Chapter 4 Control of Disk and Terminal I/O

Concepts Covered

File structure table, open file table, file status flags, auto-appending, device files, terminal devices, device drivers, line discipline, termios structure, terminal settings, canonical mode, non-canonical modes, IOCTLs, fcntl, ttyname, isatty, ctermid, getlogin, gethostname,tcgetattr, tcsetattr,tcflush, tcdrain, ioctl,

4.1 Overview

This chapter begins by examining how a program can exercise some control over the connections that it makes with disk files. It describes the data structures used by the kernel to manage open files, and explains how processes can interact with these files. It then explores device files and how they are used and structured, and device drivers and their roles and structures. From there it looks at block and character device file differences, after which it turns to terminals. The remainder of the chapter is then devoted to controlling terminals.

4.2 Open Files

The programs that we have studied so far operate on disk files, which are files that reside, of course, on disks[1]. You are by now well-acquainted with the general model of file I/O: open the file, access it, and close it. Remember that this model works for any kind of file, not just disk files. Here though, we are interested only in disk files.

In this model of file I/O, opening a file returns a reference to an object that can be used to access the file, either for reading or writing. When you use the `open()` system call, you get a *file descriptor* in return; when you use the `fopen()` C standard I/O library call, you get a `FILE` *pointer* in return. Either way, you are getting a scalar object (i.e., a small integer or a pointer) that is associated with a hidden (kernel) data structure that allows you to access the file. Here, we will explore how much control we can exercise over the way in which the program is connected to the file.

When a process opens a disk file using the `open()` system call, the kernel creates a data structure that represents the connection between the process and the file. The returned file descriptor can be used by the process to access the file. A file descriptor is simply an index into a per-process table[2] that the kernel uses to locate that specific data structure. Different versions of UNIX call this data structure different things, and the data structure may have slightly different members from one UNIX variant to another, but the basic members and purpose of the structure are invariant: its most important member is the *file pointer*, and its purpose is to store the position in the file from which the next operation will take place (whether it is a read or a write.)

[1] This is in contrast to device files, which are very different from disk files.
[2] A "per-process" table is a table of which each process has its own instance.

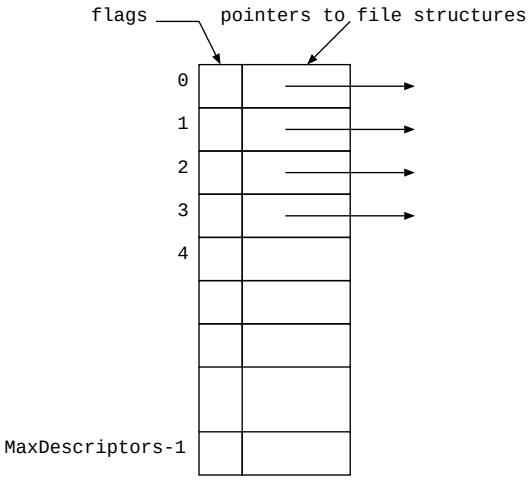

Figure 4.1: Per-process open file table

This data structure is called a *file entry* in BSD UNIX and a *file structure* in Linux. I will call it a *file structure* in these notes. The connection between a process and a file is completely characterized by the information contained in the file structure. Aside from the file pointer, the kinds of information it typically contains include:

- whether or not the file is open for reading, writing, reading and writing, or appending,
- whether or not the I/O is buffered or unbuffered, and
- whether or not the access is exclusive or whether other processes also access the file.

as well as other information that is required by the kernel. The information contained in the file structure characterizes the connection between the open file and the process; it is specific to this single connection. Other processes might have this file open with different attributes. Many of the attributes of the connection can be changed by the process; others cannot. Which can and which cannot, and how is it done? These are the questions we will answer.

Let us begin by examining all of the data structures related to open files. Each process has a table that is usually called the *open file table*. (In 4.4BSD, this table was called the *descriptor table*, and each entry was called a *filedesc* structure.) In Linux, this table is the `fd` array, which is part of a larger structure called the *files_struct*. The file descriptor returned by the `open()` system call is actually an index into the open file table of the process making the call. Recall that every process is given three file descriptors when it is created: 0, 1, and 2, respectively, for standard input, output, and error. These are the first three indices in this table, as shown in Figure 4.1.

When a process issues the `open()` system call, the kernel creates a new file structure and fills the lowest-numbered available slot in the process's open file table with a pointer to that file structure. The file structures for all open files are contained in a table called the *file structure table*. which resides in the address space of the kernel. See Figure 4.2.

The left side of the figure contains the per-process open file tables, one for each process. These tables, although in the virtual address space of the processes, are accessible only by the kernel. The right side of the table contains the file structure table, in kernel memory. The gray region at the bottom is on disk; everything above the gray area is memory-resident. You will notice in the figure

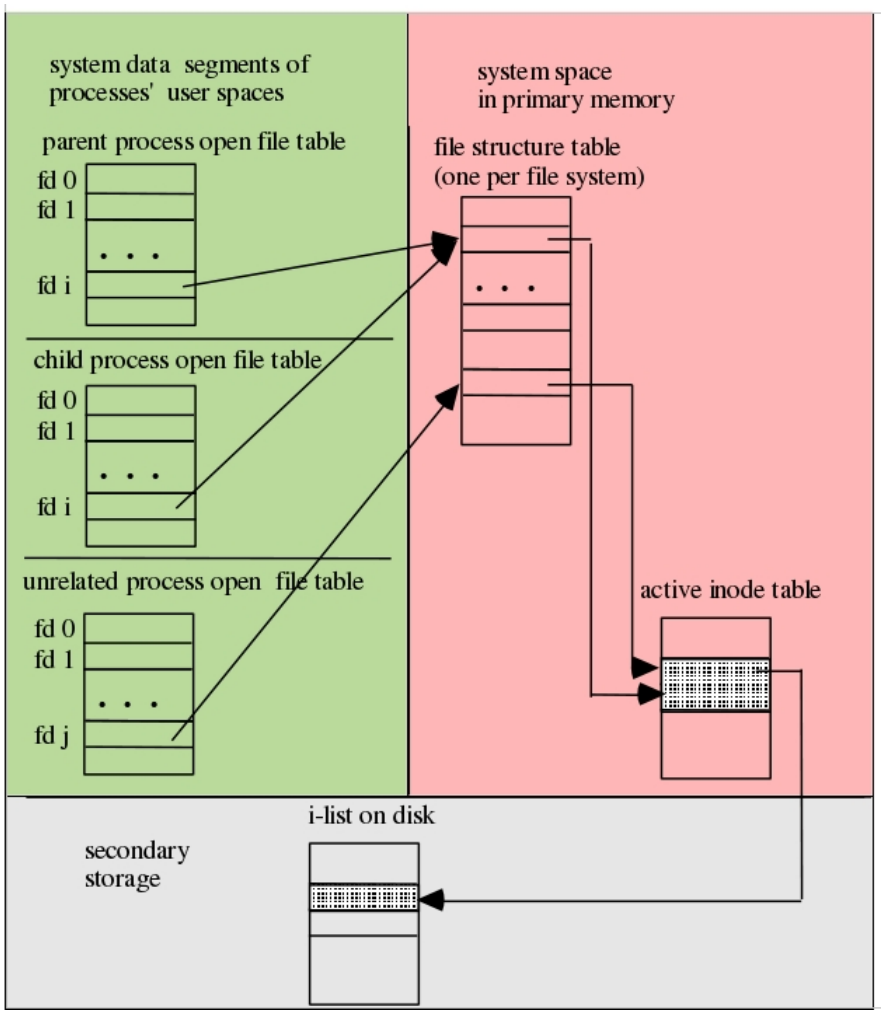

Figure 4.2: Kernel data structures related to open files.

that some processes' file descriptors point to the same file structure for the same open file, whereas others' descriptors point to separate file structures for the same file. Sometimes processes share a connection and other times, even though two or more processes may have the same file open, they access it through different connections. When different processes are connected to the same file through different file structures, the file pointers are different. When a file structure is shared by processes, they share the file pointer. You will see in the chapter on process management how these situations arise.

The file structures in the file structure table point to the active i-nodes for the open files. These i-nodes are maintained in the active i-node table in kernel memory. The active i-node table contains copies of the i-nodes on disk. If the i-nodes change in memory, the changes are written to the i-nodes on disk.

4.2.1 Using `fcntl()` to Control File Descriptor Attributes

The file structure contains a set of flags that control I/O with respect to the file. These flags are called *file status flags*, and they are shared by all processes that share that file structure. The file

descriptor is the means by which the process can modify those flags and alter the behavior of that connection to the file. The method of modifying the flags of an existing file structure is a three-step procedure:

1. The process gets a copy of the current attributes of the connection from the kernel, storing it in its own address space;
2. The process modifies the current attributes in its copy;
3. The process requests the kernel to write its copy back to the kernel's in-memory structures.

The system call that performs steps 1 and 3 is the `fcntl()` call. You can pronounce `fcntl` as "f control", short for file control. `fcntl()` is a function that operates on open files. Depending upon its arguments, `fcntl()` will either get the attributes of a file connection or set them. It has a very long man page that starts as follows:

```
NAME
       fcntl - manipulate file descriptor
SYNOPSIS
       #include <unistd.h>
       #include <fcntl.h>

       int fcntl(int fd, int cmd, ... /* arg */ );


DESCRIPTION
     fcntl performs one of various miscellaneous operations
     on fd.  The operation in question is determined by cmd.
     ...
```

Remarks

- The first parameter is the file descriptor of an already open file.
- The second parameter is an integer that `fcntl()` interprets as a command. Names for these integers are defined in `<fcntl.h>`; the names that are relevant to getting or setting file status flags are:
  - `F_GETFL` which tells `fcntl()` to return a copy of the set of flags;
  - `F_SETFL` which tells `fcntl()` to expect a third integer parameter that contains a new flag set to replace the current one.

  Some of the other commands that `fcntl()` can perform include
  - `F_DUPFD` which duplicates an existing file descriptor
  - `F_GETFD`, `F_SETFD` which get and set *file descriptor flags* (see below)
  - `F_GETOWN`, `F_SETOWN` which get and set the ownership of the `SIGIO` signal (see below)

  - `F_GETSIG, F_SETSIG` which get and set the signal that is sent when using asynchronous I/O
  - `F_GETLK`, `F_SETLK`, `F_SETLKW` which acquire, release, and test for the existence of record locks.

- Each control flag is a single bit in a long integer. To turn on an attribute, you need to set the bit. To turn it off, you need to zero it. The `<fcntl.h>` header file contains definitions of masks that can be used for this purpose. To set a bit, you can do a bitwise-or of the particular mask with the flag variable; to unset it, you can do a bitwise-and of the complement of the mask with the control flag variable. The masks are defined in `/usr/include/bits/fcntl.h`, which is included in the `<fcntl.h>` header file. They are also defined in the man page for `<fcntl.h>`.

File status flags reside in the file structure, which may be shared by multiple processes. *File descriptor flags* are part of the entry in a process's open file table and are associated with the actual file descriptor. At present there is only one such flag, `FD_CLOEXEC`, the *close-on-exec* flag. This flag is not relevant to anything that we cover in this chapter. The `F_GETOWN` and `F_SETOWN` commands will be explained when we cover asynchronous I/O, and we ignore them for now.

Similarly, commands related to record locking (`F_GETLK`, `F_SETLK`, `F_SETLKW`) will be covered when we turn to the topic of file sharing.

Not all file status flags can be changed after a file is opened. For example, if a file is opened for writing with the `O_WRONLY` flag, you cannot use `fcntl()` to change its access mode to reading. The flags that can be changed after a file has already been opened are a subset of the file status flags. The most important of them, and their mnemonic masks are:

`O_APPEND` Append mode. Before each `write()` operation, the file pointer is positioned at the end of the file, as if with `lseek()`, *atomically*. This may not work on remotely mounted file systems. Setting `O_APPEND` is done to eliminate race conditions.

`O_ASYNC` Asynchronous writes. Generate a signal when input or output becomes possible on this file descriptor. This feature is only available for terminals, pseudo-terminals, and sockets, not for disk files!

`O_NONBLOCK` or `O_NDELAY` Non-blocking mode. No subsequent operations on the file descriptor will cause the calling process to wait. This is strictly for *FIFOs* (also known as *named pipes*) and may not have any effect on files other than FIFOs. POSIX specifies the mask `O_NONBLOCK`, but some systems expect `O_NDELAY` instead. Many systems patch their header files so that they are the same. In Linux `O_NONBLOCK` may not be equivalent to `O_NDELAY`. Check the man page for your system.

`O_SYNC` Synchronous I/O. Any writes on the file descriptor will block the calling process until the data has been physically written to the underlying hardware. In Linux this attribute cannot usually be modified by `fcntl()`, and it may or may not be implemented.

From the above descriptions. you can see that there is little that we can actually do through `fcntl()` in Linux for disk files. The `O_ASYNC` flag will be important when we study terminal connections. The `O_NONBLOCK` flag often has no effect. The `O_APPEND` flag is useful, and this is one we will explore. The `O_SYNC` flag turns on synchronous writing. Synchronous writing is writing in which the process

is blocked until the data is actually written to the device, rather than to the kernel buffers. In other words, it turns off kernel buffering for this connection. There are very few reasons why a process should want to do this, as it slows down its execution significantly. If a process wants to force disk writes, it can always use `fsync()` periodically.

