3

Working with Files

In this chapter, you look at Linux files and directories and how to manipulate them. You learn how to create files, open them, read, write, and close them. You also learn how programs can manipulate directories (to create, scan, and delete them, for example). After the preceding chapter's diversion into shells, you now start programming in C.

Before proceeding to the way Linux handles file I/O, we review the concepts associated with files, directories, and devices. To manipulate files and directories, you need to make system calls (the UNIX and Linux parallel of the Windows API), but there also exists a whole range of library functions, the standard I/O library (stdio), to make file handling more efficient.

We spend the majority of the chapter detailing the various calls to handle files and directories. So this chapter covers various file-related topics:

- Files and devices
- System calls
- Library functions
- Low-level file access
- Managing files
- The standard I/O library
- Formatted input and output
- File and directory maintenance
- Scanning directories
- Errors
- The `/proc` file system
- Advanced topics: `fcntl` and `mmap`

Linux File Structure

"Why," you may be asking, "are we covering file structure? I know about that already." Well, as with UNIX, files in the Linux environment are particularly important, because they provide a simple and consistent interface to the operating system services and devices. In Linux, *everything is a file*. Well, almost!

This means that, in general, programs can use disk files, serial ports, printers, and other devices in exactly the same way they would use a file. We cover some exceptions, such as network connections, in Chapter 15, but mainly you need to use only five basic functions: `open`, `close`, `read`, `write`, and `ioctl`.

Directories, too, are special sorts of files. In modern UNIX versions, including Linux, even the superuser may not write to them directly. All users ordinarily use the high-level `opendir`/`readdir` interface to read directories without needing to know the system-specific details of directory implementation. We'll return to special directory functions later in this chapter.

Really, almost everything is represented as a file under Linux, or can be made available via special files. Even though there are, by necessity, subtle differences from the conventional files you know and love, the general principle still holds. Let's look at the special cases we've mentioned so far.

Directories

As well as its contents, a file has a name and some properties, or "administrative information"; that is, the file's creation/modification date and its permissions. The properties are stored in the file's *inode*, a special block of data in the file system that also contains the length of the file and where on the disk it's stored. The system uses the number of the file's inode; the directory structure just names the file for our benefit.

A directory is a file that holds the inode numbers and names of other files. Each directory entry is a link to a file's inode; remove the filename and you remove the link. (You can see the inode number for a file by using `ln -i`.) Using the `ln` command, you can make links to the same file in different directories.

When you delete a file all that happens is that the directory entry for the file is removed and the number of links to the file goes down by one. The data for the file is possibly still available through other links to the same file. When the number of links to a file (the number after the permissions in `ls -l`) reaches zero, the inode and the data blocks it references are then no longer in use and are marked as free.

Files are arranged in directories, which may also contain subdirectories. These form the familiar file system hierarchy. A user, say `neil`, usually has his files stored in a "home" directory, perhaps `/home/neil`, with subdirectories for e-mail, business letters, utility programs, and so on. Note that many command shells for UNIX and Linux have an excellent notation for getting straight to your home directory: the tilde (~). For another user, type ~**`user`**. As you know, home directories for each user are usually subdirectories of a higher-level directory created specifically for this purpose, in this case `/home`.

> *Note that the standard library functions unfortunately do not understand the shell's tilde shorthand notation in filename parameters, so you must always use the real name of the file in your programs.*

The `/home` directory is itself a subdirectory of the root directory, `/`, which sits at the top of the hierarchy and contains all of the system's files in subdirectories. The root directory normally includes `/bin` for system programs ("binaries"), `/etc` for system configuration files, and `/lib` for system libraries. Files that represent physical devices and provide the interface to those devices are conventionally found in a

directory called `/dev`. See Figure 3-1 for an example of part of a typical Linux hierarchy. We cover the Linux file system layout in more detail in Chapter 18, when we look at Linux File System Standard.

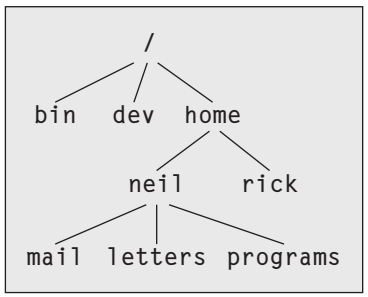

Figure 3-1

Files and Devices

Even hardware devices are very often represented (mapped) by files. For example, as the superuser, you can mount an IDE CD-ROM drive as a file:

```
# mount -t iso9660 /dev/hdc /mnt/cdrom
# cd /mnt/cdrom
```

which takes the CD-ROM device (in this case the secondary master IDE device loaded as `/dev/hdc` during boot-up; other types of device will have different `/dev` entries) and mounts its current contents as the file structure beneath `/mnt/cdrom`. You then move around within the CD-ROM's directories just as normal, except, of course, that the contents are read-only.

Three important device files found in both UNIX and Linux are `/dev/console`, `/dev/tty`, and `/dev/null`.

/dev/console

This device represents the system console. Error messages and diagnostics are often sent to this device. Each UNIX system has a designated terminal or screen to receive console messages. At one time, it might have been a dedicated printing terminal. On modern workstations, and on Linux, it's usually the "active" virtual console, and under X, it will be a special console window on the screen.

/dev/tty

The special file `/dev/tty` is an alias (logical device) for the controlling terminal (keyboard and screen, or window) of a process, if it has one. (For instance, processes and scripts run automatically by the system won't have a controlling terminal, and therefore won't be able to open `/dev/tty`.)

Where it can be used, `/dev/tty` allows a program to write directly to the user, without regard to which pseudo-terminal or hardware terminal the user is using. It is useful when the standard output has been redirected. One example is displaying a long directory listing as a group of pages with the command `ls -R | more`, where the program `more` has to prompt the user for each new page of output. You'll see more of `/dev/tty` in Chapter 5.

Note that whereas there's only one `/dev/console` device, there are effectively many different physical devices accessed through `/dev/tty`.

/dev/null

The `/dev/null` file is the null device. All output written to this device is discarded. An immediate end of file is returned when the device is read, and it can be used as a source of empty files by using the `cp` command. Unwanted output is often redirected to `/dev/null`.

> *Another way of creating empty files is to use the* `touch <filename>` *command, which changes the modification time of a file or creates a new file if none exists with the given name. It won't empty it of its contents, though.*

```
$ echo do not want to see this >/dev/null
$ cp /dev/null empty_file
```

Other devices found in `/dev` include hard and floppy disks, communications ports, tape drives, CD-ROMs, sound cards, and some devices representing the system's internal state. There's even a `/dev/zero`, which acts as a source of `null` bytes to create files full of zeros. You need superuser permissions to access some of these devices; normal users can't write programs to directly access low-level devices like hard disks. The names of the device files may vary from system to system. Linux distributions usually have applications that run as superuser to manage the devices that would otherwise be inaccessible, for example, `mount` for user-mountable file systems.

Devices are classified as either *character devices* or *block devices*. The difference refers to the fact that some devices need to be accessed a block at a time. Typically, the only block devices are those that support some type of file system, like hard disks.

In this chapter, we concentrate on disk files and directories. We cover another device, the user's terminal, in Chapter 5.

System Calls and Device Drivers

You can access and control files and devices using a small number of functions. These functions, known as *system call*s, are provided by UNIX (and Linux) directly, and are the interface to the operating system itself.

At the heart of the operating system, the kernel, are a number of *device drivers*. These are a collection of low-level interfaces for controlling system hardware. For example, there will be a device driver for a tape drive, which knows how to start the tape, wind it forward and backward, read and write to it, and so on. It will also know that tapes have to be written to in blocks of a certain size. Because tapes are sequential in nature, the driver can't access tape blocks directly, but must wind the tape to the right place. Similarly, a low-level hard disk device driver will only write whole numbers of disk sectors at a time, but will be able to access any desired disk block directly, because the disk is a random access device.

To provide a similar interface, device drivers encapsulate all of the hardware-dependent features. Idiosyncratic features of the hardware are usually available through the `ioctl` (for I/O control) system call.

Device files in `/dev` are used in the same way; they can be opened, read, written, and closed. For example, the same `open` call used to access a regular file is used to access a user terminal, a printer, or a tape drive.

The low-level functions used to access the device drivers, the system calls, include:

- `open`: Open a file or device
- `read`: Read from an open file or device
- `write`: Write to a file or device
- `close`: Close the file or device
- `ioctl`: Pass control information to a device driver

The `ioctl` system call is used to provide some necessary hardware-specific control (as opposed to regular input and output), so its use varies from device to device. For example, a call to `ioctl` can be used to rewind a tape drive or set the flow control characteristics of a serial port. For this reason, `ioctl` isn't necessarily portable from machine to machine. In addition, each driver defines its own set of `ioctl` commands.

These and other system calls are usually documented in section 2 of the manual pages. Prototypes providing the parameter lists and function return types for system calls, and associated `#define`s of constants, are provided in include files. The particular ones required for each system call will be included with the descriptions of individual calls.

