64

PSEUDOTERMINALS

A *pseudoterminal* is a virtual device that provides an IPC channel. On one end of the channel is a program that expects to be connected to a terminal device. On the other end is a program that drives the terminal-oriented program by using the channel to send it input and read its output.

This chapter describes the use of pseudoterminals, showing how they are employed in applications such as terminal emulators, the *script(1)* program, and programs such as *ssh*, which provide network login services.

64.1 Overview

Figure 64-1 illustrates one of the problems that pseudoterminals help us solve: how can we enable a user on one host to operate a terminal-oriented program (e.g., *vi*) on another host connected via a network?

As shown in the diagram, by permitting communication over a network, sockets provide part of the machinery needed to solve this problem. However, we can't connect the standard input, output, and error of a terminal-oriented program directly to a socket. This is because a terminal-oriented program expects to be connected to a terminal—to be able to perform the terminal-oriented operations described in Chapters 34 and 62. Such operations include placing the terminal in noncanonical mode, turning echoing on and off, and setting the terminal foreground process group. If a program tries to perform these operations on a socket, then the relevant system calls will fail.

Furthermore, a terminal-oriented program expects a terminal driver to perform certain kinds of processing of its input and output. For example, in canonical mode, when the terminal driver sees the end-of-file character (normally *Control-D*) at the start of a line, it causes the next *read()* to return no data.

Finally, a terminal-oriented program must have a controlling terminal. This allows the program to obtain a file descriptor for the controlling terminal by opening `/dev/tty`, and also makes it possible to generate job-control and terminal-related signals (e.g., `SIGTSTP`, `SIGTTIN`, and `SIGINT`) for the program.

From this description, it should be clear that the definition of a terminal-oriented program is quite broad. It encompasses a wide range of programs that we would normally run in an interactive terminal session.

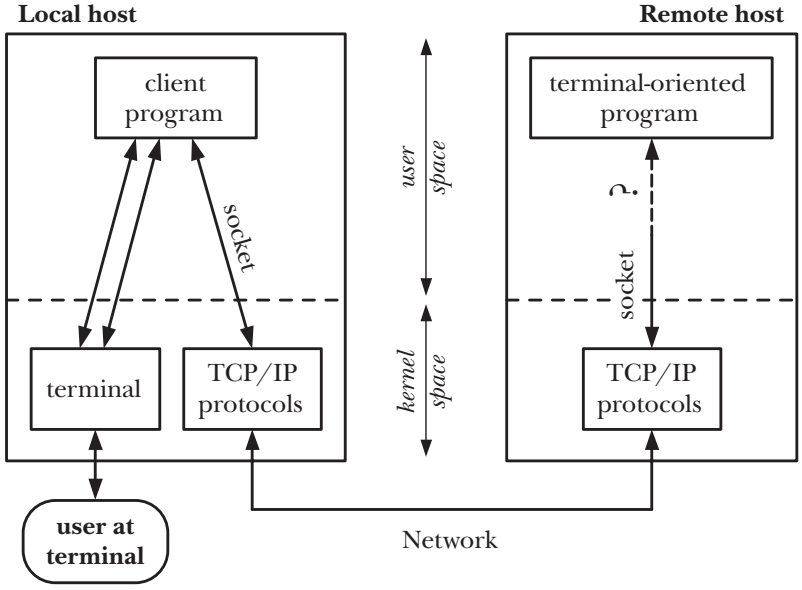

Figure 64-1: The problem: how to operate a terminal-oriented program over a network?

The pseudoterminal master and slave devices

A pseudoterminal provides the missing link for creating a network connection to a terminal-oriented program. A pseudoterminal is a pair of connected virtual devices: a *pseudoterminal master* and a *pseudoterminal slave*, sometimes jointly referred to as a *pseudoterminal pair*. A pseudoterminal pair provides an IPC channel somewhat like a bidirectional pipe—two processes can open the master and slave and then transfer data in either direction through the pseudoterminal.

The key point about a pseudoterminal is that the slave device appears just like a standard terminal. All of the operations that can be applied to a terminal device can also be applied to a pseudoterminal slave device. Some of these operations aren't meaningful for a pseudoterminal (e.g., setting the terminal line speed or parity), but that's okay, because the pseudoterminal slave silently ignores them.

How programs use pseudoterminals

Figure 64-2 shows how two programs typically employ a pseudoterminal. (The abbreviation *pty* in this diagram is a commonly used shorthand for *pseudoterminal*,

and we employ this abbreviation in various diagrams and function names in this chapter.) The standard input, output, and error of the terminal-oriented program are connected to the pseudoterminal slave, which is also the controlling terminal for the program. On the other side of the pseudoterminal, a driver program acts as a proxy for the user, supplying input to the terminal-oriented program and reading that program's output.

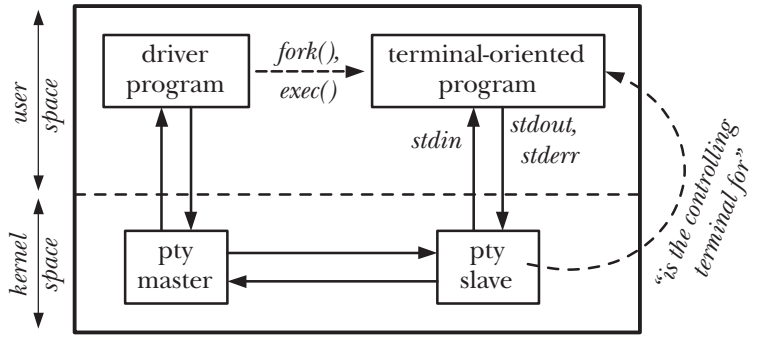

Figure 64-2: Two programs communicating via a pseudoterminal

Typically, the driver program is simultaneously reading from and writing to another I/O channel. It is acting as a relay, passing data in both directions between the pseudoterminal and another program. In order to do this, the driver program must simultaneously monitor input arriving from either direction. Typically, this is done using I/O multiplexing (*select()* or *poll()*), or using a pair of processes or threads to perform data transfer in each direction.

An application that uses a pseudoterminal typically does so as follows:

1. The driver program opens the pseudoterminal master device.
2. The driver program calls *fork()* to create a child process. The child performs the following steps:
   a) Call *setsid()* to start a new session, of which the child is the session leader (Section 34.3). This step also causes the child to lose its controlling terminal.
   b) Open the pseudoterminal slave device that corresponds to the master device. Since the child process is a session leader, and it doesn't have a controlling terminal, the pseudoterminal slave becomes the controlling terminal for the child process.
   c) Use *dup()* (or similar) to duplicate the file descriptor for the slave device on standard input, output, and error.
   d) Call *exec()* to start the terminal-oriented program that is to be connected to the pseudoterminal slave.

