11 File Handling in GNU/Linux

In This Chapter

- Understand File Handling APIs in GNU/Linux
- Explore the Character Access Mechanisms
- Explore the String Access Mechanisms
- Investigate Both Sequential and Nonsequential (Random Access) Methods
- Review Alternate APIs and Methods for File Access

INTRODUCTION

This chapter looks at the file handling APIs of GNU/Linux and explores a number of applications to demonstrate the proper use of the file handling APIs. It looks at a number of different file handling functions, including character interfaces, string interfaces, and ASCII-mode and binary interfaces. The emphasis on this chapter is to discuss the APIs and then employ them in applications to illustrate their use.

FILE HANDLING WITH GNU/LINUX

File handling within GNU/Linux is accomplished through the standard C library. You can create and manipulate ASCII text or binary files with the same API. You can append to files or seek within them.

This chapter takes a look at the `fopen` call (to open or create a file), the `fwrite` and `fread` functions (to write to or read from a file), `fseek` (to position yourself at a given position in an existing file), the `feof` call (to test whether you are at the end of a file while reading), and some other lower level calls (such as `open`, `write`, and `read`).

NOTE

The file system in Linux is based on the original Unix File System, Version 7 Unix, which was released in 1979 from Bell Labs.

FILE HANDLING API EXPLORATION

Now it's time to get your hands dirty by working through some examples of GNU/Linux stream file I/O programming.

CREATING A FILE HANDLE

To write an application that performs file handling, you first need to make visible the file I/O APIs (function prototypes). This is done by simply including the `stdio.h` header file, as follows:

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

Not doing so results in compiler errors (undeclared symbols). The next step is to declare the handle to be used in file I/O operations. This is often called a *file pointer* and is a transparent structure that should not be accessed by the developer.

```
FILE *my_fp;
```

The next sections build on this to illustrate ASCII and binary applications.

OPENING A FILE

Now it's time to open a file and illustrate the variety of modes that can be used. Recall that opening a file can also be the mechanism to create a file. This example investigates this first.

The `fopen` function is very simple and provides the following API:

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

You specify the filename that you want to access (or create) through the first argument (`filename`) and then the mode you want to use (`mode`). The result of the `fopen` operation is a `FILE` pointer, which could be `NULL`, indicating that the operation failed.

The key to the `fopen` call is the mode that is provided. Table 11.1 provides an initial list of access modes.

TABLE 11.1 Simple File Access Modes

Mode	Description
`r`	Open an existing file for read
`w`	Open a file for write (create new if a file doesn't exist)
`a`	Open a file for append (create new if a file doesn't exist)
`rw`	Open a file for read and write (create new if a file doesn't exist)

The mode is simply a string that the `fopen` call uses to determine how to open (or create) the file. If you wanted to create a new file, you could simply use the `fopen` call as follows:

```
my_fp = fopen( "myfile.txt", "w" );
```

The result would be the creation of a new file (or the destruction of the existing file) in preparation for `write` operations. If instead you wanted to read from an existing file, you'd open it as follows:

```
my_fp = fopen( "myfile.txt", "r" );
```

Note that you are simply using a different mode here. The `read` mode assumes that the file exists, and if not, a `NULL` is returned.

In both cases, it is assumed that the file `myfile.txt` either exists or is created in the current working directory. The current directory is the directory from which you invoked your application.

It's very important that the results of all file I/O operations be checked for success. For the `fopen` call, you simply test the response for `NULL`. What happens upon error is ultimately application dependent (you decide). An example of one mechanism is provided in Listing 11.1.

LISTING 11.1 Catching an Error in an `fopen` Call (on the CD-ROM at `./source/ch11/test.c`)

```
 1:     #include <stdio.h>
 2:     #include <errno.h>
 3:     #include <string.h>
 4:
 5:     #define MYFILE    "missing.txt"
 6:
 7:     main()
 8:     {
 9:
10:       FILE *fin;
11:
12:       /* Try to open the file for read */
13:       fin = fopen( MYFILE, "r" );
14:
15:       /* Check for failure to open */
16:       if (fin == (FILE *)NULL) {
17:
18:         /* Emit an error message and exit */
19:         printf("%s: %s\n", MYFILE, strerror( errno ) );
20:         exit(-1);
21:
22:       }
23:
24:       /* All was well, close the file */
25:       fclose( fin );
26:
27:     }
```