Library Functions

One problem with using low-level system calls directly for input and output is that they can be very inefficient. Why? Well:

- There's a performance penalty in making a system call. System calls are therefore expensive compared to function calls because Linux has to switch from running your program code to executing its own kernel code and back again. It's a good idea to keep the number of system calls used in a program to a minimum and get each call to do as much work as possible, for example, by reading and writing large amounts of data rather than a single character at a time.
- The hardware has limitations that can impose restrictions on the size of data blocks that can be read or written by the low-level system call at any one time. For example, tape drives often have a block size, say 10k, to which they can write. So, if you attempt to write an amount that is not an exact multiple of 10k, the drive will still advance the tape to the next 10k block, leaving gaps on the tape.

To provide a higher-level interface to devices and disk files, a Linux distribution (and UNIX) provides a number of standard libraries. These are collections of functions that you can include in your own programs to handle these problems. A good example is the standard I/O library that provides buffered output. You can effectively write data blocks of varying sizes, and the library functions arrange for the low-level system calls to be provided with full blocks as the data is made available. This dramatically reduces the system call overhead.

Library functions are usually documented in section 3 of the manual pages and often have a standard include file associated with them, such as `stdio.h` for the standard I/O library.

To summarize the discussion of the last few sections, Figure 3-2 illustrates the Linux system, showing where the various file functions exist relative to the user, the device drivers, the kernel, and the hardware.

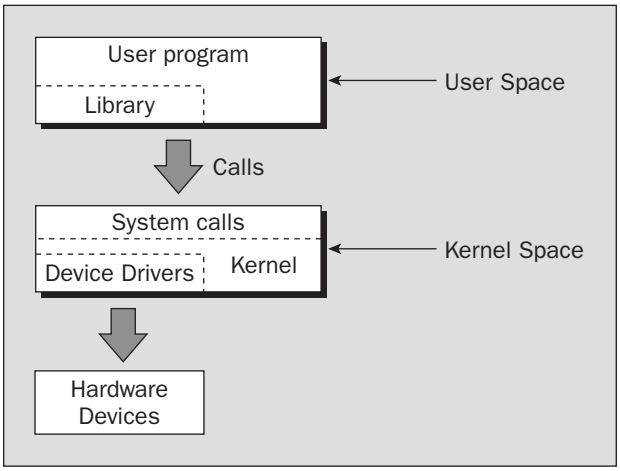

Figure 3-2

Low-Level File Access

Each running program, called a *process*, has a number of file descriptors associated with it. These are small integers that you can use to access open files or devices. How many of these are available will vary depending on how the system has been configured. When a program starts, it usually has three of these descriptors already opened. These are:

- 0: Standard input
- 1: Standard output
- 2: Standard error

You can associate other file descriptors with files and devices by using the `open` system call, which we discuss shortly. The file descriptors that are automatically opened, however, already allow you to create some simple programs using `write`.

write

The `write` system call arranges for the first `nbytes` bytes from `buf` to be written to the file associated with the file descriptor `fildes`. It returns the number of bytes actually written. This may be less than `nbytes` if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the `write` call, and the error will be specified in the `errno` global variable.

Here's the syntax:

```
#include <unistd.h>

size_t write(int fildes, const void *buf, size_t nbytes);
```

With this knowledge, you can write your first program, `simple_write.c`:

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

int main()
{
    if ((write(1, "Here is some data\n", 18)) != 18)
        write(2, "A write error has occurred on file descriptor 1\n",46);

    exit(0);
}
```

This program simply prints a message to the standard output. When a program exits, all open file descriptors are automatically closed, so you don't need to close them explicitly. This won't be the case, however, when you're dealing with buffered output.

```
$ ./simple_write
Here is some data
$
```

A point worth noting again is that `write` might report that it wrote fewer bytes than you asked it to. This is not necessarily an error. In your programs, you will need to check `errno` to detect errors and call `write` to write any remaining data.

read

The `read` system call reads up to `nbytes` bytes of data from the file associated with the file descriptor `fildes` and places them in the data area `buf`. It returns the number of data bytes actually read, which may be less than the number requested. If a `read` call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

```
#include <unistd.h>

size_t read(int fildes, void *buf, size_t nbytes);
```

This program, `simple_read.c`, copies the first 128 bytes of the standard input to the standard output. It copies all of the input if there are fewer than 128 bytes.

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

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);
```

```
    exit(0);
}
```

If you run the program, you should see the following:

```
$ echo hello there | ./simple_read
hello there
$ ./simple_read < draft1.txt
Files
In this chapter we will be looking at files and directories and how to manipulate
them. We will learn how to create files,$
```

In the first execution, you create some input for the program using `echo`, which is piped to your program. In the second execution, you redirect input from a file. In this case, you see the first part of the file `draft1.txt` appearing on the standard output.

Note how the next shell prompt appears at the end of the last line of output because, in this example, the 128 bytes don't form a whole number of lines.

open

To create a new file descriptor, you need to use the `open` system call.

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

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

Strictly speaking, you don't need to include `sys/types.h` *and* `sys/stat.h` *to use* `open` *on systems that comply with POSIX standards, but they may be necessary on some UNIX systems.*

In simple terms, `open` establishes an access path to a file or device. If successful, it returns a file descriptor that can be used in `read`, `write`, and other system calls. The file descriptor is unique and isn't shared by any other processes that may be running. If two programs have a file open at the same time, they maintain distinct file descriptors. If they both write to the file, they will continue to write where they left off. Their data isn't interleaved, but one will overwrite the other. Each keeps its own idea of how far into the file (the offset) it has read or written. You can prevent unwanted clashes of this sort by using file locking, which you'll see in Chapter 7.

The name of the file or device to be opened is passed as a parameter, `path`; the `oflags` parameter is used to specify actions to be taken on opening the file.

The `oflags` are specified as a combination of a mandatory file access mode and other optional modes. The `open` call must specify one of the file access modes shown in the following table:

Mode	Description
O_RDONLY	Open for read-only
O_WRONLY	Open for write-only
O_RDWR	Open for reading and writing

The call may also include a combination (using a bitwise OR) of the following optional modes in the oflags parameter:

- O_APPEND: Place written data at the end of the file.
- O_TRUNC: Set the length of the file to zero, discarding existing contents.
- O_CREAT: Creates the file, if necessary, with permissions given in mode.
- O_EXCL: Used with O_CREAT, ensures that the caller creates the file. The open is atomic; that is, it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

Other possible values for oflags are documented in the open manual page, which you can find in section 2 of the manual pages (use man 2 open).

open returns the new file descriptor (always a nonnegative integer) if successful, or –1 if it fails, at which time open also sets the global variable errno to indicate the reason for the failure. We look at errno more closely in a later section. The new file descriptor is always the lowest-numbered unused descriptor, a feature that can be quite useful in some circumstances. For example, if a program closes its standard output and then calls open again, the file descriptor 1 will be reused and the standard output will have been effectively redirected to a different file or device.

There is also a creat call standardized by POSIX, but it is not often used. creat doesn't only create the file, as one might expect, but also opens it. It is the equivalent of calling open with oflags equal to O_CREAT|O_WRONLY|O_TRUNC.

The number of files that any one running program may have open at once is limited. The limit, usually defined by the constant OPEN_MAX in limits.h, varies from system to system, but POSIX requires that it be at least 16. This limit may itself be subject to local system-wide limits so that a program may not always be able to open this many files. On Linux, the limit may be changed at runtime so OPEN_MAX is not a constant. It typically starts out at 256.

Initial Permissions

When you create a file using the O_CREAT flag with open, you must use the three-parameter form. mode, the third parameter, is made from a bitwise OR of the flags defined in the header file sys/stat.h. These are:

- S_IRUSR: Read permission, owner
- S_IWUSR: Write permission, owner

- S_IXUSR: Execute permission, owner
- S_IRGRP: Read permission, group
- S_IWGRP: Write permission, group
- S_IXGRP: Execute permission, group
- S_IROTH: Read permission, others
- S_IWOTH: Write permission, others
- S_IXOTH: Execute permission, others

For example,

```
open ("myfile", O_CREAT, S_IRUSR|S_IXOTH);
```

has the effect of creating a file called myfile, with read permission for the owner and execute permission for others, and only those permissions.

```
$ ls -ls myfile
0 -r-------x   1 neil     software        0 Sep 22 08:11 myfile*
```

There are a couple of factors that may affect the file permissions. First, the permissions specified are used only if the file is being created. Second, the user mask (specified by the shell's umask command) affects the created file's permissions. The mode value given in the open call is ANDed with the inverse of the user mask value at runtime. For example, if the user mask is set to 001 and the S_IXOTH mode flag is specified, the file won't be created with "other" execute permission because the user mask specifies that "other" execute permission isn't to be provided. The flags in the open and creat calls are, in fact, requests to set permissions. Whether or not the requested permissions are set depends on the runtime value of umask.

umask

The umask is a system variable that encodes a mask for file permissions to be used when a file is created. You can change the variable by executing the umask command to supply a new value. The value is a three-digit octal value. Each digit is the result of ORing values from 1, 2, or 4; the meanings are shown in the following table. The separate digits refer to "user," "group," and "other" permissions, respectively.

Digit	Value	Meaning
1	0	No user permissions are to be disallowed.
	4	User read permission is disallowed.
	2	User write permission is disallowed.
	1	User execute permission is disallowed.
2	0	No group permissions are to be disallowed.
	4	Group read permission is disallowed.