Remember that, in order to modify either the file descriptor flags or the file status flags, you cannot just issue an `F_SETFD` or an `F_SETFL` command through `fcntl()`, as this could turn off flag bits that were previously set. Instead you have to follow the three-step procedure outlined above. A typical code sequence to set a flag such as `O_APPEND`, given that `fd` is the file descriptor of a file that is open for writing, is

```
int flags, result;
flags  = fcntl(fd, F_GETFL);
flags |= (O_APPEND);
result = fcntl( fd, F_SETFL,flags);
if ( -1 == result )
    perror("Error setting O_APPEND")
return 0;
```

Notice that the mask is bitwise-or-ed with the flag variable. To turn off a bit, you would bitwise-and the complement of the mask, as in the sequence:

```
int flags, result;
flags  = fcntl(fd, F_GETFL);
flags &= ~(O_APPEND);
result = fcntl( fd, F_SETFL,flags);
if ( -1 == result )
    perror("Error unsetting O_APPEND");
return 0;
```

Although we are limited in which flags we can set in Linux for disk files, we can use `fcntl()` to check the values of all flags. The following function demonstrates how to check the state of various flags. The function requires inclusion of the `<fcntl.h>` header file. The macro `O_ACCMODE` is a mask that can be used to check which of `O_WRONLY`, `O_RDONLY`, or `O_RDWR` is set. These values are stored in the low-order two bits of the integer flagset returned by `fcntl()` and are defined below. These are not independent bits, which is why you need the two-bit mask.

```
#define O_ACCMODE 0003
#define O_RDONLY 00
#define O_WRONLY 01
#define O_RDWR   02
```

We put these ideas together in a function named `check_file_status()` which, when given the file descriptor of an open file, prints its access mode and which status flags are set on that descriptor.

```
int check_file_status ( int fd )
{
    int flags = fcntl ( fd, F_GETFL );
    if ( -1 == flags ) {
        perror ( "Could not get flags.");
        return ( -1 );
    }

    switch ( flags & O_ACCMODE ) {
    case O_WRONLY:
        printf("write-only\n");
        break;
    case O_RDONLY:
        printf("read-only\n");
        break;
    case O_RDWR:
        printf("read-write\n");
        break;
    }

    if ( flags &  O_CREAT )     printf("O_CREAT flag is set\n");
    if ( flags &  O_EXCL )      printf("O_EXCL flag is set\n");
    if ( flags &  O_NOCTTY )    printf("O_NOCTTY flag is set\n");
    if ( flags &  O_TRUNC )     printf("O_TRUNC flag is set\n");
    if ( flags &  O_APPEND )    printf("O_APPEND flag is set\n");
    if ( flags &  O_NONBLOCK )  printf("O_NONBLOCK flag is set\n");
    if ( flags &  O_NDELAY )    printf("O_NDELAY flag is set\n");
#if defined(O_SYNC)
    if ( flags &  O_SYNC )      printf("O_SYNC flag is set\n");
#endif
#if defined(O_FSYNC)
    if ( flags &  O_FSYNC )     printf("O_FSYNC flag is set\n");
#endif
    if ( flags &  O_ASYNC )     printf("O_ASYNC flag is set\n");
    printf("\n");
    return 0;
}
```

As `O_SYNC` and `O_FSYNC` are not necessarily defined on all UNIX systems, the tests for these flags are conditionally compiled. You can embed this function into a main program such as

```
int main(int argc, char *argv[])
{
    int  flags, fd;

    if ( argc != 2 ) {
        printf("usage: %s <descriptor#>\n", argv[0]);
        exit(1);
     }
    if ( ( fd = atoi(argv[1])) < 0 ) {
        printf("usage: %s <descriptor#>\n", argv[0]);
```

```
            exit(1);
        }
    check_file_status(fd);
    return 0;
}
```

and run the program redirecting standard input, output and error. In the process you will discover some surprising results. For example, if the above program is named `checkstatus`, try to predict the output of

```
$ ./checkstatus 1 > out1
$ cat out1
```

and of

```
$ ./checkstatus 0
$ ./checkstatus 2 2>errs
```

The point is to determine, when input or output is redirected, how the status flags for the three standard devices, standard input, output, and error, are changed.

4.2.2 Appending and Race Conditions

The `O_APPEND` flag controls append mode. Consider the following problem. A shared log file is used to record various system activities. Each of several different processes adds its own entry to the end of the log file to record its activity.

Let us make this concrete. Recall that UNIX uses the `wtmp` file to record each and every login and logout. Each login must be recorded at the end of this file when it occurs. When UNIX starts up, `init()` creates a `getty()` process for each terminal, or some other comparable process in the case of network logins, and when a user logs in at a terminal, the `getty()` or other similar process spawns a `login()` process for that terminal. As a consequence, multiple `login()` processes might exist at any time. Imagine that two users login on different terminals at the exact same time. The two `login()` processes each have to add an entry to the end of the table. To add an entry to the end of a file, the file must be opened for writing, using `O_WRONLY`, or reading and writing, using `O_RDWR`. Thus, a process that needs to add a record to the end of the file must perform the following steps:

1. Open the file in read/write mode.
2. Seek to the end of the file.
3. Write a login record at the position obtained by the seek.

In terms of system calls, this would look something like the following:

```
fd = open(_PATH_WTMP, O_RDWR);
// error check the open() and if successful then
// create the wtmp record to write ... and then
lseek(fd, 0, SEEK_END);
write(fd, &record, len);
```

The `lseek()` call obtains the file pointer for the file and sets it to the end of the file at the time the call is made. This pointer is stored in the file structure for the this process's connection to the file. When the process calls `write()`, the data will be written at the position of this file pointer.

Imagine now that two different processes, identified here as `login1` and `login2`, execute this same sequence of instructions independently and simultaneously. For simplicity, suppose that they run on a one-processor machine and they are time-sliced onto the processor. In particular, suppose the processor executes the following sequence of instructions. The leftmost integer represents the current time in some fixed time unit. `login1` executes instructions in the left column; `login2`, in the right.

time	**login1**	**login2**
0	`fd = open(_PATH_WTMP, O_WRONLY);`	
1	`...`	
5	`lseek(fd, 0 SEEK_END);`	
6		`fd = open(_PATH_WTMP, O_WRONLY);`
7		`...`
10		`lseek(fd, 0 SEEK_END);`
11		`write(fd, &record, len);`
12	`write(fd, &record, len);`	

`login1` sets the file pointer at time 5. It is removed from the CPU and `login2` sets the file pointer to the same position as `login1` did, at time 10. It writes to the file at time 11. Then `login1` writes to the same position, overwriting the data just written by `login2`; the record written by `login1` replaces the one just written by `login2`. This is a classic example of a *race condition*.

Race conditions are removed by using some type of mechanism that will allow a sequence of instructions to be executed as an atomic operation, which simply means that once the first instruction is started, no other process can execute any instruction that accesses any of the shared data until the last instruction is completed. UNIX solves this particular problem by providing write connections with an optional *auto-append* mode. When auto-append mode (`O_APPEND`) is enabled, every write operation is preceded immediately and atomically by a seek to the end of the file. This guarantees that each write occurs at the end of the file, regardless of how many other processes are trying to do the same thing simultaneously. Therefore the login and logout processes simply have to open the file in auto-append mode, or enable it after the file is opened with `fcntl()`. The preceding code would be reduced to the following:

```
fd = open(_PATH_WTMP, O_WRONLY | O_APPEND);
// create the record to write ...
write(fd, &record, len);
```

4.2.3 Controlling the Connection When Opening a File

Rather than using `fcntl()` to adjust the attributes of the connection to a previously opened file, one can open the file with the desired attributes in the first place. These attributes can be passed as parameters in the `open()` system call, by bitwise-or-ing them in the second argument to the call. For example, to open a file with name `foobar` with the write-only, auto-append, and synchronous I/O bits set, you would write

```
fd = open(foobar, O_WRONLY | O_APPEND | O_SYNC);
```

You can read the `open()` man page or the `<fcntl.h>` header file for a list of the flags that can be set in the `open()` call. A file must be opened either read-only, write-only, or read-write. Therefore, the bitwise-or must always contain *exactly one* of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. Other flags can be bitwise-or-ed to it. Which flags are appropriate depend upon the mode in which it is opened. In general, flags fall into two categories: *file creation flags* and *file status flags*.

The file creation flags are `O_NOCTTY`, `O_TRUNC`, `O_CREAT`, and `O_EXCL`. These are only relevant when the file is opened in a mode that allows writing, either write-only or read-write. Understanding the `O_NOCTTY` flag requires more knowledge of terminals and processes; this will be explained in the chapter on processes. The semantics of the remaining three are worthy of discussion.

`O_CREAT` When no flags are set, if a file is opened for writing and it does not exist, `open()` will return -1 and fail. The `O_CREAT` flag prevents the failure: when it is set, if a file is opened for writing and it does not exist, `open()` will create it. The ownership of the file is determined by the effective userid of the process, and the file's mode will be whatever mode is specified in the `open()` call, with the umask applied. If the file already exists, this flag has no effect, meaning that the file will be opened with the file pointer set to the start of the file.

`O_EXCL` This flag is intended to be used in conjunction with the `O_CREAT` flag. If a file is opened for writing with the `O_CREAT` flag by itself, and the file exists already, the contents of the file can be overwritten, depending on where the writes occur. To prevent this possibility, the `O_EXCL` can be bitwise-or-ed to the flagset. When both `O_CREAT` and `O_EXCL` are set, if the file exists, the open will fail. In contrast, if the file does not exist, it will be created just as if `O_EXCL` were not set. If this flag is used without `O_CREAT`, the results are the same as if no flags were set.

`O_TRUNC` Without this flag, if a file is opened for writing and it already exists, the contents of the file are not necessarily destroyed; it depends whether `O_APPEND` is set and whether the process seeks to specific places in the file prior to writing. If the program just opens a file for writing and starts writing without seeking and without appending, the file contents will be replaced, but opening for writing does not automatically zero the contents of the file. The purpose of the `O_TRUNC` flag is to force the file to be zeroed before any writes. When it is set, and the `open()` allows writing and the file exists, the file is truncated to zero length. This flag will have no effect unless all of these conditions are true. It will also have no effect if the file is anything but a regular file. If all of `O_TRUNC, O_EXCL,` and `O_CREAT` are set, `O_EXCL` will override this one – if the file exists, `open()` will fail and it if does not exist, it will be created.

The table below summarizes the effects of the possible combinations of these three flags when a file is opened for writing using the call

```
open( "file", O_WRONLY | flags );
```

flags	If the file exists	If the file does not exist
0	opens for writing and sets pointer to first byte	fails
O_CREAT	opens for writing and sets pointer to first byte	creates "file"
O_EXCL	opens for writing and sets pointer to first byte	fails
O_TRUNC	opens for writing and zeroes its contents	fails
O_CREAT | O_EXCL	fails	creates "file"
O_CREAT | O_TRUNC	opens for writing and zeroes its contents	creates "file"
O_TRUNC | O_EXCL	opens for writing and zeroes its contents	fails
O_CREAT | O_TRUNC | O_EXCL	fails	creates "file"

The program `wrflagtest.c` in the demos directory for Chapter 5 can be used to test the effects of all combinations of these flags.

4.3 Device Files

Not only is every physical device in a UNIX system associated with a device special file, but every logical device is as well. Logical devices are devices that exist as abstractions of real physical devices[3]. Although UNIX does not require this, it has been the convention that all of the device files are located in the `/dev` directory.

4.3.1 Naming and Organizing Device Files

Different versions of UNIX use different methods of organizing and naming device files. For example, under Solaris 9, you will find that almost all of the terminal files contained there are symbolic links to files in the directory `/devices/pseudo`, and that the targets of these symbolic links have names such as `pts@0:2`.

In contrast, in Linux 2.6, instead of a `/devices/pseudo` directory there is a directory `/dev/pts`, and all of the device files within `/dev/pts` have small numbers such as 1,2,3, ... that correspond to the terminals of active connections[4].

[3]Sometimes, for example, there may be more than one name for a given physical device, such as a port or a disk. A logical device is another name for that device.

[4]At installation time, Linux can be configured so that its support of pseudo terminal devices is the same as that of systems such as Solaris 9 by using the `CONFIG_UNIX98_PTYS` and `CONFIG_DEVPTS_FS` flags. The Unix98 standard specifies how `/dev/pts` is used for pseudo-terminal support.

Hard disks are represented by device files as well. Their names vary from one system to another. In Linux, the names may look like `/dev/hd1a` or `/dev/hd2c`. If the hard disks are attached via a SCSI or SATA bus, then their device files may have names such as `/dev/sda1`. In Solaris 9, on the other hand, `/dev/dsk/c0t3d0s5` would be a device file for a file system attached to Controller 0 ("`c0`"), at target position 3 ("`t3`") for that controller, on logical unit 0 ("`d0`") since that controller may have several identical units differing only by number, and in partition or "slice" 5 ("`s5`") on that disk. Thus, the device file name reveals much information about the device.

4.3.2 Accessing Devices Via Device Files

All device files support the pertinent system calls. For example, a device such as a magnetic tape that can be read and written will have a device file whose name might be `/dev/rst0` and which will support the `open()`, `read()`, `write()`, `lseek()`, and `close()` system calls. Input-only devices such as mice and keyboards will not support `write()` or `lseek()` since it would make no sense, and similar common sense reasoning applies to all devices in general.

In previous chapters you saw how you could access your terminal using its device file. For example, to write a message to the terminal device `/dev/pts/4` you would write:

```
$echo "Where are you?" > /dev/pts/4
```

If you had permission to write to this terminal, the message "`Where are you?`" would appear on the screen in the corresponding terminal window.

