical examples are the `telnetd` and `rlogind` servers. Chapter 15 of Stevens [1990] details the steps involved in the `rlogin` service. Once the login shell is running on the remote host, we have the arrangement shown in Figure 19.3. A similar arrangement is used by the `telnetd` server.

We show two calls to `exec` between the `rlogind` server and the login shell, because the `login` program is usually between the two to validate the user.

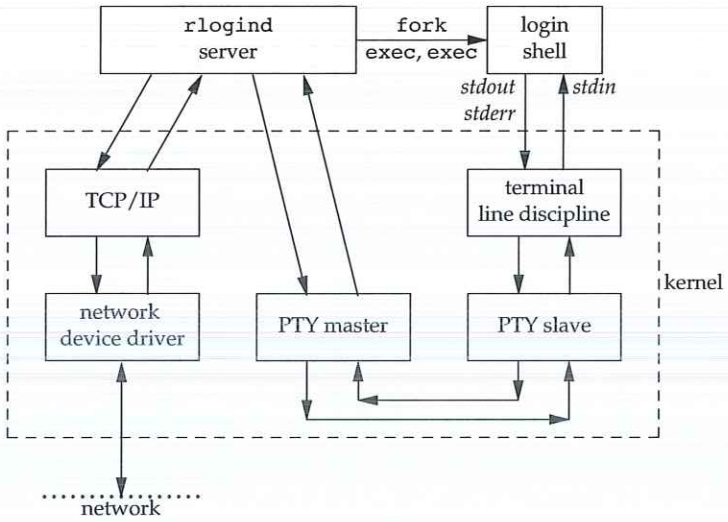

Figure 19.3 Arrangement of processes for `rlogind` server

A key point in Figure 19.3 is that the process driving the PTY master is normally reading and writing another I/O stream at the same time. In this example, the other I/O stream is the TCP/IP box. This implies that the process must be using some form of I/O multiplexing (Section 14.4), such as `select` or `poll`, or must be divided into two processes or threads.

Windowing System Terminal Emulation

Windowing systems typically provide a terminal emulator so that we can use a shell to run our programs from a familiar command-line environment. The terminal emulator acts as an intermediary between a shell and the window manager. Each shell executes in its own window. This arrangement (with two shells running in different windows) is shown in Figure 19.4.

The shell runs with its standard input, standard output, and standard error attached to the slave side of the PTY. The terminal emulator program opens the master side of the PTY. Besides acting as an interface to the windowing subsystem, the terminal emulator is responsible for emulating a particular type of terminal, which means it needs to respond to the escape codes associated with the type of device it is emulating. These codes are listed in the `termcap` and `terminfo` databases.

When a user resizes a terminal emulator window, the window manager informs the terminal emulator. The terminal emulator issues a `TIOCSWINSZ` `ioctl` command on the master side of the PTY to set the window size for the slave side. If the new size differs from the current size, the kernel sends a `SIGWINCH` signal to the foreground

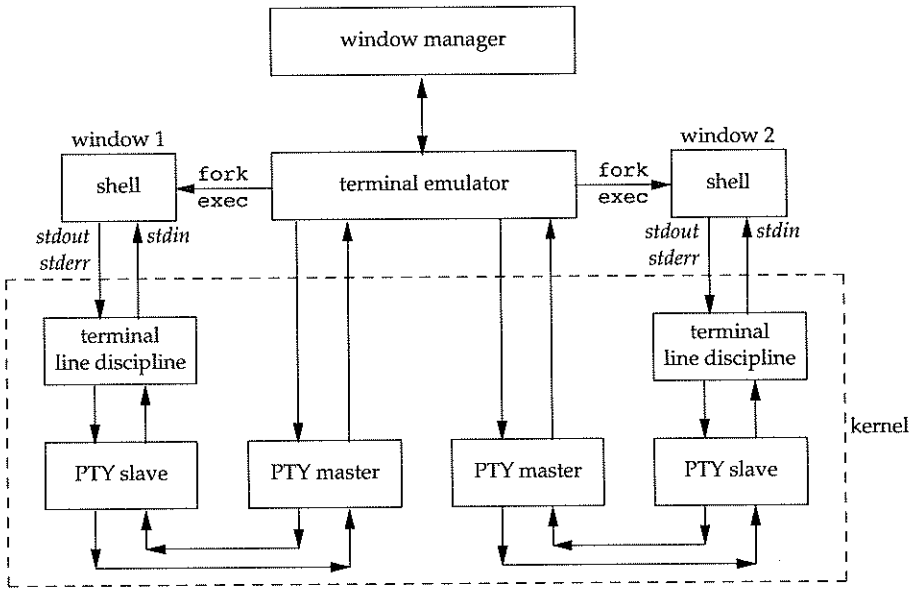

Figure 19.4 Arrangement of processes for windowing system

process group of the PTY slave. If the application needs to redraw the screen when the window is resized, it can catch the `SIGWINCH` signal, issue the `TIOCGWINSZ ioctl` command to get the new screen dimensions, and redraw the screen.

`script` Program

The `script`(1) program that is supplied with most UNIX systems makes a copy in a file of everything that is input and output during a terminal session. The program does this by placing itself between the terminal and a new invocation of our login shell. Figure 19.5 details the interactions involved in the `script` program. Here, we specifically show that the `script` program is normally run from a login shell, which then waits for `script` to terminate.

While `script` is running, everything output by the terminal line discipline above the PTY slave is copied to the script file (usually called `typescript`). Since our keystrokes are normally echoed by that line discipline module, the script file also contains our input. The script file won't contain any passwords that we enter, however, since passwords aren't echoed.