At this point, the two programs can now communicate via the pseudoterminal. Anything that the driver program writes to the master appears as input to the terminal-oriented program on the slave, and anything that the terminal-oriented program writes to the slave can be read by the driver program on the master. We consider further details of pseudoterminal I/O in Section 64.5.

> Pseudoterminals can also be used to connect an arbitrary pair of processes (i.e., not necessarily a parent and child). All that is required is that the process that opens the pseudoterminal master informs the other process of the name of the corresponding slave device, perhaps by writing that name to a file or by transmitting it using some other IPC mechanism. (When we use *fork()* in the manner described above, the child automatically inherits sufficient information from the parent to enable it to determine the name of the slave.)

So far, our discussion of the use of pseudoterminals has been abstract. Figure 64-3 shows a specific example: the use of a pseudoterminal by *ssh*, an application that allows a user to securely run a login session on a remote system connected via a network. (In effect, this diagram combines the information from Figure 64-1 and Figure 64-2.) On the remote host, the driver program for the pseudoterminal master is the *ssh* server (*sshd*), and the terminal-oriented program connected to the pseudoterminal slave is the login shell. The *ssh* server is the glue that connects the pseudoterminal via a socket to the *ssh* client. Once all of the details of logging in have been completed, the primary purpose of the *ssh* server and client is to relay characters in either direction between the user's terminal on the local host and the shell on the remote host.

> We omit describing many details of the *ssh* client and server. For example, these programs encrypt the data transmitted in either direction across the network. We show a single *ssh* server process on the remote host, but, in fact, the *ssh* server is a concurrent network server. It becomes a daemon and creates a passive TCP socket to listen for incoming connections from *ssh* clients. For each connection, the master *ssh* server forks a child process that handles all of the details for a single client login session. (We refer to this child process as the *ssh* server in Figure 64-3.) Aside from the details of pseudoterminal setup described above, the *ssh* server child authenticates the user, updates the login accounting files on the remote host (as described in Chapter 40), and then execs the login shell.

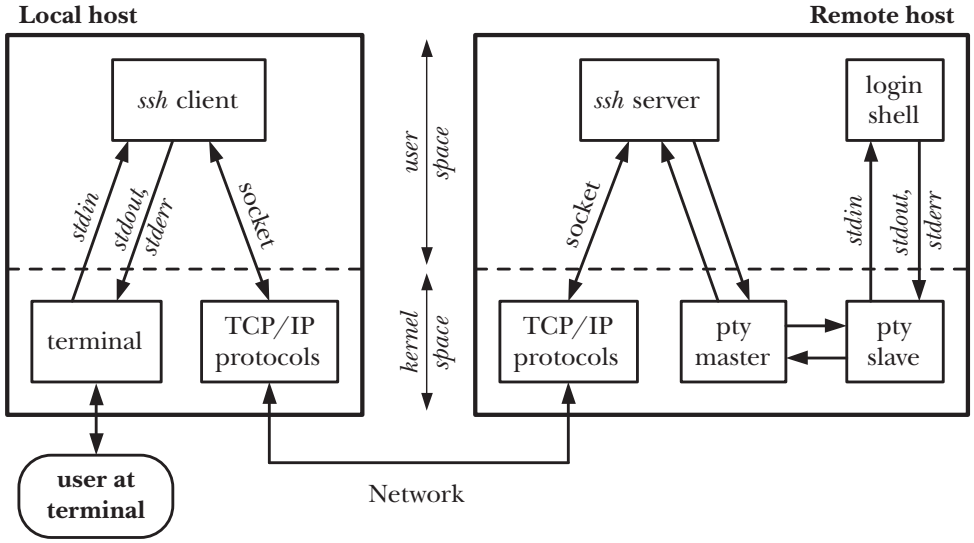

Figure 64-3: How *ssh* uses a pseudoterminal

In some cases, multiple processes may be connected to the slave side of the pseudoterminal. Our *ssh* example illustrates this point. The session leader for the slave is a shell, which creates process groups to execute the commands entered by the remote user. All of these processes have the pseudoterminal slave as their controlling terminal. As with a conventional terminal, one of these process groups can be the foreground process group for the pseudoterminal slave, and only this process group is allowed to read from the slave and (if the TOSTOP bit has been set) write to it.

Applications of pseudoterminals

Pseudoterminals are also used in many applications other than network services. Examples include the following:

- The *expect(1)* program uses a pseudoterminal to allow an interactive terminal-oriented program to be driven from a script file.
- Terminal emulators such as *xterm* employ pseudoterminals to provide the terminal-related functionality that goes with a terminal window.
- The *screen(1)* program uses pseudoterminals to multiplex a single physical terminal (or terminal window) between multiple processes (e.g., multiple shell sessions).
- Pseudoterminals are used in the *script(1)* program, which records all of the input and output that occurs during a shell session.
- Sometimes a pseudoterminal is useful to circumvent the default block buffering performed by the *stdio* functions when writing output to a disk file or pipe, as opposed to the line buffering used for terminal output. (We consider this point further in Exercise 64-7.)

System V (UNIX 98) and BSD pseudoterminals

BSD and System V provided different interfaces for finding and opening the two halves of a pseudoterminal pair. The BSD pseudoterminal implementation was historically the better known, since it was used with many sockets-based network applications. For compatibility reasons, many UNIX implementations eventually came to support both styles of pseudoterminals.

The System V interface is somewhat simpler to use than the BSD interface, and the SUSv3 specification of pseudoterminals is based on the System V interface. (A pseudoterminal specification first appeared in SUSv1.) For historical reasons, on Linux systems, this type of pseudoterminal is commonly referred to as a *UNIX 98* pseudoterminal, even though the UNIX 98 standard (i.e., SUSv2) required pseudoterminals to be STREAMS-based, and the Linux implementation of pseudoterminals is not. (SUSv3 doesn't require a STREAMS-based implementation.)

Early versions of Linux supported only BSD-style pseudoterminals, but, since kernel 2.2, Linux has supported both types of pseudoterminals. In this chapter, we focus on UNIX 98 pseudoterminals. We describe the differences for BSD pseudoterminals in Section 64.8.

64.2 UNIX 98 Pseudoterminals

Bit by bit, we'll work toward the development of a function, *ptyFork()*, that does most of the work to create the setup shown in Figure 64-2. We'll then use this function to implement the *script(1)* program. Before doing this though, we look at the various library functions used with UNIX 98 pseudoterminals:

- The *posix_openpt()* function opens an unused pseudoterminal master device, returning a file descriptor that is used to refer to the device in later calls.
- The *grantpt()* function changes the ownership and permissions of the slave device corresponding to a pseudoterminal master device.
- The *unlockpt()* function unlocks the slave device corresponding to a pseudoterminal master device, so that the slave device can be opened.
- The *ptsname()* function returns the name of the slave device corresponding to a pseudoterminal master device. The slave device can then be opened using *open()*.