If you do not know the name of the device file for the terminal in which your shell is running, the shell command `tty` will provide it to you. `tty` displays on standard output the absolute pathname of the device file representing the terminal from which the command is issued:

```
$ tty
/dev/pts/4
```

At the programming level, the library function `ttyname()` serves the same purpose, returning the full pathname of the terminal device whose file descriptor is passed to it.

```
NAME
       ttyname  - find pathname of a terminal
SYNOPSIS
       #include <unistd.h>
       char *ttyname(int fd);
DESCRIPTION
     The function ttyname() returns a pointer to the null-
     terminated pathname of the terminal device that is open on
     the file descriptor fd, or NULL on error (for example, if
     fd is not connected to a terminal). The return value may
     point to static data whose content is overwritten by the
     next call.
...
```

To find the name of the terminal device connected to standard input, for instance, a program can call `ttyname(0)`, The following program prints the name of its *control terminal*[5]. The macro `STDIN_FILENO` is defined in `<unistd.h>` and is simply the number 0.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Terminal is %s\n", ttyname(STDIN_FILENO));
    return 0;
}
```

If you name this program `show_tty` and run it, you will see something like:

```
$ show_tty
Terminal is /dev/pts/2
```

On the other hand, try running it as follows:

```
$ ls | show_tty
```

and you will see

```
Terminal is (null)
```

The problem is that in the second example, standard input was redirected, so file descriptor 0 was attached to a pipe instead. The `ttyname()` function returns the `NULL` string when there is no terminal attached to the descriptor. The same thing will happen if you try calling it with file descriptor 1 while standard output has been redirected. You can therefore use `ttyname(0)` as a test for whether or not standard input or output is redirected , as in

```
if ( ttyname(0) )
    // not redirected
else
    // is redirected
```

There is a function specifically for testing whether a file descriptor is attached to a terminal or not :

[5]The *control terminal* for a process is the terminal device from which keyboard-related signals may be generated. For example, if the user presses a Ctrl-C or Ctrl-D on terminal `/dev/pts/2`, all processes that have `/dev/pts/2` as their control terminal will receive this signal.

```
NAME
        isatty - does this descriptor refer to a terminal

SYNOPSIS
       #include <unistd.h>
       int isatty(int desc);

DESCRIPTION
     returns  1 if desc is an open descriptor connected to a
     terminal and 0 otherwise.
```

The `isatty()` function is useful for testing descriptors in this way. We can use it together with `ttyname()` as follows.

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

int main()
{
    if (isatty(0))
            printf("%s\n", ttyname(0));
        else
            printf("not a terminal\n");
    return 0;
}
```

If you run this program you will see something like

```
$ mytty
/dev/pts/1
```

and when input is redirected, you will see this:

```
$ ls | mytty
not a terminal
```

When you are writing a program that expects the standard output device to be a terminal, you can use either `isatty()` or `ttyname()` to check whether any standard stream has been redirected. However, both of these functions can only be used with an open file, since they need a file descriptor, and file descriptors exist only for open connections. In contrast, the `ctermid()` standard I/O library function will always display the pathname of the controlling terminal.

```
NAME
     ctermid  - generate path name for controlling terminal
SYNOPSIS
     #include <stdio.h>
     char *ctermid(char *s);
DESCRIPTION
    The ctermid() function generates the path name of the
    controlling terminal for the current process and stores it
    in a string. If s is a null pointer, the string is stored
    in an internal static area whose address is returned and
    whose contents are overwritten at the next call to ctermid().
        ...
BUGS
    The path returned may not uniquely identify the controlling
    terminal; it may, for example, be /dev/tty.
```

The problem is that on Linux, it will only display `/dev/tty`, just as the man page's `BUGS` section notes, so if the program needs the actual filename, using `ctermid()` is not the solution. It does work correctly in Solaris 9.

4.3.3 Device Drivers and the /dev Directory

Device files provide an interface between processes and devices. They are like regular files in the following ways: they have a mode, they have owners and groups, times of access, modification, and status change, and they have names and links to them. But they differ in one significant way. Unlike disk files, they are not storage containers; rather than storing data, they provide access to the entry points of functions. Because they are not containers, they do not have size. Because they are really just interfaces, they are associated with executable code that manages a connection to a device.

The code that manages the connection between a process and a device is inside a device driver. A device driver is a program that provides an interface between a device and the processes that communicate with it. Device files are the means in UNIX for a process to communicate with device drivers.

If you type `ls -l` in the `/dev` directory, you will see output such as this:

```
total 0
crw-rw----  1 root root     4,   0 Feb  6 11:07 tty0
crw-------  1 root root     4,   1 Feb  6 16:09 tty1
crw-rw----  1 root tty      4,  10 Feb  6 11:07 tty10
crw-rw----  1 root tty      4,  11 Feb  6 11:07 tty11
crw-rw----  1 root tty      4,  12 Feb  6 11:07 tty12
crw-rw----  1 root tty      4,  13 Feb  6 11:07 tty13
crw-rw----  1 root tty      4,  14 Feb  6 11:07 tty14
```

If you type `ls -l` in the `/dev/pts` directory, you will see something like this:

```
total 0
crw--w---- 1 root     tty 136,  1 Oct 14 14:46 1
crw--w---- 1 lsmarque tty 136, 10 Sep 12 13:13 10
crw--w---- 1 lsmarque tty 136, 11 Sep 12 18:39 11
crw--w---- 1 chays    tty 136, 12 Sep 13 20:02 12
crw--w---- 1 chays    tty 136, 13 Sep 13 20:02 13
crw--w---- 1 lsmarque tty 136, 14 Oct  3 13:22 14
crw--w---- 1 lsmarque tty 136, 15 Sep 12 13:13 15
crw--w---- 1 shixon   tty 136, 19 Oct 14 15:19 19
crw--w---- 1 sweiss   tty 136, 20 Oct 14 15:23 20
```

Notice that the first line, which always reports the total number of blocks used by the files in the directory, shows that these files do not use any storage.

Next observe that the type designator of each file listed above is '`c`', which represents the type character special file. The `c` indicates that this is a *character device*, as opposed to a block device. Every I/O device is accessed through either a block I/O interface or a character I/O interface. One major difference between them is that block I/O uses kernel buffering whereas character I/O does not. When a character I/O interface is used, the data flows in a character stream between the device and connected processes without using system buffers. In contrast, block devices use the kernel buffering system and transfer large chunks of data at a time. Some devices, such as disk partitions, may be accessed in block or character mode. Because each device file corresponds to a single access mode, devices that have more than one access mode have more than one device file.

Recall that the `write` command lets one user write to the terminal of another user, provided that the group write bit is set on the recipient's pseudo-terminal device file. As a warm-up exercise, we will eventually write an implementation of `write`. First we will examine how device drivers work.

Notice in the listings of the `/dev` and `/dev/pts` directories that the size field consists of a pair of numbers. The first number is called the *major device number* and the second, the *minor device number*. For example, `/dev/pts/12` has major device number 136 and minor device number 12. The major device number identifies the type of device, e.g., SCSI disk, pseudo-terminal, or mouse; the minor device number specifies which particular instance of this type of device is represented by the file, or the action associated with this particular interface to the device.

Each major device number is an index into the block or character device table maintained in the kernel. This table is used by the kernel to access the device drivers. Basically the table contains the address inside the kernel of the entry point into the device driver code. When a system call such as `read()` is invoked and the file descriptor passed to the `read()` system call is the file descriptor of a device file, the `read()` code (inside the kernel) finds the i-node belonging to that descriptor. The i-node contains the type of the file, which will indicate that it is a device special file, and whether it is block or character. The kernel will look up the major and minor device numbers from inside the i-node. It will then use the major device number to locate the device driver code for the particular kind of device. For example, if the device file has major number 136 and is a character device file, it will search the character device table for index 136. An attempt to access `/dev/pts/12` with major number 136 and minor number 12 will therefore result in execution of the driver whose index is 136[6]. There is usually a file in the file system that contains the mapping of major and minor device

[6]This is true in BSD and SunOs at least. In the Ext2 file system of Linux, there is more indirection, and the code that is executed might reside in a separate module. The device drivers are not necessarily part of a large executable image, but are instead in separate executable files, more like Windows, and the kernel contains stubs that are resolved dynamically at the time of the call, depending upon the type of file system on which the i-node resides.

numbers to devices. On many Linux systems, you can often find this mapping at

```
/usr/share/doc/kernel-doc-<kernel-version>/Documentation/devices.txt
```

where <kernel-version> is to be replaced by the version number, such as 2.6.9, as in

```
/usr/share/doc/kernel-doc-2.6.9/Documentation/devices.txt
```

If it is not there, then you can read the file `/proc/devices`, which contains the major device numbers for the various groups of devices.

Every I/O device with the same major device number uses the same driver. The minor device number is passed as a parameter to the device driver. The driver may use this number to select which unit of the device to use. Notice, for instance, that each of the pseudo-terminals in use in the above listing has the same major device number, 136, and a minor device number that corresponds to the name of the device file. As another example, if there are multiple disks, the minor device number would specify which disk is being accessed.

Sometimes the minor device number is used to distinguish different actions for the device to take. It is up to the device driver to decide how it will use the minor device number. The use of device numbers as indices into an array of drivers simplifies the customization of each system for the particular hardware configuration.

The device drivers in BSD UNIX, like the kernel itself, are divided into three sections:

1. Initialization and configuration routines
2. Routines to handle I/O service requests (upper half of the driver)
3. Interrupt service routines (lower half of the driver)

The lower half runs in an uninterruptible mode. The upper half runs synchronously and can block itself. They communicate through work queues. In Linux, although the terms "upper half" and "lower half" are not used explicitly, the device drivers work in a similar way. Sometimes they are called "master" and "slave" drivers.

4.3.4 Disk Drivers

An especially important group of drivers are the disk drivers. A disk driver is a software module that manages and controls the I/O that passes between a disk file and processes that have requested I/O to or from that file. This section briefly describes the role of disk drivers in handling I/O.

The purpose of a disk driver is to process requests for disk I/O. These requests are represented by transaction records, each of which consists of

1. a flag indicating whether to read or write data,
2. a primary memory address,
3. a secondary memory address, and

4. a count of the number of bytes to transfer.

The driver maintains these records in a queue. An I/O request causes the upper half of the driver to run. When the upper half is entered, it creates a transaction record and puts it into its work queue. It usually sorts the requests to reduce the latency for the particular device. (This is device dependent code designed to minimize seek time to particular cylinders. In 4.4BSD, for example, the elevator algorithm is used to sort the requests.)

When an I/O request has been satisfied or some other event occurs that warrants intervention by the kernel, the disk sends an interrupt to the processor. The interrupt service routine handles these interrupts; after saving the appropriate portion of the processor state, if the event was an I/O completion, it searches the queue to find the transaction that was completed, de-queues it, changes the state of the requesting process to indicate that it is no longer blocked on that I/O, and selects the next record to service.

Disk drivers are general enough that they can implement block I/O devices, character I/O devices, and swapping devices. To implement block I/O, the block I/O interface passes the address of a system buffer in field (2). To implement character I/O, the character I/O interface passes in field (2) an address that is in the user area, and it ensures that the process is not swapped out during the transfer. To implement a swapping device, the driver is tailored to make requests to the swapping device's controller.

4.3.5 Pseudo-Terminals

A *terminal* is a hardware device that emulates the old Teletype machines. Up until the early 1980's, most users connected to a mainframe computer through terminals. The terminals were connected to the computer via RS-232 lines, into *terminal multiplexers*, which were special-purpose devices that multiplexed multiple terminal lines into the computer[7]. The computer had device drivers whose role was to communicate with these multiplexed terminals. The terminal driver had to control all aspects of the communication path, including modem control, hardware flow control, echoing of characters, buffering of characters, and so on. When microprocessors were invented and personal computers became affordable, personal computers replaced terminals as the front ends to mainframes. Special terminal emulation software could be installed on a personal computer to make it appear to be a "dumb terminal" to the mainframe. One of the earliest such programs was Kermit, developed at Columbia University. Terminal emulation software interposed a terminal interface between the user on his or her local computer and a remote computer on a local area network or the Internet.

Whether you are working on a Linux machine or some other UNIX system locally or remotely via some remote login facility such as Telnet or SSH, if you are using the traditional command-line interface to UNIX, you are using a *pseudo-terminal*. A pseudo-terminal is a software-emulated terminal. When you open a terminal window in a desktop environment such as Gnome or KDE, you are using a pseudo-terminal. When you connect via an SSH client you are using a pseudo-terminal. The device files in the `/dev` directory that have names of the form `pts*` or `pty*` are pseudo-terminal device files. The device drivers for these files manage pseudo-terminals. Pseudo-terminals and how they work are described in more detail later.

[7]Terminals were just one of a class of devices called serial devices. RS-232 lines were serial lines, on which characters were sent one bit at a time. Modern UNIX systems, including Linux, continue to support many different types of serial lines.

4.3.6 Character I/O Interfaces

Almost all I/O devices have a character I/O interface, even if they are block devices. For example, the hard disk has a character interface to allow reads and writes of unstructured byte streams. The character interface is needed for programs such as `fsck`, which performs checks of file system integrity. The character interface translates these requests into block requests for the hardware but presents a character stream to the client. Printers, and modems have character interfaces, and these would have entries such as `/dev/tty0a`, `/dev/lp0`, `/dev/cua0`. Even physical memory has a character interface to allow memory to be treated like a RAM file. The device file for memory is `/dev/mem`.

The character I/O interface to block devices such as disks or tapes is called the *raw interface* because it allows access to the device and ignores its structure. A disk partition such as `sda1` may have two device files in `/dev`: the block interface and the character interface. The block interface is usually `/dev/sda1` in Linux but because individual partitions do not have character interfaces and only disks do, the character interface will be the "SCSI generic" device `/dev/sg1`. The `sg` interface allows byte by byte access to the entire drive. In Solaris 9, raw device names are of the form `/dev/rdsk/c0t3d0s5`.

Character device drivers do not use system buffers, except for terminal drivers, which use a linked list of very small (typically 64 byte) buffers. Character device drivers transfer characters directly to or from the user process's virtual address space. Because the transfers are directly to user memory and use DMA, the drivers must lock the memory to prevent the physical pages from being replaced during the transfer.

4.3.7 Other Character Devices

Some devices have character interfaces even though they do not fit the byte stream model of I/O. An example is a high-speed graphics display, which has such a fast transfer rate that it requires buffers in its own address space to handle the volume of data. These devices would swamp a driver that passed one character at a time, and are handled as special cases in the kernel.