In Listing 11.1, you use a couple of new calls not yet discussed. After trying to open the file at line 13, you check to see if the new file handle is `NULL` (zero). If it is, then you know that either the file is not present or you are not able to access it (you don't have proper access to the file). In this case, you emit an error message that consists of the file that you attempted to open for `read` and then the error message that resulted. You capture the error number (integer) with the `errno` variable. This is a special variable that is set by system calls to indicate the last error that occurred. You pass this value to the `strerror` function, which turns the integer error number into a string suitable for printing to standard-out. Executing the sample application results in the following:

```
$ ./app
missing.txt: No such file or directory
$
```

Now it's time to move on to writing and then reading data from a file.

TIP

The `errno` *variable is set to* `ENOENT` *if the file does not exist or* `EACCES` *if access to the file was denied because of lack of permissions.*

Reading and Writing Data

A number of methods exist for both reading and writing data to a file. More options can be a blessing, but it's also important to know where to use which mechanism. For example, you can read or write on a character basis or on a string basis (for ASCII text only). You can also use a more general API that permits reading and writing records, which supports both ASCII and binary representations. This chapter looks at each here, but focuses primarily on the latter mechanism.

The standard I/O library presents a buffered interface. This has two very important properties. First, system reads and writes are in blocks (typically 8KB in size). Character I/O is simply written to the `FILE` buffer, where the buffer is written to the media automatically when it's full. Second, `fflush` is necessary, or non-buffered I/O must be set if the data is being sent to an interactive device such as the console terminal.

Character Interfaces

The character interfaces are demonstrated in Listings 11.2 and 11.3. Listing 11.2 illustrates character output using `fputc` and Listing 11.3 illustrates character input using `fgetc`. These functions have the following prototypes:

```
int fputc( int c, FILE *stream );
int fgetc( FILE *stream );
```

This example generates an output file using `fputc` and then uses this file as the input to `fgetc`. In Listing 11.2, you open the output file at line 11 and then work your way through your sample string. The simple loop walks through the entire string until a `NULL` is detected, at which point you exit and close the file (line 21). At line 16, you use `fputc` to emit the character (as an `int`, per the `fputc` prototype) as well as specify your output stream (`fout`).

LISTING 11.2 The `fputc` Character Interface Example (on the CD-ROM at `./source/ch11/charout.c`)

```
 1:       #include <stdio.h>
 2:
 3:       int main()
 4:       {
 5:         int i;
 6:         FILE *fout;
 7:         const char string[]={"This\r\nis a test\r\nfile.\r\n\0"};
 8:
 9:         fout = fopen("inpfile.txt", "w");
10:
11:         if (fout == (FILE *)NULL) exit(-1);
12:
13:         i = 0;
14:         while (string[i] != NULL) {
15:
16:           fputc( (int)string[i], fout );
17:           i++;
18:
19:         }
20:
21:         fclose( fout );
22:
23:         return 0;
24:       }
```

The function to read this file using the character interface is shown in Listing 11.3. This function is very similar to the file creation example. You open the file for `read` at line 8 and follow with a test at line 10. You then enter a loop to get the characters from the file (lines 12–22). The loop simply reads characters from the file using `fgetc` and stops when the special `EOF` symbol is encountered. This is the indication that you've reached the end of the file. For all characters that are not `EOF` (line 16), you emit the character to standard-out using the `printf` function. Upon reaching the end of the file, you close it using `fclose` at line 24.

LISTING 11.3 The `fgetc` Character Interface Example (on the CD-ROM at `./source/ch11/charin.c`)

```
 1:       #include <stdio.h>
 2:
 3:       int main()
```

```
{
  int c;
  FILE *fin;

  fin = fopen("inpfile.txt", "r");

  if (fin == (FILE *)0) exit(-1);

  do {

    c = fgetc( fin );

    if (c != EOF) {

      printf("%c", (char)c);

    }

  } while (c != EOF);

  fclose( fin );

  return 0;
}
```

Executing the applications is illustrated as follows:

```
$ ./charout
$ ./charin
This
is a test
file.
$
```

The character interfaces are obviously simple, but they are also inefficient and should be used only if a string-based method cannot be used. We'll look at this interface next.