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

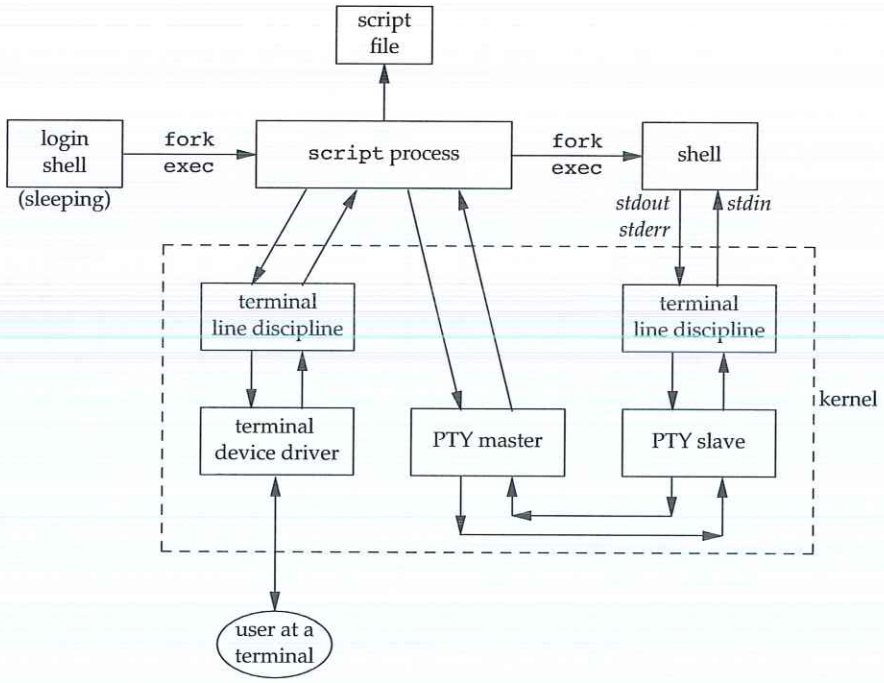

Figure 19.5 The `script` program

After developing the general `pty` program in Section 19.5, we'll see that a trivial shell script turns it into a version of the `script` program.

`expect` Program

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

Rather than modify all the interactive programs to support a batch mode of operation, a better solution is to provide a way to drive any interactive program from a script. The `expect` program [Libes 1990, 1991, 1994] provides a way to do this. It uses pseudo terminals to run other programs, similar to the `pty` program in Section 19.5. But `expect` also provides a programming language to examine the output of the program being run to make decisions about what to send the program as input. When an interactive program is being run from a script, we can't just copy everything from the script to the program, and vice versa. Instead, we have to send the program some input, look at its output, and decide what to send it next.

Running Coprocesses

In the coprocess example in Figure 15.19, we couldn't invoke a coprocess that used the standard I/O library for its input and output, because when we talked to the coprocess across a pipe, the standard I/O library fully buffered the standard input and standard output, leading to a deadlock. If the coprocess is a compiled program for which we don't have the source code, we can't add `fflush` statements to solve this problem. Figure 15.16 showed a process driving a coprocess. What we need to do is place a pseudo terminal between the two processes, as shown in Figure 19.6, to trick the coprocess into thinking that it is being driven from a terminal instead of from another process.

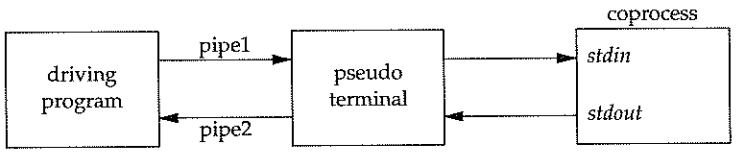

Figure 19.6 Driving a coprocess using a pseudo terminal

Now the standard input and standard output of the coprocess look like a terminal device, so the standard I/O library will set these two streams to be line buffered.

The parent can obtain a pseudo terminal between itself and the coprocess in two ways. (The parent in this case could be the program in Figure 15.18, which used two pipes to communicate with the coprocess.) One way is for the parent to call the `pty_fork` function directly (Section 19.4) instead of calling `fork`. Another is to `exec` the `pty` program (Section 19.5) with the coprocess as its argument. We'll look at these two solutions after showing the `pty` program.

Watching the Output of Long-Running Programs

If we have a program that runs for a long time, we can easily run it in the background using any of the standard shells. Unfortunately, if we redirect its standard output to a file, and if it doesn't generate much output, we can't easily monitor its progress, because the standard I/O library will fully buffer its standard output. All that we'll see are blocks of output written by the standard I/O library to the output file, possibly in chunks as large as 8,192 bytes.

If we have the source code, we can insert calls to `fflush` to force the standard I/O buffers to be flushed at select points or change the buffering mode to line buffered using `setvbuf`. If we don't have the source code, however, we can run the program under the `pty` program, making its standard I/O library think that its standard output is a terminal. Figure 19.7 shows this arrangement, where we have called the slow output program `slowout`. The `fork/exec` arrow from the login shell to the `pty` process is shown as a dashed arrow to emphasize that the `pty` process is running as a background job.

