4 Terminal I/O

4.1 Introduction

Terminal I/O is so complex that it needs most of a chapter all to itself. The problem isn't with normal terminal I/O, which I'll start with—that's even simpler than file I/O. What complicates matters are the numerous variations in terminal attributes. In this chapter I'll also explain how terminals are related to sessions and process groups. Then I'll show how to set up pseudo terminals, which allow one process to control another process that's written for a terminal.

Terminal I/O on UNIX treats a terminal like an old-fashioned hard copy Teletype, the model with the massive type box that makes a terrible racket as it bounces around, now seen only in museums or old movies. There is no specific support in the kernel for screens (character or graphical), function keys, or mice and other pointing devices. Some of these more modern devices can be handled by libraries that call on the standard device driver. The best-known such package for character displays is Curses, which is now part of [SUS2002]. So-called graphical user-interface (GUI) applications for UNIX are generally built to run on the X Window System, perhaps with one of its toolkits such as Motif, Qt, KDE, or Gnome.

Because this chapter's subject is mostly about device drivers rather than an inherent part of the kernel like the file system, specific features vary even more than usual from one UNIX version to another. Sometimes even individual UNIX sites make modifications to the terminal device driver. Remember, also, that not all attributes of terminal I/O are caused by UNIX itself. With the increasing use of smart terminals, local-area networks, and front-end processors, the character stream is subject to much processing before the UNIX kernel sees it and after it has been sent it on its way. Details of this pre- and post-processing vary much too widely to be given here. As usual, I'll concentrate on the standardized properties of terminal I/O. With this as background you should be able to figure out your own system from its manual.

4.2 Reading from a Terminal

This section explains how to read from a terminal, including how not to get blocked if the terminal has nothing ready to be read.

4.2.1 Normal Terminal I/O

I'll start by explaining how normal terminal input and output work; that is, how they work when you first log in and before you've used the `stty` command to customize them to your taste. Then, in subsequent sections, I'll show how the `fcntl` and `tcsetattr` system calls can be used to vary the normal behavior.

There are three ways to access a terminal for input or output:

1. You can open the character special file /dev/tty for reading, writing, or, most commonly, both. This special file is a synonym for the process's controlling terminal (see Section 4.3.1).
2. If you know the actual name of the special file (e.g., /dev/tty04), you can open it instead. If you just want the controlling terminal, this has no advantages over using the generic name /dev/tty. In application programs, it's mainly used to access terminals other than the controlling terminal.
3. Conventionally, each process inherits three file descriptors already open: the standard input, the standard output, and the standard error output. These may or may not be open to the controlling terminal, but normally, they should be used anyhow, since if they are open to a regular file or a pipe instead, it's because the user or parent process decided to redirect input or output.

The basic I/O system calls for terminals are `open`, `read`, `write`, and `close`. It doesn't make much sense to call `creat`, since the special file must already exist. `lseek` has no effect on terminals, as there is no file offset, and that means that `pread` and `pwrite` can't be used.

Unless you've specified the `O_NONBLOCK` flag, an attempt to open a terminal device waits until the device is connected to a terminal. This property is chiefly for the benefit of a system process that blocks in an `open` waiting for a user to connect. It then gets a return from `open` and invokes the login process so the user can log in. After login, the user's shell as specified in the password file is invoked and the user is in business.

By default, the `read` system call also acts on terminals differently from the way it acts on files: It never returns more than one line of input, and no characters are returned until the entire line is ready, even if the `read` requests only a single character. This is because until the user has ended the line, we can't assume it's in final form—the erase and kill characters can be used to revise it or to cancel it entirely. The count returned by `read` is used to determine how many characters were actually read. (What we just described is called *canonical* terminal input, but you can disable it; see Section 4.5.9.)

The user can end a line in one of two ways. Most often, a newline character is the terminator. The return key is pressed, but the device driver translates the return (octal 15) to a newline (octal 12). Alternatively, the user can generate an end-of-file (EOF) character by pressing Ctrl-d. In this case the line is made available to `read` as is, without a newline terminator. One special case is important: If the user generates an EOF at the *start* of the line, `read` will return with a zero count, since the line that the EOF terminated is empty. This looks like an end-of-file, and that's why Ctrl-d can be thought of as an end-of-file "character."

The following function can be used to read a terminal using the standard-input file descriptor `STDIN_FILENO` (defined as 0). It removes a terminating newline character if one is present, and adds a terminating null character to make the line into a C string.

```
bool getln(char *s, ssize_t max, bool *iseof)
{
    ssize_t nread;

    switch (nread = read(STDIN_FILENO, s, max - 1)) {
    case -1:
        EC_FAIL
    case 0:
        *iseof = true;
        return true;
    default:
        if (s[nread - 1] == '\n')
            nread--;
        s[nread] = '\0';
        *iseof = false;
        return true;
    }

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The return value is used to indicate an error, so an EOF is indicated with the third argument, as in this calling example:

```
ec_false( getln(s, sizeof(s), &iseof) )
if (iseof)
    printf("EOF\n");
else
    printf("Read: %s\n", s);
```

`getln` is efficient for terminals because it reads the entire line with a single system call. Furthermore, it doesn't have to search for the end—it just goes by the count returned by `read`. But it doesn't work on files or pipes at all because, since the one-line limit doesn't apply, `read` will in general read too much. Instead of `getln` reading a line, it will read the next `max - 1` characters (assuming that many characters are present).

A more universal version of `getln` would ignore that unique property of terminals—reading one line at most. It would simply examine each character, looking for a newline:

```
bool getln2(char *s, ssize_t max, bool *iseof)
{
    ssize_t n;
    char c;

    n = 0;
    while (true)
        switch (read(STDIN_FILENO, &c, 1)) {
        case -1:
            EC_FAIL
        case 0:
            s[n] = '\0';
            *iseof = true;
            return true;
        default:
            if (c == '\n') {
                s[n] = '\0';
                *iseof = false;
                return true;
            }
            if (n >= max - 1) {
                errno = E2BIG;
                EC_FAIL
            }
            s[n++] = c;
        }
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

This version treats a Ctrl-d typed anywhere as indicating an EOF, which is different from what `getln` did (treating only a Ctrl-d at the beginning of a line as an EOF). With anything other than a terminal, remember, there is no Ctrl-d; an end-of-file is simply the end of the file or a pipe with no open writing file descriptor.

Although `getln2` reads terminals, files, and pipes properly, it reads those sources more slowly than it might because it doesn't buffer the input as described in Section 2.12. This is easily fixed by changing `getln2` to call `Bgetc`, which is part of the `BUFIO` package introduced in that section. `Bopen` is already suitable for opening terminal special files (e.g., /dev/tty). However, to allow us to use `Bgetc` on the standard input, we need to add a function called `Bfdopen` (Exercise 4.2) that initializes a `BUFIO` pointer from an already-open file descriptor instead of from a path. Then we could read a character from the standard input, whether it's a terminal, file, or pipe, like this:

```
ec_null( stin = Bfdopen(STDIN_FILENO, "r") )
while ((c = Bgetc(stin)) != -1)
    /* process character */
```

We're now reading as fast as we can in each case: a block at a time on files and pipes, and a line at a time on terminals.

Our implementation of the `BUFIO` package doesn't allow the same `BUFIO` pointer to be used for both input and output, so if output is to be sent to the terminal, a second `BUFIO` must be opened using file descriptor `STDOUT_FILENO` (defined as 1).

The UNIX standard I/O Library provides three predefined, already-opened `FILE` pointers to access the terminal: `stdin`, `stdout`, and `stderr`, so its function `fdopen`, which is like our `Bfdopen`, usually need not be called for a terminal.

Output to a terminal is more straightforward than input, since nothing like erase and kill processing is done. As many characters as we output with `write` are immediately queued up for sending to the terminal, whether a newline is present or not.

`close` on a file descriptor open to a terminal doesn't do any more than it does for a file. It just makes the file descriptor available for reuse; however, since the file descriptor is most often 0, 1, or 2, no obvious reuse comes readily to mind. So no one bothers to close these file descriptors at the end of a program.[1]

4.2.2 Nonblocking Input

As I said, if a line of data isn't available when `read` is issued on a terminal, `read` waits for the data before returning. Since the process can do nothing in the meantime, this is called *blocking*. No analogous situation occurs with files: either the data is available or the end-of-file has been reached. The file may be added to later by another process, but what matters is where the end is at the time the `read` is executed.

The `O_NONBLOCK` flag, set with `open` or `fcntl`, makes `read` nonblocking. If a line of data isn't available, `read` returns immediately with a –1 return and `errno` set to `EAGAIN`.[2]

Frequently we want to turn blocking on and off at will, so we'll code a function `setblock` to call `fcntl` appropriately. (The technique I'll use is identical to what I showed in Section 3.8.3 for setting the `O_APPEND` flag.)

```
bool setblock(int fd, bool block)
{
    int flags;

    ec_neg1( flags = fcntl(fd, F_GETFL) )
    if (block)
        flags &= ~O_NONBLOCK;
    else
        flags |= O_NONBLOCK;
    ec_neg1( fcntl(fd, F_SETFL, flags) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Here's a test program for `setblock`. It turns off blocking and then reads lines in a loop. If there's nothing there, it sleeps for 5 seconds before continuing. I include

1. We will be closing them in Chapter 6 when we connect two processes with a pipe.
2. In some versions of UNIX, there is a similar flag called `O_NDELAY`. If set and no data is available, `read` returns with a 0, which is indistinguishable from an end-of-file return. Better to use `O_NONBLOCK`.

in the prompt the time since the loop started, using the `time` system call that I introduced in Section 1.7.1.

```
static void test_setblock(void)
{
    char s[100];
    ssize_t n;
    time_t tstart, tnow;

    ec_neg1( tstart = time(NULL) )
    ec_false( setblock(STDIN_FILENO, false) )
    while (true) {
        ec_neg1( tnow = time(NULL) )
        printf("Waiting for input (%.0f sec.) ...\n",
          difftime(tnow, tstart));
        switch(n = read(STDIN_FILENO, s, sizeof(s) - 1)) {
        case 0:
            printf("EOF\n");
            break;
        case -1:
            if (errno == EAGAIN) {
                sleep(5);
                continue;
            }
            EC_FAIL
        default:
            if (s[n - 1] == '\n')
                n--;
            s[n] = '\0';
            printf("Read \"%s\"\n", s);
            continue;
        }
        break;
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("test_setblock")
EC_CLEANUP_END
}
```

Here's the output from one run. I waited a while before typing "hello," and then waited awhile longer before typing Ctrl-d:

```
Waiting for input (0 sec.) ...
Waiting for input (5 sec.) ...
Waiting for input (10 sec.) ...
hello
Waiting for input (15 sec.) ...
```

```
Read "hello"
Waiting for input (15 sec.) ...
Waiting for input (20 sec.) ...
Waiting for input (25 sec.) ...
^DEOF
```

The approach of sleeping for 5 seconds is a compromise between issuing `read`s so frequently that it wastes CPU time and waiting so long that we don't process the user's input right away. As it is, you can see that several seconds elapsed between when I typed "hello" and when the program finally read the input and echoed it back. Thus, generally speaking, turning off blocking and getting input in a `read`/`sleep` loop is a lousy idea. We can do much better than that, as I'll show in the next section.

As I mentioned, you can also set the `O_NONBLOCK` flag when you open a terminal special file with `open`. In this case `O_NONBLOCK` affects `open` as well as `read`: If there is no connection, `open` returns without waiting for one.

One application for nonblocking input is to monitor several terminals. The terminals might be laboratory instruments that are attached to a UNIX computer through terminal ports. Characters are sent in sporadically, and we want to accumulate them as they arrive, in whatever order they show up. Since there's no way to predict when a given terminal might transmit a character, we can't use blocking I/O, for we might wait for one terminal that has nothing to say while other talkative terminals are being ignored. With nonblocking I/O, however, we can poll each terminal in turn; if a terminal is not ready, `read` will return –1 (`errno` set to `EAGAIN`) and we can just go on. If we make a complete loop without finding any data ready, we sleep for a second before looping again so as not to hog the CPU.

This algorithm is illustrated by the function `readany`. Its first two arguments are an array `fds` of file descriptors and a count `nfds` of the file descriptors in the array. It doesn't return until a `read` of one of those file descriptors returns a character. Then it returns via the third argument (`whichp`) the subscript in `fds` of the file descriptor from which the character was read. The character itself is the value of the function; 0 means end-of-file; –1 means error. The caller of `readany` is presumably accumulating the incoming data in some useful way for later processing.[3]

3. Assume the laboratory instruments are inputting newline-terminated lines. In Section 4.5.9 we'll see how to read data without waiting for a complete line to be ready.

```
int readany(int fds[], int nfds, int *whichp)
{
    int i;
    unsigned char c;

    for (i = 0; i < nfds; i++)
        setblock(fds[i], false); /* inefficient to do this every time */
    i = 0;
    while (true) {
        if (i >= nfds) {
            sleep(1);
            i = 0;
        }
        c = 0; /* return value for EOF */
        if (read(fds[i], &c, 1) == -1) {
            if (errno == EAGAIN) {
                i++;
                continue;
            }
            EC_FAIL
        }
        *whichp = i;
        return c;
    }

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

The comment about the call to `setblock` being inefficient means that we really don't have to do it every time `readany` is called. It would be more efficient, if less modular, to make it the caller's responsibility.

Here's a test function for `readany`. It opens two terminals: /dev/tty is the controlling terminal (a `telnet` application running elsewhere on the network), as we explained in Section 4.2.1, and /dev/pts/3 is an `xterm` window on the screen attached directly to the computer, which is running SuSE Linux. (I discovered the name /dev/pts/3 with the `tty` command.)

```
static void readany_test(void)
{
    int fds[2] = {-1, -1}, which;
    int c;
    bool ok = false;

    ec_neg1( fds[0] = open("/dev/tty", O_RDWR) )
    ec_neg1( fds[1] = open("/dev/pts/3", O_RDWR) )
    while ((c = readany(fds, 2, &which)) > 0)
        printf("Got %c from terminal %d\n", isprint(c) ? c : '?', which);
    ec_neg1( c )
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (fds[0] != -1)
        (void)close(fds[0]);
    if (fds[1] != -1)
        (void)close(fds[1]);
    if (!ok)
        EC_FLUSH("readany_test1")
EC_CLEANUP_END
}
```

Here's the output I got at the controlling terminal, which is where the `printf` function wrote:

```
$ readany_test
dog
Got d from terminal 0
Got o from terminal 0
Got g from terminal 0
Got ? from terminal 0
Got c from terminal 1
Got o from terminal 1
Got w from terminal 1
Got ? from terminal 1
```

Here's what I did to get this output: First, I typed "dog" followed by a return on the controlling terminal. That resulted in the four lines of output following the echo of "dog." (The question mark was for the newline.) Then, I went across the room to the Linux computer and typed "cow" followed by a return, which got me this output:

```
cow: command not found
```

Nothing showed up at the controlling terminal. The problem was that I was running a shell in the `xterm` window, and it was blocked in a `read` waiting for some input. That's normal for an abandoned terminal. I had two processes reading the same terminal, and the shell got the letters first, which, of course, it interpreted as a command. So then I tried again, this time like this:

```
$ sleep 10000
cow
```

The `sleep` command, run in the foreground, made the shell wait for a long time, and this time my input came to the `readany_test` command, which printed the last four lines of input. This illustrates that the terminal driver isn't owned by any particular process; just because one process with it opened isn't reading doesn't mean that another process can't.

