II

Mastering Linux

6

Devices

Linux, like most operating systems, interacts with hardware devices via modularized software components called *device drivers*. A device driver hides the peculiarities of a hardware device's communication protocols from the operating system and allows the system to interact with the device through a standardized interface.

Under Linux, device drivers are part of the kernel and may be either linked statically into the kernel or loaded on demand as kernel modules. Device drivers run as part of the kernel and aren't directly accessible to user processes. However, Linux provides a mechanism by which processes can communicate with a device driver—and through it with a hardware device—via file-like objects. These objects appear in the file system, and programs can open them, read from them, and write to them practically as if they were normal files. Using either Linux's low-level I/O operations (see Appendix B, "Low-Level I/O") or the standard C library's I/O operations, your programs can communicate with hardware devices through these file-like objects.

Linux also provides several file-like objects that communicate directly with the kernel rather than with device drivers. These aren't linked to hardware devices; instead, they provide various kinds of specialized behavior that can be of use to application and system programs.

Exercise Caution When Accessing Devices!

The techniques in this chapter provide direct access to device drivers running in the Linux kernel, and through them to hardware devices connected to the system. Use these techniques with care because misuse can cause impair or damage the GNU/Linux system.

See especially the sidebar "Dangers of Block Devices."

6.1 Device Types

Device files aren't ordinary files—they do not represent regions of data on a disk-based file system. Instead, data read from or written to a device file is communicated to the corresponding device driver, and from there to the underlying device. Device files come in two flavors:

- A *character device* represents a hardware device that reads or writes a serial stream of data bytes. Serial and parallel ports, tape drives, terminal devices, and sound cards are examples of character devices.
- A *block device* represents a hardware device that reads or writes data in fixed-size blocks. Unlike a character device, a block device provides random access to data stored on the device. A disk drive is an example of a block device.

Typical application programs will never use block devices. While a disk drive is represented as block devices, the contents of each disk partition typically contain a file system, and that file system is mounted into GNU/Linux's root file system tree. Only the kernel code that implements the file system needs to access the block device directly; application programs access the disk's contents through normal files and directories.

Dangers of Block Devices

Block devices provide direct access to disk drive data. Although most GNU/Linux systems are configured to prevent nonroot processes from accessing these devices directly, a root process can inflict severe damage by changing the contents of the disk. By writing to a disk block device, a program can modify or destroy file system control information and even a disk's partition table and master boot record, thus rendering a drive or even the entire system unusable. Always access these devices with great care.

Applications sometimes make use of character devices, though. We'll discuss several of them in the following sections.

6.2 Device Numbers

Linux identifies devices using two numbers: the *major device number* and the *minor device number*. The major device number specifies which driver the device corresponds to. The correspondence from major device numbers to drivers is fixed and part of the Linux kernel sources. Note that the same major device number may correspond to

two different drivers, one a character device and one a block device. Minor device numbers distinguish individual devices or components controlled by a single driver. The meaning of a minor device number depends on the device driver.

For example, major device no. 3 corresponds to the primary IDE controller on the system. An IDE controller can have two devices (disk, tape, or CD-ROM drives) attached to it; the "master" device has minor device no. 0, and the "slave" device has minor device no. 64. Individual partitions on the master device (if the device supports partitions) are represented by minor device numbers 1, 2, 3, and so on. Individual partitions on the slave device are represented by minor device numbers 65, 66, 67, and so on.

Major device numbers are listed in the Linux kernel sources documentation. On many GNU/Linux distributions, this documentation can be found in `/usr/src/linux/Documentation/devices.txt`. The special entry `/proc/devices` lists major device numbers corresponding to active device drivers currently loaded into the kernel. (See Chapter 7, "The `/proc` File System," for more information about `/proc` file system entries.)

6.3 Device Entries

A device entry is in many ways the same as a regular file. You can move it using the `mv` command and delete it using the `rm` command. If you try to copy a device entry using `cp`, though, you'll read bytes from the device (if the device supports reading) and write them to the destination file. If you try to overwrite a device entry, you'll write bytes to the corresponding device instead.

You can create a device entry in the file system using the `mknod` command (invoke `man 1 mknod` for the man page) or the `mknod` system call (invoke `man 2 mknod` for the man page). Creating a device entry in the file system doesn't automatically imply that the corresponding device driver or hardware device is present or available; the device entry is merely a portal for communicating with the driver, if it's there. Only superuser processes can create block and character devices using the `mknod` command or the `mknod` system call.