Digit	Value	Meaning
	2	Group write permission is disallowed.
	1	Group execute permission is disallowed.
3	0	No other permissions are to be disallowed.
	4	Other read permission is disallowed.
	2	Other write permission is disallowed.
	1	Other execute permission is disallowed.

For example, to block "group" write and execute, and "other" write, the `umask` would be

Digit	Value
1	0
2	2
	1
3	2

Values for each digit are ORed together; so the second digit will need to be `2 | 1`, giving `3`. The resulting `umask` is `032`.

When you create a file via an `open` or `creat` call, the `mode` parameter is compared with the current `umask`. Any bit setting in the `mode` parameter that is also set in the `umask` is removed. The end result is that users can set up their environment to say things like "Don't create any files with write permission for others, even if the program creating the file requests that permission." This doesn't prevent a program or user from subsequently using the `chmod` command (or `chmod` system call in a program) to add other write permissions, but it does help protect users by saving them from having to check and set permissions on all new files.

close

You use `close` to terminate the association between a file descriptor, `fildes`, and its file. The file descriptor becomes available for reuse. It returns 0 if successful and –1 on error.

```
#include <unistd.h>

int close(int fildes);
```

Note that it can be important to check the return result from `close`*. Some file systems, particularly networked ones, may not report an error writing to a file until the file is closed, because data may not have been confirmed as written when writes are performed.*

ioctl

`ioctl` is a bit of a ragbag of things. It provides an interface for controlling the behavior of devices and their descriptors and configuring underlying services. Terminals, file descriptors, sockets, and even tape drives may have `ioctl` calls defined for them and you need to refer to the specific device's `man` page for details. POSIX defines only `ioctl` for streams, which are beyond the scope of this book. Here's the syntax:

```
#include <unistd.h>

int ioctl(int fildes, int cmd, ...);
```

`ioctl` performs the function indicated by `cmd` on the object referenced by the descriptor `fildes`. It may take an optional third argument, depending on the functions supported by a particular device.

For example, the following call to `ioctl` on Linux turns on the keyboard LEDs:

```
ioctl(tty_fd, KDSETLED, LED_NUM|LED_CAP|LED_SCR);
```

Try It Out A File Copy Program

You now know enough about the `open`, `read`, and `write` system calls to write a low-level program, `copy_system.c`, to copy one file to another, character by character.

We'll do this in a number of ways during this chapter to compare the efficiency of each method. For brevity, we'll assume that the input file exists and the output file does not, and that all reads and writes succeed. Of course, in real-life programs, we would check that these assumptions are valid!

1. First you will need to make a test input file, say 1Mb in size, and name it `file.in`.

2. Then compile `copy_system.c`:

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

int main()
{
    char c;
    int in, out;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while(read(in,&c,1) == 1)
        write(out,&c,1);

    exit(0);
}
```

Note that the `#include <unistd.h>` *line must come first, because it defines flags regarding POSIX compliance that may affect other* `include` *files.*

3. Running the program will give something like the following:

```
$ TIMEFORMAT="" time ./copy_system
4.67user 146.90system 2:32.57elapsed 99%CPU
...
$ ls -ls file.in file.out
1029 -rw-r---r-   1 neil     users     1048576 Sep 17 10:46 file.in
1029 -rw-------   1 neil     users     1048576 Sep 17 10:51 file.out
```

How It Works

Here you use the `time` facility to measure how long the program takes to run. The `TIMEFORMAT` variable is used on Linux to override the default POSIX output format of `time`, which does not include the CPU usage. You can see that for this fairly old system, the 1Mb input file, `file.in`, was successfully copied to `file.out`, which was created with read/write permissions for owner only. However, the copy took two-and-a-half minutes and consumed virtually all the CPU time. It was this slow because it had to make more than two million system calls.

In recent years, Linux has seen great strides in its system call and file system performance. By comparison, a similar test using a 2.6 kernel completed in a little under 14 seconds.

```
$ TIMEFORMAT="" time ./copy_system
2.08user 10.59system 0:13.74elapsed 92%CPU
...
```

Try It Out — A Second File Copy Program

You can improve matters by copying in larger blocks. Take a look at this modified program, `copy_block.c`, which copies the files in 1K blocks, again using system calls:

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

int main()
{
    char block[1024];
    int in, out;
    int nread;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while((nread = read(in,block,sizeof(block))) > 0)
        write(out,block,nread);

    exit(0);
}
```

Now try the program, first removing the old output file:

```
$ rm file.out
$ TIMEFORMAT="" time ./copy_block
0.00user 0.02system 0:00.04elapsed 78%CPU
...
```

How It Works

Now the program takes just hundredths of a second, because it requires only around 2,000 system calls. Of course, these times are very system-dependent, but they do show that system calls have a measurable overhead, so it's worth optimizing their use.

Other System Calls for Managing Files

There are a number of other system calls that operate on these low-level file descriptors. These allow a program to control how a file is used and to return status information.

lseek

The `lseek` system call sets the read/write pointer of a file descriptor, `fildes`; that is, you can use it to set where in the file the next read or write will occur. You can set the pointer to an absolute location in the file or to a position relative to the current position or the end of file.

```
#include <unistd.h>
#include <sys/types.h>

off_t lseek(int fildes, off_t offset, int whence);
```

The `offset` parameter is used to specify the position, and the `whence` parameter specifies how the offset is used. `whence` can be one of the following:

- `SEEK_SET`: `offset` is an absolute position
- `SEEK_CUR`: `offset` is relative to the current position
- `SEEK_END`: `offset` is relative to the end of the file

`lseek` returns the `offset` measured in bytes from the beginning of the file that the file pointer is set to, or –1 on failure. The type `off_t`, used for the `offset` in seek operations, is an implementation-dependent integer type defined in `sys/types.h`.

fstat, stat, and lstat

The `fstat` system call returns status information about the file associated with an open file descriptor. The information is written to a structure, `buf`, the address of which is passed as a parameter.

Here's the syntax:

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

int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Note that the inclusion of `sys/types.h` *is optional, but we recommend it when using system calls, because some of their definitions use aliases for standard types that may change one day.*

The related functions `stat` and `lstat` return status information for a named file. They produce the same results, except when the file is a symbolic link. `lstat` returns information about the link itself, and `stat` returns information about the file to which the link refers.

The members of the structure, `stat`, may vary between UNIX-like systems, but will include those in the following table:

stat Member	Description
`st_mode`	File permissions and file-type information
`st_ino`	The inode associated with the file
`st_dev`	The device the file resides on
`st_uid`	The user identity of the file owner
`st_gid`	The group identity of the file owner
`st_atime`	The time of last access
`st_ctime`	The time of last change to permissions, owner, group, or content
`st_mtime`	The time of last modification to contents
`st_nlink`	The number of hard links to the file

The `st_mode` flags returned in the `stat` structure also have a number of associated macros defined in the header file `sys/stat.h`. These macros include names for permission and file-type flags and some masks to help with testing for specific types and permissions.

The permissions flags are the same as for the `open` system call described earlier. File-type flags include

- `S_IFBLK`: Entry is a block special device
- `S_IFDIR`: Entry is a directory
- `S_IFCHR`: Entry is a character special device
- `S_IFIFO`: Entry is a FIFO (named pipe)

- `S_IFREG`: Entry is a regular file
- `S_IFLNK`: Entry is a symbolic link

Other mode flags include

- `S_ISUID`: Entry has `setUID` on execution
- `S_ISGID`: Entry has `setGID` on execution

Masks to interpret the `st_mode` flags include

- `S_IFMT`: File type
- `S_IRWXU`: User read/write/execute permissions
- `S_IRWXG`: Group read/write/execute permissions
- `S_IRWXO`: Others' read/write/execute permissions

There are some macros defined to help with determining file types. These just compare suitably masked mode flags with a suitable device-type flag. These include

- `S_ISBLK`: Test for block special file
- `S_ISCHR`: Test for character special file
- `S_ISDIR`: Test for directory
- `S_ISFIFO`: Test for FIFO
- `S_ISREG`: Test for regular file
- `S_ISLNK`: Test for symbolic link

For example, to test that a file doesn't represent a directory and has execute permission set for the owner but no other permissions, you can use the following test:

```
struct stat statbuf;
mode_t modes;

stat("filename",&statbuf);
modes = statbuf.st_mode;

if(!S_ISDIR(modes) && (modes & S_IRWXU) == S_IXUSR)
    ...
```

dup and dup2

The `dup` system calls provide a way of duplicating a file descriptor, giving two or more different descriptors that access the same file. These might be used for reading and writing to different locations in the file. The `dup` system call duplicates a file descriptor, `fildes`, returning a new descriptor. The `dup2` system call effectively copies one file descriptor to another by specifying the descriptor to use for the copy.

Here's the syntax:

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

These calls can also be useful when you're using multiple processes communicating via pipes. We discuss the `dup` system in more depth in Chapter 13.

The Standard I/O Library

The standard I/O library (stdio) and its header file, `stdio.h`, provide a versatile interface to low-level I/O system calls. The library, now part of ANSI standard C, whereas the system calls you met earlier are not, provides many sophisticated functions for formatting output and scanning input. It also takes care of the buffering requirements for devices.