4.3.8 Writing to a Device File

As an exercise in writing to a device file, we will write our own, somewhat simplified, version of the `write` command. The `write` command writes messages to terminals. The `write` man page begins as follows:

```
NAME
        write - write to another user
SYNOPSIS
        write user [ terminal ]
DESCRIPTION
     Write allows you to communicate with other users, by copying
     lines from your terminal to theirs.
     When you run the write command, the user you are writing to gets
     a message of the form: :
             Message from yourname@yourhost on yourtty at hh:mm...
     Any further lines you enter will be copied to the specified
```

```
    user's terminal. If the other user wants to reply, they must run
    write as well.
    When you are done, type an end-of-file or interrupt character.
    The other user will see the message EOF indicating that the
    conversation is over.
...
    If the user you want to write to is logged in on more than one
    terminal, you can specify which terminal to write to by
    specifying the terminal name as the second operand to the write
    command.  Alternatively, you can let write select one of the
    terminals - it will pick the one with the shortest idle time.
    This is so that if the user is logged in at work and also dialed
    up from home, the message will go to the right place.
```

Our first version of `write` will ignore the optional line argument and will not try to pick the terminal with the smallest idle time. The main program follows. It calls two functions, `get_user_tty()` and `create_message()`, which follow thereafter.

```
int main( int argc, char *argv[] )
{
    int     fd;
    char    buf[BUFSIZ];
    char   *user_tty;
    char    eof[] = "EOF\n";

    if ( argc < 2 ){
        fprintf(stderr,"usage: write1 username\n");
        exit(1);
    }

    if ( ( user_tty = get_user_tty( argv[1] ) ) == NULL ) {
        fprintf(stderr, "User %s is not logged in.\n", argv[1]);
        return 1;
    }

    sprintf(buf, "/dev/%s", user_tty);
    fd = open( buf, O_WRONLY );
    if ( fd == -1 ){
        perror(buf); exit(1);
    }
    create_message(buf);

    if ( write(fd, buf, strlen(buf)) == -1 ) {
         perror("write");
         close(fd);
         exit(1);
    }
```

```
    while( fgets(buf, BUFSIZ, stdin) != NULL )
        if ( write(fd, buf, strlen(buf)) == -1 )
            break;
    write(fd, eof, strlen(eof));
    close( fd );
    return 0;
}
```

The most important part of the main program is the loop. The while-loop looks exactly like a loop to read from one file and write to another. The only difference is that it is using `fgets()` instead of a `read()` system call, and it "hard-wires" `stdin` into the code. The `fgets()` function reads from any `FILE` stream until it sees an end-of-line (EOL) or end-of-file (EOF) mark. Therefore, you can see that reading from the keyboard and writing to a terminal is essentially the same as reading from one file and writing to another.

The `get_user_tty()` function searches through the `utmp` file for entries that match the given login name, returning a pointer to a static string containing the line. This could have been allocated on the stack and later freed, but for demonstration purposes, this is easier.

```
char * get_user_tty( char *logname )
{
    static struct utmp utrec;
    int            utrec_size = sizeof(utrec);
    int            utmp_fd;
    int            namelen = sizeof( utrec.ut_name );
    char           *retval = NULL ;

    if ( (utmp_fd = open( UTMP_FILE, O_RDONLY )) == -1 )
        return NULL;

    while ( read( utmp_fd, &utrec, utrec_size) == utrec_size )
        if ( strncmp(logname, utrec.ut_name, namelen ) == 0 ) {
            retval = utrec.ut_line ;
            break;
        }

    close(utmp_fd);
    return retval;
}
```

The `create_message()` function constructs the message to display on the user's console. It uses the `getlogin()` function to get the real username of the owner of the calling process, the `gethostname()` function to get the hostname of the host on which the sender process is running, the `ttyname()` function to get the terminal device name of the controlling terminal of the sending process, and the `time()` function to get the current time, which it converts to a `struct tm` using `localtime()`.

```
void    create_message(char buf[] )
{
    char    *sender_tty,  *sender_name;
```

```
    char    sender_host[256];
    time_t  now;
    struct tm *timeval;

    sender_name  = getlogin();
    sender_tty   = ttyname(STDIN_FILENO);
    gethostname(sender_host, 256);
    time(&now);
    timeval      =  localtime(&now);
    sprintf(buf, "Message from %s@%s on %s at %2d:%02d:%02d ...\n",
            sender_name, sender_host, 5+sender_tty,
            timeval->tm_hour, timeval->tm_min, timeval->tm_sec);
}
```

This version is relatively easy to construct. The next version must allow the user to enter a terminal device name in the form "`pts/n`" and try to open that specific terminal for writing, if possible. If it fails it must return an error message. The changes to allow the optional terminal name are minor:

1. The main program has to check the command line arguments (not shown below).
2. The `get_user_tty()` function needs a second argument that it compares to the `ut_line` field whenever the `ut_name` field matches.

```
char * get_user_tty( char *logname, char *termname )
{
    static struct utmp utrec;
    int              utrec_size = sizeof(utrec);
    int              utmp_fd;
    int              namelen = sizeof( utrec.ut_name );
    char             *retval = NULL ;

    if ( (utmp_fd = open( UTMP_FILE, O_RDONLY )) == -1 )
        return NULL;

    /* look for a line where the user is logged in */
    while ( read( utmp_fd, &utrec, utrec_size) == utrec_size)
        if ( strncmp(logname, utrec.ut_name, namelen ) == 0 )
            if (( termname == NULL ) ||
               ( strncmp(termname, utrec.ut_line,
                           strlen(termname)) == 0 ))
            {
                retval = utrec.ut_line ;
                break;
            }
    close(utmp_fd);
    return retval;
}
```

This version of the `write` command still does not choose the terminal line with the smallest idle time, for a given username, when the terminal line is not specified. How would you go about finding

the terminal line that has been idle the least amount of time? (Hint: the least idle terminal line has been accessed most recently.)

In addition, if you experiment with the real `write` command, you will discover that this version is still lacking other functionality:

- It does not check that the sender's terminal does not have I/O redirected.
- It does not check whether the sender's messaging is turned on (which it must be for the write program to write back to it.)
- It does not check whether the receiver is actually logged in, and whether messaging is enabled on the receiver's line.
- It does not check whether the sender is trying to send to him or herself.

Although none of these checks are difficult, and this program is easy to write, in general, communicating through the terminal interface is much more complicated than this; remember that we failed to write a good version of the `more` command because we could not suppress echoing, we had trouble getting input accepted without the Enter key press, and we did not suppress the scrolling of the prompt.

4.4 Terminals and Terminal I/O

Terminal I/O is probably the messiest and most disorganized part of any operating system. This is partly due to the way that I/O interfaces have developed over the years, in an *ad hoc* fashion, responding to changes in hardware and user demands, partly due to the wide variety of I/O devices that have to be accommodated under the aegis of a single I/O system, and partly due to the general lack of standards that governed how terminal I/O should be handled.

In UNIX, part of the problem is the result of the rift between the BSD versions and System V versions of the operating system, which had very different sets of terminal I/O routines. The POSIX standard provided a unification of the two sets of interfaces, and modern systems provide POSIX compliant routines. However, there are still many parts of the I/O interface that are platform-specific.

If you understand how terminals work, then you will have a better understanding of which routines you need to use to achieve various objectives as well as how to use those routines. The kinds of questions that will be answered here include:

- Why is it that we have to press the `Enter` key in order for the typed characters to be received by a program, and is there a way to avoid this?
- Some program suppress the echoing of characters as they are typed. How can we do this?
- Some programs are able to time-out while waiting for user input. How does that happen?
- Some programs, such as `vi` and `emacs`, override the meaning of various control sequences such as Ctrl-D and Ctrl-C. How can we do that?
- Terminals have a fixed number of rows and columns. How can a program get that number dynamically and control how it wraps its output?

- Why is it that sometimes the backspace key erases characters and sometimes the delete key does, and sometimes neither does? How does the terminal erase?

Adding to the problem of understanding terminals and terminal I/O is the fact that the C Standard I/O Library adds another layer of complexity, including various forms of buffering. We have to figure out what the terminal does and what the library does.

4.4.1 An Experiment

We begin by rewriting the simple I/O program from Chapter 1 so that it does not use C `FILE` streams. This way, whatever we observe is independent of that library's semantics. Unlike the simple I/O program there, this uses the kernel's `read()` and `write()` system calls.

```
Listing copychars.c
#include  <string.h>
#include  <stdlib.h>
#include  <unistd.h>

int main( int argc, char *argv[] )
{
    char inbuf;
    char prompt[] =
           "Type any characters followed by the 'Enter' key. "
           "Use Ctrl-D to exit.\n";

    if ( -1 == write(1, prompt, strlen(prompt)) ) {
        write(1, "write failed\n", 13);
        exit(1);
    }

    while( read(0, &inbuf, 1) > 0  )
        write( 1, &inbuf, 1 );
    return 0;
}
```

Assume this has been compiled into an executable named `copychars`. The `copychars` program reads one character at a time and writes one character at a time until it receives an end-of-file indication, which is that `read()` returns 0. If the user types a `Ctrl-D`, provided that the terminal has the default settings, the `read()` on the standard input stream will return 0.

When you run this program you will notice that, even though the main loop reads a single character and immediately writes that character, nothing gets written on the screen until you press the `Enter` key. As we are not using the C Library's streams, we cannot attribute this behavior to the library's buffering. The terminal is responsible for this. Somehow the characters that we type are stored, but where, and how many can be stored before they are lost?

We could answer these questions by doing a bit of research in the manpages, but this time we will begin with an experimental answer. We can determine the maximum number of characters that the terminal buffers by supplying larger and larger sequences of characters before pressing the `Enter`

key and counting the number of characters output on the screen to make sure none were lost. We are not allowed to redirect input to `copychars` to facilitate this because redirecting input will imply that the terminal is not involved in supplying characters to the program, so we have to do this in a more tedious way. We do not have to type the characters one at a time, but can create files containing the required number of characters on a single line, and if our terminal emulator supports copy and paste, we can paste them into the terminal window and then press the `Enter` key. We are free to redirect output to the `wc` command to count the number of characters written to standard output.

We can create a file named `a_N` containing `N` consecutive 'a' characters with the command

```
$ ( for i in `seq 1 1 N` ; do echo -n 'a' ; done ) > a_N
```

where `N` is replaced by the desired number. Suppose we create such files of sizes 128, 256, 512, 1024, 2048, 4096, 8192 characters. We can open each file in a text editor, copy the sequence to the clipboard and paste it into the window after running the command

```
$ copychars | wc
```

We will see the following sequence of numbers output by `wc`

```
      2      13     198
      2      13     326
      2      13     582
      2      13    1094
      2      13    2118
      2      13    4165
      2      13    4165
```

As you can see, the maximum number of characters reported by `wc` is 4165. Subtract the length of the prompt and the following newline, 70 characters, and you see that 4095 characters seems to be the maximum size of the buffer on the system used for this experiment. We will see shortly that this number is much larger than the documentation states. Where these characters are stored cannot be answered experimentally. That question requires some research.

Try one other thing. Run `copychars`, but this time, use your backspace key to change some of the characters. When you press the `Enter` key this time, there is no trace of that backspace character. It is not part of the stream of characters that the program received. The characters seem to be stored in the buffer in such a way that we can use editing keys to remove them from the set of characters delivered to the program.

Before we go further, and as a sort of advance peek at what we are about to do, try running the program as follows. Type the commands as they appear below:

```
$ stty -icanon ; copychars
```

When you are finished observing what happens, type the command

```
$ stty icanon
```

What you should have observed was that the program behaved as the code suggests – that each time you typed a character, it was immediately echoed on the screen. The `stty` command allows us to control terminal characteristics. What we just did in part disabled buffering of input characters in the terminal. Repeat this and try deleting characters and you will see that editing seems to be disabled. We will return to this later.

Now we will look at a second example. The following program *does* use the C Standard I/O Library, and so we will need to understand how that library confounds the picture, but that will happen a bit later. In the program below, named `listchars.c`, `getchar()` is used to obtain characters typed on the keyboard and `printf()` is used to display each character as an actual character (a *glyph*) with its character code in the current encoding (ASCII).

```
#include <stdio.h>
int main(int argc  char* argv[] )
{
    int   ch;

    printf("Type any characters followed by the 'Enter' key.");
    printf(" Use Ctrl-D to exit.\n");

    while( ( ch = fgetc(stdin)) != EOF  )
        printf("char = '%c' code = %03d\n", ch, ch );
    return 0;
}
```

Notice that `ch` is declared as an `int`. This is because `fgetc()` returns an `int` rather than a `char`, because it returns `-1` to indicate `EOF`. In fact `EOF` has the value `-1`, which means that whatever variable is assigned the return value must be a `signed int`. The `char` type is `unsigned`, so we cannot use it. Notice too that `printf()` will display an integer whose value is a legal character value as a glyph if the `%c` format specification is used, and as an integer if the `%d` format specification is used, so that `printf()` can conveniently print the character code and the character itself.

When you run this program you will again see that the program does not display any characters until the `Enter` key is pressed. If we build it and name it `listchars`, and run it and type `abcde` followed by `Enter`, and `Ctrl-D` to quit, we will see the following output:

```
$ listchars
Type any characters followed by the 'Enter' key. Use Ctrl-D to exit.
abcde
char = 'a' code = 097
char = 'b' code = 098
char = 'c' code = 099
char = 'd' code = 100
char = 'e' code = 101
char = '
' code = 010
$
```

First, notice that no output took place until the `Enter` key was pressed. Once again, this means that the characters that were typed before the `Enter` key was pressed had to be stored in a buffer.

Second, unlike our `copychars` program, this lets us see the character codes of the characters that we enter. It seems that there was a character after the '`e`' but before the `^D`, and that this character caused a new line to appear on the screen and then caused "`code = 10`" to appear after it. But we pressed the `Enter` key after the '`e`'. The fact that the second quote is in the first column on the next line implies that two characters were transmitted: a *linefeed*, ASCII 10, was transmitted to the screen to advance output to the next line, and a *carriage return*, ASCII 13, was transmitted, which forced the next character to appear in column 1 of that line. The ASCII code that was printed is decimal 10, which is the linefeed character, also known as *newline*. Therefore, only one ASCII code was generated, so even though both a newline and a carriage return were sent to the screen, only a newline character was seen by the program. Therefore, our pressing the `Enter` key caused only a newline code to be sent to the program. Usually, when you type the `Enter` key, it causes both the linefeed and the carriage return to be inserted in a document, but here you can see that only the linefeed is transmitted. (If you do not remember your ASCII codes, type `man ascii` for the listing.)