String Interfaces

This section takes a look at four library functions that provide the means to read and write strings. The first two (`fputs` and `fgets`) are simple string interfaces, and the second two (`fprintf` and `fscanf`) are more complex and provide additional capabilities.

The `fputs` and `fgets` interfaces mirror the previously discussed `fputc` and `fgetc` functions. They provide the means to write and read variable-length strings to files in a very simple way. Prototypes for the `fputs` and `fgets` are defined as:

```
int fputs( int c, FILE *stream );
char *fgets( char *s, int size, FILE *stream );
```

First take a look at a sample application that accepts strings from the user (via standard-input) and then writes them to a file (see Listing 11.4). This sample then halts the input process after a blank line has been received.

LISTING 11.4 Writing Variable Length Strings to a File (on the CD-ROM at `./source/ch11/strout.c`)

```
1:      #include <stdio.h>
2:
3:      #define LEN    80
4:
5:      int main()
6:      {
7:        char line[LEN+1];
8:        FILE *fout, *fin;
9:
10:       fout = fopen( "testfile.txt", "w" );
11:       if ( fout == (FILE *)0 ) exit(-1);
12:
13:       fin = fdopen( 0, "r" );
14:
15:       while ( (fgets( line, LEN, fin )) != NULL ) {
16:
17:         fputs( line, fout );
18:
19:       }
20:
21:       fclose( fout );
22:       fclose( fin );
23:
24:       return 0;
25:     }
```

The application shown in Listing 11.4 gets a little trickier, so the next paragraphs walk through this one line by line to cover all of the points. You declare the line string (used to read user input) at line 7, called oddly enough, `line`. Next, you declare two `FILE` pointers, one for input (called `fin`) and one for output (called `fout`).

At line 10, you open the output file using `fopen` to a new file called `testfile.txt`. You check the error status of this line at line 11, exiting if a failure occurs. At line 13, you use a special function `fdopen` to associate an existing file descriptor with a stream. In this case, you associate in the standard-input descriptor with a new stream called `fin` (returned by `fdopen`). Whatever you now type in (standard-in) is routed to this file stream. Next, you enter a loop that attempts to read from the `fin` stream (standard-in) and write this out to the output stream (`fout`). At line 15, you read using `fgets` and check the return with `NULL`. The `NULL` appears when you close the descriptor (which is achieved through pressing Ctrl+D at the keyboard). The `line` read is then emitted to the output stream using `fputs`. Finally, when the input stream has closed, you exit the loop and close the two streams at lines 21 and 22.

Now take a look at another example of the `read` side, `fgets`. In this example (Listing 11.5), you read the contents of the test file using `fgets` and then `printf` it to standard-out.

LISTING 11.5 Reading Variable Length Strings from a File (on the CD-ROM at `./source/ch11/strin.c`)

```
#include <stdio.h>

#define LEN    80

int main()
{
  char line[LEN+1];
  FILE *fin;

  fin = fopen( "testfile.txt", "r" );
  if ( fin == (FILE *)0 ) exit(-1);

  while ( (fgets( line, LEN, fin )) != NULL ) {

    printf( "%s", line );

  }

  fclose( fin );

  return 0;
}
```

In this example, you open the input file and create a new input stream handle called `fin`. You use this at line 13 to read variable-length strings from the file, and when one is read, you emit it to standard-out via `printf` at line 15.

This demonstrates writing and reading strings to and from a file, but what if your data is more structured than simple strings? If your strings are actually made up of lower level structures (such as integers, floating-point values, or other types), you can use another method to more easily deal with them. This is the next topic of discussion.

Consider the problem of reading and writing data that takes a regular form but consists of various data types (such as C structures). Say that you want to store an integer item (an `id`), two floating-point values (2D coordinates), and a string (an object name). Look first at the application that creates this file (see Listing 11.6). Note that in this example you ultimately deal with strings, but using the API functions, the ability to translate to the native data types is provided.