One other comment on the test program: An EOF from any of the input terminals terminates it, which is how I decided to program it. You might want to do things differently in your own applications.

4.2.3 `select` System Call

Calling `sleep` to pass the time has two disadvantages: First, there could be a delay of as long as a second before an incoming character is processed, as I mentioned earlier, which might be a problem in a time-critical application. Second, we might wake up and make another futile polling loop, perhaps many times, before a character is ready. Best would be a system call that said, "Put me to sleep until an input character is ready on any file descriptor." If we had that, we wouldn't even need to make the `read`s nonblocking, since a `read` won't block if data is ready.

The system call we're looking for is named `select` (we'll get to the similar `pselect` shortly):

select—wait for I/O to be ready

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

int select(
    int nfds,                 /* highest fd + 1 */
    fd_set *readset,          /* read set or NULL */
    fd_set *writeset,         /* write set or NULL */
    fd_set *errorset,         /* error set or NULL */
    struct timeval *timeout   /* time-out (microseconds) or NULL */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

pselect—wait for I/O to be ready

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

int pselect(
    int nfds,                 /* highest fd + 1 */
    fd_set *readset,          /* read set or NULL */
    fd_set *writeset,         /* write set or NULL */
    fd_set *errorset,         /* error set or NULL */
    const struct timespec *timeout, /* time-out (nanoseconds) or NULL */
    const sigset_t *sigmask   /* signal mask */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

Unlike with our `readany`, where we passed in an array of file descriptors, with `select` you set a bit[4] in one of the `fd_set` arguments for each file descriptor you want to test. There are four macros for manipulating a set:

FD_ZERO—clear entire fd_set

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

void FD_ZERO(
    fd_set *fdset          /* fd_set to clear */
);
```

FD_SET—set fd_set file descriptor

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

void FD_SET(
    int fd,                /* file descriptor to set */
    fd_set *fdset          /* fd_set */
);
```

FD_CLR—clear fd_set file descriptor

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

void FD_CLR(
    int fd,                /* file descriptor to clear */
    fd_set *fdset          /* fd_set */
);
```

FD_ISSET—test fd_set file descriptor

```
#include <sys/select.h>
int FD_ISSET(
    int fd,                /* file descriptor to test */
    fd_set *fdset          /* fd_set */
);
/* Returns 1 if set or 0 if clear (no error return) */
```

You can use `select` with 0, 1, 2, or 3 sets, depending on what you're interested in waiting for (reading, writing, or an error). With all `fd_set` arguments `NULL`, `select` just blocks, until it times out or is interrupted by a signal; this isn't particularly useful.

You initialize a set with `FD_ZERO`, and then use `FD_SET` to set a bit for each file descriptor you're interested in, like this:

4. They're not required to be bits, although they usually are. Consider my use of the term in this context as just a model, not an implementation.

```
fd_set set;

FD_ZERO(&set);
FD_SET(fd1, &set);
FD_SET(fd2, &set);
```

Then you call `select`, which blocks (even if `O_NONBLOCK` is set) until one or more file descriptors are ready for reading or writing, or has an error. It modifies the sets you passed it, this time setting a bit for each file descriptor that's ready, and you have to test each file descriptor to see if it's set, like this:

```
if (FD_ISSET(fd1, &set)) {
    /* do something with fd1, depending on which fd_set */
}
```

There's no way to get the list of file descriptors; you have to ask about each one separately. You know what a file descriptor is ready for by what set you test. You can't, therefore, use the same set for more than one argument, even if the input sets are identical. You need distinct output sets so you can get the results.

A bit is set on output only if it was also set on input, even if the corresponding file descriptor was ready. That is, you only get answers for questions you asked.

On success, `select` returns the total number of bits set in all sets, a nearly useless number except as a double-check on the number of file descriptors you come up with via the calls (probably in a loop) to `FD_ISSET`.

The first argument, `nfds`, is *not* the number of bits set on input. It's the length of the sets that `select` should consider. That is, for each input set, all of the bits that are set must be in the range of file descriptors numbered 0 through `nfds - 1`, inclusive. Generally, you calculate the maximum file descriptor number and add 1. If you don't want to do that, you can use the constant `FD_SETSIZE`, which is the maximum number of file descriptors that a set can have. But, as this number is pretty large (1024, say), it's usually much more efficient to give `select` the actual number.

The last argument to `select` is a time-out interval, in terms of a `timeval` structure, which we explained in Section 1.7.1. If nothing happens during that time interval, `select` returns with a 0 value—no error and no bits set. Note that `select` might modify the structure, so reset it before calling `select` again.

If the `timeout` argument is `NULL`, there's no time-out; it's the same as an infinite time interval. If it's not `NULL` but the time is zero, `select` runs through the tests and then returns immediately; you can use this for polling, instead of waiting. This makes sense if your application has other work to do: You can poll from time to time while the other work is in progress, since your application is using the CPU productively, and then switch to blocking (a nonzero time, or a `NULL` `timeout` argument) when the work is done and there's nothing left to do but wait.

There are three common ways that programmers use `select` incorrectly:

- Setting the first argument `nfds` to the highest-numbered file descriptor, instead of that number plus one.
- Calling `select` again with the same sets that it modified, not realizing that on return only bits representing ready file descriptors are set. You have to reinitialize the input sets for each call.
- Not realizing what "ready" means. It doesn't mean that data is ready, only that a `read` or `write` with `O_NONBLOCK` clear would not block. Returning with a zero count (EOF), an error, or with data are all possibilities. It doesn't matter whether a file descriptor has `O_NONBLOCK` set, which means that it will never block; `select` considers it ready only in the hypothetical case that it would not block with `O_NONBLOCK` clear.

Now here's a *much* better implementation of `readany`:

```
int readany2(int fds[], int nfds, int *whichp)
{
    fd_set set_read;
    int i, maxfd = 0;
    unsigned char c;

    FD_ZERO(&set_read);
    for (i = 0; i < nfds; i++) {
        FD_SET(fds[i], &set_read);
        if (fds[i] > maxfd)
            maxfd = fds[i];
    }
    ec_neg1( select(maxfd + 1, &set_read, NULL, NULL, NULL) )
    for (i = 0; i < nfds; i++) {
        if (FD_ISSET(fds[i], &set_read)) {
            c = 0; /* return value for EOF */
            ec_neg1( read(fds[i], &c, 1) )
            *whichp = i;
            return c;
        }
    }
    /* "impossible" to get here */
    errno = 0;
    EC_FAIL

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

This version is very efficient. The `read`s are left blocking, there's no polling loop, and no sleeping. All the wait is in `select`, and when it returns we can read the character and immediately return with it.

If there's a chance that another thread could be reading the same file descriptor as the blocking `read`, it's possible for the data that `select` said was ready to be gone before the code gets to the `read`, which might mean that it would block forever. Perhaps the easiest way to fix this is to make the `read` nonblocking and then to loop back to the `select` if it returns –1 with `errno` set to `EAGAIN`.

It's unusual to worry about a `write` to a terminal blocking. Even if it does, because the driver's buffer is full, it will probably clear itself quickly. Usually blocked writes are more critical with other outputs, such as pipes or sockets. I'll talk about them in later chapters. In particular, there's an example in Section 8.1.3 that uses `select` in the typical way it's used with a socket that accepts connections from multiple clients.

If a bit is set in a read set, you read; if one is set in a write set, you write. What do you do if one is set in the error set? It depends on what the file descriptor is open to. For sockets, the "error" might better be termed "exceptional condition," which translates to out-of-band data being ready (Section 8.7). For terminals, which is what this chapter is about, there aren't any standard exceptional or error conditions, although it's possible for an implementation to have something nonstandard, for which you'll have to consult the documentation for that system.

There's a fancier variant of select called `pselect`; the differences between it and `select` are:

- For `pselect`, the time-out is a `timespec` structure, measured in nanoseconds, and it's not modified.
- There's a sixth argument, a signal mask (Section 9.1.5) to be set during the call to `pselect`, with the old mask restored when it returns. For reasons that are explained in Chapter 9, you should use `pselect` if you're expecting it to be interrupted by a signal, rather than `select`. (If the `sigmask` argument is

NULL, select and pselect behave identically as far as signals are concerned.)

pselect is new with SUS Version 3, so it's just now beginning to appear, and your system probably doesn't have it.

4.2.4 poll System Call

poll—wait for I/O to be ready

```
#include <poll.h>

int poll(
   struct pollfd fdinfo[], /* info on file descriptors to be tested */
   nfds_t nfds,            /* number of elements in fdinfo array */
   int timeout             /* time-out (milliseconds) */
);
/* Returns number of ready file descriptors or -1 on error (sets errno) */
```

struct pollfd—structure for poll

```
struct pollfd {
   int fd;                 /* file descriptor */
   short events;           /* event flags (see table, below) */
   short revents;          /* returned event flags (see table, below) */
};
```

poll was originally designed to be used with the STREAMS I/O facility in AT&T System V (Section 4.9), but, like select, it can be used with file descriptors open to any type of file. Despite its name, poll, like select, can be used either for waiting or for polling. Most systems have poll, but Darwin 6.6 is an exception.

Whereas with select you set up bit masks, with poll you set up pollfd structures, one for each file descriptor you want to know about. For each file descriptor, you set flags in the events field for each event (e.g., reading, writing) to be monitored. When poll returns, each structure's revents field has bits set to indicate which events occurred. So, unlike with select, the inputs aren't disturbed by the call, and you can reuse them.

If you're trying to test a file descriptor with a huge value, poll may be more efficient than select. To see why, suppose you want to test file descriptor 1000. select would have to test all bits 0 through 1000, whereas you could build an array for poll with only one element. Another problem with select is that the sets are sized according to the number of potential file descriptors, and some kernels are configured to allow really huge numbers. Bit masks with 10,000 bits are very inefficient.

The `timeout` argument is in milliseconds, rather than being a `timeval` or `timespec` structure, as with `select` and `pselect`. If the `timeout` is –1, `poll` blocks until at least one of the requested events occurs on some listed file descriptor. If it's 0, as with `select` and `pselect`, `poll` just tests the file descriptors and returns, possibly with a 0 return value, which indicates that no event has occurred. If `timeout` is positive, `poll` blocks for at most that amount of time. (So, as with `select` and `pselect`, the 0 and positive cases are really the same.)

Table 4.1 lists the event flags for `poll`. To make sense of them, you need to know that `poll` distinguishes between "normal" data, "priority" data, and "high-priority" data. The meaning of those terms varies with what the file descriptor is open to.

Table 4.1 Event Flags for `poll` System Call

Flag	Meaning
`POLLRDNORM`	Normal data ready to be read
`POLLRDBAND`	Priority data ready to be read
`POLLIN*`	Same as `POLLRDNORM | POLLRDBAND`
`POLLPRI*`	High-priority data ready to be read
`POLLWRNORM`	Normal data read to be written
`POLLOUT*`	Same as `POLLWRNORM`
`POLLWRBAND`	Priority data may be written
`POLLERR*`	I/O error has occurred; set only in `revents` (i.e., not on input)
`POLLHUP*`	Device disconnected (no longer writable, but may be readable); set only in `revents`
`POLLNVAL*`	Invalid file descriptor; set only in `revents`
* Most common flags for non-STREAMS; `POLLPRI` is generally only used with sockets.	

Except in unusual situations, you can consider `POLLIN | POLLPRI` to be equivalent to `select`'s read and `POLLOUT | POLLWRBAND` to be equivalent to `select`'s write. Note that with `poll` you don't have to ask specifically to be informed about exceptional conditions—the last three flags in the table will be set if they apply no matter what the input flags were.

If you have a `pollfd` array already set up and you just want to disable checking for a file descriptor, you can set its `fd` field to –1.

Here's a version of `readany` (from Section 4.2.2) using `poll`. (Section 4.2.3 had a version using `select`.)

```
#define MAXFDS 100

int readany3(int fds[], int nfds, int *whichp)
{
    struct pollfd fdinfo[MAXFDS] = { { 0 } };
    int i;
    unsigned char c;

    if (nfds > MAXFDS) {
        errno = E2BIG;
        EC_FAIL
    }
    for (i = 0; i < nfds; i++) {
        fdinfo[i].fd = fds[i];
        fdinfo[i].events = POLLIN | POLLPRI;
    }
    ec_neg1( poll(fdinfo, nfds, -1) )
    for (i = 0; i < nfds; i++) {
        if (fdinfo[i].revents & (POLLIN | POLLPRI)) {
            c = 0; /* return value for EOF */
            ec_neg1( read(fdinfo[i].fd, &c, 1) )
            *whichp = i;
            return c;
        }
    }
    /* "impossible" to get here */
    errno = 0;
    EC_FAIL

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

Comments on `readany3`:

- It's a good idea to initialize structures to all zeros, which is what's going on in the declaration for the `fdinfo` array, in case the implementation has defined additional fields that we don't know about. Also, should we ever have to use a debugger on the structure, zeros instead of garbage for the fields we're not initializing explicitly (e.g., `revents`) will make the display easier to read.
- Our program is limited to only 100 file descriptors. Changing it to allocate the array dynamically would be an easy improvement.
- Strictly speaking, the test on `revents` assumes that the flags use distinct bits, which doesn't seem to be stated in the SUS, but it's a safe assumption for `poll`.
- We didn't check the `POLLERR`, `POLLHUP`, or `POLLNVAL` flags, but we probably should have.
- In your own application, you probably don't want to set up the array of `pollfd` structures for every call, as the file descriptors of interest don't usually change that often.
- The blocking `read` could be a problem if another thread is reading the same file descriptor. The solution in the `select` example (Section 4.2.3) could be used here as well.

4.2.5 Testing and Reading a Single Input

In Section 4.2.2 we motivated the use of `select` and `poll` with an example where there were multiple inputs. Sometimes, though, there's just one, but you want to know if a character is ready, without reading it; you want to separate the test from the reading. You can use `select` or `poll` for that by setting the timeout to zero. But, if there's only one input, it's just as easy to do it with `read` alone, with `O_NONBLOCK` set. We'll do it with two functions: `cready` tells whether a character is ready, and `cget` reads it.

A glitch with `cready` is that if a character is ready, `read` will read it, but we wanted `cget` to read it. The simple solution is to keep the prematurely read character in a buffer. Then `cget` can look in the buffer first. If the character is there, we return it; if not, we turn on blocking and call `read`. That scheme is used in the following implementation of `cready` and `cget`:

```
#define EMPTY '\0'
static unsigned char cbuf = EMPTY;
typedef enum {CR_READY, CR_NOTREADY, CR_EOF} CR_STATUS;
```

```
bool cready(CR_STATUS *statusp)
{
    if (cbuf != EMPTY) {
        *statusp = CR_READY;
        return true;
    }
    setblock(STDIN_FILENO, false);
    switch (read(STDIN_FILENO, &cbuf, 1)) {
    case -1:
        if (errno == EAGAIN) {
            *statusp = CR_NOTREADY;
            return true;
        }
        EC_FAIL
    case 0:
        *statusp = CR_EOF;
        return true;
    case 1:
        return true;
    default: /* "impossible" case */
        errno = 0;
        EC_FAIL
    }

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool cget(CR_STATUS *statusp, int *cp)
{
    if (cbuf != EMPTY) {
        *cp = cbuf;
        cbuf = EMPTY;
        *statusp = CR_READY;
        return true;
    }
    setblock(0, true);
    switch (read(STDIN_FILENO, cp, 1)) {
    case -1:
        EC_FAIL
    case 0:
        *cp = 0;
        *statusp = CR_EOF;
        return true;
    case 1:
        *statusp = CR_READY;
        return true;
    default: /* "impossible" case */
        errno = 0;
        EC_FAIL
    }

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Both functions return `true` on success and `false` on error. The `statusp` argument gives the result for `cready`: `CR_READY`, `CR_NOTREADY`, or `CR_EOF`. For `cget`, the only possibilities are `CR_READY` and `CR_EOF`.