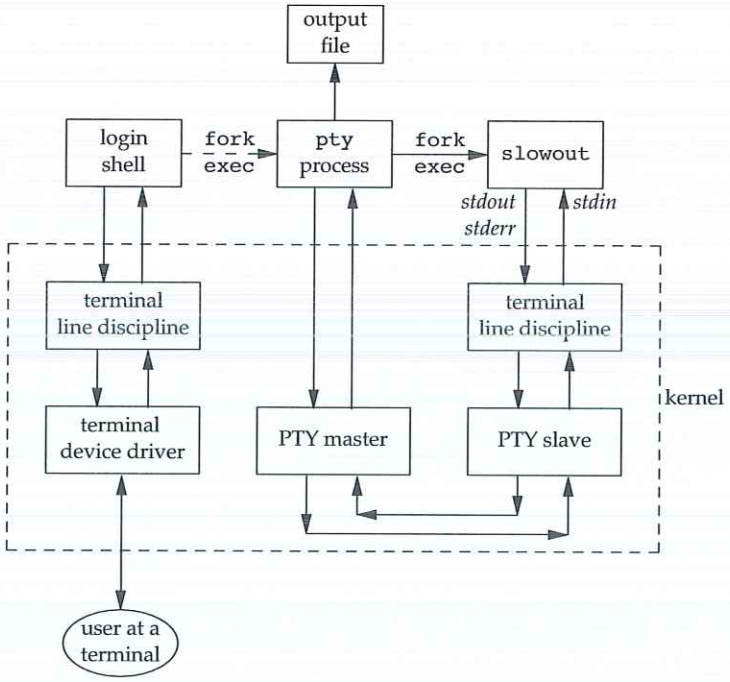

Figure 19.7 Running a slow output program using a pseudo terminal

19.3 Opening Pseudo-Terminal Devices

PTYs act like physical terminal devices so that applications are unaware of which type of device they are using. However, applications don't need to set the `O_TTY_INIT` flag when opening PTY device files. The Single UNIX Specification already requires that implementations initialize the slave side of a PTY device when it is first opened so that any nonstandard `termios` flags needed for the device to operate as expected are set. This requirement is intended to allow the PTY device to operate properly with POSIX-conforming applications that call `tcgetattr` and `tcsetattr`.

The way we open a PTY device differs among platforms. The Single UNIX Specification includes several functions as part of the XSI option in an attempt to unify the methods. These extensions are based on the functions originally provided to manage STREAMS-based PTYs in System V Release 4. The `posix_openpt` function is provided as a portable way to open an available PTY master device.

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

int posix_openpt(int oflag);
```

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

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

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

Implementations commonly set the group ownership of the slave PTY device to group `tty`. Programs that need permission to write to all active terminals on the system are set-group-ID to the group `tty`. Examples of such programs are `wall`(1) and `write`(1). Because the group write permission is enabled on slave PTY devices, these programs can write to them.

```
#include <stdlib.h>

int grantpt(int fd);

int unlockpt(int fd);
```

Both return: 0 on success, −1 on error

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

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

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

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

```
#include <stdlib.h>

char *ptsname(int fd);
```

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

Figure 19.8 summarizes the pseudo terminal functions in the Single UNIX Specification and indicates which functions are supported by the platforms discussed in this text.

On FreeBSD, `grantpt` and `unlockpt` do nothing other than argument validation; the PTYs are created dynamically with the correct permissions. Note that FreeBSD defines the `O_NOCTTY` flag only for compatibility with applications that call `posix_openpt`. FreeBSD does not allocate a controlling terminal as a side effect of opening a terminal device, so the `O_NOCTTY` flag has no effect.

Function	Description	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`grantpt`	change permissions of slave PTY device	•	•	•	•	•
`posix_openpt`	open a master PTY device	•	•	•	•	•
`ptsname`	return name of slave PTY device	•	•	•	•	•
`unlockpt`	allow slave PTY device to be opened	•	•	•	•	•

Figure 19.8 XSI pseudo terminal functions

The Single UNIX Specification has improved portability in this area, but differences remain. We provide two functions that handle all the details: `ptym_open` to open the next available PTY master device and `ptys_open` to open the corresponding slave device.

```
#include "apue.h"

int ptym_open(char *pts_name, int pts_namesz);
```

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

```
int ptys_open(char *pts_name);
```

Returns: file descriptor of PTY slave if OK, –1 on error

Normally, we don't call these two functions directly; instead, the function `pty_fork` (Section 19.4) calls them and also `fork`s a child process.

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

The reason for providing two functions to open the two devices will become obvious when we show the `pty_fork` function. Normally, a process calls `ptym_open` to open the master and obtain the name of the slave. The process then `fork`s, and the child calls `ptys_open` to open the slave after calling `setsid` to establish a new session. This is how the slave becomes the controlling terminal for the child.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#if defined(SOLARIS)
#include <stropts.h>
#endif
```

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

    if ((fdm = posix_openpt(O_RDWR)) < 0)
        return(-1);
    if (grantpt(fdm) < 0)       /* grant access to slave */
        goto errout;
    if (unlockpt(fdm) < 0)      /* clear slave's lock flag */
        goto errout;
    if ((ptr = ptsname(fdm)) == NULL)   /* get slave's name */
        goto errout;

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

int
ptys_open(char *pts_name)
{
    int fds;
#if defined(SOLARIS)
    int err, setup;
#endif

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-1);

#if defined(SOLARIS)
    /*
     * Check if stream is already set up by autopush facility.
     */
    if ((setup = ioctl(fds, I_FIND, "ldterm")) < 0)
        goto errout;

    if (setup == 0) {
        if (ioctl(fds, I_PUSH, "ptem") < 0)
            goto errout;
        if (ioctl(fds, I_PUSH, "ldterm") < 0)
            goto errout;
```