To create a device using the `mknod` command, specify as the first argument the path at which the entry will appear in the file system. For the second argument, specify `b` for a block device or `c` for a character device. Provide the major and minor device numbers as the third and fourth arguments, respectively. For example, this command makes a character device entry named `lp0` in the current directory. The device has major device no. 6 and minor device no. 0. These numbers correspond to the first parallel port on the Linux system.

```
% mknod ./lp0 c 6 0
```

Remember that only superuser processes can create block and character devices, so you must be logged in as `root` to invoke this command successfully.

The `ls` command displays device entries specially. If you invoke `ls` with the `-l` or `-o` options, the first character on each line of output specifies the type of the entry. Recall that `-` (a hyphen) designates a normal file, while `d` designates a directory. Similarly, `b` designates a block device, and `c` designates a character device. For the latter two, `ls` prints the major and minor device numbers where it would the size of an ordinary file. For example, we can display the block device that we just created:

```
% ls -l lp0
crw-r-----    1 root     root       6,   0 Mar  7 17:03 lp0
```

In a program, you can determine whether a file system entry is a block or character device and then retrieve its device numbers using `stat`. See Section B.2, "`stat`," in Appendix B, for instructions.

To remove the entry, use `rm`. This doesn't remove the device or device driver; it simply removes the device entry from the file system.

```
% rm ./lp0
```

6.3.1 The */dev* Directory

By convention, a GNU/Linux system includes a directory `/dev` containing the full complement of character and block device entries for devices that Linux knows about. Entries in `/dev` have standardized names corresponding to major and minor device numbers.

For example, the master device attached to the primary IDE controller, which has major and minor device numbers 3 and 0, has the standard name `/dev/hda`. If this device supports partitions, the first partition on it, which has minor device no. 1, has the standard name `/dev/hda1`. You can check that this is true on your system:

```
% ls -l /dev/hda /dev/hda1
brw-rw----    1 root     disk       3,   0 May  5  1998 /dev/hda
brw-rw----    1 root     disk       3,   1 May  5  1998 /dev/hda1
```

Similarly, `/dev` has an entry for the parallel port character device that we used previously:

```
% ls -l /dev/lp0
crw-rw----    1 root     daemon     6,   0 May  5  1998 /dev/lp0
```

In most cases, you should not use `mknod` to create your own device entries. Use the entries in `/dev` instead. Non-superuser programs have no choice but to use preexisting device entries because they cannot create their own. Typically, only system administrators and developers working with specialized hardware devices will need to create device entries. Most GNU/Linux distributions include facilities to help system administrators create standard device entries with the correct names.

6.3.2 Accessing Devices by Opening Files

How do you use these devices? In the case of character devices, it can be quite simple: Open the device as if it were a normal file, and read from or write to it. You can even use normal file commands such as `cat`, or your shell's redirection syntax, to send data to or from the device.

For example, if you have a printer connected to your computer's first parallel port, you can print files by sending them directly to `/dev/lp0`.[1] To print the contents of `document.txt`, invoke the following:

```
% cat document.txt > /dev/lp0
```

You must have permission to write to the device entry for this to succeed; on many GNU/Linux systems, the permissions are set so that only `root` and the system's printer daemon (`lpd`) can write to the file. Also, what comes out of your printer depends on how your printer interprets the contents of the data you send it. Some printers will print plain text files that are sent to them,[2] while others will not. PostScript printers will render and print PostScript files that you send to them.

In a program, sending data to a device is just as simple. For example, this code fragment uses low-level I/O functions to send the contents of a buffer to `/dev/lp0`.

```
int fd = open ("/dev/lp0", O_WRONLY);
write (fd, buffer, buffer_length);
close (fd);
```

6.4 Hardware Devices

Some common block devices are listed in Table 6.1. Device numbers for similar devices follow the obvious pattern (for instance, the second partition on the first SCSI drive is `/dev/sda2`). It's occasionally useful to know which devices these device names correspond to when examining mounted file systems in `/proc/mounts` (see Section 7.5, "Drives, Mounts, and File Systems," in Chapter 7, for more about this).