LISTING 11.6 Writing Structured Data in ASCII Format (on the CD-ROM at `./source/ch11/strucout.c`)

```
#include <stdio.h>

#define MAX_LINE    40

#define FILENAME "myfile.txt"

typedef struct {
  int id;
  float x_coord;
  float y_coord;
  char name[MAX_LINE+1];
} MY_TYPE_T;

#define MAX_OBJECTS    3

/* Initialize an array of three objects */
MY_TYPE_T objects[MAX_OBJECTS]={
  { 0, 1.5, 8.4, "First-object" },
  { 1, 9.2, 7.4, "Second-object" },
  { 2, 4.1, 5.6, "Final-object" }
};

int main()
{
  int i;
26:        FILE *fout;
27:
28:        /* Open the output file */
29:        fout = fopen( FILENAME, "w" );
30:        if (fout == (FILE *)0) exit(-1);
31:
32:        /* Emit each of the objects, one per line */
33:        for ( i = 0 ; i < MAX_OBJECTS ; i++ ) {
34:
35:          fprintf( fout, "%d %f %f %s\n",
36:                   objects[i].id,
37:                   objects[i].x_coord, objects[i].y_coord,
38:                   objects[i].name );
39:
40:        }
41:
42:        fclose( fout );
43:
44:        return 0;
45:      }
```

Listing 11.6 illustrates another string method for creating data files. You create a test structure (lines 7–12) to represent the data that you're going to write and then read. You initialize this structure at lines 17–21 with three rows of data. Now you can turn to the application. This one turns out to be very simple. At lines 29–30, you open and then check the `fout` file handle and then perform a `for` loop to emit our data to the file. You use the `fprintf` API function to emit this data. The format of the `fprintf` call is to first specify the output file pointer, followed by a format string, and then zero or more variables to be emitted. The format string mirrors your data structure. You're emitting an `int` (`%d`), two floating-point values (`%f`), and then finally a string (`%s`). This converts all data to string format and writes it to the output file. Finally, you close the output file at line 42 with the `fclose` call.

NOTE

You could have achieved this with a `sprintf` *call (to create your output string) and then written this out as follows:*

```
char line[81];
...
snprintf( line, 80, "%d %f %f %s\n",
             objects[i].id
             objects[i].x_coord, objects[i].y_coord,
             objects[i].name );
fputs( line, fout );
```

The disadvantage is that local space must be declared for the string being emitted. This would not be required with a call to `fprintf` *directly (the C library uses its own space internally).*

The prototypes for both the `fprintf` and `sprintf` are shown here:

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

From the file created in Listing 11.6, you read this file in Listing 11.7. This function utilizes the `fscanf` function to both read and interpret the data. After opening the input file (lines 21–22), you loop and read the data while the end of file has not been found. You detect the end of file marker using the `feof` function at line 25. The `fscanf` function utilizes the input stream (`fin`) and the format to be used to interpret the data. This string is identical to that used to write the data out (see Listing 11.6, line 35).

After a line of data has been read, it's immediately printed to standard-out using the `printf` function at lines 32–35. Finally, the input file is closed using the `fclose` call at line 39.

LISTING 11.7 Reading Structured Data in ASCII Format (on the CD-ROM at `./source/ch11/strucin.c`)

```
 1:   #include <stdio.h>
 2:
 3:   #define MAX_LINE    40
 4:
 5:   #define FILENAME "myfile.txt"
 6:
 7:   typedef struct {
 8:     int id;
 9:     float x_coord;
10:     float y_coord;
11:     char name[MAX_LINE+1];
12:   } MY_TYPE_T;
13:
14:   int main()
15:   {
16:     int i;
17:     FILE *fin;
18:     MY_TYPE_T object;
19:
20:     /* Open the input file */
```

```
    fin = fopen( FILENAME, "r" );
    if (fin == (FILE *)0) exit(-1);

    /* Read the records from the file and emit */
    while ( !feof(fin) ) {

      fscanf( fin, "%d %f %f %s\n",
               &object.id,
               &object.x_coord, &object.y_coord,
               object.name );

      printf("%d %f %f %s\n",
               object.id,
               object.x_coord, object.y_coord,
               object.name );

    }

    fclose( fin );

    return 0;
  }
```

You could have achieved this functionality with an `sscanf` *call (to parse our input string).*

```
char line[81];
...
fgets( fin, 80, line );
sscanf( line, 80, "%d %f %f %s\n",
             objects[i].id
             objects[i].x_coord, objects[i].y_coord,
             objects[i].name );
```

The disadvantage is that local space must be declared for the parse to be performed on the input string. This would not be required with a call to `fscanf` *directly.*

The `fscanf` and `sscanf` function prototypes are both shown here:

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

All of the methods discussed thus far require that you are dealing with ASCII text data. In the next section, you'll see API functions that permit dealing with binary data.