```
        if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
errout:
            err = errno;
            close(fds);
            errno = err;
            return(-1);
        }
    }
#endif
    return(fds);
}
```

Figure 19.9 Pseudo-terminal open functions

The `ptym_open` function uses the XSI PTY functions to find and open an unused PTY master device and initialize the corresponding PTY slave device. The `ptys_open` function opens the slave PTY device. On a Solaris system, however, we might need to take additional steps before the slave PTY device will behave like a terminal.

On Solaris, after opening the slave device, we might need to push three STREAMS modules onto the slave's stream. Together, the pseudo terminal emulation module (`ptem`) and the terminal line discipline module (`ldterm`) act like a real terminal. The `ttcompat` module provides compatibility for older V7, 4BSD, and Xenix `ioctl` calls. It's an optional module, but since it's automatically pushed for network logins, we push it onto the slave's stream.

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

Linux, Mac OS X, and Solaris follow the historical System V behavior: if the caller is a session leader that does not already have a controlling terminal, the call to `open` allocates the PTY slave as the controlling terminal. If we didn't want this to happen, we could specify the `O_NOCTTY` flag for `open`. However, on FreeBSD, the `open` of the slave PTY does not have the side effect of allocating the device as the controlling terminal. In the next section, we'll see how to allocate the controlling terminal when running on FreeBSD.

19.4 `pty_fork` Function

We now use the two functions from the previous section, `ptym_open` and `ptys_open`, to write a new function that we call `pty_fork`. This new function combines the opening of the master and the slave with a call to `fork`, establishing the child as a session leader with a controlling terminal.

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

pid_t pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
               const struct termios *slave_termios,
               const struct winsize *slave_winsize);
```

Returns: 0 in child, process ID of child in parent, –1 on error

The file descriptor of the PTY master is returned through the *ptrfdm* pointer.

If *slave_name* is non-null, the name of the slave device is stored at that location. The caller is responsible for allocating the storage pointed to by this argument.

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

Figure 19.10 shows the code for this function. It works on all four platforms described in this text, calling the `ptym_open` and `ptys_open` functions.

After opening the PTY master, `fork` is called. As we mentioned before, we want to wait to call `ptys_open` until in the child and after calling `setsid` to establish a new session. When it calls `setsid`, the child is not a process group leader, so the three steps listed in Section 9.5 occur: (a) a new session is created with the child as the session leader, (b) a new process group is created for the child, and (c) the child loses any association it might have had with its previous controlling terminal. Under Linux, Mac OS X, and Solaris, the slave becomes the controlling terminal of this new session when `ptys_open` is called. Under FreeBSD, we have to use the `TIOCSCTTY ioctl` command to allocate the controlling terminal. (Recall Figure 9.8—the other three platforms also support `TIOCSCTTY`, but we need to call it only on FreeBSD.)

The two structures `termios` and `winsize` are then initialized in the child. Finally, the slave file descriptor is duplicated onto standard input, standard output, and standard error in the child. This means that whatever process the caller `exec`s from the child will have these three descriptors connected to the slave PTY (its controlling terminal).

After the call to `fork`, the parent just returns the PTY master descriptor and the process ID of the child. In the next section, we use the `pty_fork` function in the `pty` program.

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

pid_t
pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
         const struct termios *slave_termios,
         const struct winsize *slave_winsize)
{
    int     fdm, fds;
```

```
    pid_t   pid;
    char    pts_name[20];

    if ((fdm = ptym_open(pts_name, sizeof(pts_name))) < 0)
        err_sys("can't open master pty: %s, error %d", pts_name, fdm);

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

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

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

#if defined(BSD)
        /*
         * TIOCSCTTY is the BSD way to acquire a controlling terminal.
         */
        if (ioctl(fds, TIOCSCTTY, (char *)0) < 0)
            err_sys("TIOCSCTTY error");
#endif
        /*
         * Set slave's termios and window size.
         */
        if (slave_termios != NULL) {
            if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                err_sys("tcsetattr error on slave pty");
        }
        if (slave_winsize != NULL) {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
                err_sys("TIOCSWINSZ error on slave pty");
        }

        /*
         * Slave becomes stdin/stdout/stderr of child.
         */
        if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
```

```
        if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
            err_sys("dup2 error to stderr");
        if (fds != STDIN_FILENO && fds != STDOUT_FILENO &&
          fds != STDERR_FILENO)
            close(fds);
        return(0);      /* child returns 0 just like fork() */
    } else {                    /* parent */
        *ptrfdm = fdm;  /* return fd of master */
        return(pid);    /* parent returns pid of child */
    }
}
```

Figure 19.10 The `pty_fork` function

19.5 `pty` Program

The goal in writing the `pty` program is to be able to type

```
pty prog arg1 arg2
```

instead of

```
prog arg1 arg2
```

When we use `pty` to execute another program, that program is executed in a session of its own, connected to a pseudo terminal.

Let's look at the source code for the `pty` program. The first file (Figure 19.11) contains the `main` function. It calls the `pty_fork` function from the previous section.

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

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

static void set_noecho(int);    /* at the end of this file */
void        do_driver(char *);  /* in the file driver.c */
void        loop(int, int);     /* in the file loop.c */

int
main(int argc, char *argv[])
{
    int             fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t           pid;
    char            *driver;
    char            slave_name[20];
```

```
    struct termios  orig_termios;
    struct winsize  size;

    interactive = isatty(STDIN_FILENO);
    ignoreeof = 0;
    noecho = 0;
    verbose = 0;
    driver = NULL;

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ((c = getopt(argc, argv, OPTSTR)) != EOF) {
        switch (c) {
        case 'd':       /* driver for stdin/stdout */
            driver = optarg;
            break;

        case 'e':       /* noecho for slave pty's line discipline */
            noecho = 1;
            break;

        case 'i':       /* ignore EOF on standard input */
            ignoreeof = 1;
            break;

        case 'n':       /* not interactive */
            interactive = 0;
            break;

        case 'v':       /* verbose */
            verbose = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }
    if (optind >= argc)
        err_quit("usage: pty [ -d driver -einv ] program [ arg ... ]");

    if (interactive) {  /* fetch current termios and window size */
        if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
            err_sys("tcgetattr error on stdin");
        if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
            err_sys("TIOCGWINSZ error");
        pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
          &orig_termios, &size);
    } else {
        pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
          NULL, NULL);
    }

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