Third, in the `printf()` statement, the program sent a "`\n`" (newline) character to the terminal, but this seems to cause both a linefeed and a carriage return to be placed on the terminal. The linefeed character, by definition, should cause the cursor to move down to the next line in the same "column" on the screen, whereas a carriage return should move the cursor to column one, the left margin, without moving it down one line. But somehow both things happen even though a single character is transmitted.

What we can conclude from this is that the terminal driver must be doing some character processing on the inputs it receives from the keyboard and on the outputs that the processor sends to the display device. (Remember that the terminal driver is controlling a logical input/output device.) We now explore what the terminal driver does and how we can control its behavior.

4.4.2 Terminal Devices: An Overview

The preceding experiment showed that the default input mode of a terminal includes assembling the input into lines, processing various special characters such as backspace, and delivering the input lines to the process after they have been processed. This mode of operation is called *canonical input mode*. Terminals can be operated in various non-canonical input modes as well. In a non-canonical mode, some part of this processing of input is turned off. Programs like `emacs`, `vi`, and `less` put the terminal into a non-canonical mode[8].

The behavior of a terminal device is controlled entirely by a piece of software called a *terminal driver*. A terminal driver is not the same thing as a *terminal device driver*. A terminal driver consists of two components:

1. a *terminal device driver*, and
2. a *line discipline*.

[8]System 7 and BSD systems supported three different input modes: cooked, raw, and cbreak. POSIX.1 does not support these, although many systems still provide support for them.

The terminal device driver is usually a part of the kernel and its main function is to transfer characters to and from the terminal device; it is the software that talks directly with the hardware at one end, and the line discipline at the other. The line discipline is the software that does the processing of input and output. It maintains an input queue and an output queue for the terminal. The relationship between these queues, the process using the terminal, and the terminal itself, is illustrated in Figure 4.3. The terminal driver is the sum of the parts in this figure; it is the combination of line discipline and device driver. Notice that

- When echoing of characters is on, characters are copied from the input queue to the output queue.
- The size of the input queue is `MAX_INPUT`. If characters are typed faster than they are removed for processing, and the queue fills, UNIX systems typically ring the bell character and discard any extra characters.

  In the experiment that we performed with the `copychars` program, we saw that the input queue was able to store 4095 characters, which would suggest that 4095 is the value of `MAX_INPUT`. `MAX_INPUT` is one of many system limits that are defined in `<limits.h>`. Its value can also be obtained by calling `pathconf(ttyname(0), _PC_MAX_INPUT)` within a program. If we do either, we will see that `MAX_INPUT` is 255. The documentation states that `MAX_INPUT` may be smaller than the actual value. In fact, the terminal driver is configured to allow 4095 characters to be queued, even though the system limit is 255.
- Even though the output queue is also finite, if a process tries to write to it faster than the driver can transfer characters to the device, the kernel will simply block the process until the queue has more room.

The figure does not display all of the data structures used by the line discipline. Another queue is not shown, the *canonical input queue*. This is the queue in which input characters are processed when the terminal is in canonical mode. The canonical processing center is part of the line discipline. The figure does not show the internal data structure that the line discipline uses to control the terminal. The kernel provides an interface to access this structure, and UNIX provides a command, `stty`, to access and modify the various attributes of the terminal that are stored in this structure. The name of the interface to this structure, in POSIX.1 compliant systems, is the `termios struct`[9].

Almost all of the terminal device characteristics that can be examined and changed are contained in this `termios` structure, which is defined in the header file `<termios.h>`. That structure is a collection of four flagsets, and an array of character codes:

```
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 */
};
```

[9] In System V, the structure used for controlling terminals was the `termio struct`, and its header file was the `termio.h` file. POSIX added an 's' to the name to distinguish the new structure from theirs. A single 's'!

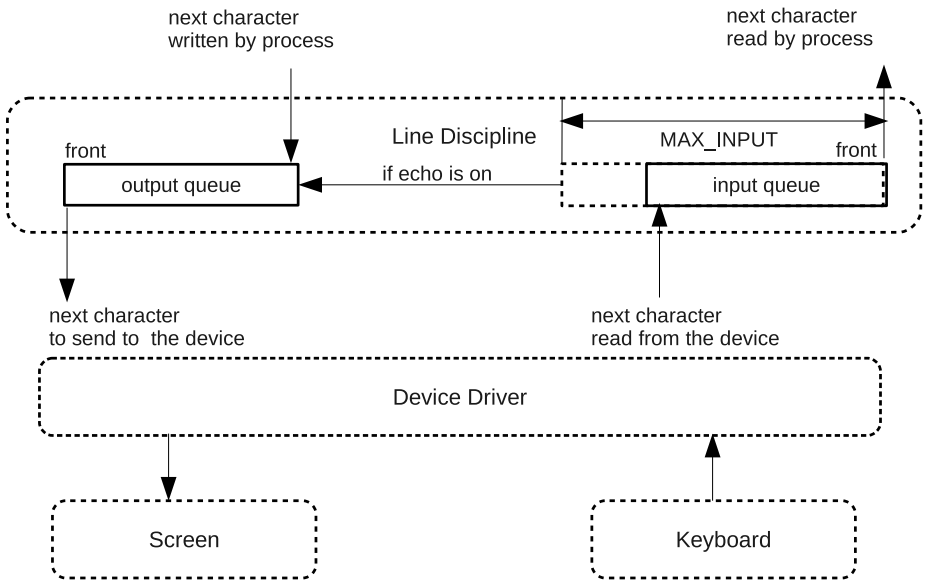

Figure 4.3: Terminal data structures

The type `tcflag_t` is an unsigned integer. The four flagsets are just four integers:

- The input flagset, `c_iflag`, controls the input of characters by the terminal device driver.
- The output flagset `c_oflag`, controls the driver output.
- The control flagset `c_cflag` controls the behavior of asynchronous serial transmission lines (such as RS-232); it may not have meaning for other kinds of terminal ports (such as pseudo-terminals).
- The local flagset `c_lflag` affects the interface between the driver and the user (echo, erase characters, etc.)

The `c_cc` array defines the special characters that can be changed. The array elements are of type `unsigned char` (`cc_t` is typedef-ed to `unsigned char`.)

Figure 4.3 shows the relationship between the terminal device and the terminal driver and its components. When you type on the keyboard, the character codes are transmitted to the device driver, which in turn passes them to the line discipline, placing them in its input queue. The line discipline, under the normal circumstances, copies the typed characters into the output queue, to be displayed on the console as you type. The line discipline is responsible for processing of input, including processing of special characters. It passes the processed characters to the kernel's `read()` routine[10], which passes them to the process that requested the read operation. When a process issues a write request, the kernel's write function passes the characters to the line discipline, which performs any processing that is supposed to be performed, and then puts the characters in the

[10]Technically, to the `sys_read()` function, which is the actual function within the kernel, as opposed to `read()`, which is a wrapper for it.

output queue. The device driver retrieves the characters from the front of the queue and displays them on the console.

Before we explore the `termios` structure at the programming level, we will look at how we can change it from the command level.

4.4.3 The `stty` Command

A user can use the `stty` command to view and alter terminal characteristics. `stty` without options or arguments displays a selection of the current settings of the terminal connected to the shell in which the command is invoked. Different systems will display different amounts of information. To see the complete list of terminal settings, use "`stty -a`". The output will look something like the following (rearranged and labeled):

```
$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
cchars:   intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
          eol = M-^?; eol2 = M-^?; start = ^Q; stop = ^S; susp = ^Z;
          rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
           min = 1; time = 0;
control flags: -parenb -parodd cs8 -hupcl -cstopb cread
               -clocal -crtscts
input flags:  -ignbrk -brkint -ignpar -parmrk -inpck -istrip
              -inlcr -igncr icrnl ixon -ixoff -iuclc ixany imaxbel
output flags:  opost -olcuc -ocrnl onlcr -onocr -onlret -ofill
              -ofdel nl0 cr0 tab0 bs0 vt0 ff0
local flags:   isig icanon iexten echo echoe echok -echonl -noflsh
              -xcase -tostop -echoprt echoctl echoke
```

It is important to understand that these settings are associated to the device file for the terminal, not to the connection between the shell or another process and the terminal. To convince yourself of this, perform the following experiment. To perform the experiment, we will play with the erase character. `stty` can be used to change the erase character used by the terminal, by typing the following, but don't do it yet:

```
$ stty erase x
```

where `x` is the character that you wish to use for erasing a single character on the command line. The default is usually either `Ctrl-H` or `Ctrl-?`, denoted `^H` and `^?` respectively. The backspace key sometimes generates `^H` and the delete key, `^?`, but not always. This experiment will not work in all shells, because some of them are designed to allow command line editing. In other words, the shell itself provides the functionality, overriding the terminal settings. If `bash` is your shell, then turn off command line editing by entering a non-editing `bash` subshell with the command line

```
$ bash --noediting
```

Then check which key is the erase character on your terminal by seeing which one actually erases, and then type

```
$ stty -a | egrep -o '\<erase = ...'
```

This will produce a line like

```
erase = ^?
```

Change the erase key to something else. For simplicity, make it the character 'X' and then verify that the change took place as follows:

```
$ stty erase X
$ stty -a | egrep -o '\<erase = ...'
erase = X;
```

Verify that it works by typing some random characters and using the `X` to erase them. Now invoke another new shell process by typing "`bash --noediting`" and then run

```
$ stty -a
```

in the new shell. You will see that the erase character is still `X`. Next, in the new shell, change the erase character back to the original, or to something else. To make it a control character, you can type the caret ('^') followed by the character.

```
$ stty erase ^H
$ stty -a | egrep -o '\<erase = ...'
erase = ^H;
```

Exit the current shell with the `exit` command and view the terminal settings again. You will see that the erase character is the one you set it to be in the shell that you just killed. This proves that you are really changing the device file settings, not those of the process's connection to it. It is not like the connection to a disk file. You can now exit the first `bash` subshell and check yet again that in your first shell the erase key is the last one you chose.

Finally note that if you have two different windows open, and one is pseudo-terminal `pts/3` and the other is `pts/4`, these may have different settings; the settings "stick" to devices (or pseudo-devices in this case), not processes.

Returning to the details of terminal settings, notice that there are two kinds of variables that are listed in the output of `stty -a`: those listed in the form

1. *var* = *value*; or *var value*; and
2. *var* or *-var*

The first type are non-Boolean variables. The second are Boolean. The Booleans are called *switches* or *flags*. I prefer the term *switch*, because they act like switches: a switch prefixed with a minus sign (-) is off, and a switch that is not prefixed with a minus sign is on. Examples of switches are `echo`, `inlcr`, `icanon`, and `icrnl`. To change the value of a switch you type

```
stty [-]switch
```

where `switch` is replaced by the name of the switch, and the minus sign is present to turn it off, absent to turn it on. Thus,

```
$ stty -echo
```

will turn off `echo` on the screen. Try it now but remember that to turn it back on you will have to type "`stty echo`" without being able to see your typing. An alternative is to open a second terminal window, and in that window, type `w` to see the which device files are associated with your two logins. Suppose that the first terminal, in which you turned off echoing, is `/dev/pts/2`. Then in the second terminal, you can type

```
$ stty --file=/dev/pts/2 echo
```

and this may turn `echo` on in the first terminal. (Remember that the `tty` command will display your terminal device filename.) In general, the `--file` option lets you specify an alternate device file for the `stty` command, provided you have permission to modify it. It may not always work, and not all systems support the `--file` option.

Non-Booleans are displayed in one of two forms

- variable = value;
- variable value;

For example, the line

```
speed 38400 baud; rows 24; columns 80; line = 0;
```

means that the baud rate is 38400, the number of rows in the terminal is 24 and the number of columns is 80. The "`line = 0`" refers to the line discipline. There are certain predefined combinations of settings that are collectively known as the line discipline. Many of these are derived from the early UNIX implementations of terminal devices; setting the line discipline to 0 means that the default behavior is the typical one wanted by most users.

The characteristics of the terminal that can be controlled via the `stty` command fall into six classes:

Class	Description
Special Characters	Characters that are used by the driver to cause specific actions to take place, such as sending signals to the process, or erasing characters or words or lines. Special characters include the `erase`, `werase`, and `kill` characters, as examples. Characters used to send signals include `Ctrl-C`, which sends the *interrupt* signal, and `Ctrl-\`, which sends the *quit* signal. Signals are covered in a later chapter, but for now it suffices to know that signals such as the interrupt and quit signal usually terminate the process that receives them.
Special Settings	Variables that control the terminal in general, such as its input and output speeds and dimensions. These include the `rows`, `cols`, `min`, and `time` values. `rows` and `cols` are the numbers of rows and columns in the window. `min` and `time` are used when the terminal is in non-canonical mode to control how characters are returned by `read()` calls; these will be discussed later. Many of these variables do not apply to pseudo-terminals.
Input Settings	Operations that process characters coming from the terminal. This includes changing their case, converting carriage returns to newlines, and ignoring various characters like breaks and carriage returns.
Output Settings	Operations that process characters sent to the terminal. Output operations include replacing tab characters by the appropriate number of spaces, converting newlines to carriage returns, carriage returns to newlines, and changing case.
Control Settings	Operations that control character representation such as parity and stop bits, hardware flow control. Several of these do not apply to pseudo-terminals.
Local Settings	Operations that control how the driver stores and processes characters internally. For example, echo is a local operation, as is processing erase and line-kill characters.
Combination Settings	Combinations of various settings that define modes such as cooked mode or raw mode.