Table 6.1 **Partial Listing of Common Block Devices**

Device	Name	Major	Minor
First floppy drive	`/dev/fd0`	2	0
Second floppy drive	`/dev/fd1`	2	1
Primary IDE controller, master device	`/dev/hda`	3	0
Primary IDE controller, master device, first partition	`/dev/hda1`	3	1

continues

1. Windows users will recognize that this device is similar to the magic Windows file `LPR1`.
2. Your printer may require explicit carriage return characters, ASCII code 14, at the end of each line, and may require a form feed character, ASCII code 12, at the end of each page.

Table 6.1 **Continued**

Device	Name	Major	Minor
Primary IDE controller, secondary device	`/dev/hdb`	3	64
Primary IDE controller, secondary device, first partition	`/dev/hdb1`	3	65
Secondary IDE controller, master device	`/dev/hdc`	22	0
Secondary IDE controller, secondary device	`/dev/hdd`	22	64
First SCSI drive	`/dev/sda`	8	0
First SCSI drive, first partition	`/dev/sda1`	8	1
Second SCSI disk	`/dev/sdb`	8	16
Second SCSI disk, first partition	`/dev/sdb1`	8	17
First SCSI CD-ROM drive	`/dev/scd0`	11	0
Second SCSI CD-ROM drive	`/dev/scd1`	11	1

Table 6.2 lists some common character devices.

Table 6.2 **Some Common Character Devices**

Device	Name	Major	Minor
Parallel port 0	`/dev/lp0` or `/dev/par0`	6	0
Parallel port 1	`/dev/lp1` or `/dev/par1`	6	1
First serial port	`/dev/ttyS0`	4	64
Second serial port	`/dev/ttyS1`	4	65
IDE tape drive	`/dev/ht0`	37	0
First SCSI tape drive	`/dev/st0`	9	0
Second SCSI tape drive	`/dev/st1`	9	1
System console	`/dev/console`	5	1
First virtual terminal	`/dev/tty1`	4	1
Second virtual terminal	`/dev/tty2`	4	2
Process's current terminal device	`/dev/tty`	5	0
Sound card	`/dev/audio`	14	4

You can access certain hardware components through more than one character device; often, the different character devices provide different semantics. For example, when you use the IDE tape device `/dev/ht0`, Linux automatically rewinds the tape in the drive when you close the file descriptor. You can use the device `/dev/nht0` to access the same tape drive, except that Linux will not automatically rewind the tape when you close the file descriptor. You sometimes might see programs using `/dev/cua0` and similar devices; these are older interfaces to serial ports such as `/dev/ttyS0`.

Occasionally, you'll want to write data directly to character devices—for example:

- A terminal program might access a modem directly through a serial port device. Data written to or read from the devices is transmitted via the modem to a remote computer.
- A tape backup program might write data directly to a tape device. The backup program could implement its own compression and error-checking format.
- A program can write directly to the first virtual terminal[3] writing data to `/dev/tty1`.

  Terminal windows running in a graphical environment, or remote login terminal sessions, are not associated with virtual terminals; instead, they're associated with pseudo-terminals. See Section 6.6, "PTYs," for information about these.
- Sometimes a program needs to access the terminal device with which it is associated.

  For example, your program may need to prompt the user for a password. For security reasons, you might want to ignore redirection of standard input and output and always read the password from the terminal, no matter how the user invokes the command. One way to do this is to open `/dev/tty`, which always corresponds to the terminal device associated with the process that opens it. Write the prompt message to that device, and read the password from it. By ignoring standard input and output, this prevents the user from feeding your program a password from a file using shell syntax such as this:

  ```
  % secure_program < my-password.txt
  ```

  If you need to authenticate users in your program, you should learn about GNU/Linux's PAM facility. See Section 10.5, "Authenticating Users," in Chapter 10, "Security," for more information.
- A program can play sounds through the system's sound card by sending audio data to `/dev/audio`. Note that the audio data must be in Sun audio format (usually associated with the `.au` extension).

  For example, many GNU/Linux distributions come with the classic sound file `/usr/share/sndconfig/sample.au`. If your system includes this file, try playing it by invoking the following:

  ```
  % cat /usr/share/sndconfig/sample.au > /dev/audio
  ```

  If you're planning on using sound in your program, though, you should investigate the various sound libraries and services available for GNU/Linux. The Gnome windowing environment uses the Enlightenment Sound Daemon (EsounD), at `http://www.tux.org/~ricdude/EsounD.html`. KDE uses aRts, at `http://space.twc.de/~stefan/kde/arts-mcop-doc/`. If you use one of these sound systems instead of writing directly to `/dev/audio`, your program will cooperate better with other programs that use the computer's sound card.