In many ways, you use this library in the same way that you use low-level file descriptors. You need to open a file to establish an access path. This returns a value that is used as a parameter to other I/O library functions. The equivalent of the low-level file descriptor is called a *stream* and is implemented as a pointer to a structure, a `FILE *`.

> *Don't confuse these file streams with either C++ iostreams or with the STREAMS paradigm of interprocess communication introduced in AT&T UNIX System V Release 3, which is beyond the scope of this book. For more information on STREAMS, check out the X/Open spec (at* `http://www.opengroup.org`*) and the AT&T STREAMS Programming Guide that accompanies System V.*

Three file streams are automatically opened when a program is started. They are `stdin`, `stdout`, and `stderr`. These are declared in `stdio.h` and represent the standard input, output, and error output, respectively, which correspond to the low-level file descriptors 0, 1, and 2.

In this section, we look at the following functions:

- `fopen`, `fclose`
- `fread`, `fwrite`
- `fflush`-
- `fseek`-
- `fgetc`, `getc`, `getchar`
- `fputc`, `putc`, `putchar`
- `fgets`, `gets`
- `printf`, `fprintf`, and `sprintf`
- `scanf`, `fscanf`, and `sscanf`

fopen

The `fopen` library function is the analog of the low-level `open` system call. You use it mainly for files and terminal input and output. Where you need explicit control over devices, you're better off with the low-level system calls, because they eliminate potentially undesirable side effects from libraries, like input/output buffering.

Here's the syntax:

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

`fopen` opens the file named by the `filename` parameter and associates a stream with it. The `mode` parameter specifies how the file is to be opened. It's one of the following strings:

- `"r"` or `"rb"`: Open for reading only
- `"w"` or `"wb"`: Open for writing, truncate to zero length
- `"a"` or `"ab"`: Open for writing, append to end of file
- `"r+"` or `"rb+"` or `"r+b"`: Open for update (reading and writing)
- `"w+"` or `"wb+"` or `"w+b"`: Open for update, truncate to zero length
- `"a+"` or `"ab+"` or `"a+b"`: Open for update, append to end of file

The `b` indicates that the file is a binary file rather than a text file.

> *Note that, unlike MS-DOS, UNIX and Linux do not make a distinction between text and binary files. UNIX and Linux treat all files exactly the same, effectively as binary files. It's also important to note that the mode parameter must be a string, and not a character. Always use double quotes and not single quotes.*

If successful, `fopen` returns a non-null `FILE *` pointer. If it fails, it returns the value `NULL`, defined in `stdio.h`.

The number of available streams is limited, in the same way that file descriptors are limited. The actual limit is `FOPEN_MAX`, which is defined through `stdio.h`, and is always at least eight and typically 16 on Linux.

fread

The `fread` library function is used to read data from a file stream. Data is read into a data buffer given by `ptr` from the stream, `stream`. Both `fread` and `fwrite` deal with data records. These are specified by a record size, `size`, and a count, `nitems`, of records to transfer. The function returns the number of items (rather than the number of bytes) successfully read into the data buffer. At the end of a file, fewer than `nitems` may be returned, including zero.

Here's the syntax:

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

As with all of the standard I/O functions that write to a buffer, it's the programmer's responsibility to allocate the space for the data and check for errors. See also `ferror` and `feof` later in this chapter.

fwrite

The `fwrite` library call has a similar interface to `fread`. It takes data records from the specified data buffer and writes them to the output stream. It returns the number of records successfully written.

Here's the syntax:

```
#include <stdio.h>

size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *stream);
```

Note that `fread` *and* `fwrite` *are not recommended for use with structured data. Part of the problem is that files written with* `fwrite` *are potentially not portable between different machine architectures.*

fclose

The `fclose` library function closes the specified `stream`, causing any unwritten data to be written. It's important to use `fclose` because the `stdio` library will buffer data. If the program needs to be sure that data has been completely written, it should call `fclose`. Note, however, that `fclose` is called automatically on all file streams that are still open when a program ends normally, but then, of course, you do not get a chance to check for errors reported by `fclose`.

Here's the syntax:

```
#include <stdio.h>

int fclose(FILE *stream);
```

fflush

The `fflush` library function causes all outstanding data on a file stream to be written immediately. You can use this to ensure that, for example, an interactive prompt has been sent to a terminal before any attempt to read a response. It's also useful for ensuring that important data has been committed to disk before continuing. You can sometimes use it when you're debugging a program to make sure that the program is writing data and not hanging. Note that an implicit flush operation is carried out when `fclose` is called, so you don't need to call `fflush` before `fclose`.

Here's the syntax:

```
#include <stdio.h>

int fflush(FILE *stream);
```

fseek

The `fseek` function is the file stream equivalent of the `lseek` system call. It sets the position in the stream for the next read or write on that stream. The meaning and values of the `offset` and `whence` parameters are the same as those we gave previously for `lseek`. However, where `lseek` returns an `off_t`, `fseek` returns an integer: 0 if it succeeds, –1 if it fails, with `errno` set to indicate the error. So much for standardization!

Here's the syntax:

```
#include <stdio.h>

int fseek(FILE *stream, long int offset, int whence);
```

fgetc, getc, and getchar

The `fgetc` function returns the next byte, as a character, from a file stream. When it reaches the end of the file or there is an error, it returns `EOF`. You must use `ferror` or `feof` to distinguish the two cases.

Here's the syntax:

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar();
```

The `getc` function is equivalent to `fgetc`, except that it may be implemented as a macro. In that case the `stream` argument may be evaluated more than once so it does not have side effects (for example, it shouldn't affect variables). Also, you can't guarantee to be able use the address of `getc` as a function pointer.

The `getchar` function is equivalent to `getc(stdin)` and reads the next character from the standard input.

fputc, putc, and putchar

The `fputc` function writes a character to an output file stream. It returns the value it has written, or `EOF` on failure.

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

As with `fgetc`/`getc`, the function `putc` is equivalent to `fputc`, but it may be implemented as a macro.

The `putchar` function is equivalent to `putc(c,stdout)`, writing a single character to the standard output. Note that `putchar` takes and `getchar` returns characters as `int`s, not `char`. This allows the end-of-file (`EOF`) indicator to take the value –1, outside the range of character codes.

fgets and gets

The `fgets` function reads a string from an input file `stream`.

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

`fgets` writes characters to the string pointed to by `s` until a newline is encountered, `n-1` characters have been transferred, or the end of file is reached, whichever occurs first. Any newline encountered is transferred to the receiving string and a terminating null byte, `\0`, is added. Only a maximum of `n-1` characters are transferred in any one call because the null byte must be added to mark the end of the string and bring the total up to `n` bytes.

When it successfully completes, `fgets` returns a pointer to the string `s`. If the stream is at the end of a file, it sets the `EOF` indicator for the stream and `fgets` returns a null pointer. If a read error occurs, `fgets` returns a null pointer and sets `errno` to indicate the type of error.

The `gets` function is similar to `fgets`, except that it reads from the standard input and discards any newline encountered. It adds a trailing null byte to the receiving string.

> *Note that* `gets` *doesn't limit the number of characters that can be transferred so it could overrun its transfer buffer. Consequently, you should avoid using it and use* `fgets` *instead. Many security issues can be traced back to functions in programs that are made to overflow a buffer of some sort or another. This is one such function, so be careful!*

Formatted Input and Output

There are a number of library functions for producing output in a controlled fashion that you may be familiar with if you've programmed in C. These functions include `printf` and friends for printing values to a file stream, and `scanf` and others for reading values from a file stream.

printf, fprintf, and sprintf

The `printf` family of functions format and output a variable number of arguments of different types. The way each is represented in the output stream is controlled by the `format` parameter, which is a string that contains ordinary characters to be printed and codes called *conversion specifiers*, which indicate how and where the remaining arguments are to be printed.

```
#include <stdio.h>

int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

The `printf` function produces its output on the standard output. The `fprintf` function produces its output on a specified `stream`. The `sprintf` function writes its output and a terminating null character into the string `s` passed as a parameter. This string must be large enough to contain all of the output.

There are other members of the `printf` family that deal with their arguments in different ways. See the `printf` manual page for more details.

Ordinary characters are passed unchanged into the output. Conversion specifiers cause `printf` to fetch and format additional arguments passed as parameters. They always start with a `%` character. Here's a simple example:

```
printf("Some numbers: %d, %d, and %d\n", 1, 2, 3);
```

This produces, on the standard output:

```
Some numbers: 1, 2, and 3
```

To print a % character, you need to use `%%`, so that it doesn't get confused with a conversion specifier.

Here are some of the most commonly used conversion specifiers:

- `%d`, `%i`: Print an integer in decimal
- `%o`, `%x`: Print an integer in octal, hexadecimal
- `%c`: Print a character
- `%s`: Print a string
- `%f`: Print a floating-point (single precision) number
- `%e`: Print a double precision number, in fixed format
- `%g`: Print a double in a general format

It's very important that the number and type of the arguments passed to `printf` match the conversion specifiers in the `format` string. An optional size specifier is used to indicate the type of integer arguments. This is either `h`, for example `%hd`, to indicate a `short int`, or `l`, for example `%ld`, to indicate a `long int`. Some compilers can check these `printf` statements, but they aren't infallible. If you are using the GNU compiler `gcc`, you can add the `-Wformat` option to your compilation command to do this.