For survivability, it's important to not leave files open over long durations of time. When I/O is complete, the file should be closed with `fclose` *(or at a minimum, flushed with* `fflush`*). This has the effect of writing any buffered data to the actual file.*

READING AND WRITING BINARY DATA

This section looks at a set of library functions that provides the ability to deal with both binary and ASCII text data. The `fwrite` and `fread` functions provide the ability to deal not only with the I/O of objects, but also with arrays of objects. The prototypes of the `fwrite` and `fread` functions are provided here:

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

Now take a look at a couple of simple examples of `fwrite` and `fread` to explore their use (see Listing 11.8). In this first example, you emit the `MY_TYPE_T` structure first encountered in Listing 11.6.

LISTING 11.8 Using `fwrite` to Emit Structured Data (on the CD-ROM at `./source/ch11/binout.c`)

```
#include <stdio.h>

#define MAX_LINE    40

#define FILENAME "myfile.bin"

typedef struct {
  int id;
  float x_coord;
  float y_coord;
  char name[MAX_LINE+1];
} MY_TYPE_T;

#define MAX_OBJECTS    3

MY_TYPE_T objects[MAX_OBJECTS]={
  { 0, 1.5, 8.4, "First-object" },
  { 1, 9.2, 7.4, "Second-object" },
  { 2, 4.1, 5.6, "Final-object" }
};

int main()
{
  int i;
  FILE *fout;

  /* Open the output file */
  fout = fopen( FILENAME, "w" );
  if (fout == (FILE *)0) exit(-1);

  /* Write out the entire object's structure */
  fwrite( (void *)objects, sizeof(MY_TYPE_T), 3, fout );

  fclose( fout );

  return 0;
}
```

What's interesting to note about Listing 11.8 is that a single `fwrite` emits the entire structure. You specify the object that you're emitting (variable object, passed as a void pointer) and then the size of a row in this structure (the type `MY_TYPE_T`) using the `sizeof` operator. You then specify the number of elements in the array of types (`3`) and finally the output stream to which you want this object to be written.

Take a look now at the invocation of this application (called `binout`) and a method for inspecting the contents of the binary file (see Listing 11.9). After executing the `binout` executable, the file `myfile.bin` is generated. Attempting to use the `more` utility to inspect the file results in a blank line. This is because the first character in the file is a `NULL` character, which is interpreted by `more` as the end. Next, you use the `od` utility (octal dump) to emit the file without interpreting it. You specify `-x` as the option to emit the file in hexadecimal format. (For navigation purposes, the integer `id` field has been underlined.)

LISTING 11.9 Inspecting the Contents of the Generated Binary File

```
$ ./binout
$ more myfile.bin
$ od -x myfile.bin
0000000 0000 0000 0000 3fc0 6666 4106 6946 7372
0000020 2d74 626f 656a 7463 0000 0000 0000 0000
```

```
0000040 0000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0001 0000 3333 4113
0000100 cccd 40ec 6553 6f63 646e 6f2d 6a62 6365
0000120 0074 0000 0000 0000 0000 0000 0000 0000
0000140 0000 0000 0000 0000 0000 0000 0000 0000
0000160 0002 0000 3333 4083 3333 40b3 6946 616e
0000200 2d6c 626f 656a 7463 0000 0000 0000 0000
0000220 0000 0000 0000 0000 0000 0000 0000 0000
0000240 0000 0000 0000 0000
0000250
$
```

NOTE

One important item to note about reading and writing binary data is the issue of portability and endianness. Consider that you create your binary data on a Pentium system, but the binary file is moved to a PowerPC system to read. The data will be in the incorrect byte order and therefore essentially corrupt. The Pentium uses little endian byte order (least significant byte first in memory), whereas the PowerPC uses big endian (most significant byte first in memory). For portability, endianness should always be considered when dealing with binary data. Also consider the use of host and network byte swapping functions, as discussed in Chapter 12, "Programming with Pipes."