Input settings start with 'i' and output settings start with 'o'. One input setting of interest is `icrnl`. This is the switch that, when set, causes input carriage returns to be converted to newline characters. Read it as `i` for input, `cr` for carriage return, `nl` for newline. This switch explains why the `Enter` key is converted to a newline character. Try running the following:

```
$ stty -icrnl ; listchars
```

and typing 'abc' followed by `Enter` then `Ctrl-D`. The `Enter` will appear as a `^M` on the screen. Then type `Ctrl-C` to quit the program. You will see that the code listed for the carriage return is now ASCII 13. You will also see that the word "`char`" is not displayed on the screen for that line:

```
abc^Mchar = 'a' code = 097
char = 'b' code = 098
char = 'c' code = 099
' code = 013
```

This is because the carriage return character causes the output to start in column 1 but there was no linefeed, so the output " code 13" is over-writing the "char = ". To prove this, modify the program to force a newline just before the '%c' in the `printf()` call. You will see the word "char" reappear. Restore your settings by typing

```
$ stty icrnl
```

The output setting of interest is the `onlcr` switch. This is what adds a carriage return to each newline character sent to the terminal. Try turning it off and looking at the output:

```
$ stty -onlcr ; listchars
[stewart@harpo chapter05]$ stty -onlcr; listchars
Type any characters followed by the 'Enter' key; Ctrl-D to exit.
                                                                    abc
                                                                       c
       har = 'a' code = 097
                           char = 'b' code = 098
                                                 char = 'c' code = 099
                                                                     char
       = '
           ' code = 010
```

Notice that each new line starts just below the end of the previous line; there were no carriage returns inserted into the output stream. To restore your settings, type

```
$ reset
```

or

```
$ stty onlcr
```

Local settings include whether or not canonical mode and echo are enabled. By default, the terminal is in canonical mode. The output of the `stty` command indicates this:

```
$ stty -a | grep canon
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
```

In canonical mode, the driver processes the line editing keys:

- *character erase* (`erase`),
- *word-erase* (`werase`),
- *line-kil*l (`kill`), and
- *line redraw* (`rprnt`)

and recognizes the following special input characters[11]: *line delimiter* (`eol` and `eol2`) and *end-of-file* (`eof`). The `eol` and `eol2` characters are equivalent. It also buffers the input characters until the `Enter` key is pressed, at which point it delivers them to the process. Shells that support line editing natively (by the shell itself) such as `bash` and `tcsh` ignore the `stty` command to turn off canonical mode because they process the characters themselves. Canonical mode is turned off, but within the shell, there will be no effect. To see the effect of disabling canonical mode, you have to run `bash` with the `--noediting` command line switch.

In canonical mode, typed characters are processed and placed into the canonical input queue. In non-canonical mode, they are delivered to the `read()` system call directly. We will demonstrate this now. We will run a simple program similar to the `copychars` program but, just to make it different, it will display not the typed character, but its transpose, defined here by *transpose(c) = islowercase(c)? 'a' + 'z' - c:c*. The program is therefore

```
Listing transpose.c
#include  <unistd.h>
#include   <ctype.h>

int main(int argc, char* argv[])
{
    char ch;
    while ( read(0, &ch, 1) > 0  ) {
        if (islower(ch))
            ch = 'a' + 'z' - ch;
        write( 1, &ch, 1 );
    }
    return 0;
}
```

We turn off canonical mode with the command, `stty -icanon`, and run the `transpose` program in non-canonical mode. In my terminal, the erase character is the Backspace key, which echos as "^?" (control-?). As soon as canonical mode is disabled, it will be hard to correct typing mistakes, so it is easier to turn off canonical mode and issue the `transpose` command as a single job. The interaction is as follows:

```
$ stty -icanon; transpose
azbycxdw^? ev

^C
```

For each character that I type, the transpose is printed immediately after, because the character is given to the `transpose` program as soon as it is typed. This shows that when canonical mode is off, there is no input buffering. Also, when I try to erase using the Backspace key, the character pair "^?" appears on the screen but nothing is erased. Following it is a non-printing character. The character code for the Backspace key is something that cannot be printed, so some type of symbol

[11] There are many more special input characters than these. These are just the ones that are processed in canonical mode.

will appear after the "`^?`", which is what is echoed for Backspace. If you try entering the line-kill character, `Ctrl-U`, it will not erase the line. Nor will `Ctrl-D` send an `EOF` signal. The program must be killed with `Ctrl-C`. Furthermore, when I typed the `Enter` key, that was echoed and then printed again, so two newlines were displayed. In a later chapter, we will explore other effects of disabling canonical mode.

Terminals are complex structures. Most of their attributes should not be changed unless you really understand how they are used. Some are communication characteristics, such as parity, start and stop bits, handshake methods, hang-up methods, and so on. These are often not relevant to your terminal settings; they are part of the `stty` command because it is also a general command for controlling serial devices and real tty's, for which they are necessary. You can read the man pages for more detailed explanations of all of these attributes.

4.4.4 Programming the Terminal Driver

The `stty` command allows you to modify terminal settings from the shell, but it cannot be used from within a program. To control most of the terminal characteristics from within a program, we can use the pair of system calls, `tcgetattr()` and `tcsetattr()`. These get and set driver attributes respectively. There is an alternative function, `ioctl()`, that can be used for controlling terminal settings, but it is not preferred, because it is not supported by the standard. We will look at `ioctl()` later. The `ioctl()` function is necessary for controlling devices other than terminals.

`tcgetattr()` and `tcsetattr()` are not the only functions that operate on the terminal settings. There are actually 13 different functions in the POSIX standard. The remaining ones are

`cfgetispeed()`	gets input speed
`cfgetospeed()`	gets output speed
`cfsetispeed()`	sets input speed
`cfsetospeed()`	sets output speed
`tcdrain()`	waits for all output to be transmitted
`tcflow()`	suspends transmission
`tcflush()`	flushes input and/or output queues
`tcsendbreak()`	sends a break character
`tcgetpgrp()`	gets foreground process groupid
`tcsetpgrp()`	sets foreground process groupid
`tcgetsid()`	gets process group ID of session leader for control of tty

Some of these act on the line discipline; others act on the device driver settings. We will only explore a few of these.

Whereas `fcntl()` allows you to control disk file connections, these allow you to program terminal driver attributes. Unlike `fcntl()`, which can be used for both getting and setting attributes, the work of getting terminal attributes is in one function, and setting, in another. The method of making changes is the same though; you have to

- retrieve the current settings into a structure in the process's address space using `tcgetattr()`,
- modify that structure locally, and
- write it back to the driver using the `tcsetattr()` call.

The attributes are passed back and forth in a `termios` structure. There is a single man page for `termios`, `tcgetattr()`, and `tcsetattr()` and all of the other functions listed above except the last three. The following excerpts the relevant information:

```
NAME
     termios,  tcgetattr, tcsetattr, tcsendbreak, tcdrain,
     tcflush,tcflow, cfmakeraw, cfgetospeed, cfgetispeed,
     cfsetispeed, cfsetospeed
     get and set terminal attributes,
     line control,get and set baud rate
SYNOPSIS
     #include <termios.h>
     #include <unistd.h>
     int tcgetattr(int fd, struct termios *termios_p);
     int tcsetattr(int fd, int optional_actions,
                    struct termios *termios_p);
          ...
DESCRIPTION
The termios functions describe a general terminal interface
that is provided to control asynchronous communications ports.

Many of the functions described here have a termios_p argument
that is a pointer to a termios structure. This structure
contains at least the following members:
               tcflag_t c_iflag;      /* input modes */
               tcflag_t c_oflag;      /* output modes */
               tcflag_t c_cflag;      /* control modes */
               tcflag_t c_lflag;      /* local modes */
               cc_t c_cc[NCCS];       /* control chars */
      ...
```

The `tcgetattr()` system call is given a file descriptor and a pointer to a `termios` structure. The file descriptor must refer to a terminal device file otherwise it is an error[12]. The `tcgetattr()` call will store the attributes of the referenced terminal device in the `termios` structure. The `tcsetattr()` system call is also given a file descriptor and a pointer to a `termios` structure, but its second parameter is a set of optional actions. These optional actions specify when to apply the changes to the terminal device. There are three possible values for this parameter. From the man page (with my corrections):

`TCSANOW` The change occurs immediately.

`TCSADRAIN` The change occurs after all output written to `fd` has been transmitted. This function should be used when changing parameters that affect output.

`TCSAFLUSH` The change occurs after all output written to the object referred to by `fd` has been transmitted; furthermore, before the change takes place, all input data that has not been read is discarded.

[12]This is yet another way to test whether a particular file descriptor points to a terminal device, along with `isatty()` and `ttyname()`.

`TCSANOW` forces the change immediately; this can cause problems if the terminal driver is writing to the terminal and the changes modify the output flags. `TCSADRAIN` forces the changes, but only after the output queue has been emptied by the driver. This is the action that should be chosen whenever the changes affect output to the terminal. `TCSAFLUSH` forces the changes only after the output queues are emptied and after it causes the input data sitting in the queue to be discarded. It is safest, when restoring the terminal to its original state, to use `TCSAFLUSH`.

Different versions of UNIX have different definitions of the `termios` structure. The definition found in Solaris 9 complies with the Single UNIX Specification, Version 3, of the Open Group[13] (also known as IEEE Std 1003.1), the most generally accepted UNIX standard. The definition found in the `/usr/include/bits/termios.h` header file on Linux 2.6 contains other fields (which is allowed by POSIX):

```
struct termios
{
    tcflag_t c_iflag;           /* input mode flags */
    tcflag_t c_oflag;           /* output mode flags */
    tcflag_t c_cflag;           /* control mode flags */
    tcflag_t c_lflag;           /* local mode flags */
    cc_t c_line;                /* line discipline */
    cc_t c_cc[NCCS];            /* control characters */
    speed_t c_ispeed;           /* input speed */
    speed_t c_ospeed;           /* output speed */
}
```

The `c_line`, `c_ispeed`, and `c_ospeed` members are not part of the standard, which allows the structure to contain additional members. Although `tcflag_t` is an integer whose individual bits are flags, you do not need to know the individual bit positions of the flags, because the header file defines masks for each one. The order of the bits may vary from one implementation to another. POSIX specifies one set of flags, XOpen another, Open Source yet another, and so on.

The flagsets are illustrated in Figure 4.4 Each box is a single integer flagset. The names inside each are the names of the individual bit masks. Each of these flags is described in the man page for `termios`. The `c_iflag` member contains flags that define input processing. The `c_oflag` member contains flags that define output processing. The `c_cflag` has flags that define control characteristics, and the `c_lflag` member has flags that define how characters are processed locally, i.e., internally in the driver. The `c_cc` array is an array that stores control character assignments. This is where the map of erase key, backspace key, and so on, is stored. POSIX requires that the following subscript names must exist:

[13] You can download version 3, the latest version, from http://www.unix.org/version3.

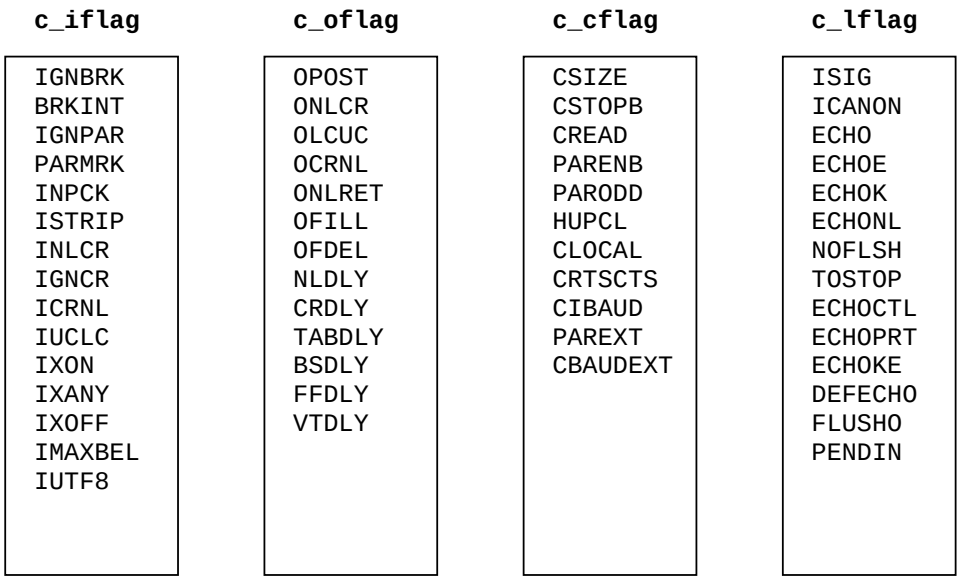

Figure 4.4: Flagset bitmasks of the `termios` struct

Canonical Mode	**Non-Canonical Mode**	**Description**
VEOF	EOF	character
VEOL	EOL	character
VERASE	ERASE	character
VINTR	VINTR	INTR character
VKILL	KILL	character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value

The size of the `c_cc` array is defined to be `NCCS`. We will cover this array in a later chapter.