We've preempted the NUL byte as our empty-buffer indicator (`EMPTY`), which means that NUL bytes read by `cready` will be ignored. If this isn't acceptable, a separate `bool` variable can be used as an empty-buffer flag.

Notice the way `cready` and `cget` shift back and forth between blocking and nonblocking input. Since we're reading only one input stream, there's no need to stay in the nonblocking state while we test for an available character, as we had to do in the first, inefficient, version of `readany` in Section 4.2.2. We revert to blocking and issue a `read`. After all, waiting for a character is what blocking means.

`cready` and `cget` are most useful when a program has some discretionary work to do, work that can be postponed if a character is ready. A good example is a program that uses Curses (Section 4.8). The way Curses works, all the screen output is held in a buffer until `refresh` is called to send it to the physical screen. This is time-consuming because `refresh` compares what is currently on the screen with the new image so as to minimize the number of characters transmitted. A fast typist can easily get ahead of the updates, especially if the screen must be updated on each keystroke, as it must with a screen editor. A neat solution is to call `refresh` from within the input routine, but only if input is not waiting. If input is waiting, then the user has obviously typed ahead without waiting for the screen to catch up, so the screen update is skipped. The screen will be updated the next time the input routine is executed, unless a character is waiting then too. When the user stops typing, the screen will become current. On the other hand, if the program can process characters faster than the typist types them, the screen will get updated with each keystroke.

This may sound complex, but by using functions we've already developed, it takes only a few lines of code:

```
ec_false( cready(&status) )
if (status == CR_NOTREADY)
    refresh();
ec_false( cget(&status, &c) )
```

4.3 Sessions and Process Groups (Jobs)

This section explains sessions and process groups (also called jobs), which are mainly for use by shells.

4.3.1 Terminology

When a user logs in, a new *session* is created, which consists of a new *process group,* comprising a single process that is running the login shell. That process is the *process-group leader,* and its process ID (73056, say) is the *process-group ID*. That process is also the *session leader,* and the process ID is also the *session ID*.[5] The terminal from which the user logged in becomes the *controlling terminal* of the session. The session leader is also the *controlling process*. This arrangement is shown in the right part of Figure 4.1, with the process group marked FG, for foreground.

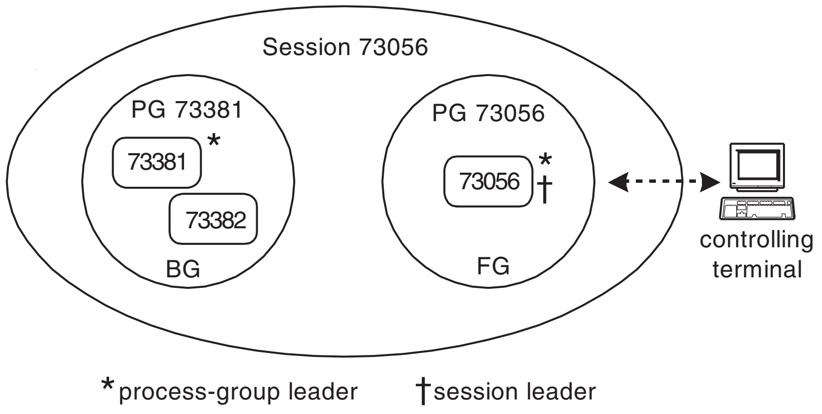


Figure 4.1 Session, process groups, processes, and controlling terminal.

5. The various standards use the phrase "process-group ID of the session leader," but I prefer to just call it the session ID.

If *job control* is enabled by the shell and a command or pipeline of commands is then run in the background, like this:

```
$ du -a | grep tmp >out.tmp&
```

a new process group is formed, this time with two processes in it, as shown at the left of Figure 4.1, with the process group marked BG, for background. One of the two new processes is the leader of this new process group.

Both process groups (73056 and 73381) are in the same session and have the same controlling terminal. With job control, a new process group is created for each command line, and each such process group is also called a *job*.

Just because a process group runs in the background, it doesn't mean its standard file descriptors aren't still directed to and from the terminal. In the previous example pipeline, `du`'s standard output is connected via a pipe to the standard input of `grep`, and `grep`'s standard output is redirected to a file, but that's because the user specifically set them up that way, not because they're running in the background.

What job control does mean is that if certain signals are generated from the controlling terminal, such as interrupt (`SIGINT`), quit (`SIGQUIT`), or suspend (`SIGTSTP`), they are only sent to the foreground process group, leaving any background process groups alone. The "control" part means that the user can execute shell commands to move process groups (jobs) back and forth between foreground and background. Usually, if a process group is running the foreground, a Ctrl-z sends it a `SIGTSTP` signal, which causes the shell to move it to the background, bringing one of the background process groups to the foreground. Or the `fg` shell command can bring a specific background process group to the foreground, which also then sends the old foreground process group to the background.

If job control is not enabled, the two new processes in the example pipeline (running `du` and `grep`) don't run in their own process group, but run in process group 73056, along with the first process, the shell, as shown in Figure 4.2. They still run in the background, but all that means is that the shell doesn't wait for them to complete. As there is only one process group, there is no way to move process groups between foreground and background, and any signals generated from the controlling terminal go to all processes in the only process group in the session. Tying Ctrl-c at the terminal will send a `SIGINT` to all three processes, including the two background processes, and, unless those processes have set themselves up to catch or ignore the signal, it will terminate them, which is the default action for this particular signal. (Much more about signals in Chapter 9.)

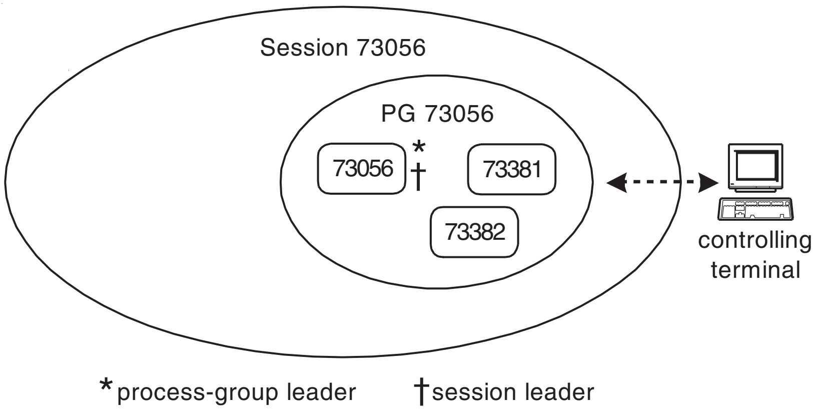


Figure 4.2 No job control; only one process group per session.

With or without job control, when the controlling terminal hangs up or is disconnected, a `SIGHUP` signal is sent to the controlling process, and the default for that signal is to terminate it. For a typical session, this terminates the login shell and logs out the user. The controlling terminal is no longer the controlling terminal for the session, which then has no controlling terminal. Whether other processes in the session can continue to access the terminal is not specified by any of the standards, and most implementations will probably prevent such access by, for example, returning an error from a `read` or `write`.

However, once the controlling process terminates for *any* reason (not just receiving a `SIGHUP`), every process in the foreground (or only) process group gets a `SIGHUP`, which by default causes them to terminate. Unless they've arranged to catch or ignore this signal, they never get far enough to attempt to access the former controlling terminal.

What happens if a process in a background process group attempts to access the controlling terminal? Regardless of whether the controlling process is running, the basic rule is that any attempt by a background process to read the controlling terminal causes it to be sent a `SIGTTIN` signal, and any attempt to write it generates a `SIGTTOU` signal. Here "write" also means using the terminal-control system calls `tcsetattr`, `tcdrain`, `tcflow`, `tcflush`, and `tcsendbreak` (Sections 4.5 and 4.6).

The default behavior for these signals is to suspend the process, which makes a lot of sense: Background processes that need access to the controlling terminal stop, and you move them to the foreground when you're ready to interact with them.

The exceptions to the basic rule are:

- A background process attempting to read with `SIGTTIN` signals ignored or blocked instead gets an `EIO` error from `read`.
- An orphaned background process (process-group leader has terminated) also gets an `EIO` from `read`.
- If the `TOSTOP` terminal attribute (see Section 4.5.6) is clear, the process is allowed to write.
- If `TOSTOP` is set but `SIGTTOU` signals are ignored or blocked, the process is allowed to write.
- If `TOSTOP` is set, an orphaned background process gets an `EIO` error from `write`.

So, to summarize the rules: A background process can't read the controlling terminal no matter what; it gets an error or is stopped. An orphaned background process gets an error if it tries to write the terminal. Nonorphaned background processes can write it if `TOSTOP` is clear or if `SIGTTOU`s are ignored or blocked; otherwise they are stopped.

The asymmetry between reading and writing is because it never makes sense for two processes to both read characters—it's much too dangerous. (Imagine typing `rm *.o` and having the shell only get `rm *`.) Two processes writing is only confusing, and if that's what the user really wants, then so be it.

4.3.2 System Calls for Sessions

setsid—create session and process group

```
#include <unistd.h>

pid_t setsid(void);
/* Returns process-group ID or -1 on error (sets errno) */
```

getsid—get session ID

```
#include <unistd.h>

pid_t getsid(
    pid_t pid          /* process ID or 0 for calling process */
);
/* Returns session ID or -1 on error (sets errno) */
```

I said that each login started in a new session, but where do sessions come from? Actually, any process that isn't already a session leader can call `setsid` to become the leader of a new session, and the process-group leader of the only process group in that session. Importantly, the new session has no controlling terminal. This is great for daemons, which don't need one, and also for sessions that want to establish a different controlling terminal from the one established at login time. The first terminal device opened by the new session becomes its controlling terminal.

4.3.3 System Calls for Process Groups

setpgid—set or create process-group

```
#include <unistd.h>

int setpgid(
    pid_t pid,             /* process ID or 0 for calling process */
    pid_t pgid             /* process-group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

getpgid—get process-group ID

```
#include <unistd.h>

pid_t getpgid(
    pid_t pid              /* process ID or 0 for calling process */
);
/* Returns process-group ID or -1 on error (sets errno) */
```

With a call to `setpgid`, a process that isn't already a process-group leader can be made the leader of a new process group. This occurs if the two arguments are equal and no process-group with that ID exists. Or, if the second argument specifies an existing process-group ID, the process-group indicated by the `pid` argument is changed. However, there are a bunch of restrictions:

- `pid` must be the calling process or a child of the calling process that has not yet done an "exec" system call (explained in Section 5.3).
- `pid` must be in the same session as the calling process.
- If `pgid` exists, it must be in the same session as the calling process.

Practically speaking, these restrictions don't amount to much. Typically, a shell creates child processes for each command in a pipeline, chooses one as the process-group leader, creates the new process-group with a call to `setpgid`, and then puts the other commands in the pipeline into that group with additional calls to `setpgid`. Once this is all set up, process-group assignments don't change, although it's theoretically possible.

Two older calls set and get process-group IDs, `setpgrp` and `getpgrp`, but they're less functional than `setpgid` and `getpgid` and are obsolete.

4.3.4 System Calls for Controlling Terminals

tcsetpgrp—set foreground process-group ID

```
#include <unistd.h>

int tcsetpgrp(
     int fd,            /* file descriptor */
     pid_t pgid         /* process-group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

tcgetpgrp—get foreground process-group ID

```
#include <unistd.h>

pid_t tcgetpgrp(
     int fd             /* file descriptor */
);
/* Returns process-group ID or -1 on error (sets errno) */
```

tcgetsid—get session ID

```
#include <termios.h>

pid_t tcgetsid(
     int fd             /* file descriptor */
);
/* Returns session ID or -1 on error (sets errno) */
```

The `tcsetpgrp` system brings a process-group to the foreground, which means that it receives signals generated from the controlling terminal. Whatever used to be in the foreground is moved to the background. The process group must be in the same session as the calling process.

The shell `fg` command uses `tcsetpgrp` to bring the requested process to the foreground. Typing Ctrl-z doesn't directly move the foreground process to the background; what actually happens is that it sends a `SIGTSTP` signal to the process, which stops it, and then its parent, the shell, gets a return from the `waitpid` system call that tells it what happened. The shell then executes `tcsetpgrp` to move itself to the foreground.

`tcgetsid` is only on SUS systems, so it isn't available on FreeBSD. You can get the session ID from a file descriptor open to the controlling terminal, however, first by calling `tcgetpgrp` to get the foreground process group ID, and then, since that must be the process ID of a process in the session, you can call `getsid` to get the session ID.