Here's another example:

```
char initial = 'A';
char *surname = "Matthew";
double age = 13.5;

printf("Hello Mr %c %s, aged %g\n", initial, surname, age);
```

This produces

```
Hello Mr A Matthew, aged 13.5
```

You can gain greater control over the way items are printed by using field specifiers. These extend the conversion specifiers to include control over the spacing of the output. A common use is to set the number of decimal places for a floating-point number or to set the amount of space around a string.

Field specifiers are given as numbers immediately after the `%` character in a conversion specifier. The following table contains some more examples of conversion specifiers and resulting output. To make things a little clearer, we'll use vertical bars to show the limits of the output.

Format	Argument	|Output|
`%10s`	`"Hello"`	`|     Hello|`
`%-10s`	`"Hello"`	`|Hello     |`
`%10d`	`1234`	`|      1234|`
`%-10d`	`1234`	`|1234      |`
`%010d`	`1234`	`|0000001234  |`
`%10.4f`	`12.34`	`|   12.3400|`
`%*s`	`10,"Hello"`	`|     Hello|`

All of these examples have been printed in a field width of 10 characters. Note that a negative field width means that the item is written left-justified within the field. A variable field width is indicated by using an asterisk (`*`). In this case, the next argument is used for the width. A leading zero indicates the item is written with leading zeros. According to the POSIX specification, `printf` doesn't truncate fields; rather, it expands the field to fit. So, for example, if you try to print a string longer than the field, the field grows:

Format	Argument	|Output|
`%10s`	`"HelloTherePeeps"`	`|HelloTherePeeps|`

The `printf` functions return an integer, the number of characters written. This doesn't include the terminating `null` in the case of `sprintf`. On error, these functions return a negative value and set `errno`.

scanf, fscanf, and sscanf

The `scanf` family of functions works in a way similar to the `printf` group, except that these functions read items from a stream and place values into variables at the addresses they're passed as pointer parameters. They use a format string to control the input conversion in the same way, and many of the conversion specifiers are the same.

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

It's very important that the variables used to hold the values scanned in by the `scanf` functions are of the correct type and that they match the format string precisely. If they don't, your memory could be corrupted and your program could crash. There won't be any compiler errors, but if you're lucky, you might get a warning!

The format string for `scanf` and friends contains both ordinary characters and conversion specifiers, as for `printf`. However, the ordinary characters are used to specify characters that must be present in the input.

Here is a simple example:

```
int num;
scanf("Hello %d", &num);
```

This call to `scanf` will succeed only if the next five characters on the standard input are `Hello`. Then, if the next characters form a recognizable decimal number, the number will be read and the value assigned to the variable `num`. A space in the format string is used to ignore all whitespace (spaces, tabs, form feeds, and newlines) in the input between conversion specifiers. This means that the call to `scanf` will succeed and place 1234 into the variable `num` given either of the following inputs:

```
Hello     1234
Hello1234
```

Whitespace is also usually ignored in the input when a conversion begins. This means that a format string of `%d` will keep reading the input, skipping over spaces and newlines until a sequence of digits is found. If the expected characters are not present, the conversion fails and `scanf` returns.

> *This can lead to problems if you are not careful. An infinite loop can occur in your program if you leave a non-digit character in the input while scanning for integers.*

Other conversion specifiers are

- `%d`: Scan a decimal integer
- `%o`, `%x`: Scan an octal, hexadecimal integer
- `%f`, `%e`, `%g`: Scan a floating-point number
- `%c`: Scan a character (whitespace not skipped)
- `%s`: Scan a string
- `%[]`: Scan a set of characters (see the following discussion)
- `%%`: Scan a % character

Like `printf`, `scanf` conversion specifiers may also have a field width to limit the amount of input consumed. A size specifier (either `h` for short or `l` for long) indicates whether the receiving argument is shorter or longer than the default. This means that `%hd` indicates a `short int`, `%ld` a `long int`, and `%lg` a double precision floating-point number.

A specifier beginning with an asterisk indicates that the item is to be ignored. This means that the information is not stored and therefore does not need a variable to receive it.

Use the `%c` specifier to read a single character in the input. This doesn't skip initial whitespace characters.

Use the `%s` specifier to scan strings, but take care. It skips leading whitespace, but stops at the first whitespace character in the string; so, you're better off using it for reading words rather than general strings. Also, without a field-width specifier, there's no limit to the length of string it might read, so the receiving

string must be sufficient to hold the longest string in the input stream. It's better to use a field specifier, or a combination of `fgets` and `sscanf`, to read in a line of input and then scan it. This will prevent possible buffer overflows that could be exploited by a malicious user.

Use the `%[]` specifier to read a string composed of characters from a set. The format `%[A-Z]` will read a string of capital letters. If the first character in the set is a caret, `^`, the specifier reads a string that consists of characters not in the set. So, to read a string with spaces in it, but stopping at the first comma, you can use `%[^,]`.

Given the input line,

```
Hello, 1234, 5.678, X, string to the end of the line
```

this call to `scanf` will correctly scan four items:

```
char s[256];
int n;
float f;
char c;

scanf("Hello,%d,%g, %c, %[^\n]", &n,&f,&c,s);
```

The `scanf` functions return the number of items successfully read, which will be zero if the first item fails. If the end of the input is reached before the first item is matched, `EOF` is returned. If a read error occurs on the file stream, the stream error flag will be set and the error variable, `errno`, will be set to indicate the type of error. See the "Stream Errors" section later in this chapter for more details.

In general, `scanf` and friends are not highly regarded; this is for three reasons:

- Traditionally, the implementations have been buggy.
- They're inflexible to use.
- They can lead to code where it's difficult to work out what is being parsed.

As an alternative, try using other functions, like `fread` or `fgets`, to read input lines and then use the string functions to break the input into the items you need.

Other Stream Functions

There are a number of other `stdio` library functions that use either stream parameters or the standard streams `stdin`, `stdout`, `stderr`:

- `fgetpos`: Get the current position in a file stream.
- `fsetpos`: Set the current position in a file stream.
- `ftell`: Return the current file offset in a stream.
- `rewind`: Reset the file position in a stream.
- `freopen`: Reuse a file stream.

- `setvbuf`: Set the buffering scheme for a stream.
- `remove`: Equivalent to `unlink` unless the `path` parameter is a directory, in which case it's equivalent to `rmdir`.

These are all library functions documented in section 3 of the manual pages.

You can use the file stream functions to re-implement the file copy program, using library functions instead. Take a look at `copy_stdio.c` in the following Try It Out exercise.

Try It Out A Third File Copy Program

This program is very similar to earlier versions, but the character-by-character copy is accomplished using calls to the functions referenced in `stdio.h`:

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

int main()
{
    int c;
    FILE *in, *out;

    in = fopen("file.in","r");
    out = fopen("file.out","w");

    while((c = fgetc(in)) != EOF)
        fputc(c,out);

    exit(0);
}
```

Running this program as before, you get

```
$ TIMEFORMAT="" time ./copy_stdio
0.06user 0.02system 0:00.11elapsed 81%CPU
...
```

How It Works

This time, the program runs in 0.11 seconds, not as fast as the low-level block version, but a great deal better than the other single-character-at-a-time version. This is because the stdio library maintains an internal buffer within the `FILE` structure and the low-level system calls are made only when the buffer fills. Feel free to experiment with testing line-by-line and block stdio copying code to see how they perform relative to the three examples we've tested.

Stream Errors

To indicate an error, many stdio library functions return out-of-range values, such as null pointers or the constant `EOF`. In these cases, the error is indicated in the external variable `errno`:

```
#include <errno.h>

extern int errno;
```

Note that many functions may change the value of `errno`*. Its value is valid only when a function has failed. You should inspect it immediately after a function has indicated failure. You should always copy it into another variable before using it, because printing functions, such as* `fprintf`*, might alter* `errno` *themselves.*

You can also interrogate the state of a file stream to determine whether an error has occurred, or the end of file has been reached.

```
#include <stdio.h>

int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

The `ferror` function tests the error indicator for a stream and returns nonzero if it's set, but zero otherwise.

The `feof` function tests the end-of-file indicator within a stream and returns nonzero if it is set, zero otherwise. Use it like this:

```
if(feof(some_stream))
    /* We're at the end */
```

The `clearerr` function clears the end-of-file and error indicators for the stream to which `stream` points. It has no return value and no errors are defined. You can use it to recover from error conditions on streams. One example might be to resume writing to a stream after a "disk full" error has been resolved.

Streams and File Descriptors

Each file stream is associated with a low-level file descriptor. You can mix low-level input and output operations with higher-level stream operations, but this is generally unwise, because the effects of buffering can be difficult to predict.

```
#include <stdio.h>

int fileno(FILE *stream);
FILE *fdopen(int fildes, const char *mode);
```

You can determine which low-level file descriptor is being used for a file stream by calling the `fileno` function. It returns the file descriptor for a given stream, or –1 on failure. This function can be useful if you need low-level access to an open stream, for example, to call `fstat` on it.

You can create a new file stream based on an already-opened file descriptor by calling the `fdopen` function. Essentially, this function provides stdio buffers around an already-open file descriptor, which might be an easier way to explain it.