64.2.1 Opening an Unused Master: *posix_openpt()*

The *posix_openpt()* function finds and opens an unused pseudoterminal master device, and returns a file descriptor that can later be used to refer to this device.

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int flags);
```

Returns file descriptor on success, or −1 on error

The *flags* argument is constructed by ORing zero or more of the following constants together:

`O_RDWR`
: Open the device for both reading and writing. Normally, we would always include this constant in *flags*.

`O_NOCTTY`
: Don't make this terminal the controlling terminal for the process. On Linux, a pseudoterminal master can't become a controlling terminal for a process, regardless of whether the `O_NOCTTY` flag is specified when calling *posix_openpt()*. (This makes sense because the pseudoterminal master isn't really a terminal; it is the other side of a terminal to which the slave is connected.) However, on some implementations, `O_NOCTTY` is required if we want to prevent a process from acquiring a controlling terminal as a consequence of opening a pseudoterminal master device.

Like *open()*, *posix_openpt()* uses the lowest available file descriptor to open the pseudoterminal master.

Calling *posix_openpt()* also results in the creation of a corresponding pseudoterminal slave device file in the `/dev/pts` directory. We say more about this file when we describe the *ptsname()* function below.

The *posix_openpt()* function is new in SUSv3, and was an invention of the POSIX committee. In the original System V pseudoterminal implementation, obtaining an available pseudoterminal master was accomplished by opening the *pseudoterminal master clone device*, `/dev/ptmx`. Opening this virtual device automatically locates and opens the next unused pseudoterminal master, and returns a file descriptor for it. This device is provided on Linux, where *posix_openpt()* is implemented as follows:

```
int
posix_openpt(int flags)
{
    return open("/dev/ptmx", flags);
}
```

Limits on the number of UNIX 98 pseudoterminals

Because each pseudoterminal pair in use consumes a small amount of nonswappable kernel memory, the kernel imposes a limit on the number of UNIX 98 pseudoterminal pairs on the system. In kernels up to 2.6.3, this limit is controlled by a kernel configuration option (`CONFIG_UNIX98_PTYS`). The default value for this option is 256, but we can change the limit to any value in the range 0 to 2048.

From Linux 2.6.4 onward, the `CONFIG_UNIX98_PTYS` kernel configuration option is discarded in favor of a more flexible approach. Instead, the limit on the number of pseudoterminals is defined by the value in the Linux-specific `/proc/sys/kernel/pty/max` file. The default value for this file is 4096, and it can be set to any value up to 1,048,576. A related read-only file, `/proc/sys/kernel/pty/nr`, shows how many UNIX 98 pseudoterminals are currently in use.

64.2.2 Changing Slave Ownership and Permissions: *grantpt()*

SUSv3 specifies the use of *grantpt()* to change the ownership and permissions of the slave device that corresponds to the pseudoterminal master referred to by the file descriptor *mfd*. On Linux, calling *grantpt()* is not actually necessary. However, the use of *grantpt()* is required on some implementations, and portable applications should call it after calling *posix_openpt()*.

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

int grantpt(int mfd);
```

Returns 0 on success, or −1 on error

On systems where *grantpt()* is required, this function creates a child process that executes a set-user-ID-*root* program. This program, usually called *pt_chown*, performs the following operations on the pseudoterminal slave device:

- change the ownership of the slave to be the same as the effective user ID of the calling process;
- change the group of the slave to *tty*; and

- change the permissions on the slave so that the owner has read and write permissions, and group has write permission.

The reason for changing the group of the terminal to *tty* and enabling group write permission is that the *wall(1)* and *write(1)* programs are set-group-ID programs owned by the *tty* group.

On Linux, a pseudoterminal slave is automatically configured in the above manner, which is why calling *grantpt()* isn't needed (but still should be done).

> Because it may create a child process, SUSv3 says that the behavior of *grantpt()* is unspecified if the calling program has installed a handler for `SIGCHLD`.

64.2.3 Unlocking the Slave: *unlockpt()*

The *unlockpt()* function removes an internal lock on the slave corresponding to the pseudoterminal master referred to by the file descriptor *mfd*. The purpose of this locking mechanism is to allow the calling process to perform whatever initialization is required for the pseudoterminal slave (e.g., calling *grantpt()*) before another process is allowed to open it.

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

int unlockpt(int mfd);
```

Returns 0 on success, or −1 on error

An attempt to open a pseudoterminal slave before it has been unlocked with *unlockpt()* fails with the error `EIO`.

64.2.4 Obtaining the Name of the Slave: *ptsname()*

The *ptsname()* function returns the name of the pseudoterminal slave corresponding to the pseudoterminal master referred to by the file descriptor *mfd*.

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

char *ptsname(int mfd);
```

Returns pointer to (possibly statically allocated) string on success, or `NULL` on error

On Linux (as on most implementations), *ptsname()* returns a name of the form `/dev/pts/`*nn*, where *nn* is replaced by a number that uniquely identifies this pseudoterminal slave.

The buffer used to return the slave name is normally statically allocated. It is thus overwritten by subsequent calls to *ptsname()*.

The GNU C library provides a reentrant analog of *ptsname()* in the form of *ptsname_r(mfd, strbuf, buflen)*. However, this function is nonstandard and is available on few other UNIX implementations. The `_GNU_SOURCE` feature test macro must be defined in order to obtain the declaration of *ptsname_r()* from `<stdlib.h>`.

Once we have unlocked the slave device with *unlockpt()*, we can open it using the traditional *open()* system call.

On System V derivatives that employ STREAMS, it may be necessary to perform some further steps (pushing STREAMS modules onto the slave device after opening it). An example of how to perform these steps can be found in [Stevens & Rago, 2005].

64.3 Opening a Master: *ptyMasterOpen()*

We now present a function, *ptyMasterOpen()*, that employs the functions described in the previous sections to open a pseudoterminal master and obtain the name of the corresponding pseudoterminal slave. Our reasons for providing such a function are twofold:

- Most programs perform these steps in exactly the same way, so it is convenient to encapsulate them in a single function.
- Our *ptyMasterOpen()* function hides all of the details that are specific to UNIX 98 pseudoterminals. In Section 64.8, we present a reimplementation of this function that uses BSD-style pseudoterminals. All of the code that we present in the remainder of this chapter can work with either of these implementations.

```
#include "pty_master_open.h"

int ptyMasterOpen(char *slaveName, size_t snLen);
```

Returns file descriptor on success, or −1 on error

The *ptyMasterOpen()* function opens an unused pseudoterminal master, calls *grantpt()* and *unlockpt()* on it, and copies the name of the corresponding pseudoterminal slave into the buffer pointed to by *slaveName*. The caller must specify the amount of space available in this buffer in the argument *snLen*. We show the implementation of this function in Listing 64-1.