4.3.5 Using the Session-Related System Calls

Here's a function that prints lots of session- and process-group-related information using the previous system calls:

```
#include <termios.h>

static void showpginfo(const char *msg)
{
    int fd;

    printf("%s\n", msg);
    printf("\tprocess ID = %ld; parent = %ld\n",
      (long)getpid(), (long)getppid());
    printf("\tsession ID = %ld; process-group ID = %ld\n",
      (long)getsid(0), (long)getpgid(0));
    ec_neg1( fd = open("/dev/tty", O_RDWR) )
    printf("\tcontrolling terminal's foreground process-group ID = %ld\n",
      (long)tcgetpgrp(fd));
#if _XOPEN_VERSION >= 4
    printf("\tcontrolling-terminal's session ID = %ld\n",
      (long)tcgetsid(fd));
#else
    printf("\tcontrolling-terminal's session ID = %ld\n",
      (long)getsid(tcgetpgrp(fd)));
#endif
    ec_neg1( close(fd) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("showpginfo")
EC_CLEANUP_END
}
```

We want to call `showpginfo` when the process starts up and when it gets a `SIGCONT` signal so we can see what's going on when it's running in the background. I'm not going to tell much about signal catching until Chapter 9, and all you need to know for now is that the initializing of the structure and the call of `sigaction` arranges for the function `catchsig` to be executed when a

`SIGCONT` signal arrives.[6] After that, the main program sleeps. If it returns from `sleep` because a signal arrived, it goes back to sleeping again.

```
int main(void)
{
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = catchsig;
    ec_neg1( sigaction(SIGCONT, &act, NULL) )
    showpginfo("initial call");
    while (true)
        sleep(10000);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void catchsig(int signo)
{
    if (signo == SIGCONT)
        showpginfo("got SIGCONT");
}
```

Interacting with `pginfo` is illuminating:

```
$ echo $$
5140
$ pginfo
initial call
        process ID = 6262; parent = 5140
        session ID = 5140; process-group ID = 6262
        controlling terminal's foreground process-group ID = 6262
        controlling-terminal's session ID = 5140

^Z[1]+  Stopped                 pginfo
$ bg %1
[1]+ pginfo &
got SIGCONT
        process ID = 6262; parent = 5140
        session ID = 5140; process-group ID = 6262
        controlling terminal's foreground process-group ID = 5140
        controlling-terminal's session ID = 5140
```

6. In Section 9.1.7 I'm going to say that there are restrictions on what system calls can be used in a signal handler, and, therefore, some of what `showpginfo` does is technically illegal. We're only in Chapter 4, though, and what we don't know won't hurt us too badly to get through this example.

First, we see that the shell's process ID is 5140. Initially, `pginfo` is running in the foreground. It's in its own process group, whose ID is 6262, which is the same as `pginfo`'s process ID. This means it's the process-group leader. The session ID is the same as the shell's process ID, which means that `pginfo` is in the same session as the shell. Then when the user typed Ctrl-z the shell got the status of its child, `pginfo`, reported as stopped, and brought itself to the foreground, allowing us to type another command to the shell, `bg`, which restarted `pginfo` in the background. That sent a `SIGCONT` signal to `pginfo` which caused it to print the information again. It's much the same, except this time the foreground process group is 5140, that of the shell.

Most of the system calls related to sessions and process groups are used by shells and not by other application programs. However, one of them, `setsid`, is very important to applications, as we'll see in Section 4.10.1, when we use it to switch a process's controlling terminal to a pseudo terminal.

4.4 `ioctl` System Call

Recall that there are two kinds of UNIX devices: block and character. Block devices were dealt with in Chapter 3, and one important kind of character device, that for terminals, is what we've been talking about in this chapter. There's a general-purpose system call for controlling character devices of all types called `ioctl`:

ioctl—control character device

```
#include <...>

int ioctl(
     int fd,              /* file descriptor */
     int req,             /* request */
     ...                  /* arguments that depend on request */
);
/* Returns -1 on error (sets errno); some other value on success */
```

With two exceptions, because the POSIX and SUS standards don't specify devices, the include file, the various requests, and the associated third arguments used with `ioctl` are implementation dependent, and you generally find the details in the documentation for the driver you're trying to control.

The two exceptions are:

- Almost everything you can do to a terminal with `ioctl` also has its own standardized function, primarily so that the types of the arguments can be checked at compile time. These functions are `tcgetattr`, `tcsetattr`, `tcdrain`, `tcflow`, `tcflush`, and `tcsendbreak`, and they're described in the next section.
- The SUS does specify how `ioctl` is to be used with STREAMS, which I touch on briefly in Section 4.9.

Outside of Section 4.9, I won't discuss `ioctl` any further in this book.

4.5 Setting Terminal Attributes

The two important system calls for controlling terminals are `tcgetattr`, which gets the current attributes, and `tcsetattr`, which sets new attributes. Four other functions, `tcdrain`, `tcflow`, `tcflush`, and `tcsendbreak`, are explained in Section 4.6.

4.5.1 Basic `tcgetattr` and `tcsetattr` Usage

tcgetattr—get terminal attributes

```
#include <termios.h>

int tcgetattr(
    int fd,                     /* file descriptor */
    struct termios *tp          /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

tcsetattr—set terminal attributes

```
#include <termios.h>

int tcsetattr(
    int fd,                     /* file descriptor */
    int actions,                /* actions on setting */
    const struct termios *tp    /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct termios—structure for terminal-control functions

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

The terminal-information structure (`termios`) contains about 50 flag bits that tell the driver how to process characters coming in and going out, how to set communication-line parameters, such as the baud rate, and so on. The structure also defines several control characters, such as erase (normally Del or backspace), kill (normally Ctrl-u), and EOF (normally Ctrl-d).

Before you can change attributes, you need to call `tcgetattr` to initialize the structure. It's complicated and possibly loaded with implementation-defined flags, so it's impractical to just initialize a structure from scratch. After you get the structure, you adjust the flags and other fields as you like and then call `tcsetattr` to effect the changes. Its second argument, `action`, controls how and when the changes occur; one of these symbols is used:

`TCSANOW`	Set the terminal immediately, according to the information in the structure.
`TCSADRAIN`	Similar to `TCSANOW`, but wait for all pending output characters to be sent first. This should be used when output-affecting changes are made, to ensure characters previously output are processed under the rules in effect when they were written.
`TCSAFLUSH`	Similar to `TCSADRAIN` but in addition to waiting for pending output to be drained, the input queue is flushed (characters are discarded). When beginning a new interactive mode, as when starting a screen editor, this is the safest command to use because it prevents characters that may have been typed ahead from causing an unintended action.

The following subsections describe most of the commonly used flags and how they're generally combined for typical uses (e.g., "raw" mode). For a complete list of standardized flags, see [SUS2002]. Or, you can execute `man termios` or `man termio` to see your implementation's flags.

4.5.2 Character Size and Parity

Flags in `c_cflag` represent the character size (`CS7` for 7 bits; `CS8` for 8), the number of stop bits (`CSTOPB` for 2, clear for 1), whether parity should be checked (`PARENB` if so), and the parity (`PARODD` for odd, clear for even). There's no creativity here; one is usually quite satisfied to find a combination that works.

You can also get more information about characters that fail the parity check. If the `PARMRK` flag of `c_iflag` is on, bad characters are preceded with the two characters 0377 and 0. Otherwise, characters with incorrect parity are input as NUL bytes. Alternatively, flag `IGNPAR` of `c_iflag` can be set to ignore characters with bad parity entirely.

4.5.3 Speed

Nowadays most terminals run pretty fast, but there still is a list of symbols (not integers) defined for various standard speeds, including some very low ones. Their names are of the form `B`n, where n is `50`, `75`, `110`, `134`, `150`, `200`, `300`, `600`, `1200`, `1800`, `2400`, `4800`, `9600`, `19200`, or `38400`. The units are bits-per-second, which is called "baud" in UNIX documentation and standards.

Some implementations encode the speed in the `c_cflag` field, but for portability you use four standard functions to get or set the input and output speeds, rather than manipulating the `termios` structure directly. These functions only manipulate the structure—you have to first populate the structure with a call to `tcgetattr`, and then you have to call `tcsetattr` to make a speed change effective. Also, keep in mind that the only standardized values for a `speed_t` are the `B`n symbols. You can't portably pass in a plain integer, although some implementations might allow that.

cfgetispeed—get input speed from termios structure

```
#include <termios.h>

speed_t cfgetispeed(
     const struct termios *tp  /* attributes */
);
/* Returns speed (no error return) */
```

cfgetospeed—get output speed from termios structure

```
#include <termios.h>

speed_t cfgetospeed(
     const struct termios *tp  /* attributes */
);
/* Returns speed (no error return) */
```

cfsetispeed—set input speed in termios structure

```
#include <termios.h>

int cfsetispeed(
    struct termios *tp,       /* attributes */
    speed_t speed             /* speed */
);
/* Returns 0 on success or -1 on error (may set errno) */
```

cfsetospeed—set output speed in termios structure

```
#include <termios.h>

int cfsetospeed(
    struct termios *tp,       /* attributes */
    speed_t speed             /* speed */
);
/* Returns 0 on success or -1 on error (may set errno) */
```

The speed `B0` is special: If it's set as the output speed, the call to `tcsetattr` will disconnect the terminal. If it's set as the input speed, it means that the output and input speeds are the same.

4.5.4 Character Mapping

The mapping of newlines to returns and returns to newlines on input is controlled by flags `INLCR` and `ICRNL` in `c_oflag`; normally the first is clear and the second is set. For terminals that input both a return and a newline when the return key is pressed, the return can be ignored (`IGNCR`).

On output we usually want to map a newline to a return-newline pair; flag `ONLCR` in `c_oflag` does this job. Other flags cause a return to be changed to a newline (`OCRNL`), and a return at column 0 to be suppressed (`ONOCR`). If a newline also causes a return, there's a flag (`ONLRET`) to tell the terminal driver this so it can keep track of the column (for tabs and backspaces).

The driver can also handle uppercase-only terminals, which are much less common than they once were.[7] If `XCASE` in `c_lflag` is set, `IUCLC` in `c_iflag` is set, and `OLCUC` in `c_oflag` is set, then uppercase letters are mapped to lowercase on input, and lowercase letters are mapped to uppercase on output. Since UNIX uses lowercase more than upper, this is desirable. To input or output uppercase, the letter is preceded with a backslash (\).

7. The words "much less common" appeared in the 1985 edition of this book. Now they probably don't even exist outside of museums.

If the `ISTRIP` flag in `c_iflag` is set, input characters are stripped to seven bits (after parity is checked). Otherwise, all eight bits are input. Some terminals use ASCII, which is a seven-bit code, so stripping is normally desirable on those. Modern devices and programs such as `xterm`, `telnet`, or `ssh` may, however, transmit a full eight bits. On output, all bits written by the process are sent. If eight data bits are to be sent to the terminal, then parity generation must be turned off by clearing the `PARENB` flag in `c_cflag`.

4.5.5 Delays and Tabs

Terminals used to be mechanical and lacked big buffers so they required time to perform various motions, such as to return the carriage. Flags in `c_oflag` can be set to adjust delays for newlines, returns, backspaces, horizontal and vertical tabs, and form-feeds, but they're unlikely to be needed any more.

Another flag, `TAB3`, causes output tabs to be replaced by an appropriate number of spaces. This is useful when the terminal has no tabs of its own, when it's too much trouble to set them, or when the terminal is really another computer that is downloading the output and only spaces are wanted.

4.5.6 Flow Control

Output to the terminal can be stopped momentarily either by the user pressing Ctrl-s or by a process calling `tcflow` (see Section 4.6). Flow is restarted by the user typing Ctrl-q or by `tcflow`.

If the `IXANY` flag in `c_iflag` is set, the user can type any character to restart the flow, not just Ctrl-q. If the `IXON` flag is clear, no output flow control is available to the user at all; Ctrl-s and Ctrl-q have no special meaning.

The terminal driver also supports input flow control. If the `IXOFF` flag is set, then, when the input queue gets full, the driver will send a Ctrl-s to the terminal to suspend input. When the queue length drops because a process read some queued-up characters, a Ctrl-q will be sent to tell the terminal to resume input. Of course, this feature can be used only with those terminals that support it.

If the `TOSTOP` flag in `c_lflag` is set, a `SIGTTOU` signal is sent if a process in a background process group attempts to write to the controlling terminal, as I explained in Section 4.3.1.

4.5.7 Control Characters

Several control characters can be changed from their defaults by setting elements of the `c_cc` array in the `termios` structure. Table 4.2 gives, for each settable control character, the subscript in `c_cc` and the default value. In the array a character is represented by its internal value.

The ASCII control characters are as follows: Ctrl-a through Ctrl-z have the values 1 through 26; Ctrl-[, Ctrl-\, Ctrl-], Ctrl-^, and Ctrl-_ have values 27 through 31; Ctrl-? (Del) has value 127; Ctrl-@ has value 0. Values 32 through 126 are the 95 printable ASCII characters and aren't useful for control characters. Values above 127 cannot be generated from a standard US English keyboard, but may be available on other keyboards. There's no standard UNIX way to use function keys that generate multiple-character sequences.

Table 4.2 `c_cc` Subscripts

Subscript	Meaning	Typical Default
`VEOF`	end-of-file	Ctrl-d
`VEOL`	alternative end-of-line (rarely used)	undefined
`VERASE`	erase character	Ctrl-?[†]
`VINTR`	interrupt; generates `SIGINT`	Ctrl-c
`VKILL`	kill line	Ctrl-u
`VQUIT`	quit; generates `SIGQUIT`	Ctrl-\
`VSUSP`	stop process; generates `SIGTSTP`[*]	Ctrl-z
`VSTART`	resume input or output	Ctrl-q
`VSTOP`	suspend input or output	Ctrl-s

[*] Switching the names `VSUSP` and `VSTOP` might make sense, but the table is correct.
[†] Because some terminals generate Ctrl-? (ASCII DEL) when the backspace key is pressed, not Ctrl-h (ASCII BS).

Most implementations define additional characters for such things as word-erase, reprint, and discard output; the only standardized ones are those in the table, however. The total number of elements in the array is `NCCS`.

The characters specified by the c_cc subscripts VINTR, VQUIT, and VSUSP generate signals, as I said in Section 4.3. By default, SIGINT terminates a process, SIGQUIT terminates it with a core dump, and SIGTSTP causes it to stop until a SIGCONT signal is received. There's lots more on signals in Chapter 9.

To suppress a control character, you can set it to _POSIX_VDISABLE. Alternatively, you can suppress interrupt, quit, and stop (c_cc[VSUSP]) by clearing the ISIG flag in c_lflag, which disables signal generation. If _POSIX_VDISABLE is not in the header unistd.h, you get its value from pathconf or fpathconf (Section 1.5.6). It's usually defined as zero.

4.5.8 Echo

Terminals normally run in full duplex, which means that data can flow across the communication line in both directions simultaneously. Consequently, the computer, not the terminal, echoes typed characters to provide verification that they were correctly received. The usual output character mapping applies, so, for example, a return is mapped to a newline and then, when echoed, the newline is mapped to both a return and a newline.

To turn echo off, the ECHO flag in c_lflag is cleared. This is done either to preserve secrecy, as when typing a password, or because the process itself must decide if, what, and where to echo, as when running a screen editor.

Two special kinds of echo are available for the erase and kill characters. The flag ECHOE can be set so erase character echoes as backspace-space-backspace. This has the pleasing effect of clearing the erased character from a CRT screen (it has no effect at all on hard-copy terminals, except to wiggle the type element). The flag ECHOK can be set to echo a newline (possibly mapped to return and newline) after a kill character; this gives the user a fresh line to work on.

4.5.9 Punctual vs. Canonical Input

Normally, input characters are queued until a line is complete, as indicated by a newline or an EOF. Only then are any characters made available to a read, which might only ask for and get one character. In many applications, such as screen editors and form-entry systems, the reading process wants the characters as they are typed, without waiting for a line to be assembled. Indeed, the notion of "lines" may have no meaning.