3. On most GNU/Linux systems, you can switch to the first virtual terminal by pressing Ctrl+Alt+F1. Use Ctrl+Alt+F2 for the second virtual terminal, and so on.

6.5 Special Devices

Linux also provides several character devices that don't correspond to hardware devices. These entries all use the major device no. 1, which is associated with the Linux kernel's memory device instead of a device driver.

6.5.1 */dev/null*

The entry `/dev/null`, the *null device*, is very handy. It serves two purposes; you are probably familiar at least with the first one:

- Linux discards any data written to `/dev/null`. A common trick is to specify `/dev/null` as an output file in some context where the output is unwanted.

  For example, to run a command and discard its standard output (without printing it or writing it to a file), redirect standard output to `/dev/null`:

  ```
  % verbose_command > /dev/null
  ```

- Reading from `/dev/null` always results in an end-of-file. For instance, if you open a file descriptor to `/dev/null` using `open` and then attempt to `read` from the file descriptor, `read` will read no bytes and will return 0. If you copy from `/dev/null` to another file, the destination will be a zero-length file:

  ```
  % cp /dev/null empty-file
  % ls -l empty-file
  -rw-rw----    1 samuel   samuel          0 Mar  8 00:27 empty-file
  ```

6.5.2 */dev/zero*

The device entry `/dev/zero` behaves as if it were an infinitely long file filled with 0 bytes. As much data as you'd try to read from `/dev/zero`, Linux "generates" enough 0 bytes.

To illustrate this, let's run the hex dump program presented in Listing B.4 in Section B.1.4, "Reading Data," of Appendix B. This program prints the contents of a file in hexadecimal form.

```
% ./hexdump /dev/zero
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

Hit Ctrl+C when you're convinced that it will go on indefinitely.

Memory mapping `/dev/zero` is an advanced technique for allocating memory. See Section 5.3.5, "Other Uses for `mmap`," in Chapter 5, "Interprocess Communication," for more information, and see the sidebar "Obtaining Page-Aligned Memory" in Section 8.9, "`mprotect`: Setting Memory Permissions," in Chapter 8, "Linux System Calls," for an example.

6.5.3 */dev/full*

The entry `/dev/full` behaves as if it were a file on a file system that has no more room. A write to `/dev/full` fails and sets `errno` to `ENOSPC`, which ordinarily indicates that the written-to device is full.

For example, you can try to write to `/dev/full` using the `cp` command:

```
% cp /etc/fstab /dev/full
cp: /dev/full: No space left on device
```

The `/dev/full` entry is primarily useful to test how your program behaves if it runs out of disk space while writing to a file.

6.5.4 Random Number Devices

The special devices `/dev/random` and `/dev/urandom` provide access to the Linux kernel's built-in random number–generation facility.

Most software functions for generating random numbers, such as the `rand` function in the standard C library, actually generate *pseudorandom* numbers. Although these numbers satisfy some properties of random numbers, they are reproducible: If you start with the same seed value, you'll obtain the same sequence of pseudorandom numbers every time. This behavior is inevitable because computers are intrinsically deterministic and predictable. For certain applications, though, this behavior is undesirable; for instance, it is sometimes possible to break a cryptographic algorithm if you can obtain the sequence of random numbers that it employs.

To obtain better random numbers in computer programs requires an external source of randomness. The Linux kernel harnesses a particularly good source of randomness: *you!* By measuring the time delay between your input actions, such as keystrokes and mouse movements, Linux is capable of generating an unpredictable stream of high-quality random numbers. You can access this stream by reading from `/dev/random` and `/dev/urandom`. The data that you read is a stream of randomly generated bytes.

The difference between the two devices exhibits itself when Linux exhausts its store of randomness. If you try to read a large number of bytes from `/dev/random` but don't generate any input actions (you don't type, move the mouse, or perform a similar action), Linux blocks the read operation. Only when you provide some randomness does Linux generate some more random bytes and return them to your program.

For example, try displaying the contents of `/dev/random` using the `od` command.[4] Each row of output shows 16 random bytes.

4. We use `od` here instead of the `hexdump` program presented in Listing B.4, even though they do pretty much the same thing, because `hexdump` terminates when it runs out of data, while `od` waits for more data to become available. The `-t x1` option tells `od` to print file contents in hexadecimal.