It would be equally possible to omit the use of the *slaveName* and *snLen* arguments, and have the caller of *ptyMasterOpen()* call *ptsname()* directly in order to obtain the name of the pseudoterminal slave. However, we employ the *slaveName* and *snLen* arguments because BSD pseudoterminals don't provide an equivalent of the *ptsname()* function, and our implementation of the equivalent function for BSD-style pseudoterminals (Listing 64-4) encapsulates the BSD technique for obtaining the name of the slave.

Listing 64-1: Implementation of *ptyMasterOpen()*

pty/pty_master_open.c

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <fcntl.h>
#include "pty_master_open.h"            /* Declares ptyMasterOpen() */
#include "tlpi_hdr.h"

int
ptyMasterOpen(char *slaveName, size_t snLen)
{
    int masterFd, savedErrno;
    char *p;

    masterFd = posix_openpt(O_RDWR | O_NOCTTY); /* Open pty master */
    if (masterFd == -1)
        return -1;

    if (grantpt(masterFd) == -1) {              /* Grant access to slave pty */
        savedErrno = errno;
        close(masterFd);                        /* Might change 'errno' */
        errno = savedErrno;
        return -1;
    }

    if (unlockpt(masterFd) == -1) {             /* Unlock slave pty */
        savedErrno = errno;
        close(masterFd);                        /* Might change 'errno' */
        errno = savedErrno;
        return -1;
    }

    p = ptsname(masterFd);                      /* Get slave pty name */
    if (p == NULL) {
        savedErrno = errno;
        close(masterFd);                        /* Might change 'errno' */
        errno = savedErrno;
        return -1;
    }

    if (strlen(p) < snLen) {
        strncpy(slaveName, p, snLen);
    } else {                    /* Return an error if buffer too small */
        close(masterFd);
        errno = EOVERFLOW;
        return -1;
    }

    return masterFd;
}
```

pty/pty_master_open.c

64.4 Connecting Processes with a Pseudoterminal: *ptyFork()*

We are now ready to implement a function that does all of the work of setting up a connection between two processes using a pseudoterminal pair, as shown in Figure 64-2. The *ptyFork()* function creates a child process that is connected to the parent by a pseudoterminal pair.

```
#include "pty_fork.h"

pid_t ptyFork(int *masterFd, char *slaveName, size_t snLen,
              const struct termios *slaveTermios, const struct winsize *slaveWS);
```

In parent: returns process ID of child on success, or −1 on error;
in successfully created child: always returns 0

The implementation of *ptyFork()* is shown in Listing 64-2. This function performs the following steps:

- Open a pseudoterminal master using *ptyMasterOpen()* (Listing 64-1) ①.
- If the *slaveName* argument is not NULL, copy the name of the pseudoterminal slave into this buffer ②. (If *slaveName* is not NULL, then it must point to a buffer of at least *snLen* bytes.) The caller can use this name to update the login accounting files (Chapter 40), if appropriate. Updating the login accounting files would be appropriate for applications that provide login services—for example, *ssh*, *rlogin*, and *telnet*. On the other hand, programs such as *script(1)* (Section 64.6) do not update the login accounting files, because they don't provide login services.
- Call *fork()* to create a child process ③.
- All that the parent does after the *fork()* is to ensure that the file descriptor for the pseudoterminal master is returned to the caller in the integer pointed to by *masterFd* ④.
- After the *fork()*, the child performs the following steps:
  - Call *setsid()*, to create a new session (Section 34.3) ⑤. The child is the leader of the new session and loses its controlling terminal (if it had one).
  - Close the file descriptor for the pseudoterminal master, since it is not required in the child ⑥.
  - Open the pseudoterminal slave ⑦. Since the child lost its controlling terminal in the previous step, this step causes the pseudoterminal slave to become the controlling terminal for the child.
  - If the TIOCSCTTY macro is defined, perform a TIOCSCTTY *ioctl()* operation on the file descriptor for the pseudoterminal slave ⑧. This code allows our *ptyFork()* function to work on BSD platforms, where a controlling terminal can be acquired only as a consequence of an explicit TIOCSCTTY operation (refer to Section 34.4).
  - If the *slaveTermios* argument is non-NULL, call *tcsetattr()* to set the terminal attributes of the slave to the values in the *termios* structure pointed to by

this argument ⑨. Use of this argument is a convenience for certain interactive programs (e.g., *script(1)*) that use a pseudoterminal and need to set the attributes of the slave device to be the same as those of the terminal under which the program is run.

- If the *slaveWS* argument is non-NULL, perform an *ioctl()* TIOCSWINSZ operation to set the window size of the pseudoterminal slave to the values in the *winsize* structure pointed to by this argument ⑩. This step is performed for the same reason as the previous step.
- Use *dup2()* to duplicate the slave file descriptor to be the standard input, output, and error for the child ⑪. At this point, the child can now exec an arbitrary program, and that program can use the standard file descriptors to communicate with the pseudoterminal. The execed program can perform all of the usual terminal-oriented operations that can be performed by a program running on a conventional terminal.

As with *fork()*, *ptyFork()* returns the process ID of the child in the parent process, 0 in the child process, or −1 on error.

Eventually, the child process created by *ptyFork()* will terminate. If the parent doesn't terminate at the same time, then it must wait on the child to eliminate the resulting zombie. However, this step can often be eliminated, since applications that employ pseudoterminals are commonly designed so that the parent does terminate at the same time as the child.

> BSD derivatives provide two related, nonstandard functions for working with pseudoterminals. The first of these is *openpty()*, which opens a pseudoterminal pair, returns the file descriptors for the master and slave, optionally returns the name of the slave device, and optionally sets the terminal attributes and window size from arguments analogous to *slaveTermios* and *slaveWS*. The other function, *forkpty()*, is the same as our *ptyFork()*, except that it doesn't provide an analog of the *snLen* argument. On Linux, both of these functions are provided by *glibc* and are documented in the *openpty(3)* manual page.

Listing 64-2: Implementation of *ptyFork()*

pty/pty_fork.c

```
#include <fcntl.h>
#include <termios.h>
#include <sys/ioctl.h>
#include "pty_master_open.h"
#include "pty_fork.h"                   /* Declares ptyFork() */
#include "tlpi_hdr.h"

#define MAX_SNAME 1000

pid_t
ptyFork(int *masterFd, char *slaveName, size_t snLen,
        const struct termios *slaveTermios, const struct winsize *slaveWS)
{
    int mfd, slaveFd, savedErrno;
    pid_t childPid;
    char slname[MAX_SNAME];
```