If the flag `ICANON` ("canonical") in `c_lflag` is clear, input characters are not assembled into lines before they are read; therefore, erase and kill editing are unavailable. The erase and kill characters lose their special meaning. Two parameters, MIN and TIME, determine when a `read` is satisfied. When the queue becomes MIN characters long, or when TIME tenths of a second have elapsed after a byte has been received, the characters in the queue become available. TIME uses an inter-byte timer that's reset when a byte is received, and it doesn't start until the first byte is received so you won't get a timeout with no bytes received. (An exception when MIN is zero follows.)

The subscripts `VMIN` and `VTIME` of the `c_cc` array hold MIN and TIME. As those positions may be the same ones used by VEOF and VEOL, make sure you set MIN and TIME explicitly or you might get whatever the current EOF and EOL characters work out to, which will cause very strange results. If you ever find a process getting input on every fourth character typed, this is probably what's happened. (Ctrl-d, the usual EOF character, is 4.)

The idea behind MIN and TIME is to allow a process to get characters as, or soon after, they are typed without losing the benefits of reading several characters with a single `read` system call. Unless you code your input routine to buffer input, however, you might as well set MIN to 1 (TIME is then irrelevant, as long as it's greater than zero), since your `read` system calls will be reading only a single character anyhow.

What I just described applies to MIN and TIME both greater than zero. If either or both are zero, it's one of these special cases:

- If TIME is zero, the timer is turned off, and MIN applies, assuming it's greater than zero.
- If MIN is zero, TIME is no longer treated as an inter-byte timer, and timing starts as soon as the `read` is issued. It returns when one byte is read or TIME expires, whichever comes first. So, in this case, `read` could return with a count of zero.
- In both are zero, `read` returns when the minimum of the number of bytes requested (by `read`'s third argument) and the number available in the input queue. If no bytes are available, `read` returns with a count of zero, so this acts like a nonblocking `read`.

The standards don't specify exactly what happens if `O_NONBLOCK` is set and MIN and/or TIME are nonzero. `O_NONBLOCK` may have precedence (causing an immediate return if no characters are available), or MIN/TIME may have precedence. To avoid this uncertainty, you should not set `O_NONBLOCK` when `ICANON` is clear.

Here's a buffering, but punctual, input routine called `tc_keystroke`. Since it will change the terminal flags, we've also provided the companion routine `tc_restore` to restore the terminal to the way it was before the first call to `tc_keystroke`. The calling program must call `tc_restore` before terminating.

```
static struct termios tbufsave;
static bool have_attr = false;

int tc_keystroke(void)
{
    static unsigned char buf[10];
    static ssize_t total = 0, next = 0;
    static bool first = true;
    struct termios tbuf;

    if (first) {
        first = false;
        ec_neg1( tcgetattr(STDIN_FILENO, &tbuf) )
        have_attr = true;
        tbufsave = tbuf;
        tbuf.c_lflag &= ~ICANON;
        tbuf.c_cc[VMIN] = sizeof(buf);
        tbuf.c_cc[VTIME] = 2;
        ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
    }
    if (next >= total)
        switch (total = read(0, buf, sizeof(buf))) {
        case -1:
            syserr("read");
        case 0:
            fprintf(stderr, "Mysterious EOF\n");
            exit(EXIT_FAILURE);
        default:
            next = 0;
        }
    return buf[next++];

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

```
bool tc_restore(void)
{
    if (have_attr)
        ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbufsave) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Note that we've cleared only `ICANON` (and set MIN and TIME, of course). Typically, we would want to clear more flags, to turn off echo, output character mapping, and so on. This is addressed in the next section. In a specific application, you will want to experiment with values for MIN and TIME. We've used 10 characters and .2 seconds, which seems to work well.

Here's some test code that calls `tc_keystroke` in a loop:

```
setbuf(stdout, NULL);
while (true) {
    c = tc_keystroke();
    putchar(c);
    if (c == 5) { /* ^E */
        ec_false( tc_restore() )
        printf("\n%s\n", "Exiting...");
        exit(EXIT_SUCCESS);
    }
}
```

The important thing about this test code is that characters are echoed by the terminal driver when they're typed and then again by the `putchar` function when they're returned by `tc_keystroke`. This allows us to see the effect of MIN and TIME, as in the following example output where the letters in "slow" were typed very slowly and the letters in "fast" were typed very fast. The typed letters are underlined.

```
sslloowwfastfast^E
Exiting...
```

Thus, MIN and TIME cause no change in how `tc_keystoke` operates, in that it's still punctual (within .2 seconds, with our setting of TIME) yet cuts down on `read`s by a factor of 10 (or whatever we set MIN to).

4.5.10 Raw Terminal I/O

Punctual, or noncanonical, input, as described in the previous section, is useful, but sometimes we want echo turned off as well, along with most everything else.

That's normally called *raw* mode, which means that no special input or output processing is done and that characters are readable immediately, without waiting for a line to be assembled. There's no single flag to set raw mode; instead you have to set the various attributes one-by-one. (The `stty` *command* does have a `raw` option.)

We want raw terminal I/O to have the following attributes, although there's no hard-and-fast definition, and you may want to vary this a bit for your own purposes:

1. *Punctual input.* Clear `ICANON` and set MIN and TIME.
2. *No character mapping.* Clear `OPOST` to turn off output processing. For input, clear `INLCR` and `ICRNL`. Set the character size to `CS8`. Clear `ISTRIP` to get all eight bits and clear `INPCK` and `PARENB` to turn off parity checking. Clear `IEXTEN` to turn off extended-character processing.
3. *No flow control.* Clear `IXON`.
4. *No control characters.* Clear `BRKINT` and `ISIG` and set all the control characters to the disabled value, even those that are implementation defined.
5. *No echo.* Clear `ECHO`.

These operations are encapsulated in the function `tc_setraw`. Note that it saves the old `termios` structure for use by `tc_restore` (previous section).

```
bool tc_setraw(void)
{
    struct termios tbuf;
    long disable;
    int i;

#ifdef _POSIX_VDISABLE
    disable = _POSIX_VDISABLE;
#else
    /* treat undefined as error with errno = 0 */
    ec_neg1( (errno = 0, disable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) )
#endif
    ec_neg1( tcgetattr(STDIN_FILENO, &tbuf) )
    have_attr = true;
    tbufsave = tbuf;
    tbuf.c_cflag &= ~(CSIZE | PARENB);
    tbuf.c_cflag |= CS8;
    tbuf.c_iflag &= ~(INLCR | ICRNL | ISTRIP | INPCK | IXON | BRKINT);
    tbuf.c_oflag &= ~OPOST;
    tbuf.c_lflag &= ~(ICANON | ISIG | IEXTEN | ECHO);
    for (i = 0; i < NCCS; i++)
        tbuf.c_cc[i] = (cc_t)disable;
    tbuf.c_cc[VMIN] = 5;
    tbuf.c_cc[VTIME] = 2;
    ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Here's a test program for `tc_setraw` that uses the `stty` command to display the terminal settings:

```
setbuf(stdout, NULL);
printf("Initial attributes:\n");
system("stty | fold -s -w 60");
printf("\r\nRaw attributes:\n");
tc_setraw();
system("stty | fold -s -w 60");
tc_restore();
printf("\r\nRestored attributes:\n");
system("stty | fold -s -w 60");
```

What follows is the output I got (on Solaris). Note that `tc_setraw` turned off all the `c_cc` characters, not just the standard ones that we know about. But, it didn't turn off all the flags that might make the terminal nonraw, such as `imaxbel`.[8] You may find that you have to adjust your version of `tc_setraw` for each system you port your code to, or else use the Curses library (Section 4.8), which has its own way of setting the terminal to raw.

```
Initial attributes:
speed 38400 baud; evenp
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
swtch = <undef>;
brkint -inpck icrnl -ixany imaxbel onlcr
echo echoe echok echoctl echoke iexten

Raw attributes:
speed 38400 baud; -parity
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
min = 5; time = 2;
intr = <undef>; quit = <undef>; erase = <undef>; kill =
<undef>; eof = ^e; eol = ^b; swtch = <undef>; start =
```

8. A nonstandard flag that controls ringing of the bell when an input line is too long.

```
<undef>; stop = <undef>; susp = <undef>; dsusp = <undef>;
rprnt = <undef>; flush = <undef>; werase = <undef>; lnext =
<undef>;
-inpck -istrip -ixon imaxbel -opost
-isig -icanon -echo echoe echok echoctl echoke

Restored attributes:
speed 38400 baud; evenp
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
swtch = <undef>;
brkint -inpck icrnl -ixany imaxbel onlcr
echo echoe echok echoctl echoke iexten
```

Sometimes during (or after!) debugging, a program that's put the terminal in raw mode aborts before it can restore the original settings. Users sometimes think the computer has crashed or that their terminal has "locked up"—they can't even use EOF to log off. But it's possible to recover from raw mode. First, recall that `ICRNL` is clear; this means you'll have to end your input lines with Ctrl-j instead of using the return key.[9] Second, you won't see what you type because `ECHO` is off, too. Start by typing a few line feeds; you should see a series of shell prompts but not at the left margin because output processing (`OPOST`) is turned off. Then type `stty sane`, followed by a line feed, and all should be well.

4.6 Additional Terminal-Control System Calls

As we saw in Section 4.5.1, when you set terminal attributes with `tcsetattr`, you can drain (wait for) the output queue before the attributes are set with the `TCSADRAIN` action, and you can in addition flush (throw away) any characters in the input queue with the `TCSAFLUSH` action. Two system calls allow you to control draining and flushing separately, without also setting attributes:

tcdrain—drain (wait for) terminal output

```
#include <termios.h>

int tcdrain(
    int fd            /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

9. A nice enhancement to the shell would be for it to accept a return as a line terminator as well as a newline. (Footnote first appeared in 1985—no progress so far.)

tcflush—flush (throw away) terminal output, input, or both

```
#include <termios.h>

int tcflush(
    int fd,             /* file descriptor */
    int queue           /* queue to be affected */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

These functions act on the queue—that is, characters received from the terminal but not yet read by any process, or characters written by a process but not yet transmitted to the terminal.

The second argument (`queue`) for `tcflush` is one of:

`TCIFLUSH`	Flush the input queue.
`TCOFLUSH`	Flush the output queue.
`TCIOFLUSH`	Flush both queues.

The `TCSADRAIN` argument of `tcsetattr` is therefore equivalent to calling `tcdrain` before the attributes have been set, and the `TCSAFLUSH` argument of `tcsetattr` is equivalent to calling both `tcdrain` and `tcflush` with a queue argument of `TCIFLUSH`.

The next system call, `tcflow`, allows an application to suspend or restart terminal input or output:

tcflow—suspend or restart flow of terminal input or output

```
#include <termios.h>

int tcflow(
    int fd,             /* file descriptor */
    int action          /* direction and suspend/restart */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `action` argument is one of:

`TCOOFF`	Suspend output.
`TCOON`	Restart suspended output.

`TCIOFF`	Send a STOP character, intended to make the terminal suspend input.
`TCION`	Send a START character, intended to make the terminal restart suspended input.

`TCOOF` could be used by an application that's writing multiple pages to the terminal to suspend output after each page so the user can read it before continuing by typing the START character from the keyboard, which is Ctrl-q by default (Section 4.5.7). After calling `tcflow` with `TCOOF`, the application can keep writing pages of output; when the terminal's queue is full, the next `write` will block until it gets drained a bit.

However, this approach is not what users expect, and it's not how page-at-a-time applications typically work. Commands like `more` and `man` output a page and then prompt the user to type an ordinary character (e.g., space or return) to get the next page. Internally, such applications output one page, output the prompt, and block on a `read` (typically with canonical input and echo turned off), waiting for the user to type the go-ahead character.

On the input side, `TCIOFF` and `TCION` had a historical use in preventing a terminal from overflowing the input queue, but nowadays this is handled entirely by the driver and/or the hardware. Today `tcflow` would probably be used only for specialized devices, not actual terminals.

In the same category would be `tcsendbreak`, which sends a break to a terminal:

tcsendbreak—send break to terminal

```
#include <termios.h>

int tcsendbreak(
     int fd,            /* file descriptor */
     int duration       /* duration of break */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

A "break" is a stream of 0 bits sent to a terminal for a period of time, often as a way of sending the terminal into a special "attention" mode.

If the `duration` argument is 0, the period is between a quarter and half second. If it's nonzero, what it means is implementation dependent. Don't worry about portability, though—you'll probably never have occasion to send a break to a terminal.

4.7 Terminal-Identification System Calls

As I said in Section 4.2.1, the generic pathname /dev/tty provides a way to access a process's controlling terminal without knowing its name. That wasn't always a requirement, however, and instead there was a system call for getting such a synonym:

ctermid—get pathname for controlling terminal

```
#include <stdio.h>

char *ctermid(
    char *buf          /* buffer of size L_ctermid or NULL */
);
/* Returns pathname or empty string on error (errno not defined) */
```

You can call `ctermid` with a `NULL` argument, in which case it uses a static buffer of the appropriate size. Or, you can supply a buffer of size `L_ctermid` (which includes room for the NUL byte) to avoid any conflicts in case there are multiple threads calling the function. Note that on error `ctermid` returns the empty string, not `NULL`. `ctermid` isn't required to return anything more useful than the string `/dev/tty`, and on most (if not all) systems that's all it does.

Somewhat more useful would be a system call that returned an actual terminal name, and that's what `ttyname` and `ttyname_r` are for, only you have to give them an open file descriptor as input:

ttyname—find pathname of terminal

```
#include <unistd.h>

char *ttyname(
    int fd             /* file descriptor */
);
/* Returns string or NULL on error (sets errno) */
```

ttyname_r—find pathname of terminal

```
#include <unistd.h>

int ttyname_r(
    int fd,            /* file descriptor */
    char *buf,         /* buffer for pathname */
    size_t bufsize     /* size of buffer */
);
/* Returns 0 on success or error number on error (errno not set) */
```

As we've seen with other functions, the _r suffix means that ttyname_r is reentrant—it uses no static storage but rather the buffer you pass in. And, as we've seen with other functions, it's a pain in the neck to size that buffer: You can use the macro TTY_NAME_MAX if it's defined (in limits.h), and if it isn't you have to call sysconf (Section 1.5.5) with the argument _SC_TTY_NAME_MAX. Or, you can just call sysconf all the time. If you know that only one thread is calling ttyname, it's the easier of the two to use by far.

Here's my version of the tty command, which prints the name of the terminal connected to the standard input or "not a tty" if it's not a terminal:

```
int main(void)
{
    char *ctty;

    if ((ctty = ttyname(STDIN_FILENO)) == NULL) {
        printf("not a tty\n");
        exit(1);
    }
    printf("%s\n", ctty);
    exit(0);
}
```

Here I used the numbers 1 and 0 instead of the symbols EXIT_FAILURE and EXIT_SUCCESS because the SUS explicitly says that the return code shall be 1 or 0. (POSIX specifies only that EXIT_FAILURE be nonzero, not that it be 1.)

If you really want the name of the controlling terminal, even if the standard input is somehow opened to a different terminal or a nonterminal input, you might think that you could open /dev/tty and pass that file descriptor to ttyname. But, alas, it doesn't work that way on the systems where I tried it—ttyname just returns /dev/tty instead of the actual name. In fact, there seems to be no portable way to get the name of the controlling terminal, although if the standard input is a terminal, it's probably the controlling terminal.

You can test whether a file descriptor is opened to a terminal with the isatty system call:

isatty—test for terminal

```
#include <unistd.h>

int isatty(
    int fd                /* file descriptor */
);
/* Returns 1 if a terminal and 0 if not (may set errno on 0 return) */
```

You can't depend on `errno` being set if `isatty` returns 0, but in most cases you don't care about the reason if it returns anything other than 1.

4.8 Full-Screen Applications

Traditional UNIX commands are sometimes interactive (e.g., `dc`, `more`), but they still read and write text lines. There are at least three other interesting categories of applications that more fully exploit the capabilities of display screens and their associated input devices (e.g., mice):

- Character-oriented full-screen applications that run on cheap terminals (hardly made anymore) and terminal emulators such as `telnet` and `xterm`. The best-known such application is probably the `vi` text editor.
- Graphical-User-Interface (GUI) applications written for the X Window System, perhaps using a higher-level toolkit such as Motif, Gnome, KDE, or even Tcl/Tk. (There are also some other GUI systems besides X.)
- Web applications, in which the user interface is written with technologies such as HTML, JavaScript, and Java.

Only the first group makes use of the terminal functionality that's the subject of this chapter. X applications talk to the so-called X server over a network, and the GUI processing actually occurs in the server, which may or may not be running on a UNIX-based computer. If it is, it talks directly to the display and input devices via their device drivers, not through the character-terminal driver. Similarly, Web applications communicate with the browser over a network (Section 8.4.3), and the browser handles the user interaction. If the browser is running on UNIX, it's often an X application.

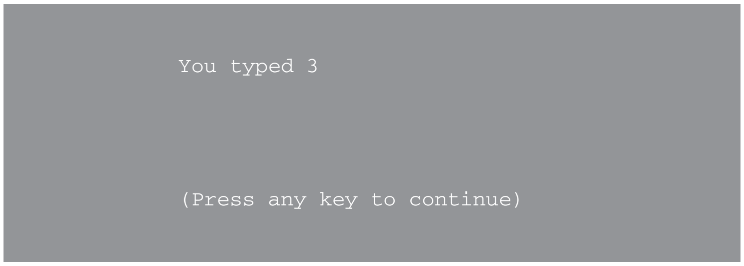
I'm going to present an example of a very simple character-oriented full-screen application that clears the screen and puts up a menu that looks like this:

```
What do you want to do?
1. Check out tape/DVD
2. Reserve tape/DVD
3. Register new member
4. Search for title/actor
5. Quit

(Type item number to continue)
```

Figure 4.3 Menu.

The number that the user types isn't echoed and doesn't have to be followed by a return. In my little example, all you get if you type, say, 3, is the screen in Figure 4.4.



```
You typed 3



(Press any key to continue)
```

Figure 4.4 Response.

I'll show two implementations of this application. The first is just for ANSI terminals and works fine on anything that can emulate a VT100 terminal. ANSI terminals (and VT100s) have dozens of control sequences, of which we're only going to use two:

- The sequence **`ε[r;cH`** positions the cursor to row r, column c. (ε is the escape character.)
- The sequence **`ε[2J`** clears the screen.

Two functions use these control sequences; note that `clear` also moves the cursor to the upper-left of the screen:

```
#define ESC "\033"

bool mvaddstr(int y, int x, const char *str)
{
    return printf(ESC "[%d;%dH%s", y, x, str) >= 0;
}

bool clear(void)
{
    return printf(ESC "[2J") >= 0 &&
      mvaddstr(0, 0, "");
}
```

We want to ring the terminal's bell if the user types a wrong character, and we can use an ASCII code for that:

```
#define BEL "\007"

int beep(void)
{
    return printf(BEL) >= 0;
}
```

Assuming the terminal is in raw mode from a call to `tc_setraw`, we call `getch` to read a character:

```
int getch(void)
{
    char c;

    switch(read(STDIN_FILENO, &c, 1)) {
    default:
        errno = 0;
        /* fall through */
    case -1:
        return -1;
    case 1:
        break;
    }
    return c;
}
```

Those are all the user-interface service routines we need. The whole application is in the `main` function. Obviously, if it did anything it would be much longer.

```
int main(void)
{
    int c;
    char s[100];
    bool ok = false;

    ec_false( tc_setraw() )
    setbuf(stdout, NULL);
    while (true) {
        ec_false( clear() )
        ec_false( mvaddstr(2, 9, "What do you want to do?") )
        ec_false( mvaddstr(3, 9, "1. Check out tape/DVD") )
        ec_false( mvaddstr(4, 9, "2. Reserve tape/DVD") )
        ec_false( mvaddstr(5, 9, "3. Register new member") )
        ec_false( mvaddstr(6, 9, "4. Search for title/actor") )
        ec_false( mvaddstr(7, 9, "5. Quit") )
        ec_false( mvaddstr(9, 9, "(Type item number to continue)") )
        ec_neg1( c = getch() )
        switch (c) {
        case '1':
        case '2':
        case '3':
        case '4':
            ec_false( clear() )
            snprintf(s, sizeof(s), "You typed %c", c);
            ec_false( mvaddstr(4, 9, s) )
            ec_false( mvaddstr(9, 9, "(Press any key to continue)") )
            ec_neg1( getch() )
            break;
        case '5':
            ok = true;
            EC_CLEANUP
        default:
            ec_false( beep() )
        }
    }

EC_CLEANUP_BGN
    (void)tc_restore();
    (void)clear();
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

You probably don't want to code for just one kind of terminal, no matter how ubiquitous. Instead, you'll want to use a standardized library called Curses, which was introduced in Section 4.1. It keeps a database of the control-sequences for every terminal ever made so it can switch to the appropriate control sequence at run-time.

I cleverly named the functions `mvaddstr`, `clear`, `beep`, and `getch` after Curses functions of the same names, which do the same thing. So all we have to show for the Curses version is the `main` function:

```
#include <curses.h>

/* "ec" macro for ERR (used by Curses) */
#define ec_ERR(x) ec_cmp(x, ERR)

int main(void)
{
    int c;
    char s[100];
    bool ok = false;
    (void)initscr();
    ec_ERR( raw() )
    while (true) {
        ec_ERR( clear() )
        ec_ERR( mvaddstr( 2, 9, "What do you want to do?") )
        ec_ERR( mvaddstr( 3, 9, "1. Check out tape/DVD") )
        ec_ERR( mvaddstr( 4, 9, "2. Reserve tape/DVD") )
        ec_ERR( mvaddstr( 5, 9, "3. Register new member") )
        ec_ERR( mvaddstr( 6, 9, "4. Search for title/actor") )
        ec_ERR( mvaddstr( 7, 9, "5. Quit") )
        ec_ERR( mvaddstr( 9, 9, "(Type item number to continue)") )
        ec_ERR( c = getch() )
        switch (c) {
        case '1':
        case '2':
        case '3':
        case '4':
            ec_ERR( clear() )
            snprintf(s, sizeof(s), "You typed %c", c);
            ec_ERR( mvaddstr( 4, 9, s) )
            ec_ERR( mvaddstr( 9, 9, "(Press any key to continue)") )
            ec_ERR( getch() )
            break;
        case '5':
            ok = true;
            EC_CLEANUP
        default:
            ec_ERR( beep() )
        }
    }

EC_CLEANUP_BGN
    (void)clear();
    (void)refresh();
    (void)endwin();
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

Some comments on this program:

- Most Curses functions return `ERR` on an error, so I defined an `ec_ERR` macro just for them. The documentation for Curses doesn't say that it uses `errno`, but I left it alone, knowing that if I do get a message its value may be meaningless.
- Curses requires that you begin with `initscr` and end with `endwin`.
- The Curses version of `tc_setraw` is `raw`.

I won't cover any more of Curses in this book, as it's not a kernel service.[10] Curses is part of SUS2 and SUS3, so you can read the specification online at the SUS2 site at *www.unix-systems.org/version2/online.html*. On most systems you can also type `man curses` or `man ncurses` (a free version of Curses).

4.9 STREAMS I/O

STREAMS are a mechanism on some implementations of UNIX that allow character device drivers to be implemented in a modular fashion. Applications can, to some extent, chain various STREAMS modules together to get the kind of driver capabilities they want. It's somewhat analogous to the way UNIX filters work at the shell level; for example, the `who` command may not have an option to sort its output, but you can pipe it into `sort` if that's what you want.

I'll show an example of STREAMS-based pseudo terminals in the next section. What I'll do there is open a device file to get a pseudo terminal and then use `ioctl` to "push" two STREAMS modules onto it (`ldterm` and `ptem`) to make it behave like a terminal. Unlike with shell pipelines, where you have dozens of filters to choose from that can be combined hundreds of creative ways, STREAMS modules are usually used in cookbook fashion—you push the modules the documentation tells you to push, and, as with cooking, you won't get anything delicious if you just push things willy-nilly. (STREAMS should not be confused with the standard I/O concept of streams, used by `fopen`, `fwrite`, etc.. The two are totally unrelated.)

The STREAMS feature was required by SUS1 and SUS2, but not by SUS3. Linux, which sets `_XOPEN_VERSION` to 500 (indicating it's SUS2), doesn't provide STREAMS, which makes it nonconforming in that sense.

Aside from my use of STREAMS for pseudo terminals in the next section and a few other mentions in this book, I won't provide a full treatment of STREAMS. The easiest way to find out more about them is from the SUS or from Sun's site at *www.sun.com,* where you can find the *STREAMS Programming Guide*.

10. Besides, if we included Curses, we'd have to include X, and that would kill us. (In this context, "us" means "you and me.")

4.10 Pseudo Terminals

A normal terminal device driver connects a process to an actual terminal, as shown in Figure 4.5a, where the user at the terminal is running `vi`. A *pseudo terminal* device driver acts just like a terminal as far as the interactive (*slave*) process (`vi`) is concerned, but the other end is connected to a *master* process, not to an actual device. This allows the master process to feed input to the slave process as though it were coming from an actual terminal, and to capture the output from the slave process as though it were going to an actual terminal. The slave process can use `tcgetattr`, `tcsetattr`, and other terminal-only system calls just as it normally does, and they will all work exactly as it expects them to.

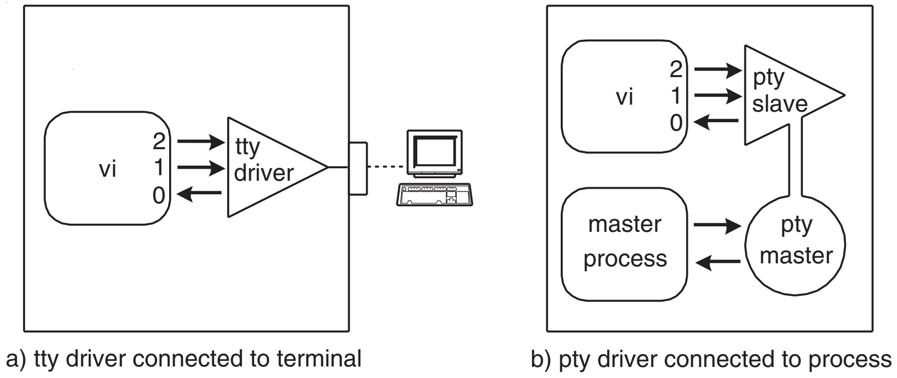

vi
2
1
0
tty
driver
pty
slave
master
process
pty
master
a) tty driver connected to terminal
b) pty driver connected to process

Figure 4.5 Terminal driver vs. pseudo-terminal driver.

You can think of connecting a pseudo terminal to a process as similar in concept to redirecting its input and output with the shell but doing it with a kind of terminal instead of with pipes. Indeed, `vi` won't work with pipes:

```
$ vi >tmp
ex/vi: Vi's standard input and output must be a terminal
```

Perhaps the most common use of pseudo terminals is by the `telnetd` server as shown in Figure 4.6. It communicates with the `telnet` client over a network and with whatever the `telnet` user is doing via a pseudo terminal. In practice, `telnet` sessions usually start with a shell, just as actual-terminal sessions; in the

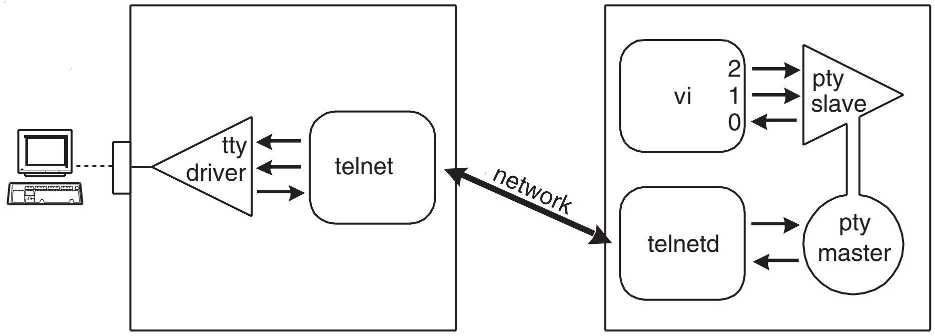


Figure 4.6 Pseudo terminal used by `telnet` server (`telnetd`).

picture `vi` is running because the user typed the `vi` command to the shell. In fact, these days, using `telnet` or `xterm` is far more common than using an actual terminal, as most users connect to a UNIX computer over a network, not by dialing it up directly. (They may dial up their Internet ISP, but that's just to get them onto the Internet.) I won't show how to implement a `telnetd` server (see Exercise 4.6); we're just concerned with the pseudo-terminal part of the picture.

On the pseudo-terminal master side, the master process gets whatever the slave process is writing. In the case of `vi` and other screen-oriented programs, this data includes escape sequences (e.g., to clear the screen), as we saw in Section 4.8. Neither the terminal driver nor the pseudo-terminal driver care about what those sequences are. On the other hand, control operations such as those done with `tcsetattr` are handled entirely by the drivers and don't get to the master process. So, in Figure 4.6, the escape sequences generated by `vi` get all the way to the `telnet` process running on the computer at the left. If it's running on the full screen, it can just send them straight to the actual terminal. It's even more common for the `telnet` process not to be connected to a terminal driver at all but to run under a window system. In this case it has to know how to translate the escape sequences to whatever the window system requires to display characters in a window. Such programs are called *terminal emulators*. So remember that terminal emulators and pseudo terminals are two very different things: the first processes escape sequences, and the second replaces the terminal driver so as to reroute the I/O to a master process.

4.10.1 Pseudo-Terminal Library

As this section uses a few system calls (e.g., `fork`*) from Chapter 5, you may want to read this section after you've read that chapter.*

Pseudo terminals (called "ptys" from now on), like all other devices, are represented by special files whose names vary from system to system. Unfortunately, connecting a process to a pty isn't nearly as straightforward as just doing something like this:

```
$ vi </dev/pty01 >/dev/pty01
```

It's more complicated: We have to get the name of the pty, execute some system calls to prepare it, and make it the controlling terminal. Some of the system calls involved are standardized by SUS1 (Section 1.5.1) and later standards, but pre-SUS systems like FreeBSD do things a completely different way. Worse, systems that use STREAMS to implement ptys require some extra steps, and there were some changes between SUS2 and SUS3.

I'm going to encapsulate the differences between the systems into a little library of functions, each of which starts with `pt_`. Then I'll use this pt-library to implement a record/playback application as an example.

Here's the overall scheme for using ptys presented as nine steps:

1. Open the master side of the pty for reading and writing.
2. Get access to the pty (explained below).
3. From the name or file descriptor of the master side, get the name of the slave side. Don't open it yet.
4. Execute the `fork` system call (Section 5.5) to create a child process.
5. In the child, call `setsid` (Section 4.3.2) to make it relinquish its controlling terminal.
6. In the child, open the slave side of the pty. It will become the new controlling terminal. On systems that support STREAMS (e.g., Solaris), you have to set up the STREAM. On BSD systems, you have to explicitly make it the controlling terminal with:

   ```
   ec_neg1( ioctl(fd, TIOCSCTTY) )
   ```

7. Redirect the child's standard input, output, and error file descriptors to the pty.

8. Execute the `execvp` system call (Section 5.3) to make the child run the desired program (e.g., `vi`). (Any of the six variants of the "exec" system call will work; `execvp` is the one I use in my example.)
9. Now the parent can read and write the master side of the pty using the file descriptor it got in step 1. The child will read from its standard input and write to its standard output and standard error output as it normally would (it knows nothing about the context in which it was executed), and those file descriptors will act as though they are open to a terminal device.

All of this plumbing is shown in Figure 4.5b.

There are three methods used to open the master side of a pty (step 1):

A. On a SUS3 system you just call `posix_openpt` (see below).

B. On most SUS1 and SUS2 systems (including Solaris and Linux), you open the clone file /dev/ptmx, and that will provide you with a unique pty, although you won't know the actual file name, if there even is one. That's OK, as all you need is the open file descriptor.

C. On BSD-based systems, there are a collection of special files whose names are of the form /dev/ptyXY, where X and Y are a digit or letter. You have to try them all until you find one you can open. (No kidding!)

We can use the `_XOPEN_VERSION` macro to distinguish between methods A and B, and we'll define the macro `MASTER_NAME_SEARCH` for those systems that use method C. In step 6 (of the 9 steps listed previously), we'll define the macro `NEED_STREAM_SETUP` for systems that need the STREAM to be set up, and the macro `NEED_TIOCSCTTY` for those systems that need the `ioctl` call. Here's the code in the pt-library that sets these macros for Solaris and FreeBSD; Linux doesn't need any of them set:

```
#if defined(SOLARIS) /* add to this as necessary */
#define NEED_STREAM_SETUP
#endif

#if defined(FREEBSD) /* add to this as necessary */
#define NEED_TIOCSCTTY
#endif

#ifndef _XOPEN_UNIX
#define MASTER_NAME_SEARCH
#endif
```

You'll want to augment this code for your own system if it's not already taken into account.

Here's the synopsis for posix_openpt, available only in SUS3-conforming systems:

posix_openpt—open pty

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

int posix_openpt(
    int oflag          /* O_RDWR optionally ORed with O_NOCTTY */
);
/* Returns file descriptor on success or -1 on error (sets errno) */
```

You use the flag O_NOCTTY to prevent the master side from being the controlling terminal, in case the process has no controlling terminal already. (Recall that the first terminal device opened for a process without a controlling terminal becomes the controlling terminal.) We usually don't want the master side to be the controlling terminal, although we will want exactly that for the slave side (step 5), so we will use the O_NOCTTY flag.

Now, getting on with the pt-library code, it needs some includes that aren't in defs.h:

```
#ifdef _XOPEN_UNIX
#include <stropts.h> /* for STREAMS */
#endif
#ifdef NEED_TIOCSCTTY
#include <sys/ttycom.h> /* for TIOCSCTTY */
#endif
```

The pt-library functions all operate on a PTINFO structure that holds file descriptors for the master and slave sides and their path names, if known:

```
#define PT_MAX_NAME 20

typedef struct {
    int pt_fd_m;                    /* master file descriptor */
    int pt_fd_s;                    /* slave file descriptor */
    char pt_name_m[PT_MAX_NAME];    /* master file name */
    char pt_name_s[PT_MAX_NAME];    /* slave file name */
} PTINFO;

#define PT_GET_MASTER_FD(p)  ((p)->pt_fd_m)
#define PT_GET_SLAVE_FD(p)   ((p)->pt_fd_s)
```

The two macros are for use by applications that call the pt-library, since they need the file descriptors.

The function `pt_open_master`, which I'll show shortly, allocates a `PTINFO` structure and returns a pointer to it for use with the other library functions, just as the standard I/O library uses a `FILE` structure. Internally, `pt_open_master` calls `find_and_open_master` to implement step 1, using one of the three methods previously outlined:

```
#if defined(MASTER_NAME_SEARCH)
#define PTY_RANGE \
  "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
#define PTY_PROTO    "/dev/ptyXY"
#define PTY_X        8
#define PTY_Y        9
#define PTY_MS       5 /* replace with 't' to get slave name */
#endif /* MASTER_NAME_SEARCH */

static bool find_and_open_master(PTINFO *p)
{
#if defined(_XOPEN_UNIX)
#if _XOPEN_VERSION >= 600
    p->pt_name_m[0] = '\0'; /* don't know or need name */
    ec_neg1( p->pt_fd_m = posix_openpt(O_RDWR | O_NOCTTY) )
#else
    strcpy(p->pt_name_m, "/dev/ptmx"); /* clone device */
    ec_neg1( p->pt_fd_m = open(p->pt_name_m, O_RDWR | O_NOCTTY) )
#endif
#elif defined(MASTER_NAME_SEARCH)
    int i, j;
    char proto[] = PTY_PROTO;

    if (p->pt_fd_m != -1) {
        (void)close(p->pt_fd_m);
        p->pt_fd_m = -1;
    }
    for (i = 0; i < sizeof(PTY_RANGE) - 1; i++) {
        proto[PTY_X] = PTY_RANGE[i];
        proto[PTY_Y] = PTY_RANGE[0];
        if (access(proto, F_OK) == -1) {
            if (errno == ENOENT)
                continue;
            EC_FAIL
        }
        for (j = 0; j < sizeof(PTY_RANGE) - 1; j++) {
            proto[PTY_Y] = PTY_RANGE[j];
            if ((p->pt_fd_m = open(proto, O_RDWR)) == -1) {
                if (errno == ENOENT)
                    break;
            }
            else {
                strcpy(p->pt_name_m, proto);
                break;
            }
        }
        if (p->pt_fd_m != -1)
            break;
    }
    if (p->pt_fd_m == -1) {
        errno = EAGAIN;
        EC_FAIL
    }
#else
    errno = ENOSYS;
    EC_FAIL
#endif
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Most of the code in `find_and_open_master` is for that silly name search—as many as 3844 names! To speed it up a bit, we use `access` (Section 3.8.1) to test whether each name of the form /dev/ptyX0 exists, and, if not, we don't bother with /dev/ptyX1, /dev/ptyX2, and the other 59 names in that series but just move to the next X.

Step 2 is to get access to the pty. On SUS systems, you have to call `grantpt` to get access to the slave side and `unlockpt` in case the pty was locked:

grantpt—get access to slave side of pty

```
#include <stdlib.h>

int grantpt(
    int fd              /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

unlockpt—unlock pty

```
#include <stdlib.h>

int unlockpt(
    int fd             /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Step 3 is to get the name of the slave side. On SUS1 systems there's a system call for it:

ptsname—get name of slave side of pty

```
#include <stdlib.h>

char *ptsname(
    int fd             /* file descriptor */
);
/* Returns name or NULL on error (errno not defined) */
```

On BSD systems (`MASTER_NAME_SEARCH` defined) you get the name from the name of the master by replacing the "p" in /dev/ptyXY with a "t." That is, if you discovered that /dev/ptyK4 could be opened as the master side, the corresponding slave side's name will be /dev/ttyK4.

Here's the code for the function `pt_open_master` that calls the function `find_and_open_master` shown previously for step 1 and then does steps 2 and 3, ending up with the slave side named but not open:

```
PTINFO *pt_open_master(void)
{
    PTINFO *p = NULL;
    char *s;

    ec_null( p = calloc(1, sizeof(PTINFO)) )
    p->pt_fd_m = -1;
    p->pt_fd_s = -1;
    ec_false( find_and_open_master(p) )
#ifdef _XOPEN_UNIX
    ec_neg1( grantpt(p->pt_fd_m) )
    ec_neg1( unlockpt(p->pt_fd_m) )
    ec_null( s = ptsname(p->pt_fd_m) )
    if (strlen(s) >= PT_MAX_NAME) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->pt_name_s, s);
#elif defined(MASTER_NAME_SEARCH)
    strcpy(p->pt_name_s, p->pt_name_m);
    p->pt_name_s[PTY_MS] = 't';
#else
    errno = ENOSYS;
    EC_FAIL
#endif
    return p;

EC_CLEANUP_BGN
    if (p != NULL) {
        (void)close(p->pt_fd_m);
        (void)close(p->pt_fd_s);
        free(p);
    }
    return NULL;
EC_CLEANUP_END
}
```

Step 4, `fork`ing to create a child process, is done by the program that uses the library—there's no library function to do it.

Step 5 is to call `setsid` (Section 4.3.2) in the child to make it relinquish its controlling terminal because we want the pty to be the controlling terminal. Step 6, also done in the child, is to open the slave side, whose name we already have. Those two steps are done by `pt_open_slave`:

```
bool pt_open_slave(PTINFO *p)
{
    ec_neg1( setsid() )
    if (p->pt_fd_s != -1)
        ec_neg1( close(p->pt_fd_s) )
    ec_neg1( p->pt_fd_s = open(p->pt_name_s, O_RDWR) )
#if defined(NEED_TIOCSCTTY)
    ec_neg1( ioctl(p->pt_fd_s, TIOCSCTTY, 0) )
#endif
#if defined(NEED_STREAM_SETUP)
    ec_neg1( ioctl(p->pt_fd_s, I_PUSH, "ptem") )
    ec_neg1( ioctl(p->pt_fd_s, I_PUSH, "ldterm") )
#endif
    /*
        Changing mode not that important, so don't fail if it doesn't
        work only because we're not superuser.
    */
    if (fchmod(p->pt_fd_s, PERM_FILE) == -1 && errno != EPERM)
        EC_FAIL
    return true;
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Again, `pt_open_slave` must be called in the child, not in the parent. (We'll see an example shortly.) Systems for which we defined `NEED_STREAM_SETUP`, such as Solaris, require two modules to be pushed onto the stream: `ptem`, the pseudo-terminal emulator, and `ldterm`, the normal terminal module.[11] (The `ioctl` system calls and the `I_PUSH` command are standardized but not the module names that are used to set up a pty.)

At the end of `pt_open_slave` we change the mode of the pty, since on some systems it may not have suitable permissions. However, as it may not be owned by the user-ID of the process (that's system-dependent), the `fchmod` call (Section 3.7.1) may fail, in which case we proceed anyway hoping for the best.

Steps 7, 8, and 9 aren't done with the pt-library but with system calls introduced in Chapters 5 and 6. I'll show the code in the example that appears later in this section, however.

As the child (slave) and parent (master) processes are running independently, the parent has to make sure the child is completely set up before starting to read and write the pty in step 9. So, the pt-library has a call for the parent's use that doesn't return until it's safe to proceed:

```
bool pt_wait_master(PTINFO *p)
{
    fd_set fd_set_write;

    FD_ZERO(&fd_set_write);
    FD_SET(PT_GET_MASTER_FD(p), &fd_set_write);
    ec_neg1( select(PT_GET_MASTER_FD(p) + 1, NULL, &fd_set_write, NULL,
      NULL) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

11. To read about these on a Solaris system (and perhaps others that implement STREAMS), run `man ptem` and `man ldterm`.

All we're doing is using `select` (Section 4.2.3) to wait until the pty is writable.

The next function is `pt_close_master`, called in the parent to close the file descriptors when the pty is no longer needed and to free the `PTINFO` structure.

```
bool pt_close_master(PTINFO *p)
{
    ec_neg1( close(p->pt_fd_m) )
    free(p);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

There's also a `pt_close_slave`, but it's usually not called because the child process is overlaid by another program in step 8. Also, since we redirected file descriptors, the child's end of the pty is by this time represented by the standard file descriptors (0, 1, and 2), and those are almost always closed automatically when the process terminates. Here it is anyway:

```
bool pt_close_slave(PTINFO *p)
{
    (void)close(p->pt_fd_s); /* probably already closed */
    free(p);
    return true;
}
```

If you find yourself calling `pt_close_slave` in your code, you're probably doing something wrong.

Note that both `pt_close_master` and `pt_close_slave` free the `PTINFO` structure. That's the way it should be because when they are executed the structure is allocated in both parent and child. This will be more clear when `fork` is explained in Section 5.5.

Putting it all together, here's the overall framework for using the pt-library calls:

```
    PTINFO *p = NULL;
    bool ok = false;

    ec_null( p = pt_open_master() ) /* Steps 1, 2, and 3 */
    switch (fork()) { /* Step 4 */
    case 0:
        ec_false( pt_open_slave(p) ) /* Steps 5 and 6 */
        /*
            Redirect fds and exec (not shown) - steps 7 and 8
        */
```

```
            break;
        case -1:
            EC_FAIL
        }
        ec_false( pt_wait_master(p) ) /* Synchronize before step 9 */
        /*
            Parent (master) now reads and writes pty as desired - step 9
        */
        ok = true;
        EC_CLEANUP

    EC_CLEANUP_BGN
        if (p != NULL)
            (void)pt_close_master(p);
        /*
            Other clean-up goes here
        */
        exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
    EC_CLEANUP_END
```

This is what the application I'm about to present will look like.

4.10.2 Record and Playback Example

I'm going to use a pty to build a record/playback system that can record the output of a command and then play the recorded output back just as if the command were running. This means not only that the screen has to be written identically to the original but that the playback takes place at the same speed. To see what I mean, here's a script that shows the time, waits 5 seconds, and shows it again:

```
$ cat >script1
date
sleep 5
date
$ chmod +x script1
$ script1
Wed Nov  6 10:46:31 MST 2002 [5 second pause]
Wed Nov  6 10:46:36 MST 2002
```

Of course, as you can see from the times, there was a 5-second pause between the first and second executions of `date`. (The comment in brackets isn't part of the output.)