```
    } else if (pid == 0) {      /* child */
        if (noecho)
            set_noecho(STDIN_FILENO);   /* stdin is slave pty */

        if (execvp(argv[optind], &argv[optind]) < 0)
            err_sys("can't execute: %s", argv[optind]);
    }

    if (verbose) {
        fprintf(stderr, "slave name = %s\n", slave_name);
        if (driver != NULL)
            fprintf(stderr, "driver = %s\n", driver);
    }

    if (interactive && driver == NULL) {
        if (tty_raw(STDIN_FILENO) < 0)  /* user's tty to raw mode */
            err_sys("tty_raw error");
        if (atexit(tty_atexit) < 0)     /* reset user's tty on exit */
            err_sys("atexit error");
    }

    if (driver)
        do_driver(driver);  /* changes our stdin/stdout */

    loop(fdm, ignoreeof);   /* copies stdin -> ptym, ptym -> stdout */

    exit(0);
}

static void
set_noecho(int fd)      /* turn off echo (for slave pty) */
{
    struct termios  stermios;

    if (tcgetattr(fd, &stermios) < 0)
        err_sys("tcgetattr error");

    stermios.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);

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

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

Figure 19.11 The `main` function for the `pty` program

In the next section, we'll look at the various command-line options when we examine different uses of the `pty` program. The `getopt` function helps us parse command-line arguments in a consistent manner. To enforce POSIX behavior on Linux systems, we set the first character of the option string to a plus sign.

Before calling `pty_fork`, we fetch the current values for the `termios` and `winsize` structures, passing these as arguments to `pty_fork`. This way, the PTY slave assumes the same initial state as the current terminal.

After returning from `pty_fork`, the child optionally turns off echoing for the slave PTY and then calls `execvp` to execute the program specified on the command line. All remaining command-line arguments are passed as arguments to this program.

The parent optionally sets the user's terminal to raw mode. In this case, the parent also sets an exit handler to reset the terminal state when `exit` is called. We describe the `do_driver` function in the next section.

The parent then calls the function `loop` (Figure 19.12), which copies everything received from the standard input to the PTY master and everything from the PTY master to standard output. For variety, we have coded it in two processes this time, although a single process using `select`, `poll`, or multiple threads would also work.

```
#include "apue.h"

#define BUFFSIZE    512

static void sig_term(int);
static volatile sig_atomic_t    sigcaught;  /* set by signal handler */

void
loop(int ptym, int ignoreeof)
{
    pid_t   child;
    int     nread;
    char    buf[BUFFSIZE];

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

        /*
         * We always terminate when we encounter an EOF on stdin,
         * but we notify the parent only if ignoreeof is 0.
         */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM);   /* notify parent */
        exit(0);    /* and terminate; child can't return */
    }

    /*
     * Parent copies ptym to stdout.
     */
```

```
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
        err_sys("signal_intr error for SIGTERM");

    for ( ; ; ) {
        if ((nread = read(ptym, buf, BUFFSIZE)) <= 0)
            break;      /* signal caught, error, or EOF */
        if (writen(STDOUT_FILENO, buf, nread) != nread)
            err_sys("writen error to stdout");
    }

    /*
     * There are three ways to get here: sig_term() below caught the
     * SIGTERM from the child, we read an EOF on the pty master (which
     * means we have to signal the child to stop), or an error.
     */
    if (sigcaught == 0) /* tell child if it didn't send us the signal */
        kill(child, SIGTERM);

    /*
     * Parent returns to caller.
     */
}

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

Figure 19.12 The loop function

Note that because we use two processes, one has to notify the other when it terminates. We use the SIGTERM signal for this notification.

19.6 Using the pty Program

We'll now look at various examples with the pty program, seeing the need for the command-line options.

If our shell is the Korn shell, we can execute the command

```
pty ksh
```

and get a brand-new invocation of the shell, running under a pseudo terminal.

If the file ttyname is the program we showed in Figure 18.16, we can run the pty program as follows:

```
$ who
sar  console  May 19 16:47
sar  ttys000  May 19 16:47
sar  ttys001  May 19 16:48
sar  ttys002  May 19 16:48
sar  ttys003  May 19 16:49
sar  ttys004  May 19 16:49          ttys004 is the highest PTY currently in use
$ pty ttyname                       run program in Figure 18.16 from PTY
fd 0: /dev/ttys005                  ttys005 is the next available PTY
fd 1: /dev/ttys005
fd 2: /dev/ttys005
```

utmp File

In Section 6.8, we described the `utmp` file that records all users currently logged in to a UNIX system. The question is whether a user running a program on a pseudo terminal is considered logged in. In the case of remote logins, with `telnetd` and `rlogind`, obviously an entry should be made in the `utmp` file for the user logged in on the pseudo terminal. There is little agreement, however, whether users running a shell on a pseudo terminal from a window system or from a program, such as `script`, should have entries made in the `utmp` file. Some systems record these; others don't. If a system doesn't record these entries in the `utmp` file, the `who`(1) program normally won't show the corresponding pseudo terminals as being used.

Unless the `utmp` file has other-write permission enabled (which is considered to be a security hole), random programs that use pseudo terminals won't be able to write to this file.

Job Control Interaction

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

```
pty ksh
```

runs the Korn shell under `pty`. We can run programs under this new shell and use job control just as we do with our login shell. But if we run an interactive program other than a job-control shell under `pty`, as in