The `fdopen` function operates in the same way as the `fopen` function, but instead of a filename it takes a low-level file descriptor. This can be useful if you have used `open` to create a file, perhaps to get fine control over the permissions, but want to use a stream for writing to it. The `mode` parameter is the same as for the `fopen` function and must be compatible with the file access modes established when the file was originally opened. `fdopen` returns the new file stream or `NULL` on failure.

File and Directory Maintenance

The standard libraries and system calls provide complete control over the creation and maintenance of files and directories.

chmod

You can change the permissions on a file or directory using the `chmod` system call. This forms the basis of the `chmod` shell program.

Here's the syntax:

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

int chmod(const char *path, mode_t mode);
```

The file specified by `path` is changed to have the permissions given by `mode`. The modes are specified as in the `open` system call, a bitwise OR of required permissions. Unless the program has been given appropriate privileges, only the owner of the file or a superuser can change its permissions.

chown

A superuser can change the owner of a file using the `chown` system call.

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

The call uses the numeric values of the desired new user and group IDs (culled from `getuid` and `getgid` calls) and a system value that is used to restrict who can change file ownership. The owner and group of a file are changed if the appropriate privileges are set.

> *POSIX actually allows systems where non-superusers can change file ownerships. All "proper" POSIX systems won't allow this, but, strictly speaking, it's an extension (for FIPS 151-2). The kinds of systems we deal with in this book conform to the XSI (X/Open System Interface) specification and do enforce ownership rules.*

unlink, link, and symlink

You can remove a file using `unlink`.

The `unlink` system call removes the directory entry for a file and decrements the link count for it. It returns 0 if the unlinking was successful, –1 on an error. You must have write and execute permissions in the directory where the file has its directory entry for this call to function.

```
#include <unistd.h>

int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

If the count reaches zero and no process has the file open, the file is deleted. In fact, the directory entry is always removed immediately, but the file's space will not be recovered until the last process (if any) closes it. The `rm` program uses this call. Additional links represent alternative names for a file, normally created by the `ln` program. You can create new links to a file programmatically by using the `link` system call.

> *Creating a file with* `open` *and then calling* `unlink` *on it is a trick some programmers use to create transient files. These files are available to the program only while they are open; they will effectively be automatically deleted when the program exits and the file is closed.*

The `link` system call creates a new link to an existing file, `path1`. The new directory entry is specified by `path2`. You can create symbolic links using the `symlink` system call in a similar fashion. Note that symbolic links to a file do not increment a file's reference count and so do not prevent the file from being effectively deleted as normal (hard) links do.

mkdir and rmdir

You can create and remove directories using the `mkdir` and `rmdir` system calls.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

The `mkdir` system call is used for creating directories and is the equivalent of the `mkdir` program. `mkdir` makes a new directory with `path` as its name. The directory permissions are passed in the parameter `mode` and are given as in the `O_CREAT` option of the `open` system call and, again, subject to `umask`.

```
#include <unistd.h>

int rmdir(const char *path);
```

The `rmdir` system call removes directories, but only if they are empty. The `rmdir` program uses this system call to do its job.

chdir and getcwd

A program can navigate directories in much the same way as a user moves around the file system. As you use the `cd` command in the shell to change directory, so a program can use the `chdir` system call.

```
#include <unistd.h>

int chdir(const char *path);
```

A program can determine its current working directory by calling the `getcwd` function.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

The `getcwd` function writes the name of the current directory into the given buffer, `buf`. It returns `NULL` if the directory name would exceed the size of the buffer (an `ERANGE` error), given as the parameter `size`. It returns `buf` on success.

`getcwd` may also return `NULL` if the directory is removed (`EINVAL`) or permissions changed (`EACCESS`) while the program is running.

Scanning Directories

A common problem on Linux systems is scanning directories, that is, determining the files that reside in a particular directory. In shell programs, it's easy — just let the shell expand a wildcard expression. In the past, different UNIX variants have allowed programmatic access to the low-level file system structure. You can still open a directory as a regular file and directly read the directory entries, but different file system structures and implementations have made this approach nonportable. A standard suite of library functions has now been developed that makes directory scanning much simpler.

The directory functions are declared in a header file `dirent.h`. They use a structure, `DIR`, as a basis for directory manipulation. A pointer to this structure, called a *directory stream* (a `DIR *`), acts in much the same way as a file steam (`FILE *`) does for regular file manipulation. Directory entries themselves are returned in `dirent` structures, also declared in `dirent.h`, because one should never alter the fields in the `DIR` structure directly.

We'll review these functions:

- `opendir`, `closedir`
- `readdir`
- `telldir`
- `seekdir`
- `closedir`

opendir

The `opendir` function opens a directory and establishes a directory stream. If successful, it returns a pointer to a `DIR` structure to be used for reading directory entries.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
```

`opendir` returns a null pointer on failure. Note that a directory stream uses a low-level file descriptor to access the directory itself, so `opendir` could fail with too many open files.

readdir

The `readdir` function returns a pointer to a structure detailing the next directory entry in the directory stream `dirp`. Successive calls to `readdir` return further directory entries. On error, and at the end of the directory, `readdir` returns `NULL`. POSIX-compliant systems leave `errno` unchanged when returning `NULL` at end of directory and set it when an error occurs.

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

Note that `readdir` scanning isn't guaranteed to list all the files (and subdirectories) in a directory if there are other processes creating and deleting files in the directory at the same time.

The `dirent` structure containing directory entry details includes the following entries:

- `ino_t d_ino`: The inode of the file
- `char d_name[]`: The name of the file

To determine further details of a file in a directory, you need to make a call to `stat`, which we covered earlier in this chapter.

telldir

The `telldir` function returns a value that records the current position in a directory stream. You can use this in subsequent calls to `seekdir` to reset a directory scan to the current position.

```
#include <sys/types.h>
#include <dirent.h>

long int telldir(DIR *dirp);
```

seekdir

The `seekdir` function sets the directory entry pointer in the directory stream given by `dirp`. The value of `loc`, used to set the position, should have been obtained from a prior call to `telldir`.

```
#include <sys/types.h>
#include <dirent.h>

void seekdir(DIR *dirp, long int loc);
```

closedir

The `closedir` function closes a directory stream and frees up the resources associated with it. It returns 0 on success and –1 if there is an error.

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

In the next program, `printdir.c`, you will put together a lot of the file manipulation functions to create a simple directory listing. Each file in a directory is listed on a line by itself. Each subdirectory has its name followed by a slash and the files listed in it are indented by four spaces.

The program changes a directory into the subdirectories so that the files it finds have usable names, that is, they can be passed directly to `opendir`. The program will fail on very deeply nested directory structures because there's a limit on the allowed number of open directory streams.

We could, of course, make it more general by taking a command-line argument to specify the start point. Check out the Linux source code of such utilities as `ls` and `find` for ideas on a more general implementation.

Try It Out — A Directory-Scanning Program

1. Start with the appropriate headers and then a function, `printdir`, which prints out the current directory. It will recurse for subdirectories using the `depth` parameter for indentation.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>

void printdir(char *dir, int depth)
{
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;

    if((dp = opendir(dir)) == NULL) {
        fprintf(stderr,"cannot open directory: %s\n", dir);
```

```
        return;
    }
    chdir(dir);
    while((entry = readdir(dp)) != NULL) {
        lstat(entry->d_name,&statbuf);
        if(S_ISDIR(statbuf.st_mode)) {
            /* Found a directory, but ignore . and .. */
            if(strcmp(".",entry->d_name) == 0 ||
                strcmp("..",entry->d_name) == 0)
                continue;
            printf("%*s%s/\n",depth,"",entry->d_name);
            /* Recurse at a new indent level */
            printdir(entry->d_name,depth+4);
        }
        else printf("%*s%s\n",depth,"",entry->d_name);
    }
    chdir("..");
    closedir(dp);
}
```

2. Now move onto the `main` function:

```
int main()
{
    printf("Directory scan of /home:\n");
    printdir("/home",0);
    printf("done.\n");

    exit(0);
}
```

The program scans the home directories and produces output like that following (edited for brevity). To see into other users' directories you may need superuser permissions.

```
$ ./printdir
Directory scan of /home:
neil/
    .Xdefaults
    .Xmodmap
    .Xresources
    .bash_history
    .bashrc
    .kde/
        share/
            apps/
                konqueror/
                    dirtree/
                        public_html.desktop
                    toolbar/
                    bookmarks.xml
                    konq_history
                kdisplay/
                    color-schemes/
```

```
        BLP4e/
            Gnu_Public_License
            chapter04/
                argopt.c
                args.c
            chapter03/
                file.out
                mmap.c
                printdir
done.
```

How It Works

Most of the action is within the `printdir` function. After some initial error checking using `opendir` to see that the directory exists, `printdir` makes a call to `chdir` to the directory specified. While the entries returned by `readdir` aren't null, the program checks to see whether the entry is a directory. If it isn't, it prints the file entry with indentation `depth`.

If the entry *is* a directory, you meet a little bit of recursion. After the `.` and `..` entries (the current and parent directories) have been ignored, the `printdir` function calls itself and goes through the same process again. How does it get out of these loops? Once the `while` loop has finished, the call `chdir("..")` takes it back up the directory tree and the previous listing can continue. Calling `closedir(dp)` makes sure that the number of open directory streams isn't higher than it needs to be.