```
% od -t x1 /dev/random
0000000 2c 9c 7a db 2e 79 3d 65 36 c2 e3 1b 52 75 1e 1a
0000020 d3 6d 1e a7 91 05 2d 4d c3 a6 de 54 29 f4 46 04
0000040 b3 b0 8d 94 21 57 f3 90 61 dd 26 ac 94 c3 b9 3a
0000060 05 a3 02 cb 22 0a bc c9 45 dd a6 59 40 22 53 d4
```

The number of lines of output that you see will vary—there may be quite a few—but the output will eventually pause when Linux exhausts its store of randomness. Now try moving your mouse or typing on the keyboard, and watch additional random numbers appear. For even better randomness, let your cat walk on the keyboard.

A read from `/dev/urandom`, in contrast, will never block. If Linux runs out of randomness, it uses a cryptographic algorithm to generate pseudorandom bytes from the past sequence of random bytes. Although these bytes are random enough for many purposes, they don't pass as many tests of randomness as those obtained from `/dev/random`.

For instance, if you invoke the following, the random bytes will fly by forever, until you kill the program with Ctrl+C:

```
% od -t x1 /dev/urandom
0000000 62 71 d6 3e af dd de 62 c0 42 78 bd 29 9c 69 49
0000020 26 3b 95 bc b9 6c 15 16 38 fd 7e 34 f0 ba ce c3
0000040 95 31 e5 2c 8d 8a dd f4 c4 3b 9b 44 2f 20 d1 54
...
```

Using random numbers from `/dev/random` in a program is easy, too. Listing 6.1 presents a function that generates a random number using bytes read from in `/dev/random`. Remember that `/dev/random` blocks a read until there is enough randomness available to satisfy it; you can use `/dev/urandom` instead if fast execution is more important and you can live with the potential lower quality of random numbers.

Listing 6.1 **(*random_number.c*) Function to Generate a Random Number Using */dev/random***

```
#include <assert.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

/* Return a random integer between MIN and MAX, inclusive.  Obtain
   randomness from /dev/random.  */

int random_number (int min, int max)
{
  /* Store a file descriptor opened to /dev/random in a static
     variable.  That way, we don't need to open the file every time
     this function is called.  */
  static int dev_random_fd = -1;
```

```
  char* next_random_byte;
  int bytes_to_read;
  unsigned random_value;

  /* Make sure MAX is greater than MIN.  */
  assert (max > min);

  /* If this is the first time this function is called, open a file
     descriptor to /dev/random.  */
  if (dev_random_fd == -1) {
    dev_random_fd = open ("/dev/random", O_RDONLY);
    assert (dev_random_fd != -1);
  }

  /* Read enough random bytes to fill an integer variable.  */
  next_random_byte = (char*) &random_value;
  bytes_to_read = sizeof (random_value);
  /* Loop until we've read enough bytes.  Because /dev/random is filled
     from user-generated actions, the read may block and may only
     return a single random byte at a time.  */
  do {
    int bytes_read;
    bytes_read = read (dev_random_fd, next_random_byte, bytes_to_read);
    bytes_to_read -= bytes_read;
    next_random_byte += bytes_read;
  } while (bytes_to_read > 0);