```
pty cat
```

everything is fine until we type the job-control suspend character. At that point, the job-control character is echoed as `^Z` and is ignored. Under earlier BSD-based systems, the `cat` process terminates, the `pty` process terminates, and we're back to our original shell. To understand what's going on here, we need to examine all the processes involved, their process groups, and sessions. Figure 19.13 shows the arrangement when `pty cat` is running.

When we type the suspend character (Control-Z), it is recognized by the line discipline module beneath the `cat` process, since `pty` puts the terminal (beneath the `pty` parent) into raw mode. But the kernel won't stop the `cat` process, because it

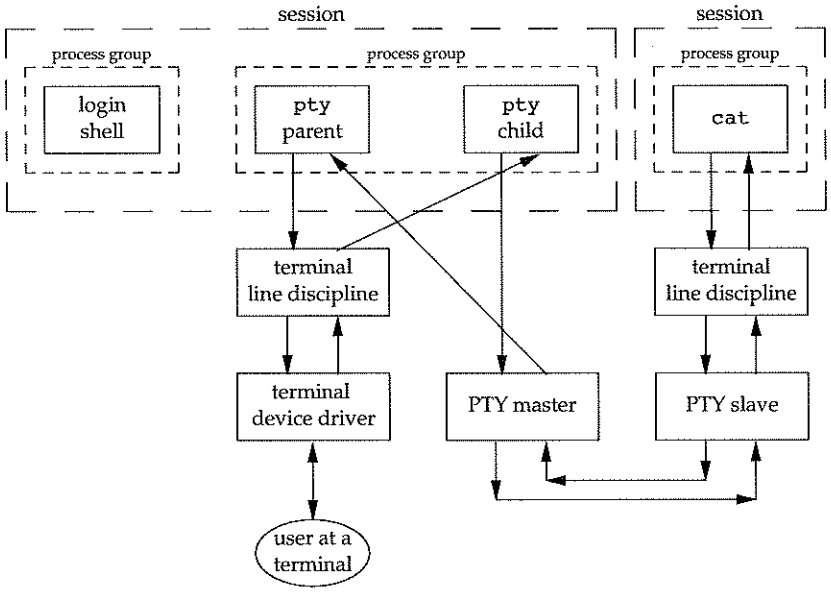

Figure 19.13 Process groups and sessions for `pty cat`

belongs to an orphaned process group (Section 9.10). The parent of `cat` is the `pty` parent, and it belongs to another session.

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

When we use `pty` to run a job-control shell, the jobs invoked by this new shell are never members of an orphaned process group, because the job-control shell always belongs to the same session. In that case, the Control-Z that we type is sent to the process invoked by the shell, not to the shell itself.

The only way to avoid this inability of the process invoked by `pty` to handle job-control signals is to add yet another command-line flag to `pty`, telling it to recognize the job control suspend character itself (in the `pty` child) instead of letting the character get all the way through to the other line discipline.

Watching the Output of Long-Running Programs

Another example of job control interaction with the `pty` program is found in the configuration illustrated in Figure 19.7. If we run the program that generates output slowly as

```
pty slowout > file.out &
```

the `pty` process is stopped immediately when the child tries to read from its standard input (the terminal). The reason is that the job is a background job and gets job-control stopped when it tries to access the terminal. If we redirect standard input so that `pty` doesn't try to read from the terminal, as in

```
pty slowout < /dev/null > file.out &
```

the `pty` program stops immediately because it reads an end of file on its standard input and terminates. The solution for this problem is the `-i` option, which says to ignore an end of file on the standard input:

```
pty -i slowout < /dev/null > file.out &
```

This flag causes the `pty` child in Figure 19.12 to exit when the end of file is encountered, but the child doesn't tell the parent to terminate. Instead, the parent continues copying the PTY slave output to standard output (the file `file.out` in the example).

`script` Program

Using the `pty` program, we can implement the `script`(1) program as the following shell script:

```
#!/bin/sh
pty "${SHELL:-/bin/sh}" | tee typescript
```

Once we run this shell script, we can execute the `ps` command to see all the process relationships. Figure 19.14 details these relationships.

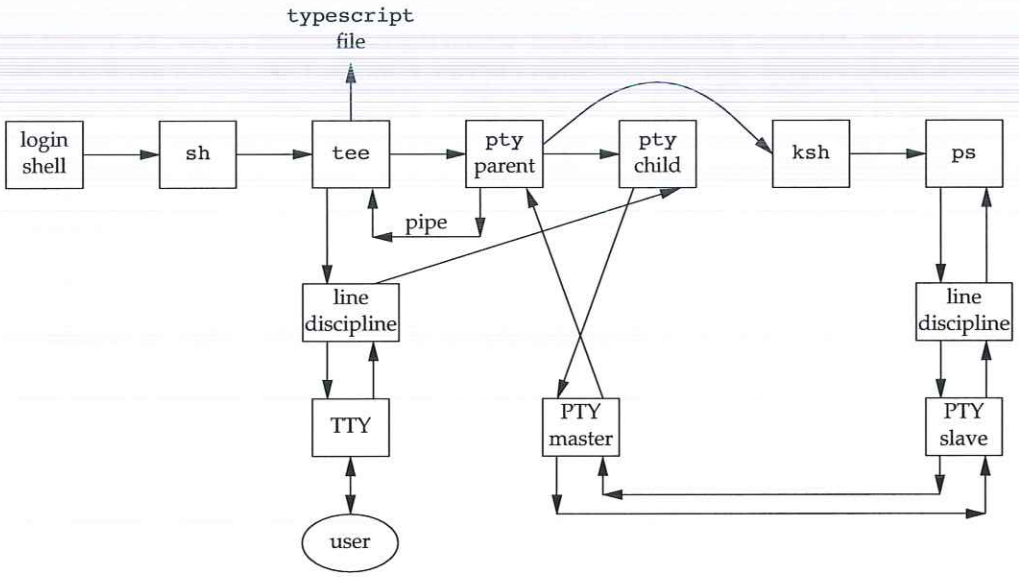

Figure 19.14 Arrangement of processes for `script` shell script

In this example, we assume that the SHELL variable is the Korn shell (probably /bin/ksh). As we mentioned earlier, script copies only what is output by the new shell (and any processes that it invokes), but since the line discipline module above the PTY slave normally has echo enabled, most of what we type is also written to the typescript file.

Running Coprocesses

In Figure 15.18, the coprocess couldn't use the standard I/O functions, because standard input and standard output do not refer to a terminal, so the standard I/O functions treat them as fully buffered. If we run the coprocess under pty by replacing the line