You can change individual bits in the flagsets with bitwise operations. If `MASK` represents an arbitrary bit mask, then the following test, set, and clear the bits in a flagset:

```
if ( flagset & MASK ) ... //tests the masked bit
flagset |= MASK           //sets the masked bit
flagset &= ~MASK          //clears the masked bit
```

For example, to turn off terminal echo, assuming `flagset` is a local copy of the `c_lflags` flagset from the `termios` structure, we would use

```
flagset = flagset & ~ECHO;
```

Following is a program that prompts the user to enter a password and makes the typing invisible when the password is entered, as the `login` program does. The program turns off the echo switch and resets it afterward.

```
Listing login.c
#include   <stdio.h>
#include   <stdlib.h>
#include   <termios.h>

/*************************************************************************
                                Main Program
*************************************************************************/

int main(int argc, char* argv[] )
{
    struct termios info, orig;
    char username[33];
    char passwd[33];
    FILE   *fp;

    /* get a FILE* to the control terminal --   don't assume stdin  */
    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
         return(1);

    printf("login: ");                    /* display message          */
    fgets(username, 32, stdin);           /* get user's typing        */

    /* Now turn off echo */
    tcgetattr(fileno(fp), &info);         /* Get current terminal state */
    orig = info;                          /* Save a copy of it */
    info.c_lflag &= ~ECHO ;               /* Turn off echo bit   */
    tcsetattr(fileno(fp),TCSANOW,&info); /* Use this state in line discipline */

    printf("password: ");
    fgets( passwd, 32, stdin);            /* Get user's non-echoed typing
*/
    tcsetattr(fileno(fp),TCSANOW,&orig); /* Restore saved settings
*/

    printf("\n");                         /* Print a fake message
*/
    printf("Last login: Tue Apr 31 21:29:54 2088 from the twilight zone.\n");
    return 0;
}
```

Comments

- This is the standard procedure: use the `tcgetattr()` to get the old state, save a copy, make the change, set it with `tcsetattr()`, do the work, and then restore.
- We read from and write to only the control terminal and return an error if we can't open this device for reading and writing. We can get the name of the control terminal with `ctermid()`, and get the file descriptor from the name with the `fileno()` function. This is more secure.
- We don't use `"/dev/tty"` ; it is not as reliable as calling `ctermid()`.

4.4.5 Implementing `stty`: `showtty`

Implementing a user-friendly, perhaps more verbose, version of `stty` is a good exercise in using the `termios` structure. We will write a POSIX compliant version, so it should display the correct values of any flags that it does attempt to display.

The basic idea is very straightforward: `tcgetattr()` is used to retrieve the settings of the flagsets. The `show_flagset()` function prints the status of each flag in the four flagsets and the `show_cc_array()` function prints the mapping of control characters to character codes. Their designs are similar and are based on the use of an array of structures that contain a value and a textual description or name associated with that value:

```
/* Mapping of flag bit position to name */
typedef struct _flaginfo  flaginfo;
struct _flaginfo
{
    int   fl_value;   // flag value
    char  *fl_name;   // string describing flag
};

/* Mapping of index to description */
typedef struct _cc_entry  cc_entry;
struct _cc_entry {
    int  index;
    char *description;
};
```

A single flagset is then represented by an array of `flaginfo` structures, and the array of control character codes is represented in a similar way by an array of `cc_entry` structures. For example, the input flags are defined by the constant array

```
flaginfo input_flags[] = {
    {IGNBRK  , "Ignore break condition" },
    {BRKINT  , "Signal interrupt on break" },
    {IGNPAR  , "Ignore chars with parity errors" },
    {PARMRK  , "Mark parity errors" },
    {INPCK   , "Enable input parity check" },
    {ISTRIP  , "Strip character" },
    {INLCR   , "Map NL to CR on input" },
    {IGNCR   , "Ignore CR" },
    {ICRNL   , "Map CR to NL on input" },
    {IXON    , "Enable start/stop output control" },
    {IXOFF   , "Enable start/stop input control" },
    {-1      ,  NULL }
};
```

and the control characters, by the constant array

```
cc_entry  cc_values[] = {
    { VEOF,     "The EOF         character"},
    { VEOL,     "The EOL         character"},
    { VERASE,   "The ERASE character"},
#if defined VWERASE
    { VWERASE,  "The WERASE character"},
#endif
    { VINTR, "The INTR character"},
    { VKILL,    "The KILL character"},
    { VQUIT,    "The QUIT character"},
    { VSTART,   "The START character"},
    { VSTOP,    "The STOP character"},
    { VSUSP,    "The SUSP character"},
    { VMIN,     "The MIN value "},
    { VTIME,    "The TIME value"},
    { -1,      NULL}
};
```

Notice that each array is terminated by a sentinel value -1. Notice too that the `VWERASE` code is only conditionally compiled, because POSIX does not define it. The logic in processing either array is similar. The function to display the character codes for the `c_cc` array is

```
void show_cc_array( struct termios ttyinfo,
                    cc_entry       controlchars[] )
{
    int  i = 0;

    while ( -1 != controlchars[i].index ) {
        printf( "%s is ", controlchars[i].description);
        printf( "Cntl-%c\n",
                ttyinfo.c_cc[controlchars[i].index] -1 +'A');
        i++;
    }
}
```

For each character code in the array whose index is not -1, it prints the description from the structure and the value of the code as a "control-something" name. These "control-something" names, e.g., "`Ctrl-A`", "`Ctrl-B`", and so on, through "`Ctrl-_`", derived from the fact that on a keyboard, it is often possible to generate a control code using the control (Ctrl) key and a normal key. By adding the code for 'A', less one, to the index stored in the `c_cc` array, we get the keyboard key that should be typed to enter that code from the keyboard.

The `show_flagset()` function is similar. It is called by a function, `show_flags()`, that simply calls `show_flagset()` for each different flagset.

```
void show_flagset( int flag, flaginfo bitnames[] )
{
```

```
    int  i = 0;

    while (-1 != bitnames[i].fl_value ) {
        printf( "%s is ", bitnames[i].fl_name);
        if ( flag & bitnames[i].fl_value )
                printf("ON\n");
        else
                printf("OFF\n");
        i++;
    }
}
```

The main program calls a function, `show_baud_rate()`, that when given the `termios` structure `ttyinfo`, converts the symbolical values of the baud rate into numerical values. The `show_baud_rate()` function calls `cfgetospeed()` to extract the baud rate from the structure. The main program is below.

```
int main(int argc, char* argv[])
{
    struct      termios ttyinfo;
    FILE   *fp;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL )
        exit (1);
    if ( tcgetattr( fileno(fp) , &ttyinfo ) == -1 ){
        perror( "Cannot get info about this terminal.");
        exit (1);
    }

    show_baud_rate (ttyinfo);
    printf("\n");
    show_cc_array(ttyinfo, cc_values);
    printf("\n");
    show_flags(ttyinfo, input_flags, output_flags, local_flags);
    return 0;
}
```

4.4.6 Non-Canonical Terminal Modes

Recall that in canonical mode, the terminal driver assembles input into lines, interpreting various special characters such as backspace, and then delivering it to the process. Above we noted that in non-canonical mode, input buffering is turned off and typed characters are delivered immediately to the reading process. Here we will explore the features of the line discipline that allow us to control more precisely how many characters are delivered to the process and when they are delivered. We will use the `transpose` program for demonstrating this.

When canonical mode is turned off, the `min` option to `stty` controls the minimum number of characters that must be typed before they are transmitted to the program. For example

```
$ stty -icanon min 4
```

puts the terminal into non-canonical mode and sets the minimum number of characters that must be typed before anything is transmitted to the program to 4. Try typing the following:

```
$ stty -icanon min 3; transpose
abczyxdefwvu
^C
```

Notice that three characters at a time are sent to the program, and that until you type three characters, none will be sent. This shows that although there is no buffering until an `EOL` is entered, a read operation is not considered to be complete unless the minimum number of characters is present in the driver.

Within the `termios` structure, the `MIN` member of the `c_cc` array (`c_cc[VMIN]`) specifies the minimum number of bytes before a `read()` call returns. `MIN` can be zero or positive. The `TIME` member of the `c_cc` array (`c_cc[VTIME]`) specifies the number of tenths of a second that a `read()` call must wait for data to arrive. It too can be zero or positive. There are thus four combinations of values for this pair of variables, each with different semantics. Assume in the following that the call to `read()` is

```
read(STDIN_FILENO, buf, numbytes_to_read);
```

Case 1. `MIN` $>$ 0, `TIME` $>$ 0

> The `read()` call will block (and hence the calling process will block) until it receives the first byte. `TIME` specifies an *inter-byte timer* that is activated only when the first byte is received. In other words, until the user types, the timer is inactive. After each byte is received, the timer is reset to 0. Once the user starts to type, if the lesser of `numbytes_to_read` and `MIN` bytes are received before the timer expires, `read()` returns that many bytes. Remember that the timer is reset after each byte. If the timer expires before `MIN` bytes are received, `read()` returns whatever bytes it received. (At least one byte is returned if the timer expires, because the timer is not started until the first byte is received.) If data is already available when `read()` is called, it is as if the data had been received immediately after the `read()`. This means that the `read()` can return immediately if enough data was in the input queue of the line discipline (inside the terminal driver).

Case 2. `MIN` $>$ 0, `TIME` $==$ 0

> The `read()` does not return until `MIN` bytes have been received. Setting `TIME` to zero means that the timer is turned off. It is given an infinite amount of time, in effect and so this can cause a `read()` to block indefinitely, waiting for input. The lesser of `numbytes_to_read` and `MIN` bytes are returned to the user process. This means that if `numbytes_to_read` $<$ `MIN`, then `read()` is not satisfied until `MIN` bytes are received, but it only gives `numbytes_to_read` to the caller. If `numbytes_toread` $>$ `MIN`, then `read()` returns when `MIN` bytes are received and `MIN` bytes are given to the caller.

Case 3. `MIN` == 0, `TIME` > 0

> Unlike Case 1, `TIME` specifies a timer that is started when `read()` is called. *It is not an inter-byte timer.* In this case, `read()` returns when a single byte is received or when the timer expires. If the timer expires, `read()` returns 0, otherwise it returns the number of bytes read. If there are characters in the terminal's input queue before the `read()` call is processed, the minimum of the number of characters in the queue and `numbytes_to_read` bytes will be delivered to `read()` immediately, satisfying the call, and `read()` will return the number of bytes actually read.

Case 4. `MIN` == 0, `TIME` == 0

> The minimum of either the number of bytes requested, i.e. `numbytes_to_read`, or the number of bytes currently available is returned to the caller without waiting for more bytes to be input. If no characters are available, `read()` returns 0, having read no data. Otherwise the number of bytes actually read is returned. In both cases, `read()` returns immediately.

To demonstrate, first, in non-canonical mode we will set `min` = 0 and `time` = 30. What you will see is that if a character is typed within 3 seconds (30 tenths of a second), it will be delivered immediately, but if no character is typed within 3 seconds, the read will return with an `EOF` character. Type some characters with 3 seconds each and then let 3 seconds elapse. You will see the program terminate.

```
$ stty -icanon min 0 time 30 ; transpose
```

You may also discover that your shell terminates. Before you do this, make sure that you set the `IGNOREEOF` environment variable so that you do not lose your shell. If you see repeated messages of the form "Use exit to logout" then turn off the terminal settings with `reset`.

Next, test the effect of setting `min` and `time` to be non-zero. Type the following command sequence:

```
$ stty -icanon min 4 time 30 ; transpose
```

Wait at least 3 seconds and notice that nothing happens. The program is waiting for input. There is no timeout and the read has not returned. Now type a single character and wait several seconds. You will see its transpose after 3 seconds, demonstrating that the read returned with only 1 character after a delay of 30 time units. Now type 4 characters and see how quickly the program prints their transposes. The driver is waiting 3 seconds for the minimum number of characters to be entered; if they are entered before 3 seconds, it returns, otherwise it returns after 3 seconds with as many characters as were entered. Kill the process with `Ctrl-C` and turn canonical mode back on.

These experiments demonstrate that with canonical mode off, you can control input precisely. So far we have been using the `stty` command to control the terminal driver. In a program you can use `tcgetattr()` and `tcsetattr()` instead. The following listing is of a program that allows you to test the effects of various combinations of terminal settings, giving specific values to the `MIN` and `TIME` parameters in non-canonical mode.

Listing canon_mode_test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define  SLEEPTIME   2

/**
   Gets  input under the mode specified by the values of iscanon, vmin, and
   vtime. See the notes or the termios man page for the meaning of the vmin
   and vtime parameters.
 */
void get_response( int iscanon, int vmin, int vtime);

/* Sets the VMIN value to vmin, VTIME to vtime, turns off icanon */
void set_non_canonical(int vmin, int vtime);

/* if how == 0 this saves the termios state for later restoring     */
/* if how == 1 this restores the termios state from the saved state */
/* CANNOT CALL with how == 1 before first calling with how == 0     */
void tty_mode(int how);

int main(int argc, char* argv[])
{
    int  bNoCanon = 1;  /* default is non-canonical mode */
    int  vmin     = 1;  /* default is one char */
    int  vtime    = 0;  /* default is to force reads to wait for vmin chars */
    int  ch;
    char optstring[] = ":hcm:t:";

    while (1) {
        ch = getopt(argc, argv, optstring);
        if ( -1 == ch )
            break;
        switch (ch) {
        case 'h' :
            printf("Usage: %s [-hbc] [ -m MIN ] [ -t TIME ]\n",argv[0]);
            printf("       -h : help\n");
            printf("       -c : turn on canonical mode\n");
            printf("       -m : set VMIN value \n");
            printf("       -t : set VTIME value \n");
            exit(0);
        case 'c' :
            bNoCanon = 0;  /* meaning canonical mode is ON */
            break;
        case 'm' :
            vmin = strtol(optarg,'\0',0);
            break;
        case 't' :
            vtime = strtol(optarg, '\0',0);
            break;
```

```
        case '?' :
            break;
        case ':' :
            break;
        default:
           fprintf(stderr, " ");
        }
    }

    /* Report on state that we will put terminal in */
    if ( bNoCanon )
        printf("Canonical mode off\n");
    else
        printf("Canonical mode on\n");

    printf("VMIN set to %d\n",vmin);
    printf("VTIME set to %d\n",vtime);

    /* save current terminal state */
    tty_mode(0);

    /* if canonical mode flag, unset icanon */
    if ( bNoCanon )
        set_non_canonical(vmin, vtime);

    /* call function to get input */
    get_response(!bNoCanon, vmin, vtime);

    /* restore terminal state */
    tty_mode(1);

    /* flush input stream */
    tcflush(0,TCIFLUSH);
    return 0;
}

void get_response( int iscanon, int vmin, int vtime)
{
    int    num_bytes_read;
    char   input[128];

    fflush(stdout);
    sleep(SLEEPTIME);
    if ( iscanon )
        printf("About to call read() in canonical mode\n");
    else
        printf("About to call read() with MIN = %d and TIME = %d\n",
               vmin, vtime);

    printf("Enter some characters or wait to see what happens.\n");
    num_bytes_read = read(0, input, 10) ;
    if ( num_bytes_read >= 0 ) {
        input[num_bytes_read] = '\0';
        if ( num_bytes_read > 0 )
```