```
①      mfd = ptyMasterOpen(slname, MAX_SNAME);
       if (mfd == -1)
           return -1;

②      if (slaveName != NULL) {            /* Return slave name to caller */
           if (strlen(slname) < snLen) {
               strncpy(slaveName, slname, snLen);

           } else {                        /* 'slaveName' was too small */
               close(mfd);
               errno = EOVERFLOW;
               return -1;
           }
       }

③      childPid = fork();

       if (childPid == -1) {               /* fork() failed */
           savedErrno = errno;             /* close() might change 'errno' */
           close(mfd);                     /* Don't leak file descriptors */
           errno = savedErrno;
           return -1;
       }

④      if (childPid != 0) {                /* Parent */
           *masterFd = mfd;                /* Only parent gets master fd */
           return childPid;                /* Like parent of fork() */
       }

       /* Child falls through to here */

⑤      if (setsid() == -1)                 /* Start a new session */
           err_exit("ptyFork:setsid");

⑥      close(mfd);                         /* Not needed in child */

⑦      slaveFd = open(slname, O_RDWR);     /* Becomes controlling tty */
       if (slaveFd == -1)
           err_exit("ptyFork:open-slave");

⑧  #ifdef TIOCSCTTY                        /* Acquire controlling tty on BSD */
       if (ioctl(slaveFd, TIOCSCTTY, 0) == -1)
           err_exit("ptyFork:ioctl-TIOCSCTTY");
   #endif

⑨      if (slaveTermios != NULL)           /* Set slave tty attributes */
           if (tcsetattr(slaveFd, TCSANOW, slaveTermios) == -1)
               err_exit("ptyFork:tcsetattr");

⑩      if (slaveWS != NULL)                /* Set slave tty window size */
           if (ioctl(slaveFd, TIOCSWINSZ, slaveWS) == -1)
               err_exit("ptyFork:ioctl-TIOCSWINSZ");
```

```
    /* Duplicate pty slave to be child's stdin, stdout, and stderr */

⑪  if (dup2(slaveFd, STDIN_FILENO) != STDIN_FILENO)
        err_exit("ptyFork:dup2-STDIN_FILENO");
    if (dup2(slaveFd, STDOUT_FILENO) != STDOUT_FILENO)
        err_exit("ptyFork:dup2-STDOUT_FILENO");
    if (dup2(slaveFd, STDERR_FILENO) != STDERR_FILENO)
        err_exit("ptyFork:dup2-STDERR_FILENO");

    if (slaveFd > STDERR_FILENO)        /* Safety check */
        close(slaveFd);                 /* No longer need this fd */

    return 0;                           /* Like child of fork() */
}
```

pty/pty_fork.c

64.5 Pseudoterminal I/O

A pseudoterminal pair is similar to a bidirectional pipe. Anything that is written on the master appears as input on the slave, and anything that is written on the slave appears as input on the master.

The point that distinguishes a pseudoterminal pair from a bidirectional pipe is that the slave side operates like a terminal device. The slave interprets input in the same way as a normal controlling terminal would interpret keyboard input. For example, if we write a *Control-C* character (the usual terminal *interrupt* character) to the pseudoterminal master, the slave will generate a SIGINT signal for its foreground process group. Just as with a conventional terminal, when a pseudoterminal slave operates in canonical mode (the default), input is buffered line by line. In other words, the program reading from the pseudoterminal slave will see (a line of) input only when we write a newline character to the pseudoterminal master.

Like pipes, pseudoterminals have a limited capacity. If we exhaust this capacity, then further writes are blocked until the process on the other side of the pseudoterminal has consumed some bytes.

> On Linux, the pseudoterminal capacity is about 4 kB in each direction.

If we close all file descriptors referring to the pseudoterminal master, then:

- If the slave device has a controlling process, a SIGHUP signal is sent to that process (see Section 34.6).
- A *read()* from the slave device returns end-of-file (0).
- A *write()* to the slave device fails with the error EIO. (On some other UNIX implementations, *write()* fails with the error ENXIO in this case.)

If we close all file descriptors referring to the pseudoterminal slave, then:

- A *read()* from the master device fails with the error EIO. (On some other UNIX implementations, a *read()* returns end-of-file in this case.)

- A *write()* to the master device succeeds, unless the input queue of the slave device is full, in which case the *write()* blocks. If the slave device is subsequently reopened, these bytes can be read.

UNIX implementations vary widely in their behavior for the last case. On some UNIX implementations, *write()* fails with the error `EIO`. On other implementations, *write()* succeeds, but the output bytes are discarded (i.e., they can't be read if the slave is reopened). In general, these variations don't present a problem. Normally, the process on the master side detects that the slave has been closed because a *read()* from the master returns end-of-file or fails. At this point, the process performs no further writes to the master.

Packet mode

Packet mode is a mechanism that allows the process running above a pseudoterminal master to be informed when the following events related to software flow control occur on the pseudoterminal slave:

- the input or output queue is flushed;
- terminal output is stopped or started (*Control-S/Control-Q*); or
- flow control was enabled or disabled.

Packet mode helps with handling software flow control in certain pseudoterminal applications that provide network login services (e.g., *telnet* and *rlogin*).

Packet mode is enabled by applying the *ioctl()* `TIOCPKT` operation to the file descriptor referring to the pseudoterminal master:

```
int arg;

