Ayisha Farhath Abbas
Assistant Professor
Department of Computer Science & IT

WORKING WITH FILES

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.

The various file-related topics are:

- ❑ 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

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. 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.

Directories, too, are special sorts of files. In 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. Really, almost everything is represented as a file under Linux, or can be made available via special files.

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. 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 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.

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. 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

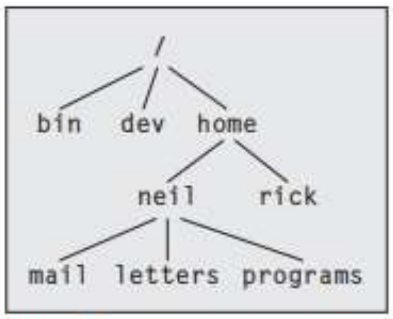


Figure 3-1

typical Linux hierarchy.

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.

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.

/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.

/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 command, which changes the modification time of a file or creates a new file if none exists with the given name.

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

$ cp /dev/null empty_file
```

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.

LIBRARY FUNCTIONS

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

❑ There’s a performance penalty in making a system call. System calls are therefore expensive compared to function calls. 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.

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.

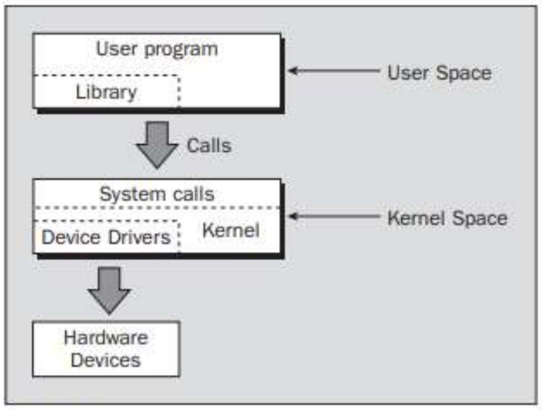


Figure 3-2

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.

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 *.

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. 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.

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.

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);
```

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);
```

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. 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 ints, not char. This allows the end-offile (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.

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 sprint

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.

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

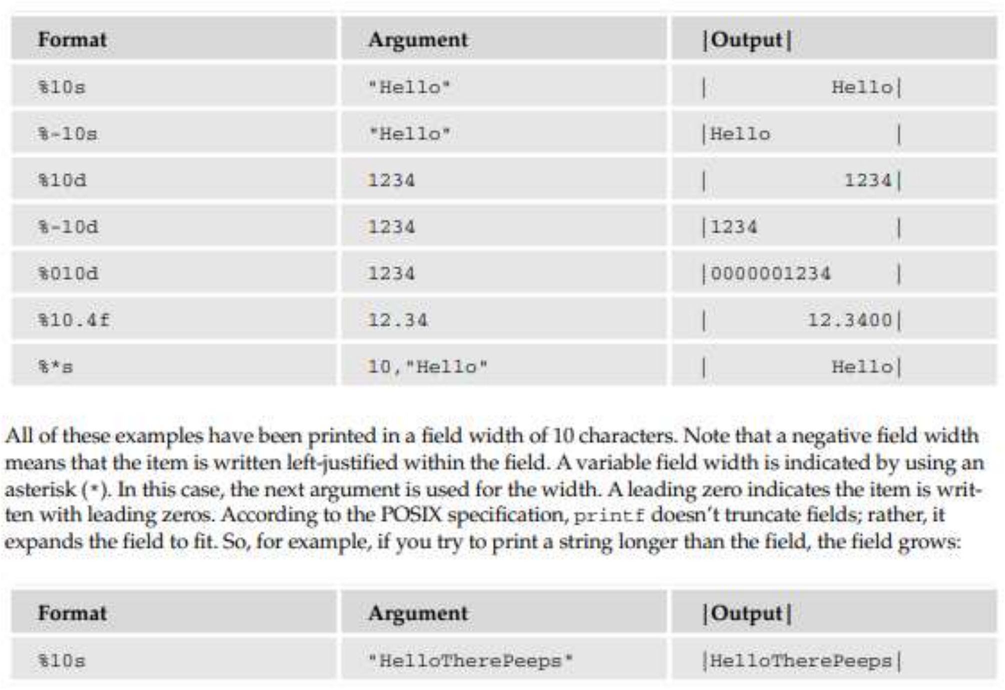
It's very important that the number and type of the arguments passed to printf match the conversion specifiers in the format string. An optional size specifier is used to indicate the type of integer arguments. This is either h, for example %hd, to indicate a short int, or l, for example %ld, to indicate 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|`

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.

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.

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. 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.

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.

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.

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;
```

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.

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.
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.

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.

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.

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
        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. 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.

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

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

```
#include
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. 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

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. It fails to open many 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

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);
```

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. 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 <string.h>
void perror(const char *s);
```

For example,

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

might give the following on the standard error output:

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