Now it's time to take a look at reading this file using `fread`, but rather than reading it sequentially, read it in a nonsequential way (otherwise known as random access).This example reads the records of the file in reverse order. This requires the use of two new functions that permit you to seek into a file (`fseek`) and also rewind back to the start (`rewind`):

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

The `rewind` function simply resets the file read pointer back to the start of the file, whereas the `fseek` function moves you to a new position given an index. The `whence` argument defines whether the position is relative to the start of the file (`SEEK_SET`), the current position (`SEEK_CUR`), or the end of the file (`SEEK_END`). See Table 11.2. The `lseek` function operates like `fseek`, but instead on a file descriptor.

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

In this example (Listing 11.10), you open the file using `fopen`, which automatically sets the `read` index to the start of the file. Because you want to read the last record first, you seek into the file using `fseek` (line 26). The index that you specify

TABLE 11.2 Function `fseek/lseek` `whence` Arguments

Name	Description
`SEEK_SET`	Moves the file position to the position defined by `offset`.
`SEEK_CUR`	Moves the file position the number of bytes defined by `offset` from the current file position.
`SEEK_END`	Moves the file position to the number of bytes defined by `offset` from the end of the file.

is twice the size of the record size (`MY_TYPE_T`). This puts you at the first byte of the third record, which is then read with the `fread` function at line 28. The `read` index is now at the end of the file, so you reset the `read` position to the top of the file using the `rewind` function.

You repeat this process, setting the file `read` position to the second element at line 38, and then read again with `fread`. The final step is reading the first element in the file. This requires no `fseek` because after the `rewind` (at line 48), you are at the top of the file. You can then `fread` the first record at line 50.

LISTING 11.10 Using `fread` and `fseek/rewind` to Read Structured Data (on the CD-ROM at `./source/ch11/nonseq.c`)

```
1:      #include <stdio.h>
2:
3:      #define MAX_LINE    40
4:
5:      #define FILENAME "myfile.txt"
6:
7:      typedef struct {
8:        int id;
9:        float x_coord;
10:       float y_coord;
11:       char name[MAX_LINE+1];
12:     } MY_TYPE_T;
13:
14:     MY_TYPE_T object;
15:
16:     int main()
17:     {
18:       int i;
19:       FILE *fin;
```

```

  /* Open the input file */
  fin = fopen( FILENAME, "r" );
  if (fin == (FILE *)0) exit(-1);

  /* Get the last entry */
  fseek( fin, (2 * sizeof(MY_TYPE_T)), SEEK_SET );

  fread( &object, sizeof(MY_TYPE_T), 1, fin );

  printf("%d %f %f %s\n",
           object.id,
           object.x_coord, object.y_coord,
           object.name );

  /* Get the second to last entry */
  rewind( fin );

  fseek( fin, (1 * sizeof(MY_TYPE_T)), SEEK_SET );

  fread( &object, sizeof(MY_TYPE_T), 1, fin );

  printf("%d %f %f %s\n",
           object.id,
           object.x_coord, object.y_coord,
           object.name );

  /* Get the first entry */
  rewind( fin );

  fread( &object, sizeof(MY_TYPE_T), 1, fin );

  printf("%d %f %f %s\n",
           object.id,
           object.x_coord, object.y_coord,
           object.name );

  fclose( fin );

  return 0;
}
```

The process of reading the third record is illustrated graphically in Figure 11.1. It illustrates `fopen`, `fseek`, `fread`, and finally `rewind`.

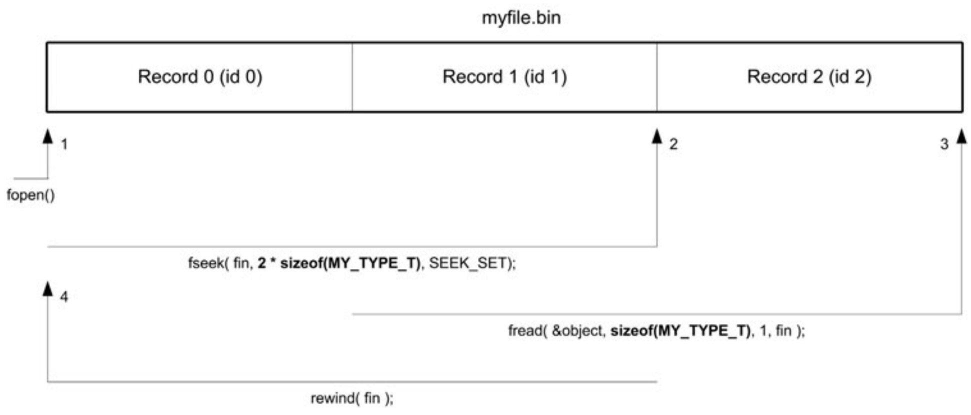
myfile.bin
Record 0 (id 0)
Record 1 (id 1)
Record 2 (id 2)
1
2
3
4
fopen()
fseek(fin, 2 * sizeof(MY_TYPE_T), SEEK_SET);
fread(&object, sizeof(MY_TYPE_T), 1, fin);
rewind(fin);