arg = 1;                /* 1 == enable; 0 == disable */
if (ioctl(mfd, TIOCPKT, &arg) == -1)
    errExit("ioctl");
```

When packet mode is in operation, reads from the pseudoterminal master return either a single nonzero control byte, which is a bit mask indicating the state change(s) that occurred on the slave device, or a 0 byte followed by one or more bytes of data that were written on the pseudoterminal slave.

When a state change occurs on a pseudoterminal that is operating in packet mode, *select()* indicates that an exceptional condition (the *exceptfds* argument) has occurred on the master, and *poll()* returns `POLLPRI` in the *revents* field. (Refer to Chapter 63 for descriptions of *select()* and *poll()*.)

Packet mode is not standardized in SUSv3, and some details vary on other UNIX implementations. Further details of packet mode on Linux, including the bit-mask values used to indicate state changes, can be found in the *tty_ioctl(4)* manual page.

64.6 Implementing *script(1)*

We are now ready to implement a simple version of the standard *script(1)* program. This program starts a new shell session, and records all input and output from the session to a file. Most of the shell sessions shown in this book were recorded using *script*.

In a normal login session, the shell is connected directly to the user's terminal. When we run *script*, it places itself between the user's terminal and the shell, and uses a pseudoterminal pair to create a communication channel between itself and the shell (see Figure 64-4). The shell is connected to the pseudoterminal slave. The *script* process is connected to the pseudoterminal master. The *script* process acts as a proxy for the user, taking input entered at the terminal and writing it to the pseudoterminal master, and reading output from the pseudoterminal master and writing it to the user's terminal.

In addition, *script* produces an output file (named `typescript` by default) that contains a copy of all bytes that are output on the pseudoterminal master. This has the effect of recording not only the output produced by the shell session, but also the input that is supplied to it. The input is recorded because, just as with a conventional terminal device, the kernel echoes input characters by copying them to the terminal output queue (see Figure 62-1, on page 1291). However, when terminal echoing is disabled, as is done by programs that read passwords, the pseudoterminal slave input is not copied to the slave output queue, and thus is not copied to the script output file.

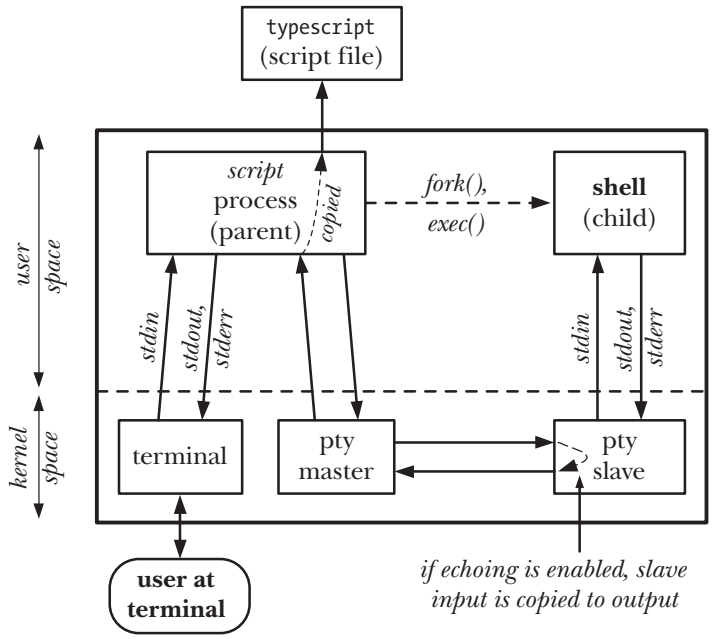

Figure 64-4: The *script* program

Our implementation of *script* is shown in Listing 64-3. This program performs the following steps:

- Retrieve the attributes and window size of the terminal under which the program is run ①. These are passed to the subsequent call to *ptyFork()*, which uses them to set the corresponding values for the pseudoterminal slave device.
- Call our *ptyFork()* function (Listing 64-2) to create a child process that is connected to the parent via a pseudoterminal pair ②.
- After the *ptyFork()* call, the child execs a shell ④. The choice of shell is determined by the setting of the `SHELL` environment variable ③. If the `SHELL` variable is not set or its value is an empty string, then the child execs `/bin/sh`.
- After the *ptyFork()* call, the parent performs the following steps:
  - Open the output script file ⑤. If a command-line argument is supplied, this is used as the name of the script file. If no command-line argument is supplied, the default name `typescript` is used.
  - Place the terminal in raw mode (using the *ttySetRaw()* function shown in Listing 62-3, on page 1310), so that all input characters are passed directly to the *script* program without being modified by the terminal driver ⑥. Characters output by the *script* program are likewise not modified by the terminal driver.

    > The fact that the terminal is in raw mode doesn't mean that raw, uninterpreted control characters will be transmitted to the shell, or whatever other process group is in the foreground for the pseudoterminal slave device, nor that output from that process group is passed raw to the user's terminal. Instead, interpretation of terminal special characters is taking place within the slave device (unless the slave was also explicitly placed in raw mode by an application). By placing the user's terminal in raw mode, we prevent a *second* round of interpretation of input and output characters from occurring.

  - Call *atexit()* to install an exit handler that resets the terminal to its original mode when the program terminates ⑦.
  - Execute a loop that transfers data in both directions between the terminal and the pseudoterminal master ⑧. In each loop iteration, the program first uses *select()* (Section 63.2.1) to monitor both the terminal and the pseudoterminal master for input ⑨. If the terminal has input available, then the program reads some of that input and writes it to the pseudoterminal master ⑩. Similarly, if the pseudoterminal master has input available, the program reads some of that input and writes it to the terminal and to the script file ⑪. The loop executes until end-of-file or an error is detected on one of the monitored file descriptors.

Listing 64-3: A simple implementation of *script(1)*

pty/script.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <libgen.h>
#include <termios.h>
#include <sys/select.h>
#include "pty_fork.h"           /* Declaration of ptyFork() */
#include "tty_functions.h"      /* Declaration of ttySetRaw() */
#include "tlpi_hdr.h"

#define BUF_SIZE 256
#define MAX_SNAME 1000

struct termios ttyOrig;

static void             /* Reset terminal mode on program exit */
ttyReset(void)
{
    if (tcsetattr(STDIN_FILENO, TCSANOW, &ttyOrig) == -1)
        errExit("tcsetattr");
}

int
main(int argc, char *argv[])
{
    char slaveName[MAX_SNAME];
    char *shell;
    int masterFd, scriptFd;
    struct winsize ws;
    fd_set inFds;
    char buf[BUF_SIZE];
    ssize_t numRead;
    pid_t childPid;

①   if (tcgetattr(STDIN_FILENO, &ttyOrig) == -1)
        errExit("tcgetattr");
    if (ioctl(STDIN_FILENO, TIOCGWINSZ, &ws) < 0)
        errExit("ioctl-TIOCGWINSZ");

②   childPid = ptyFork(&masterFd, slaveName, MAX_SNAME, &ttyOrig, &ws);
    if (childPid == -1)
        errExit("ptyFork");

    if (childPid == 0) {        /* Child: execute a shell on pty slave */
③       shell = getenv("SHELL");
        if (shell == NULL || *shell == '\0')
            shell = "/bin/sh";

④       execlp(shell, shell, (char *) NULL);
        errExit("execlp");      /* If we get here, something went wrong */
    }
```

```
        /* Parent: relay data between terminal and pty master */

⑤       scriptFd = open((argc > 1) ? argv[1] : "typescript",
                        O_WRONLY | O_CREAT | O_TRUNC,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                                S_IROTH | S_IWOTH);
        if (scriptFd == -1)
            errExit("open typescript");

⑥       ttySetRaw(STDIN_FILENO, &ttyOrig);

⑦       if (atexit(ttyReset) != 0)
            errExit("atexit");

⑧       for (;;) {
            FD_ZERO(&inFds);
            FD_SET(STDIN_FILENO, &inFds);
            FD_SET(masterFd, &inFds);

⑨           if (select(masterFd + 1, &inFds, NULL, NULL, NULL) == -1)
                errExit("select");

⑩           if (FD_ISSET(STDIN_FILENO, &inFds)) {   /* stdin --> pty */
                numRead = read(STDIN_FILENO, buf, BUF_SIZE);
                if (numRead <= 0)
                    exit(EXIT_SUCCESS);

                if (write(masterFd, buf, numRead) != numRead)
                    fatal("partial/failed write (masterFd)");
            }

⑪          if (FD_ISSET(masterFd, &inFds)) {       /* pty --> stdout+file */
                numRead = read(masterFd, buf, BUF_SIZE);
                if (numRead <= 0)
                    exit(EXIT_SUCCESS);

                if (write(STDOUT_FILENO, buf, numRead) != numRead)
                    fatal("partial/failed write (STDOUT_FILENO)");
                if (write(scriptFd, buf, numRead) != numRead)
                    fatal("partial/failed write (scriptFd)");
            }
        }
    }
```