  /* Compute a random number in the correct range.  */
  return min + (random_value % (max - min + 1));
}
```

6.5.5 Loopback Devices

A *loopback device* enables you to simulate a block device using an ordinary disk file. Imagine a disk drive device for which data is written to and read from a file named `disk-image` rather than to and from the tracks and sectors of an actual physical disk drive or disk partition. (Of course, the file `disk-image` must reside on an actual disk, which must be larger than the simulated disk.) A loopback device enables you to use a file in this manner.

Loopback devices are named `/dev/loop0`, `/dev/loop1`, and so on. Each can be used to simulate a single block device at one time. Note that only the superuser can set up a loopback device.

A loopback device can be used in the same way as any other block device. In particular, you can construct a file system on the device and then mount that file system as you would mount the file system on an ordinary disk or partition. Such a file system, which resides in its entirety within an ordinary disk file, is called a *virtual file system*.

To construct a virtual file system and mount it with a loopback device, follow these steps:

1. Create an empty file to hold the virtual file system. The size of the file will be the apparent size of the loopback device after it is mounted.

   One convenient way to construct a file of a fixed size is with the `dd` command. This command copies blocks (by default, 512 bytes each) from one file to another. The `/dev/zero` file is a convenient source of bytes to copy from.

   To construct a 10MB file named `disk-image`, invoke the following:

   ```
   % dd if=/dev/zero of=/tmp/disk-image count=20480
   20480+0 records in
   20480+0 records out
   % ls -l /tmp/disk-image
   -rw-rw----    1 root     root     10485760 Mar  8 01:56 /tmp/disk-image
   ```

2. The file that you've just created is filled with 0 bytes. Before you mount it, you must construct a file system. This sets up the various control structures needed to organize and store files, and builds the root directory.

   You can build any type of file system you like in your disk image. To construct an `ext2` file system (the type most commonly used for Linux disks), use the `mke2fs` command. Because it's usually run on a block device, not an ordinary file, it asks for confirmation:

   ```
   % mke2fs -q /tmp/disk-image
   mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09
   disk-image is not a block special device.
   Proceed anyway? (y,n) y
   ```

   The `-q` option suppresses summary information about the newly created file system. Leave this option out if you're curious about it.

   Now `disk-image` contains a brand-new file system, as if it were a freshly initialized 10MB disk drive.

3. Mount the file system using a loopback device. To do this, use the `mount` command, specifying the disk image file as the mount device. Also specify `loop=`*`loopback-device`* as a mount option, using the `-o` option to `mount` to tell `mount` which loopback device to use.

   For example, to mount our `disk-image` file system, invoke these commands. Remember, only the superuser may use a loopback device. The first command creates a directory, `/tmp/virtual-fs`, to use as the mount point for the virtual file system.

   ```
   % mkdir /tmp/virtual-fs
   % mount -o loop=/dev/loop0 /tmp/disk-image /tmp/virtual-fs
   ```

   Now your disk image is mounted as if it were an ordinary 10MB disk drive.

   ```
   % df -h /tmp/virtual-fs
   Filesystem             Size  Used Avail Use% Mounted on
   /tmp/disk-image        9.7M   13k  9.2M   0% /tmp/virtual-fs
   ```

   You can use it like any other disk:

   ```
   % cd /tmp/virtual-fs
   % echo 'Hello, world!' > test.txt
   % ls -l
   total 13
   drwxr-xr-x    2 root     root         12288 Mar  8 02:00 lost+found
   -rw-rw----    1 root     root            14 Mar  8 02:12 test.txt
   % cat test.txt
   Hello, world!
   ```

   Note that `lost+found` is a directory that was automatically added by `mke2fs`.[5]

5. If the file system is ever damaged, and some data is recovered but not associated with a file, it is placed in `lost+found`.

When you're done, unmount the virtual file system.

```
% cd /tmp
% umount /tmp/virtual-fs
```

You can delete `disk-image` if you like, or you can mount it later to access the files on the virtual file system. You can also copy it to another computer and mount it there—the whole file system that you created on it will be intact.

Instead of creating a file system from scratch, you can copy one directly from a device. For instance, you can create an image of the contents of a CD-ROM simply by copying it from the CD-ROM device.

If you have an IDE CD-ROM drive, use the corresponding device name, such as `/dev/hda`, described previously. If you have a SCSI CD-ROM drive, the device name will be `/dev/scd0` or similar. Your system may also have a symbolic link `/dev/cdrom` that points to the appropriate device. Consult your `/etc/fstab` file to determine what device corresponds to your computer's CD-ROM drive.

Simply copy that device to a file. The resulting file will be a complete disk image of the file system on the CD-ROM in the drive—for example:

```
% cp /dev/cdrom /tmp/cdrom-image
```

This may take several minutes, depending on the CD-ROM you're copying and the speed of your drive. The resulting image file will be quite large—as large as the contents of the CD-ROM.

Now you can mount this CD-ROM image without having the original CD-ROM in the drive. For example, to mount it on `/mnt/cdrom`, use this line:

```
% mount -o loop=/dev/loop0 /tmp/cdrom-image /mnt/cdrom
```

Because the image is on a hard disk drive, it'll perform much faster than the actual CD-ROM disk. Note that most CD-ROMs use the file system type `iso9660`.

6.6 PTYs

If you run the `mount` command with no command-line arguments, which displays the file systems mounted on your system, you'll notice a line that looks something like this:

```
none on /dev/pts type devpts (rw,gid=5,mode=620)
```

This indicates that a special type of file system, `devpts`, is mounted at `/dev/pts`. This file system, which isn't associated with any hardware device, is a "magic" file system that is created by the Linux kernel. It's similar to the `/proc` file system; see Chapter 7 for more information about how this works.

Like the `/dev` directory, `/dev/pts` contains entries corresponding to devices. But unlike `/dev`, which is an ordinary directory, `/dev/pts` is a special directory that is created dynamically by the Linux kernel. The contents of the directory vary with time and reflect the state of the running system.

The entries in `/dev/pts` correspond to *pseudo-terminals* (or *pseudo-TTYs*, or *PTYs*). Linux creates a PTY for every new terminal window you open and displays a corresponding entry in `/dev/pts`. The PTY device acts like a terminal device—it accepts input from the keyboard and displays text output from the programs that run in it. PTYs are numbered, and the PTY number is the name of the corresponding entry in `/dev/pts`.

You can display the terminal device associated with a process using the `ps` command. Specify `tty` as one of the fields of a custom format with the `-o` option. To display the process ID, TTY, and command line of each process sharing the same terminal, invoke `ps -o pid,tty,cmd`.

6.6.1 A PTY Demonstration

For example, you can determine the PTY associated with a given terminal window by invoking in the window this command:

```
% ps -o pid,tty,cmd
  PID TT       CMD
28832 pts/4    bash
29287 pts/4    ps -o pid,tty,cmd
```

This particular terminal window is running in PTY 4.

The PTY has a corresponding entry in `/dev/pts`:

```
% ls -l /dev/pts/4
crw--w----    1 samuel   tty      136,   4 Mar  8 02:56 /dev/pts/4
```

Note that it is a character device, and its owner is the owner of the process for which it was created.

You can read from or write to the PTY device. If you read from it, you'll hijack keyboard input that would otherwise be sent to the program running in the PTY. If you write to it, the data will appear in that window.

Try opening a new terminal window, and determine its PTY number by invoking `ps -o pid,tty,cmd`. From another window, write some text to the PTY device. For example, if the new terminal window's PTY number is 7, invoke this command from another window:

```
% echo 'Hello, other window!' > /dev/pts/7
```

The output appears in the new terminal window. If you close the new terminal window, the entry `7` in `/dev/pts` disappears.

If you invoke `ps` to determine the TTY from a text-mode virtual terminal (press Ctrl+Alt+F1 to switch to the first virtual terminal, for instance), you'll see that it's running in an ordinary terminal device instead of a PTY:

```
% ps -o pid,tty,cmd
  PID TT       CMD
29325 tty1     -bash
29353 tty1     ps -o pid,tty,cmd
```

6.7 *ioctl*

The `ioctl` system call is an all-purpose interface for controlling hardware devices. The first argument to `ioctl` is a file descriptor, which should be opened to the device that you want to control. The second argument is a request code that indicates the operation that you want to perform. Various request codes are available for different devices. Depending on the request code, there may be additional arguments supplying data to `ioctl`.

Many of the available requests codes for various devices are listed in the `ioctl_list` man page. Using `ioctl` generally requires a detailed understanding of the device driver corresponding to the hardware device that you want to control. Most of these are quite specialized and are beyond the scope of this book. However, we'll present one example to give you a taste of how `ioctl` is used.

Listing 6.2 **(*cdrom-eject.c*) Eject a CD-ROM**

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  /* Open a file descriptor to the device specified on the command line.  */
  int fd = open (argv[1], O_RDONLY);
  /* Eject the CD-ROM.  */
  ioctl (fd, CDROMEJECT);
  /* Close the file descriptor.  */
  close (fd);

  return 0;
}
```

Listing 6.2 presents a short program that ejects the disk in a CD-ROM drive (if the drive supports this). It takes a single command-line argument, the CD-ROM drive device. It opens a file descriptor to the device and invokes `ioctl` with the request code `CDROMEJECT`. This request, defined in the header `<linux/cdrom.h>`, instructs the device to eject the disk.

For example, if your system has an IDE CD-ROM drive connected as the master device on the secondary IDE controller, the corresponding device is `/dev/hdc`. To eject the disk from the drive, invoke this line:

```
% ./cdrom-eject /dev/hdc
```