FIGURE 11.1 Nonsequential Reads in a Binary File

The function `ftell` provides the means to identify the current position. This function returns the current position as a long type and can be used to pass as the offset to `fseek` (with `SEEK_SET`) to reset to that position. The `ftell` prototype is provided here:

```
long ftell( FILE *stream );
```

An alternate API exists to `ftell` and `fseek`. The `fgetpos` and `fsetpos` provide the same functionality, but in a different form. Rather than an absolute position, an opaque type is used to represent the position (returned by `fgetpos`, passed into `fsetpos`). The prototypes for these functions are provided here:

```
int fgetpos( FILE *stream, fpos_t *pos );
int fsetpos( FILE *stream, fops_t *pos );
```

A sample code snippet of these functions is shown here:

```
fpos_t file_pos;
...
/* Get desired position */
fgetpos( fin, &file_pos );
...
rewind( fin );
/* Return to desired position */
fsetpos( fin, &file_pos );
```

It's recommended to use the fgetpos and fsetpos APIs over the ftell and fseek methods. Because the ftell and fseek methods don't abstract the details of the mechanism, the fgetpos and fsetpos functions are less likely to be deprecated in the future.

BASE API

The open, read, and write functions can also be used for file I/O. The API differs somewhat, but this section also looks at how to switch between file and stream mode with fdopen.

NOTE

These functions are referred to as the base API because they are the platform from which the standard I/O library is built.

The open function allows you to open or create a new file. Two variations are provided, with their APIs listed here:

```
int open( const char *pathname, int flags );
int open( const char *pathname, int flags, mode_t mode );
```

The pathname argument defines the file (with path) to be opened or created (such as temp.txt or /tmp/myfile.txt). The flags argument is one of O_RDONLY, O_WRONLY, or O_RDWR. One or more of the flags shown in Table 11.3 might also be OR'd (bitwise OR operation) in, depending on the needs of the open call.

TABLE 11.3 Additional Flags for the open Function

Flag	Description
O_CREAT	Create the file if it doesn't exist.
O_EXCL	If used with O_CREAT, will return an error if the file already exists; otherwise the file is created.
O_NOCTTY	If the file descriptor refers to a TTY device, this process will not become the controlling terminal.
O_TRUNC	The file will be truncated (if it exists) and the length reset to zero if write privileges are permitted.
O_APPEND	The file pointer is repositioned to the end of the file prior to each write.

→

Flag	Description
O_NONBLOCK	Opens the file in nonblocking mode. Operations on the file will not block (such as read, write, and so on).
O_SYNC	write functions are blocked until the data is written to the physical device.
O_NOFOLLOW	Fail following symbolic links.
O_DIRECTORY	Fail the open if the file being opened is not a directory.
O_DIRECT	Attempts to minimize cache effects by doing I/O directly to/from user space buffers (synchronously as with O_SYNC).
O_ASYNC	Requests a signal when data is available on input or output of this file descriptor.
O_LARGEFILE	Request a large filesystem file to be opened on a 32-bit system whose size cannot be represented in 32 bits.

The third argument for the second open instance is a mode. This mode defines the permissions to be used when the file is created (used only with the flag O_CREAT). Table 11.4 lists the possible symbolic constants that can be OR'd together.

TABLE 11.4 Mode Arguments for the open System Call

Constant	Use
S_IRWXU	User has read/write/execute permissions.
S_IREAD	User has read permission.
S_IWRITE	User has write permission.
S_IEXEC	User has execute permission.
S_IRWXG	Group has read/write/execute permissions.
S_IRGRP	Group has read permission.
S_IWGRP	Group has write permission.
S_IXGRP	Group has execute permission.
S_IRWXO	Others have read/write/execute permissions.
S_IROTH	Others have read permission.
S_IWOTH	Others have write permission.
S_IXOTH	Others have execute permission.