```
if (execl("./add2", "add2", (char *)0) < 0)
```

with

```
if (execl("./pty", "pty", "-e", "add2", (char *)0) < 0)
```

the program now works, even if the coprocess uses standard I/O.

Figure 19.15 shows the arrangement of processes when we run the coprocess with a pseudo terminal as its input and output. It is an expansion of Figure 19.6, showing all the process connections and data flow. The box labeled "driving program" is the program from Figure 15.18, with the execl changed as described previously.

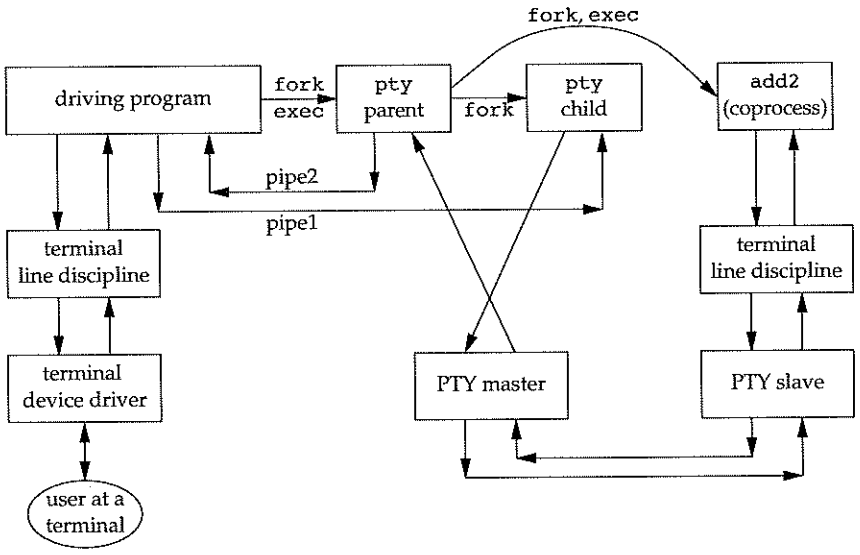

Figure 19.15 Running a coprocess with a pseudo terminal as its input and output

This example shows the need for the -e (no echo) option for the pty program. The pty program is not running interactively, because its standard input is not connected to a terminal. In Figure 19.11, the interactive flag defaults to false, since the call to

`isatty` returns false. This means that the line discipline above the actual terminal remains in canonical mode with echo enabled. By specifying the `-e` option, we turn off echo in the line discipline module above the PTY slave. If we don't do this, everything we type is echoed twice—by both line discipline modules.

We also have the `-e` option turn off the `ONLCR` flag in the `termios` structure to prevent all the output from the coprocess from being terminated with a carriage return and a newline.

Testing this example on different systems revealed another problem that we alluded to in Section 14.7 when we described the `readn` and `writen` functions. The amount of data returned by a `read`, when the descriptor refers to something other than an ordinary disk file, can differ between implementations. This coprocess example using `pty` gave unexpected results that were tracked down to the `read` function on the pipe in the program from Figure 15.18, which returned less than a line. The solution was to not use the program shown in Figure 15.18, but rather to use the version of this program from Exercise 15.5 that was modified to use the standard I/O library, with the standard I/O streams for both pipes set to line buffering. With this approach, the `fgets` function does as many `read`s as required to obtain a complete line. The `while` loop in Figure 15.18 assumes that each line sent to the coprocess causes one line to be returned.

Driving Interactive Programs Noninteractively

Although it's tempting to think that `pty` can run any coprocess, even a coprocess that is interactive, it doesn't work. The problem is that `pty` just copies everything on its standard input to the PTY and everything from the PTY to its standard output, never looking at what it sends or what it gets back.

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

```
pty telnet 192.168.1.3
```

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

```
sar
passwd
uptime
exit
```

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

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

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

When we run the telnet program interactively, we wait for the remote host to prompt for a password before we type it, but the pty program doesn't know to do this. This is why it takes a more sophisticated program than pty, such as expect, to drive an interactive program from a script file.

Even running pty from the program in Figure 15.18, as we showed earlier, doesn't help, because the program in Figure 15.18 assumes that each line it writes to the pipe generates exactly one line on the other pipe. With an interactive program, one line of input may generate many lines of output. Furthermore, the program in Figure 15.18 always sent a line to the coprocess before reading from it. This strategy won't work when we want to read from the coprocess before sending it anything.

There are a few ways to proceed from here to be able to drive an interactive program from a script. We could add a command language and interpreter to pty, but a reasonable command language would probably be ten times larger than the pty program. Another option is to take a command language and use the pty_fork function to invoke interactive programs. This is what the expect program does.

We'll take a different path here and just provide an option (-d) to allow pty to be connected to a driver process for its input and output. The standard output of the driver is pty's standard input, and vice versa. This is similar to a coprocess, but on "the other side" of pty. The resulting arrangement of processes is almost identical to Figure 19.15, but in the current scenario, pty does the fork and exec of the driver process. Also, instead of two half-duplex pipes, we'll use a single bidirectional pipe between pty and the driver process.

Figure 19.16 shows the source for the do_driver function, which is called by the main function of pty (Figure 19.11) when the -d option is specified.