For a brief taste of the discussion of the Linux environment in Chapter 4, let's look at one way you can make the program more general. The program is limited because it's specific to the directory `/home`. With the following changes to `main`, you could turn it into a more useful directory browser:

```
int main(int argc, char* argv[])
{
    char *topdir = ".";
    if (argc >= 2)
      topdir=argv[1];

    printf("Directory scan of %s\n",topdir);
    printdir(topdir,0);
    printf("done.\n");

    exit(0);
}
```

Three lines were changed and five added, but now it's a general-purpose utility with an optional parameter of the directory name, which defaults to the current directory. You can run it using the following command:

```
$ ./printdir2 /usr/local | more
```

The output will be paged so that the user can page through the output. Hence, the user has quite a convenient little general-purpose directory tree browser. With very little effort, you could add space usage statistics, limit depth of display, and so on.

Errors

As you've seen, many of the system calls and functions described in this chapter can fail for a number of reasons. When they do, they indicate the reason for their failure by setting the value of the external variable `errno`. Many different libraries use this variable as a standard way to report problems. It bears repeating that the program must inspect the `errno` variable immediately after the function giving problems because it may be overwritten by the next function called, even if that function itself doesn't fail.

The values and meanings of the errors are listed in the header file `errno.h`. They include

- `EPERM`: Operation not permitted
- `ENOENT`: No such file or directory
- `EINTR`: Interrupted system call
- `EIO`: I/O Error
- `EBUSY`: Device or resource busy
- `EEXIST`: File exists
- `EINVAL`: Invalid argument
- `EMFILE`: Too many open files
- `ENODEV`: No such device
- `EISDIR`: Is a directory
- `ENOTDIR`: Isn't a directory

There are a couple of useful functions for reporting errors when they occur: `strerror` and `perror`.

strerror

The `strerror` function maps an error number into a string describing the type of error that has occurred. This can be useful for logging error conditions.

Here's the syntax:

```
#include <string.h>

char *strerror(int errnum);
```

perror

The `perror` function also maps the current error, as reported in `errno`, into a string and prints it on the standard error stream. It's preceded by the message given in the string `s` (if not `NULL`), followed by a colon and a space.

Here's the syntax:

```
#include <stdio.h>

void perror(const char *s);
```

For example,

```
perror("program");
```

might give the following on the standard error output:

```
program: Too many open files
```

The /proc File System

Earlier in the chapter we mentioned that Linux treats most things as files and that there are entries in the file system for hardware devices. These `/dev` files are used to access hardware in a specific way using low-level system calls.

The software drivers that control hardware can often be configured in certain ways, or are capable of reporting information. For example, a hard disk controller may be configured to use a particular DMA mode. A network card might be able to report whether it has negotiated a high-speed, duplex connection.

Utilities for communicating with device drivers have been common in the past. For example, `hdparm` is used to configure some disk parameters and `ifconfig` can report network statistics. In recent years, there has been a trend toward providing a more consistent way of accessing driver information, and, in fact, to extend this to include communication with various elements of the Linux kernel.

Linux provides a special file system, `procfs`, that is usually made available as the directory `/proc`. It contains many special files that allow higher-level access to driver and kernel information. Applications can read and write these files to get information and set parameters as long as they are running with the correct access permissions.

The files that appear in `/proc` will vary from system to system, and more are included with each Linux release as more drivers and facilities support the `procfs` file system. Here, we look at some of the more common files and briefly consider their use.

A directory listing of `/proc` on the computer being used to write this chapter shows the following entries:

```
1/      10514/  20254/  6/     9057/  9623/        ide/        mtrr
10359/  10524/  29/     698/   9089/  9638/        interrupts  net/
10360/  10530/  2983/   699/   9118/  acpi/        iomem       partitions
10381/  10539/  3/      710/   9119/  asound/      ioports     scsi/
10438/  10541/  30/     711/   9120/  buddyinfo    irq/        self@
10441/  10555/  3069/   742/   9138/  bus/         kallsyms    slabinfo
10442/  10688/  3098/   7808/  9151/  cmdline      kcore       splash
10478/  10689/  3099/   7813/  92/    config.gz    keys        stat
10479/  10784/  31/     8357/  9288/  cpuinfo      key-users   swaps
10482/  113/    3170/   8371/  93/    crypto       kmsg        sys/
10484/  115/    3171/   840/   9355/  devices      loadavg     sysrq-trigger
10486/  116/    3177/   8505/  9407/  diskstats    locks       sysvipc/
10495/  1167/   32288/  8543/  9457/  dma          mdstat      tty/
10497/  1168/   3241/   8547/  9479/  driver/      meminfo     uptime
```

```
10498/  1791/   352/    8561/  9618/  execdomains  misc        version
10500/  19557/  4/      8677/  9619/  fb           modules     vmstat
10502/  19564/  4010/   888/   9621/  filesystems  mounts@     zoneinfo
10510/  2/      5/      8910/  9622/  fs/          mpt/
```

In many cases, the files can just be read and will give status information. For example, `/proc/cpuinfo` gives details of the processors available:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Pentium(R) 4 CPU 2.66GHz
stepping        : 8
cpu MHz         : 2665.923
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss up
bogomips        : 5413.47
clflush size    : 64
```

Similarly, `/proc/meminfo` and `/proc/version` give information about memory usage and kernel version, respectively:

```
$ cat /proc/meminfo
MemTotal:       776156 kB
MemFree:         28528 kB
Buffers:        191764 kB
Cached:         369520 kB
SwapCached:         20 kB
Active:         406912 kB
Inactive:       274320 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       776156 kB
LowFree:         28528 kB
SwapTotal:     1164672 kB
SwapFree:      1164652 kB
Dirty:              68 kB
Writeback:           0 kB
AnonPages:       95348 kB
Mapped:          49044 kB
Slab:            57848 kB
SReclaimable:    48008 kB
SUnreclaim:       9840 kB
PageTables:       1500 kB
```

```
NFS_Unstable:        0 kB
Bounce:              0 kB
CommitLimit:   1552748 kB
Committed_AS:   189680 kB
VmallocTotal:   245752 kB
VmallocUsed:     10572 kB
VmallocChunk:   234556 kB
HugePages_Total:     0
HugePages_Free:      0
HugePages_Rsvd:      0
Hugepagesize:     4096 kB
$ cat /proc/version
Linux version 2.6.20.2-2-default (geeko@buildhost) (gcc version 4.1.3 20070218
(prerelease) (SUSE Linux)) #1 SMP Fri Mar 9 21:54:10 UTC 2007
```

The information given by these files is generated each time the file is read. So rereading the `meminfo` file at a later time will give up-to-the-second results.

You can find more information from specific kernel functions in subdirectories of `/proc`. For example, you can get network socket usage statistics from `/proc/net/sockstat`:

```
$ cat /proc/net/sockstat
sockets: used 285
TCP: inuse 4 orphan 0 tw 0 alloc 7 mem 1
UDP: inuse 3
UDPLITE: inuse 0
RAW: inuse 0
FRAG: inuse 0 memory 0
```

Some of the `/proc` entries can be written to as well as read. For example, the total number of files that all running programs can open at the same time is a Linux kernel parameter. The current value can be read at `/proc/sys/fs/file-max`:

```
$ cat /proc/sys/fs/file-max
76593
```

Here the value is set to 76,593. If you need to increase this value, you can do so by writing to the same file. You may need to do this if you are running a specialist application suite — such as a database system that uses many tables — that needs to open many files at once.

> *Writing `/proc` files requires superuser access. You must take great care when writing `/proc` files; it's possible to cause severe problems including system crashes and loss of data by writing inappropriate values.*

To increase the system-wide file handle limit to 80,000, you can simply write the new limit to the `file-max` file:

```
# echo 80000 >/proc/sys/fs/file-max
```

Now, when you reread the file, you see the new value:

```
$ cat /proc/sys/fs/file-max
80000
```

The subdirectories of `/proc` that have numeric names are used to provide access to information about running programs. You learn more about how programs are executed as processes in Chapter 11.

For now, just notice that each process has a unique identifier: a number between 1 and about 32,000. The `ps` command provides a list of currently running processes. For example, as this chapter is being written:

```
neil@suse103:~/BLP4e/chapter03> ps -a
  PID TTY          TIME CMD
 9118 pts/1    00:00:00 ftp
 9230 pts/1    00:00:00 ps
10689 pts/1    00:00:01 bash
neil@suse103:~/BLP4e/chapter03>
```

Here, you can see several terminal sessions running the bash shell and a file transfer session running the `ftp` program. You can get more details about the `ftp` session by looking in `/proc`.

The process identifier for `ftp` here is given as 9118, so you need to look in `/proc/9118` for details about it:

```
$ ls -l /proc/9118
total 0
0 dr-xr-xr-x 2 neil users 0 2007-05-20 07:43 attr
0 -r-------- 1 neil users 0 2007-05-20 07:43 auxv
0 -r--r--r-- 1 neil users 0 2007-05-20 07:35 cmdline
0 -r--r--r-- 1 neil users 0 2007-05-20 07:43 cpuset
0 lrwxrwxrwx 1 neil users 0 2007-05-20 07:43 cwd -> /home/neil/BLP4e/chapter03
0 -r-------- 1 neil users 0 2007-05-20 07:43 environ
0 lrwxrwxrwx 1 neil users 0 2007-05-20 07:43 exe -> /usr/bin/pftp
0 dr-x------ 2 neil users 0 2007-05-20 07:19 fd
0 -rw-r--r-- 1 neil users 0 2007-05-20 07:43 loginuid
0 -r--r--r-- 1 neil users 0 2007-05-20 07:43 maps
0 -rw------- 1 neil users 0 2007-05-20 07:43 mem
0 -r--r--r-- 1 neil users 0 2007-05-20 07:43 mounts
0 -r-------- 1 neil users 0 2007-05-20 07:43 mountstats
0 -rw-r--r-- 1 neil users 0 2007-05-20 07:43 oom_adj
0 -r--r--r-- 1 neil users 0 2007-05-20 07:43 oom_score
0 lrwxrwxrwx 1 neil users 0 2007-05-20 07:43 root -> /
0 -rw------- 1 neil users 0 2007-05-20 07:43 seccomp
0 -r--r--r-- 1 neil users 0 2007-05-20 07:43 smaps
0 -r--r--r-- 1 neil users 0 2007-05-20 07:33 stat
0 -r--r--r-- 1 neil users 0 2007-05-20 07:43 statm
0 -r--r--r-- 1 neil users 0 2007-05-20 07:33 status
0 dr-xr-xr-x 3 neil users 0 2007-05-20 07:43 task
0 -r--r--r-- 1 neil users 0 2007-05-20 07:43 wchan
```

Here, you can see various special files that can tell us what is happening with this process.

You can tell that the program `/usr/bin/pftp` is running and that its current working directory is `/home/neil/BLP4e/chapter03`. It is possible to read the other files in this directory to see the command line used to start it as well as the shell environment it has. The `cmdline` and `environ` files provide this information as a series of null-terminated strings, so you need to take care when viewing them. We discuss the Linux environment in depth in Chapter 4.

```
$ od -c /proc/9118/cmdline
0000000   f   t   p  \0   1   9   2   .   1   6   8   .   0   .   1   2
```

```
0000020  \0
0000021
```

Here, you can see that `ftp` was started with the command line `ftp 192.168.0.12`.

The `fd` subdirectory provides information about the open file descriptors in use by the process. This information can be useful in determining how many files a program has open at one time. There is one entry per open descriptor; the name matches the number of the descriptor. In this case, you can see that `ftp` has open descriptors 0, 1, 2, and 3, as we might expect. These are the standard input, output, and error descriptors plus a connection to the remote server.

```
$ ls /proc/9118/fd
0  1  2  3
```

Advanced Topics: fcntl and mmap

Here, we cover a couple of topics that you might like to skip because they're seldom used. Having said that, we've put them here for your reference because they can provide simple solutions to some tricky problems.

fcntl

The `fcntl` system call provides further ways to manipulate low-level file descriptors.

```
#include <fcntl.h>

int fcntl(int fildes, int cmd);
int fcntl(int fildes, int cmd, long arg);
```

You can perform several miscellaneous operations on open file descriptors with the `fcntl` system call, including duplicating them, getting and setting file descriptor flags, getting and setting file status flags, and managing advisory file locking.

The various operations are selected by different values of the command parameter `cmd`, as defined in `fcntl.h`. Depending on the command chosen, the system call will require a third parameter, `arg`:

- `fcntl(fildes, F_DUPFD, newfd)`: This call returns a new file descriptor with a numerical value equal to or greater than the integer `newfd`. The new descriptor is a copy of the descriptor `fildes`. Depending on the number of open files and the value of `newfd`, this can be effectively the same as `dup(fildes)`.
- `fcntl(fildes, F_GETFD)`: This call returns the file descriptor flags as defined in `fcntl.h`. These include `FD_CLOEXEC`, which determines whether the file descriptor is closed after a successful call to one of the `exec` family of system calls.
- `fcntl(fildes, F_SETFD, flags)`: This call is used to set the file descriptor flags, usually just `FD_CLOEXEC`.
- `fcntl(fildes, F_GETFL)` and `fcntl(fildes, F_SETFL, flags)`: These calls are used, respectively, to get and set the file status flags and access modes. You can extract the file access modes by using the mask `O_ACCMODE` defined in `fcntl.h`. Other flags include those passed in a third argument to `open` when used with `O_CREAT`. Note that you can't set all flags. In particular, you can't set file permissions using `fcntl`.

You can also implement advisory file locking via `fcntl`. Refer to section 2 of the manual pages for more information, or see Chapter 7, where we discuss file locking.

mmap

UNIX provides a useful facility that allows programs to share memory, and the good news is that it's been included in versions 2.0 and later of the Linux kernel. The `mmap` (for memory map) function sets up a segment of memory that can be read or written by two or more programs. Changes made by one program are seen by the others.

You can use the same facility to manipulate files. You can make the entire contents of a disk file look like an array in memory. If the file consists of records that can be described by C structures, you can update the file using structure array accesses.

This is made possible by the use of virtual memory segments that have special permissions set. Reading from and writing to the segment causes the operating system to read and write the appropriate part of the disk file.

The `mmap` function creates a pointer to a region of memory associated with the contents of the file accessed through an open file descriptor.

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

void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

You can alter the start of the file data that is accessed by the shared segment by passing the `off` parameter. The open file descriptor is passed as `fildes`. The amount of data that can be accessed (that is, the length of the memory segment) is set via the `len` parameter.

You can use the `addr` parameter to request a particular memory address. If it's zero, the resulting pointer is allocated automatically. This is the recommended usage, because it is difficult to be portable otherwise; systems vary as to the available address ranges.

The `prot` parameter is used to set access permissions for the memory segment. This is a bitwise OR of the following constant values:

- `PROT_READ`: The segment can be read
- `PROT_WRITE`: The segment can be written
- `PROT_EXEC`: The segment can be executed
- `PROT_NONE`: The segment can't be accessed

The `flags` parameter controls how changes made to the segment by the program are reflected elsewhere; these options are displayed in the following table.

`MAP_PRIVATE`	The segment is private, changes are local
`MAP_SHARED`	The segment changes are made in the file
`MAP_FIXED`	The segment must be at the given address, `addr`

The `msync` function causes the changes in part or all of the memory segment to be written back to (or read from) the mapped file.

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

int msync(void *addr, size_t len, int flags);
```

The part of the segment to be updated is given by the passed start address, `addr`, and length, `len`. The `flags` parameter controls how the update should be performed using the options shown in the following table.

`MS_ASYNC`	Perform asynchronous writes
`MS_SYNC`	Perform synchronous writes
`MS_INVALIDATE`	Read data back in from the file

The `munmap` function releases the memory segment.

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

int munmap(void *addr, size_t len);
```

The following program, `mmap.c`, shows a file of structures being updated using `mmap` and array-style accesses. Linux kernels before 2.0 don't fully support this use of `mmap`. The program does work correctly on Sun Solaris and other systems.

Try It Out Using mmap

1. Start by defining a `RECORD` structure and then creating `NRECORDS` versions, each recording their number. These are appended to the file `records.dat`.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>

typedef struct {
    int integer;
    char string[24];
} RECORD;

#define NRECORDS (100)

int main()
{
    RECORD record, *mapped;
    int i, f;
    FILE *fp;

    fp = fopen("records.dat","w+");
```

```
    for(i=0; i<NRECORDS; i++) {
        record.integer = i;
        sprintf(record.string,"RECORD-%d",i);
        fwrite(&record,sizeof(record),1,fp);
    }
    fclose(fp);
```

2. Next, change the integer value of record 43 to 143 and write this to the 43rd record's string:

```
    fp = fopen("records.dat","r+");
    fseek(fp,43*sizeof(record),SEEK_SET);
    fread(&record,sizeof(record),1,fp);

    record.integer = 143;
    sprintf(record.string,"RECORD-%d",record.integer);

    fseek(fp,43*sizeof(record),SEEK_SET);
    fwrite(&record,sizeof(record),1,fp);
    fclose(fp);
```

3. Now map the records into memory and access the 43rd record in order to change the integer to 243 (and update the record string), again using memory mapping:

```
    f = open("records.dat",O_RDWR);
    mapped = (RECORD *)mmap(0, NRECORDS*sizeof(record),
                        PROT_READ|PROT_WRITE, MAP_SHARED, f, 0);

    mapped[43].integer = 243;
    sprintf(mapped[43].string,"RECORD-%d",mapped[43].integer);

    msync((void *)mapped, NRECORDS*sizeof(record), MS_ASYNC);
    munmap((void *)mapped, NRECORDS*sizeof(record));
    close(f);

    exit(0);
}
```

In Chapter 13, you meet another shared memory facility: System V shared memory.

Summary

In this chapter, you've seen how Linux provides direct access to files and devices. You've seen how library functions build upon these low-level functions to provide flexible solutions to programming problems. As a result, you can write a fairly powerful directory-scanning routine in just a few lines of code.

You've also learned enough about file and directory handling to convert the fledgling CD application created at the end of Chapter 2 to a C program using a more structured file-based solution. At this stage, however, you can add no new functionality to the program, so we'll postpone the next rewrite until you've learned how to handle the screen and keyboard, which are the subjects of the next two chapters.