pty/script.c

In the following shell session, we demonstrate the use of the program in Listing 64-3. We begin by displaying the name of the pseudoterminal used by the *xterm* on which the login shell is running and the process ID of the login shell. This information is useful later in the shell session.

```
$ tty
/dev/pts/1
$ echo $$
7979
```

We then start an instance of our *script* program, which invokes a subshell. Once more, we display the name of the terminal on which the shell is running and the process ID of the shell:

```
$ ./script
$ tty
/dev/pts/24                    Pseudoterminal slave opened by script
$ echo $$
29825                          PID of subshell process started by script
```

Now we use *ps(1)* to display information about the two shells and the process running *script*, and then terminate the shell started by *script*:

```
$ ps -p 7979 -p 29825 -C script -o "pid ppid sid tty cmd"
  PID  PPID   SID TT       CMD
 7979  7972  7979 pts/1    /bin/bash
29824  7979  7979 pts/1    ./script
29825 29824 29825 pts/24   /bin/bash
$ exit
```

The output of *ps(1)* shows the parent-child relationships between the login shell, the process running *script*, and the subshell started by *script*.

At this point, we have returned to the login shell. Displaying the contents of the file `typescript` shows a record of all input and output that was produced while *script* was running:

```
$ cat typescript
$ tty
/dev/pts/24
$ echo $$
29825
$ ps -p 7979 -p 29825 -C script -o "pid ppid sid tty cmd"
  PID  PPID   SID TT       CMD
 7979  7972  7979 pts/1    /bin/bash
29824  7979  7979 pts/1    ./script
29825 29824 29825 pts/24   /bin/bash
$ exit
```

64.7 Terminal Attributes and Window Size

The master and slave device share terminal attributes (*termios*) and window size (*winsize*) structures. (Both of these structures are described in Chapter 62.) This means that the program running above the pseudoterminal master can change these attributes for the pseudoterminal slave by applying *tcsetattr()* and *ioctl()* to the file descriptor of the master device.

One example of where changing terminal attributes can be useful is in the *script* program. Suppose we are running *script* in a terminal emulator window, and we change the size of the window. In this case, the terminal emulator program will inform the kernel of the change in the size of the corresponding terminal device, but this change is not reflected in the separate kernel record for the pseudoterminal slave (see Figure 64-4). As a consequence, screen-oriented programs (e.g., *vi*) running above the pseudoterminal slave will produce confusing output, since their

understanding of the terminal window size differs from the actual size of the terminal. We can solve this problem as follows:

1. Install a handler for SIGWINCH in the *script* parent process, so that it is signaled when the size of the terminal window changes.
2. When the *script* parent receives a SIGWINCH signal, it uses an *ioctl()* TIOCGWINSZ operation to retrieve a *winsize* structure for the terminal window associated with its standard input. It then uses this structure in an *ioctl()* TIOCSWINSZ operation that sets the window size of the pseudoterminal master.
3. If the new pseudoterminal window size is different from the old size, then the kernel generates a SIGWINCH signal for the foreground process group of the pseudoterminal slave. Screen-handling programs such as *vi* are designed to catch this signal and perform an *ioctl()* TIOCGWINSZ operation to update their understanding of the terminal window size.

We described the details of terminal window sizes and the *ioctl()* TIOCGWINSZ and TIOCSWINSZ operations in Section 62.9.

64.8 BSD Pseudoterminals

For most of this chapter, we have focused on UNIX 98 pseudoterminals, since this is the style of pseudoterminal that is standardized in SUSv3 and thus should be used in all new programs. However, we may sometimes encounter BSD pseudoterminals in older applications or when porting programs to Linux from other UNIX implementations. Therefore, we now consider the details of BSD pseudoterminals.

> The use of BSD pseudoterminals is deprecated on Linux. From Linux 2.6.4 onward, BSD pseudoterminal support is an optional kernel component that can be configured via the CONFIG_LEGACY_PTYS option.

BSD pseudoterminals differ from their UNIX 98 counterparts only in the details of how pseudoterminal master and slave devices are found and opened. Once the master and slave have been opened, BSD pseudoterminals operate in the same way as UNIX 98 pseudoterminals.

With UNIX 98 pseudoterminals, we obtain an unused pseudoterminal master by calling *posix_openpt()*, which opens /dev/ptmx, the pseudoterminal master clone device. We then obtain the name of the corresponding pseudoterminal slave using *ptsname()*. By contrast, with BSD pseudoterminals, the master and slave device pairs are precreated entries in the /dev directory. Each master device has a name of the form /dev/pty*xy*, where *x* is replaced by a letter in the 16-letter range [p-za-e] and *y* is replaced by a letter in the 16-letter range [0-9a-f]. The slave corresponding to a particular pseudoterminal master has a name of the form /dev/tty*xy*. Thus, for example, the devices /dev/ptyp0 and /dev/ttyp0 constitute a BSD pseudoterminal pair.

> UNIX implementations vary in the number and names of BSD pseudoterminal pairs that they supply, with some supplying as few as 32 pairs by default. Most implementations provide at least the 32 master devices with names in the range /dev/pty[pq][0-9a-f], along with the corresponding slave devices.

To find an unused pseudoterminal pair, we execute a loop that attempts to open each master device in turn, until one of them is opened successfully. While executing this loop, there are two errors that we may encounter when calling *open()*:

- If a given master device name doesn't exist, *open()* fails with the error ENOENT. Typically, this means we've run through the complete set of pseudoterminal master names on the system without finding a free device (i.e., there was not the full range of devices listed above).
- If the master device is in use, *open()* fails with the error EIO. We can just ignore this error and try the next device.

> On HP-UX 11, *open()* fails with the error EBUSY on an attempt to open a BSD pseudoterminal master that is in use.