```
            printf("\nReturn value of read() is %d; chars read are %s\n",
                num_bytes_read, input );
        else
            printf("\nReturn value of read() is %d;"
                    " no chars were read\n",num_bytes_read);
    }
    else
        printf("read() returned -1\n");
}

void set_non_canonical(int vmin, int vtime)
{
    struct termios ttystate;

    tcgetattr( 0, &ttystate);               /*  read curr. setting    */
    ttystate.c_lflag        &= ~ICANON;     /*  no buffering          */
    ttystate.c_cc[VMIN]      =  vmin;       /*  get min of vmin chars */
    ttystate.c_cc[VTIME]     =  vtime;      /*  set timeout to vtime  */
    tcsetattr( 0 , TCSANOW, &ttystate);     /*  install settings      */
}

void tty_mode(int how)
/* if how == 0 saves the termios state  and fd flags for later restoring    */
/* if how == 1 restores the termios state and fd flags from the saved state */
/* CANNOT CALL with how == 1 before first calling with how == 0 */
{
    static struct termios original_mode;
    static int            original_flags = -1;

    if ( 0 == how ){
        tcgetattr(0, &original_mode);
        original_flags = fcntl(0, F_GETFL);
    }
    else {
        if ( -1 == original_flags ) {
            fprintf(stderr, "tty_mode(1) called without saving first.\n");
            exit(1);
        }
        tcsetattr(0, TCSANOW, &original_mode);
        fcntl( 0, F_SETFL, original_flags);
    }
}
```

Before we leave this topic, let us see what happens when we disable the ability to process signals from the terminal. When you type a `Ctrl-C` at the keyboard, you usually do so to kill the foreground process. Why does this happen? The answer is a bit deeper than we explain here, but the general idea is that the `Ctrl-C` is mapped by the terminal driver to a signal named `SIGINT`, and that signal is delivered to the foreground process. This usually results in its being terminated. A similar sequence takes place when you type other keyboard signals, such as `Ctrl-\` and `Ctrl-Z`. These do not result in the same signals, but they are delivered to the process nonetheless.

The terminal is aware of three special control characters, known as the *interrupt*, *quit*, and *suspend* characters, usually `Ctrl-C`, `Ctrl-\`, and `Ctrl-Z` respectively. The `isig` switch to `stty` enables

detection and transmission of these characters. We will experiment with this idea. Have a second terminal window open before you start this. Make sure you restore the terminal to canonical mode and type the following:

```
$ stty -isig ; transpose
```

You can now type a line of characters and transpose will correctly transpose them. But try to terminate the program with `Ctrl-C` and you will be disappointed. Nor can you cannot stop it with `Ctrl-Z` or `Ctrl-\`. The terminal will not interpret these character codes as signals. Now you will have to issue a `kill -9` to terminate the `transpose` process in the other terminal window. You can use `pkill` to do this; for example:

```
$  pkill -9 transpose
```

You can remap other characters to these special characters using the terminal driver. It is a good exercise to try.

4.4.7 Other Terminal Processing Modes

There are a large number of possible combinations of terminal settings that make the terminal behave somewhere between the extremes of canonical mode and *raw mode*, in which most, but not all processing is turned off. For example, the driver can buffer characters but not allow editing. It can buffer characters, allow editing, but not echo anything. Most of these other modes are nameless. Raw mode was the name of a mode defined in Version 7 UNIX, as was *cbreak* mode. Programs that need complete control of their windows or the screen operate in raw mode usually, since they have to turn off all processing and do it on their own.

Regardless of the input mode of the terminal, the terminal driver must manage flow control of output. A process will typically generate characters faster than the hardware can display them. The driver has to block the process when the buffers are full.

4.5 I/O Control Using `ioctl`

The `fcntl()` call accesses disk driver attributes and the `tcgetattr()` call accesses terminal driver attributes. UNIX provides a more general-purpose device-control system call, `ioctl()`, which can be used to access and control any I/O device for which the manufacturer has provided a device driver. Although most operations on devices can be achieved with the previously described system calls, most devices also have some device-specific operations that do not fit into the general model, such as

- changing the character font used on a terminal,
- telling a magnetic tape system to rewind or fast forward[14],
- ejecting a disk from a drive,

[14]Since tapes do not move in byte increments, `lseek()` cannot do this.

- playing an audio track from a CD-ROM drive, and
- maintaining routing tables for a network.

The `ioctl()` function was designed to overcome these problems. It was first introduced in Version 7 AT&T UNIX as a general purpose I/O control program, to allow user level programs to access device drivers, hidden within the depths of the kernel. As time went on, more and more devices were handled through `ioctl()` calls. However, POSIX separated out the terminal control functions into the `termios` structure and functions, and for the most part, the `ioctl()` function is used today for accessing other devices. POSIX does not define `ioctl()`, but GNU includes the `ioctl()` function in its distribution of the C Standard Library.

The various device operations, known as *IOCTLs*, are assigned code numbers and multiplexed through the `ioctl()` function, which is defined in `<sys/ioctl.h>`. The code numbers themselves are defined in many different headers.

The `ioctl()` system call is given a file descriptor of a device file, and an IOCTL, which is an integer that represents a particular request that `ioctl()` should execute. If the request requires arguments, these are included after the command number. The man page begins as follows:

```
NAME
       ioctl - control device
SYNOPSIS
       #include <sys/ioctl.h>
       int ioctl(int fildes, int request, /* arg */ ...);
DESCRIPTION
    The  ioctl() function manipulates the underlying device
    parameters of special files. In particular, many operating
    characteristics of character special files (e.g.,terminals)
    may be controlled with ioctl() requests. The argument d
    must be an open file descriptor.

    The second argument is a device-dependent request code. The
    third argument is an untyped  pointer to memory.  ...
```

There are over 400 different request types, each defined by a different mnemonic code. The `ioctl_list` man page contains the list of valid request mnemonics that can be passed to `ioctl()` together with their argument types. Each device driver can define its own set of `ioctl` commands. The system, however, provides generic `ioctl` commands for different classes of devices. Generally speaking, the commands for the different classes have a prefix that identifies the type of command. For example, the magnetic tape commands are of the form `MTIOxxx` and can be found in `<sys/mtio.h>`, and the terminal I/O commands are of the form `TIOxxx`.

A simple example of an `ioctl()` is the call to retrieve the size of a terminal window. Terminal window sizes are defined by the `winsize` structure in the `<ioctl-types.h>` header file, which is included in `<sys/ioctl.h>`. The `winsize` structure looks like:

```
struct winsize
{
    unsigned short int ws_row;
    unsigned short int ws_col;
    unsigned short int ws_xpixel;
    unsigned short int ws_ypixel;
};
```

The pixel width and height are not normally assigned values by the kernel, but the rows and columns fields are. Therefore, we can obtain the rows and columns with a simple program such as the following.

```
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

void get_winsize(int fd,
                 unsigned short *rows,
                 unsigned short *cols )
{
    struct winsize size;

    if (ioctl(fd, TIOCGWINSZ, &size) < 0) {
        perror("TIOCGWINSZ error");
        return;
    }
    *rows = size.ws_row;
    *cols = size.ws_col;
}

int main(int argc, char* argv[])
{
    unsigned short int rows, cols;

    if (isatty(STDIN_FILENO) == 0) {
        fprintf(stderr, "Not a terminal\n");
        exit(1);
    }
    get_winsize(STDIN_FILENO, &rows, &cols);
    if ( rows > 0 )
        printf("%d rows, %d columns\n", rows, cols);
```

```
    return 0;
}
```

In general, one should try to avoid the use of `ioctl()` if there is an alternative means of performing the same task, as it is not completely portable. There are many IOCTLs for which alternative functions exist. For example, there are IOCTLS for the terminal interface. The `tty_ioctl` manpage has a complete list of them. Some have equivalents and others do not:

IOCTL	**argument**	**equivalent to**
`TCGETS`	`struct termios *argp`	`tcgetattr(fd, argp)`
`TCSETS`	`const struct termios *argp`	`tcsetattr(fd, TCSANOW, argp)`
`TCSETSW`	`const struct termios *argp`	`tcsetattr(fd, TCSADRAIN, argp)`
`TCSETSF`	`const struct termios *argp`	`tcsetattr(fd, TCSAFLUSH, argp)`
`TCSBRK`	`int arg`	`tcsendbreak(fd, arg)`
`TIOCINQ`	`int *argp`	*none - counts bytes in the input buffer*
`TIOCOUTQ`	`int *argp`	*none - counts bytes in output buffer*
`TCFLSH`	`int arg`	`tcflush(fd, arg).`
`TIOCSTI`	`char *argp`	*none – inserts *argp into input queue*

We could rewrite the `setecho` program to use these IOCTLs instead of using `tcgetattr()` and `tcsetattr()`, simply as an exercise. It is displayed in the listing that follows.

```
Listing setecho_ioctl.c

int main(int argc, char *argv[])
{
    struct termios info;
    FILE   *fp;

    if ( argc < 2 ) {
        printf("usage: %s [y|n]\n", argv[0]);
        exit(1);
    }

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
         return(1);

    /* retrieve termios struct */
    if ( ioctl( fileno(fp) , TCGETS, &info ) == -1 )
        die("ioctl", "1");

    /* If second argument starts with 'y' it is yes, echo on otherwise off */
    if ( 'y' == argv[1][0] )
        info.c_lflag |= ECHO ;
    else
        info.c_lflag &= ~ECHO ;

    /* replace termios with the modified copy */
    if ( ioctl(fileno(fp),TCSETS, &info) == -1 )
        die("ioctl","2");

    return 0;
}
```

Another use of `ioctl()` is to manipulate the terminal input queue. One can use the `TIOCSTI` IOCTL to insert bytes into a terminal input queue. The following listing demonstrates how one can insert bytes into the input queue of a pseudoterminal.

It repeatedly inserts a single 'x' into the terminal input queue of the controlling terminal of the process, `BUFFERSIZE-1` times, after which it inserts a newline so that if the terminal is in canonical mode, a call to `read()` will deliver the characters. It then removes the characters from the queue by entering a loop that reads a single character at a time. To allow a second program to be run to watch the queue size, it sleeps a bit before starting to read from the terminal.

During the read loop, it sleeps enough time so that the second program can watch the queue size diminish. Without this bit of delay between each read, the watching program would not see the queue decrease by one character at a time, but would just see it go straight to zero. When it is finished reading, it writes it to the standard output.

```
Listing ioctl2.c
#include <unistd.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdio.h>

#define BUFFERSIZE   30
int main ()
{
    int  i;
    char ch      = 'x';
    char newline = '\n';
    char buf[BUFFERSIZE];

    /* Insert BUFFERSIZE-1 'x's into the terminal input queue */
    for ( i = 0; i < BUFFERSIZE-1; i++ )
        ioctl(0, TIOCSTI, &ch);

    /* Because we assume canonical mode is on, we put a newline there
       so that when the read is issued, the characters will be
       transmitted */
    ioctl(0,TIOCSTI, &newline);

    /* Delay to allow queue-reading demo to start up */
    sleep(10);

    /* Read the queue one char at a time */
    i = 0;
    while ( read(0 , &ch, 1) > 0 ) {
        buf[i] = ch;
        i++;
        if ( ch == '\n' ) /* to exit loop */
            break;
        /* Delay a bit so other program can see queue size drop */
        usleep(100000);
    }

   /* Write the buffer contents to the terminal */
```

```
    write(1, &buf, i);
    return 0;
}
```

The last demo program shows how the `TIOCINQ` IOCTL can be used to query the size of the input queue. It opens the terminal device file specified on the command line, and then enters a loop of fixed duration in which it uses the `ioctl()` function to query the size of that terminal's input queue. It writes the size of the queue into a file named `queue_log` in the current working directory.

```
Listing ioctl3.c
#include   <stdio.h>
#include   <unistd.h>
#include   <stdlib.h>
#include   <termios.h>
#include   <fcntl.h>
#include   <sys/ioctl.h>
#include  "utils.h"

int main(int argc, char* argv[] )
{
    FILE   *fp;
    int     count, i;
    int     fd;

    /* Try to open given terminal device file */
    if ((fd = open(argv[1], O_RDONLY)) == -1)
        die("open", argv[1]);

    /* Open queue log in current working directory */
    if ((fp = fopen("queue_log", "w")) == NULL) {
        printf("Could not open queue_log\n");
        exit(1);
    }

    /* For a fixed amount of time (for simplicity) count queue size */
    for ( i = 0; i < 400; i++ ) {
        if ( ioctl( fd, TIOCINQ, &count ) == -1 )
            die( "ioctl", "TIOCINQ");
        fprintf(fp, "%d chars in queue\n", count);
        /* delay to see changes */
        usleep(100000);
    }

    fclose(fp);
    return 0;
}
```

If you start up `ioctl2` and note which terminal it is in, and then start `ioctl3` in a different terminal, specifying the first terminal as its argument, when `ioctl3` terminates, look at the contents of

`queue_log` and you will see the queue size diminish with each read.

One could write a program that kept track of the input queue size of all open terminals, with root privilege, for administrative purposes, using this idea.

4.6 Summary

A process's connnection to a file is encapsulated in a data structure maintained by the kernel, invaraiably called something like a file structure or a file entry. That structure contains flags that determine how the I/O is performed. A process can set various flags when it first opens the file, using the `open()` system call, and it can modify various flags using the `fcntl()` system call.

Terminals and pseudo-terminals are represented by device files, and as such, `fcntl()` can also be used to modify a process's connection to them ,but in addition, there is an API for altering the characteristics of terminals directly. This chapter explored the ways in which a process could control the terminal characteristics, both by altering them directly and by altering the process's connection to terminals. It also showed how the non-POSIX `ioctl()` function could be used for controlling terminals and other devices.