But if we just capture the output on a file and display that file, the text in the file is displayed very rapidly. Sure, the times in the file were exactly what the `date` command wrote, but there's no pause when the `cat` command prints the output:

```
$ script1 >tmp                  [5 second pause]
$ cat tmp
Wed Nov  6 10:55:24 MST 2002    [no pause]
Wed Nov  6 10:55:29 MST 2002
```

However, this is not what we want. We want to record the output so it plays back at the same speed as the recording, with a command we're going to show called `record` (its `-p` option plays the recording back):

```
$ record script1
Wed Nov  6 10:57:18 MST 2002    [5 second pause]
Wed Nov  6 10:57:23 MST 2002
$ record -p
Wed Nov  6 10:57:18 MST 2002    [5 second pause]
Wed Nov  6 10:57:23 MST 2002
```

You can't see it on the page, but after displaying the first line, there was a 5 second pause before displaying the second, which was the same speed at which the output was recorded.

So, at a minimum, what `record` needs to do is save *events:* the time along with the characters so the playback part will know how fast to write them to the standard output. We want it to work with interactive programs, too, so it can't just capture the output with a pipe—it has to set up the command with a pty because interesting commands, like `vi`, `emacs`, and other full-screen applications, will work only on terminals. How the recorder, the recording, and the command being recorded are all connected as shown in Figure 4.7. Whatever `record` reads from its standard input it writes to the pty, and whatever it reads from the pty it writes both to its standard output *and* to a file of events.

First, I'll show the data structures and functions that record the output from the command as a series of events. Then, I'll show the playback functions that read

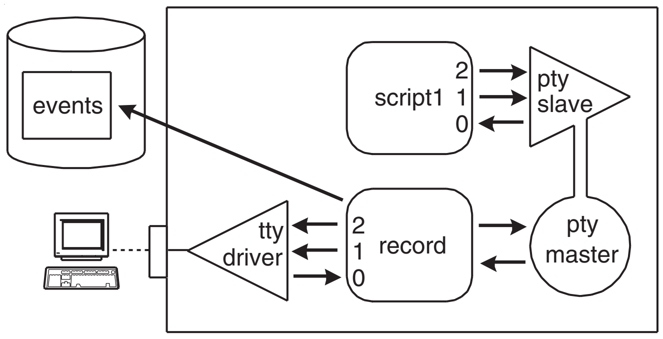


Figure 4.7 Recording interaction with `script1`.

the events and display the data in them at the appropriate times. Playback doesn't involve ptys, since the command isn't actually running. Finally, I'll show the recorder, which follows the framework for using the pt-library that I showed in the previous section.

Each time the process to be recorded writes some data, the master side of the pty reads it and treats it as one event. It stores the relative time since the recording began and the length of the data in an `event` structure. Then it writes the structure and the data itself to a file named recording.tmp. Figure 4.8 shows three such events in the file.

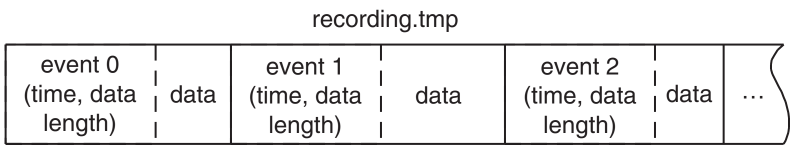

recording.tmp
event 0 (time, data length)
data
event 1 (time, data length)
data
event 2 (time, data length)
data
...

Figure 4.8 Layout of recording file.

This is the `event` structure along with a global file descriptor and macros for the file name and a buffer size for writing and reading data:

```
#define EVFILE "recording.tmp"
#define EVBUFSIZE 512

struct event {
    struct timeval e_time;
    unsigned e_datalen;
};
static int fd_ev = -1;
```

The `timeval` structure was explained in Section 1.7.1.

The recorder uses `ev_creat` to open the recording file (creating it if necessary), `ev_write` to write an event to it, and `ev_close` to close it:

```
static bool ev_creat(void)
{
    ec_neg1( fd_ev = open(EVFILE, O_WRONLY | O_CREAT | O_TRUNC,
      PERM_FILE) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

```
static bool ev_write(char *data, unsigned datalen)
{
    struct event ev = { { 0 } };

    get_rel_time(&ev.e_time);
    ev.e_datalen = datalen;
    ec_neg1( writeall(fd_ev, &ev, sizeof(ev)) )
    ec_neg1( writeall(fd_ev, data, datalen) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void ev_close(void)
{
    (void)close(fd_ev);
    fd_ev = -1;
}
```

`writeall` is a simple way of writing a full buffer, even if it takes multiple calls to `write`; it's from Section 2.9.

The function `get_rel_time` does the work of calling `gettimeofday` (Section 1.7.1) to get the current time and subtracting it from the starting time (which it saves on its first call) using another function, `timeval_subtract`:

```
static void get_rel_time(struct timeval *tv_rel)
{
    static bool first = true;
    static struct timeval starttime;
    struct timeval tv;

    if (first) {
        first = false;
        (void)gettimeofday(&starttime, NULL);
    }
    (void)gettimeofday(&tv, NULL);
    timeval_subtract(&tv, &starttime, tv_rel);
}

static void timeval_subtract(const struct timeval *x,
  const struct timeval *y, struct timeval *diff)
{
    if (x->tv_sec == y->tv_sec || x->tv_usec >= y->tv_usec) {
        diff->tv_sec = x->tv_sec - y->tv_sec;
        diff->tv_usec = x->tv_usec - y->tv_usec;
    }
    else {
        diff->tv_sec = x->tv_sec - 1 - y->tv_sec;
        diff->tv_usec = 1000000 + x->tv_usec - y->tv_usec;
    }
}
```

Note that `timeval_subtract` assumes that `x` >= `y`.

Those are all the service functions needed to make a recording. Playing one back needs functions for opening the recording file and reading the `event` structure and the data that follows it in the file:

```
static bool ev_open(void)
{
    ec_neg1( fd_ev = open(EVFILE, O_RDONLY) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool ev_read(struct event *ev, char *data, unsigned datalen)
{
    ssize_t nread;

    ec_neg1( nread = read(fd_ev, ev, sizeof(*ev)) )
    if (nread != sizeof(*ev)) {
        errno = EIO;
        EC_FAIL
    }
    ec_neg1( nread = read(fd_ev, data, ev->e_datalen) )
    if (nread != ev->e_datalen) {
        errno = EIO;
        EC_FAIL
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

One more thing: Once the player has an event, it needs to wait for the appropriate time before displaying it. This is the principal difference between real-time playback and just dumping the data out. The function `ev_sleep` takes a time from an `event` structure, calculates how much time to wait, and then sleeps for that interval. As a `timeval` stores the time in seconds and microseconds, we use the

Standard C function `sleep` for the seconds and the `usleep` system call (Section 9.7.3) for the microseconds.

```
static bool ev_sleep(struct timeval *tv)
{
    struct timeval tv_rel, tv_diff;

    get_rel_time(&tv_rel);
    if (tv->tv_sec > tv_rel.tv_sec ||
      (tv->tv_sec == tv_rel.tv_sec && tv->tv_usec >= tv_rel.tv_usec)) {
        timeval_subtract(tv, &tv_rel, &tv_diff);
        (void)sleep(tv_diff.tv_sec);
        ec_neg1( usleep(tv_diff.tv_usec) )
    }
    /* else we are already running late */
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

That's all we need to record and play back events. Assuming that a recording has been made already, here's the function that plays it back:

```
static bool playback(void)
{
    bool ok = false;
    struct event ev;
    char buf[EVBUFSIZE];
    struct termios tbuf, tbufsave;

    ec_neg1( tcgetattr(STDIN_FILENO, &tbuf) )
    tbufsave = tbuf;
    tbuf.c_lflag &= ~ECHO;
    ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
    ec_false( ev_open() )
    while (true) {
        ec_false( ev_read(&ev, buf, sizeof(buf)) )
        if (ev.e_datalen == 0)
            break;
        ev_sleep(&ev.e_time);
        ec_neg1( writeall(STDOUT_FILENO, buf, ev.e_datalen) )
    }
    ec_neg1( write(STDOUT_FILENO, "\n", 1) )
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
```

```
    (void)tcdrain(STDOUT_FILENO);
    (void)sleep(1); /* Give the terminal a chance to respond. */
    (void)tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbufsave);
    ev_close();
    return ok;
EC_CLEANUP_END
}
```

The reading of events and the writing to `STDOUT_FILENO` is pretty straightforward in this function. Certain escape sequences written to a terminal cause it to respond, however, and while we don't care about the response during playback (whatever it might have meant has already been captured in the recording), we do have to turn off echo to make sure the response doesn't mess up the output. At the end, we call `tcdrain` to get the last of the output out, wait a second for the terminal to process it, and then flush any remaining input when we restore the terminal's attributes. The functions `tcgetattr`, `tcsetattr`, and `tcdrain` are from Sections 4.5.1 and 4.6.

Now we can use the pt-library from the previous section along with the "ev" routines to make a recording. (You might want to review the framework at the end of the previous section.) I'll present the `main` function in pieces; it starts like this:

```
int main(int argc, char *argv[])
{
    bool ok = false;
    PTINFO *p = NULL;

    if (argc < 2) {
        fprintf(stderr, "Usage: record cmd ...\n       record -p\n");
        exit(EXIT_FAILURE);
    }
    if (strcmp(argv[1], "-p") == 0) {
        playback();
        ok = true;
        EC_CLEANUP
    }
```

If it's called with the `-p` option, it just calls the `playback` function I already showed and exits. Otherwise, for recording, it follows the framework:

```
    ec_null( p = pt_open_master() )
    switch (fork()) {
    case 0:
        ec_false( pt_open_slave(p) )
        ec_false( exec_redirected(argv[1], &argv[1], PT_GET_SLAVE_FD(p),
          PT_GET_SLAVE_FD(p), PT_GET_SLAVE_FD(p)) )
        break;
```

```
case -1:
    EC_FAIL
}
ec_false( ev_creat() )
ec_false( pt_wait_master(p) )
```

I put the stuff having to do with redirecting file descriptors and executing a command in the function `exec_redirected` because I'm going to explain it all in Chapters 5 and 6. I will show the code for `exec_redirected`, however, without explaining it. You can come back to it after reading those chapters if you need to.

The code so far in `main` takes us to step 9 (explained in the previous section), which is when we begin reading and writing the pty's master-side file descriptor. Bear in mind that the subject command (`script1`, `vi`, or whatever) is already running and is probably blocked in a `read` waiting for some input from what it thinks is a terminal. Here's the rest of the `main` function:

```
tc_setraw();
while (true) {
    fd_set fd_set_read;
    char buf[EVBUFSIZE];
    ssize_t nread;

    FD_ZERO(&fd_set_read);
    FD_SET(STDIN_FILENO, &fd_set_read);
    FD_SET(PT_GET_MASTER_FD(p), &fd_set_read);
    ec_neg1( select(FD_SETSIZE, &fd_set_read, NULL, NULL, NULL) )
    if (FD_ISSET(STDIN_FILENO, &fd_set_read)) {
        ec_neg1( nread = read(STDIN_FILENO, &buf, sizeof(buf)) )
        ec_neg1( writeall(PT_GET_MASTER_FD(p), buf, nread) )
    }
    if (FD_ISSET(PT_GET_MASTER_FD(p), &fd_set_read)) {
        if ((nread = read(PT_GET_MASTER_FD(p), &buf,
          sizeof(buf))) > 0) {
            ec_false( ev_write(buf, nread) )
            ec_neg1( writeall(STDOUT_FILENO, buf, nread) )
        }
        else if (nread == 0 || (nread == -1 && errno == EIO))
            break;
        else
            EC_FAIL
    }
}
ec_neg1( ev_write(NULL, 0) )
fprintf(stderr,
  "EOF or error reading stdin or master pseudo-terminal; exiting\n");
ok = true;
```

```
    EC_CLEANUP

EC_CLEANUP_BGN
    if (p != NULL)
        (void)pt_close_master(p);
    tc_restore();
    ev_close();
    printf("\n");
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

Comments on the last part of `main`:

- As `record` is mostly used for screen-oriented commands, we put the terminal in raw mode with `tc_setraw` (in Section 4.5.10). The attributes are restored by `tc_restore` in the clean-up code. We didn't use those functions in `playback` because all we wanted there was to turn off echo.
- `record` has two inputs: what the user types and what the pty sends back, as shown in Figure 4.5 in the previous section. It uses `select` to wait until one of those two inputs has some data, deals with it, and then loops back to the `select`. Note that it sets up `fd_set_read` each time, since `select` changes the bits to report its results.
- We break out of the loop when we get an end-of-file or I/O error from the pty, not from `record`'s standard input. After all, it's up to the subject command to decide when we're done, not us. An end-of-file or I/O error means that the slave side of the pty has all its file descriptors closed; that is, the subject command has terminated. Which indicator we get is implementation dependent, so we take either.

Finally, as promised, here's `exec_redirected`, which you will understand once you've finished Chapter 6:

```
bool exec_redirected(const char *file, char *const argv[], int fd_stdin,
  int fd_stdout, int fd_stderr)
{
    if (fd_stdin != STDIN_FILENO)
        ec_neg1( dup2(fd_stdin, STDIN_FILENO) )
    if (fd_stdout != STDOUT_FILENO)
        ec_neg1( dup2(fd_stdout, STDOUT_FILENO) )
    if (fd_stderr != STDERR_FILENO)
        ec_neg1( dup2(fd_stderr, STDERR_FILENO) )
    if (fd_stdin != STDIN_FILENO)
        (void)close(fd_stdin);
    if (fd_stdout != STDOUT_FILENO)
```

```
        (void)close(fd_stdout);
    if (fd_stderr != STDERR_FILENO)
        (void)close(fd_stderr);
    ec_neg1( execvp(file, argv) )

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

It's hard to really appreciate `record` in action in a book—we'd need a video to do it justice. Try it out for yourself! You'll smile when you see how it plays back the recording with the same timing as the original—almost as though you're watching a ghost at the keyboard.

Exercises

4.1. Change `getln` to return lines ended with an EOF, as discussed in Section 4.2.1.

4.2. Implement `Bfdopen`, as discussed in Section 4.2.1.

4.3. Implement a simplified version of the `stty` command that just prints out the current terminal status.

4.4. Implement a version of the `stty` command that allows the user to specify only the 10 most commonly used operands. You have to decide which 10 these are.

4.5. Implement a simple screen editor. Decide on a small selection of functions that allow reasonable editing but without too many bells and whistles. Keep the text being edited in internal memory. Use Curses (or equivalent) to update the screen and read the keyboard. Write some documentation using the `man`-page macros if you can.

4.6. Implement a `telnetd` server. For full details of what it's supposed to do, see [RFC854]. But, for this exercise, you can implement just enough so that you can connect from another machine that's running `telnet` and, through your server, execute both line-oriented commands and `vi` or `emacs`.