```
#include "apue.h"

void
do_driver(char *driver)
{
    pid_t   child;
    int     pipe[2];

    /*
     * Create a full-duplex pipe to communicate with the driver.
     */
    if (fd_pipe(pipe) < 0)
        err_sys("can't create stream pipe");

    if ((child = fork()) < 0) {
        err_sys("fork error");
    } else if (child == 0) {        /* child */
        close(pipe[1]);

        /* stdin for driver */
        if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");

        /* stdout for driver */
        if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
```

```
            err_sys("dup2 error to stdout");
        if (pipe[0] != STDIN_FILENO && pipe[0] != STDOUT_FILENO)
            close(pipe[0]);

        /* leave stderr for driver alone */
        execlp(driver, driver, (char *)0);
        err_sys("execlp error for: %s", driver);
    }

    close(pipe[0]);     /* parent */
    if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
        err_sys("dup2 error to stdin");
    if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("dup2 error to stdout");
    if (pipe[1] != STDIN_FILENO && pipe[1] != STDOUT_FILENO)
        close(pipe[1]);

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

Figure 19.16 The `do_driver` function for the `pty` program

By writing our own driver program that is invoked by `pty`, we can drive interactive programs in any way desired. Even though it has its standard input and standard output connected to `pty`, the driver process can still interact with the user by reading and writing `/dev/tty`. This solution still isn't as general as the `expect` program, but it provides a useful option to `pty` for fewer than 50 lines of code.

19.7 Advanced Features

Pseudo terminals have some additional capabilities that we briefly mention here. These capabilities are further documented in Sun Microsystems [2005] and the BSD `pts`(4) manual page.

Packet Mode

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

The details of packet mode differ between Solaris and the other platforms. Under Solaris, the process reading the PTY master has to call `getmsg` to fetch the messages from the stream head, because the `pckt` module converts certain events into nondata STREAMS messages. With the other platforms, each `read` from the PTY master returns a status byte followed by optional data.

Regardless of the implementation details, the purpose of packet mode is to inform the process reading the PTY master when the following events occur at the line discipline module above the PTY slave: when the read queue is flushed, when the write queue is flushed, when output is stopped (e.g., Control-S), when output is restarted, when XON/XOFF flow control is enabled after being disabled, and when XON/XOFF flow control is disabled after being enabled. These events are used, for example, by the `rlogin` client and the `rlogind` server.

Remote Mode

A PTY master can set the PTY slave to remote mode by issuing the `TIOCREMOTE ioctl` command. Although Mac OS X 10.6.8 and Solaris 10 use the same command to enable and disable this feature, under Solaris the third argument to `ioctl` is an integer, whereas with Mac OS X, it is a pointer to an integer. (FreeBSD 8.0 and Linux 3.2.0 don't support this command.)

When it sets this mode, the PTY master is telling the PTY slave's line discipline not to perform any processing of the data that it receives from the PTY master, regardless of the canonical/noncanonical flag in the slave's `termios` structure. Remote mode is intended for an application, such as a window manager, that does its own line editing.

Window Size Changes

The process above the PTY master can issue the `TIOCSWINSZ ioctl` command to set the window size of the slave. If the new size differs from the current size, a `SIGWINCH` signal is sent to the foreground process group of the PTY slave.

Signal Generation

The process reading and writing the PTY master can send signals to the process group of the PTY slave. Under Solaris 10, this is done using the `TIOCSIGNAL ioctl` command. With FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, the `ioctl` command is `TIOCSIG`. In both cases, the third argument is set to the signal number.

19.8 Summary

We started this chapter with an overview of how to use pseudo terminals and a look at some use cases. We continued by examining the code required to set up a pseudo terminal under the four platforms discussed in this text. We then used this code to provide the generic `pty_fork` function that can be used by many different applications. We used this function as the basis for a small program (`pty`), which we then used to explore many of the properties of pseudo terminals.

Pseudo terminals are used daily on most UNIX systems to provide network logins. We've examined other uses for pseudo terminals as well, ranging from the `script` program to driving interactive programs from a batch script.

Exercises

19.1 When we remotely log in to a BSD system using either `telnet` or `rlogin`, the ownership of the PTY slave and its permissions are set, as we described in Section 19.3. How does this happen?

19.2 Use the `pty` program to determine the values used by your system to initialize a slave PTY's `termios` structure and `winsize` structure.

19.3 Recode the `loop` function (Figure 19.12) as a single process using either `select` or `poll`.

19.4 In the child process after `pty_fork` returns, standard input, standard output, and standard error are all open for read–write. Can you change standard input to be read-only and the other two to be write-only?

19.5 In Figure 19.13, identify which process groups are in the foreground and which are in the background, and identify the session leaders.

19.6 In Figure 19.13, in what order do the processes terminate when we type the end-of-file character? Verify this with process accounting, if possible.

19.7 The `script`(1) program normally adds to the beginning of the output file a line with the starting time, and to the end of the output file another line with the ending time. Add these features to the simple shell script that we showed.

19.8 Explain why the contents of the file `data` are output to the terminal in the following example, even though the program `ttyname` (Figure 18.16) only generates output and never reads its input.

```
$ cat data                      a file with two lines
hello,
world
$ pty -i < data ttyname         -i says ignore eof on stdin
hello,                          where did these two lines come from?
world
fd 0: /dev/ttys005              we expect these three lines from ttyname
fd 1: /dev/ttys005
fd 2: /dev/ttys005
```

19.9 Write a program that calls `pty_fork` and have the child `exec` another program that you will write. The new program that the child `exec`s must catch `SIGTERM` and `SIGWINCH`. When it catches a signal, the program should print that it did; for the latter signal, it should also print the terminal's window size. Then have the parent process send the `SIGTERM` signal to the process group of the PTY slave with the `ioctl` command we described in Section 19.7. Read back from the slave to verify that the signal was caught. Follow this with the parent setting the window size of the PTY slave, and then read back the slave's output again. Have the parent `exit` and determine whether the slave process also terminates; if so, how does it terminate?