Once we have found an available master device, we can obtain the name of the corresponding slave by substituting tty for pty in the name of the master. We can then open the slave using *open()*.

> With BSD pseudoterminals, there is no equivalent of *grantpt()* to change the ownership and permissions of the pseudoterminal slave. If we need to do this, then we must make explicit calls to *chown()* (only possible in a privileged program) and *chmod()*, or write a set-user-ID program (like *pt_chown*) that performs this task for an unprivileged program.

Listing 64-4 shows a reimplementation of the *ptyMasterOpen()* function of Section 64.3 using BSD pseudoterminals. Substituting this implementation is all that is required to make our *script* program (Section 64.6) work with BSD pseudoterminals.

Listing 64-4: Implementation of *ptyMasterOpen()* using BSD pseudoterminals

pty/pty_master_open_bsd.c

```
#include <fcntl.h>
#include "pty_master_open.h"            /* Declares ptyMasterOpen() */
#include "tlpi_hdr.h"

#define PTYM_PREFIX     "/dev/pty"
#define PTYS_PREFIX     "/dev/tty"
#define PTY_PREFIX_LEN  (sizeof(PTYM_PREFIX) - 1)
#define PTY_NAME_LEN    (PTY_PREFIX_LEN + sizeof("XY"))
#define X_RANGE         "pqrstuvwxyzabcde"
#define Y_RANGE         "0123456789abcdef"

int
ptyMasterOpen(char *slaveName, size_t snLen)
{
    int masterFd, n;
    char *x, *y;
    char masterName[PTY_NAME_LEN];

    if (PTY_NAME_LEN > snLen) {
        errno = EOVERFLOW;
        return -1;
    }
```

```
    memset(masterName, 0, PTY_NAME_LEN);
    strncpy(masterName, PTYM_PREFIX, PTY_PREFIX_LEN);

    for (x = X_RANGE; *x != '\0'; x++) {
        masterName[PTY_PREFIX_LEN] = *x;

        for (y = Y_RANGE; *y != '\0'; y++) {
            masterName[PTY_PREFIX_LEN + 1] = *y;

            masterFd = open(masterName, O_RDWR);

            if (masterFd == -1) {
                if (errno == ENOENT)    /* No such file */
                    return -1;          /* Probably no more pty devices */
                else                    /* Other error (e.g., pty busy) */
                    continue;

            } else {            /* Return slave name corresponding to master */
                n = snprintf(slaveName, snLen, "%s%c%c", PTYS_PREFIX, *x, *y);
                if (n >= snLen) {
                    errno = EOVERFLOW;
                    return -1;
                } else if (n == -1) {
                    return -1;
                }

                return masterFd;
            }
        }
    }

    return -1;                  /* Tried all ptys without success */
}
```

pty/pty_master_open_bsd.c

64.9 Summary

A pseudoterminal pair consists of a connected master device and slave device. Together, these two devices provide a bidirectional IPC channel. The benefit of a pseudoterminal is that, on the slave side of the pair, we can connect a terminal-oriented program that is driven by the program that has opened the master device. The pseudoterminal slave behaves just like a conventional terminal. All of the operations that can be applied to a conventional terminal can be applied to the slave, and input transmitted from the master to the slave is interpreted in the same manner as keyboard input is interpreted on a conventional terminal.

One common use of pseudoterminals is in applications that provide network login services. However, pseudoterminals are also used in many other programs, such as terminal emulators and the *script(1)* program.

Different pseudoterminals APIs arose on System V and BSD. Linux supports both APIs, but the System V API forms the basis for the pseudoterminal API that is standardized in SUSv3.

64.10 Exercises

64-1. In what order do the *script* parent process and the child shell process terminate when the user types the end-of-file character (usually *Control-D*) while running the program in Listing 64-3? Why?

64-2. Make the following modifications to the program in Listing 64-3 (`script.c`):

a) The standard *script(1)* program adds lines to the beginning and the end of the output file showing the time the script started and finished. Add this feature.

b) Add code to handle changes to the terminal window size as described in Section 64.7. You may find the program in Listing 62-5 (`demo_SIGWINCH.c`) useful for testing this feature.

64-3. Modify the program in Listing 64-3 (`script.c`) to replace the use of *select()* by a pair of processes: one to handle data transfer from the terminal to the pseudoterminal master, and the other to handle data transfer in the opposite direction.

64-4. Modify the program in Listing 64-3 (`script.c`) to add a time-stamped recording feature. Each time the program writes a string to the `typescript` file, it should also write a time-stamped string to a second file (say, `typescript.timed`). Records written to this second file might have the following general form:

```
<timestamp> <space> <string> <newline>
```

The *timestamp* should be recorded in text form as the number of milliseconds since the start of the script session. Recording the timestamp in text form has the advantage that the resulting file is human-readable. Within *string*, real newline characters will need to be escaped. One possibility is to record a newline as the 2-character sequence `\n` and a backslash as `\\`.

Write a second program, `script_replay.c`, that reads the time-stamped script file and displays its contents on standard output at the same rate at which they were originally written. Together, these two programs provide a simple recording and playback feature for shell session logs.

64-5. Implement client and server programs to provide a simple *telnet*-style remote login facility. Design the server to handle clients concurrently (Section 60.1). Figure 64-3 shows the setup that needs to be established for each client login. What isn't shown in that diagram is the parent server process, which handles incoming socket connections from clients and creates a server child to handle each connection. Note that all of the work of authenticating the user and starting a login shell can be dealt with in each server child by having the (grand)child created by *ptyFork()* go on to exec *login(1)*.

64-6. Add code to the program developed in the previous exercise to update the login accounting files at the start and end of the login session (Chapter 40).

64-7. Suppose we execute a long-running program that slowly generates output that is redirected to a file or pipe, as in this example:

```
$ longrunner | grep str
```

One problem with the above scenario is that, by default, the *stdio* package flushes standard output only when the *stdio* buffer is filled. This means that the output from the *longrunner* program will appear in bursts separated by long intervals of time. One way to circumvent this problem is to write a program that does the following:

a) Create a pseudoterminal.

b) Exec the program named in its command-line arguments with the standard file descriptors connected to the pseudoterminal slave.

c) Read output from the pseudoterminal master and write it immediately to standard output (`STDOUT_FILENO`, file descriptor 1), and, at the same time, read input from the terminal and write it to the pseudoterminal master, so that it can be read by the execed program.

Such a program, which we'll call *unbuffer*, would be used as follows:

```
$ ./unbuffer longrunner | grep str
```

Write the *unbuffer* program. (Much of the code for this program will be similar to that of Listing 64-3.)

64-8. Write a program that implements a scripting language that can be used to drive *vi* in a noninteractive mode. Since *vi* expects to be run from a terminal, the program will need to employ a pseudoterminal.