To open a new file in the tmp directory, you could do the following:

```
int fd;
fd = open( "/tmp/newfile.txt", O_CREAT | O_WRONLY );
```

To instead open an existing file for read, you could open as follows:

```
int fd;
fd = open( "/tmp/newfile.txt", O_RDONLY );
```

Reading and writing to these files is done very simply with the read and write API functions:

```
ssize_t read( int fd, void *buf, size_t count );
ssize_t write( int fd, const void *buf, size_t count );
```

These are used simply with a buffer and a size to represent the number of bytes to read or write, such as:

```
unsigned char buffer[MAX_BUF+1];
int fd, ret;
...
ret = read( fd, (void *)buffer, MAX_BUF );
...
ret = write( fd, (void *)buffer, MAX_BUF );
```

You'll see more examples of these in Chapter 12 "Programming with Pipes" and Chapter 13 "Introduction to Sockets Programming." What's interesting here is that the same set of API functions to read and write data to a file can also be used for pipes and sockets. This represents a unique aspect of the UNIX-like operating systems, where many types of devices are represented as files. The result is that a common API can be used over a broad range of devices.

TIP

File and string handling are some of the strengths of the object-oriented scripting languages. This book explores two such languages (Ruby and Python) in Chapter 26, "Scripting with Ruby," and Chapter 27, "Scripting with Python."

Finally, a file descriptor can be attached to a stream by using the fdopen system call. This call has the following prototype:

```
FILE *fdopen( int filedes, const char *mode );
```

Therefore, if you've opened a device using the `open` function call, you can associate a stream with it using `fdopen` and then use stream system calls on the device (such as `fscanf` or `fprintf`). Consider the following example:

```
FILE *fp;
int fd;
fd = open( "/tmp/myfile.txt", O_RDWR );
fp = fdopen( fd, "rw" );
```

After this is done, you can use `read/write` with the `fd` descriptor or `fscanf/fprintf` with the `fp` descriptor.

One other useful API to consider is the `pread/pwrite` API. These functions require an offset into the file to read or write, but they do not affect the file pointer. These functions have the following prototype:

```
ssize_t pread( int filedes, void *buf, size_t nbyte, off_t offset );
ssize_t pwrite( int filedes, void *buf, size_t nbyte, off_t offset );
```

These functions require that the target be seekable (in other words, regular files) and used regularly for record I/O in databases.

SUMMARY

In this chapter, the file handling APIs were discussed with examples provided for each. The character interfaces were first explored (`fputc`, `fgetc`), followed by the string interfaces (`fputs`, `fgets`). Some of the more structured methods for generating and parsing files were then investigated (such as the `fprintf` and `fscanf` functions), in addition to some of the other possibilities (`sprintf` and `sscanf`). Finally, the topics of binary files and random (nonsequential) access were discussed, including methods for saving and restoring file positions.

FILE HANDLING APIS

```
FILE *fopen( const char *filename, const char *mode );
FILE *fdopen( int filedes, const char *type );
int fputc( int c, FILE *stream );
int fgetc( FILE *stream );
int fputs( int c, FILE *stream );
char *fgets( char *s, int size, FILE *stream );
int fprintf( FILE* stream, const char *format, ... );
int sprintf( char *str, const char *format, ... );
int fscanf( FILE *stream, const char *format, ... );
int sscanf( const char *str, const char *format, ... );
void rewind( FILE *stream );
int fseek( FILE *stream, long offset, int whence );
int lseek( in filedes, long offset, int whence );
long ftell( FILE *stream );
int fgetpos( FILE *stream, fpos_t *pos );
int fsetpos( FILE *stream, fops_t *pos );
int fclose( FILE *stream );
int open( const char *pathname, int flags );
int open( const char *pathname, int flags, mode_t mode );
ssize_t read( int fd, void *buf, size_t count );
ssize_t write( int fd, const void *buf, size_t count );
ssize_t pread( int filedes, void *buf, size_t count, off_t offset );
ssize_t pwrite( int filedes, const void *buf,
                size_t count, off_t offset );
```