CHAPTER 3

Buffered I/O

Recall from Chapter 1 that the *block* is an abstraction representing the smallest unit of storage on a filesystem. Inside the kernel, all filesystem operations occur in terms of blocks. Indeed, the block is the lingua franca of I/O. Consequently, no I/O operation may execute on an amount of data less than the block size or that is not an integer multiple of the block size. If you only want to read a byte, too bad: you'll have to read a whole block. Want to write 4.5 blocks worth of data? You'll need to write 5 blocks, which implies reading that partial block in its entirety, updating only the half you've changed, and then writing back the whole block.

You can see where this is leading: partial block operations are inefficient. The operating system has to "fix up" your I/O by ensuring that everything occurs on block-aligned boundaries and rounding up to the next largest block. Unfortunately, this is not how user-space applications are generally written. Most applications operate in terms of higher-level abstractions, such as fields and strings, whose size varies independently of the block size. At its worst, a user-space application might read and write but a single byte at a time! That's a lot of waste. Each of those one-byte writes is actually writing a whole block.

The situation is exacerbated by the extraneous system calls required to read, say, a single byte 1,024 times rather than a single 1,024-byte block all at once. The solution to this pathological performance problem is *user-buffered I/O*, a way for applications to read and write data in whatever amounts feel natural but have the actual I/O occur in units of the filesystem block size.

User-Buffered I/O

Programs that have to issue many small I/O requests to regular files often perform user-buffered I/O. This refers to buffering done in user space, either manually by the application or transparently in a library, not to buffering done by the kernel. As discussed in

Chapter 2, for reasons of performance, the kernel buffers data internally by delaying writes, coalescing adjacent I/O requests, and reading ahead. Through different means, user buffering also aims to improve performance.

Consider an example using the user-space program *dd*:

```
dd bs=1 count=2097152 if=/dev/zero of=pirate
```

Because of the `bs=1` argument, this command will copy two megabytes from the device */dev/zero* (a virtual device providing an endless stream of zeros) to the file *pirate* in 2,097,152 one-byte chunks. That is, it will copy the data via about two million read and write operations—one byte at a time.

Now consider the same two megabyte copy, but using 1,024 byte blocks:

```
dd bs=1024 count=2048 if=/dev/zero of=pirate
```

This operation copies the same two megabytes to the same file, yet issues 1,024 times fewer read and write operations. The performance improvement is huge, as you can see in Table 3-1. Here, I've recorded the time taken (using three different measures) by four *dd* commands that differed only in block size. Real time is the total elapsed wall clock time, user time is the time spent executing the program's code in user space, and system time is the time spent executing system calls in kernel space on the process's behalf.

Table 3-1. Effects of block size on performance

Block size	Real time	User time	System time
1 byte	18.707 seconds	1.118 seconds	17.549 seconds
1,024 bytes	0.025 seconds	0.002 seconds	0.023 seconds
1,130 bytes	0.035 seconds	0.002 seconds	0.027 seconds

Using 1,024 byte chunks results in an *enormous* performance improvement compared to the single byte chunk. However, the table also demonstrates that using a larger block size—which implies even fewer system calls—can result in performance degradation if the operations are not performed in multiples of the disk's block size. Despite requiring fewer calls, the 1,130 byte requests end up generating unaligned requests, and are therefore less efficient than the 1,024 byte requests.

Taking advantage of this performance boon requires prior knowledge of the physical block size. The results in the table show the block size is most likely 1,024, an integer multiple of 1,024, or a divisor of 1,024. In the case of */dev/zero*, the block size is actually 4,096 bytes.

Block Size

In practice, blocks are usually 512, 1,024, 2,048, 4,096, or 8,192 bytes in size.

As Table 3-1 demonstrates, a large performance gain is realized simply by performing operations in chunks that are integer multiples or divisors of the block size. This is because the kernel and hardware speak in terms of blocks. Thus, using the block size or a value that fits neatly inside of a block guarantees block-aligned I/O requests and prevents extraneous work inside the kernel.

Figuring out the block size for a given device is easy using the `stat()` system call (covered in Chapter 8) or the *stat*(1) command. It turns out, however, that you generally do not need to know the actual block size.

The primary goal in picking a size for your I/O operations is to not pick an oddball size such as 1,130. No block in the history of Unix has been 1,130 bytes, and choosing such a size for your operations will result in unaligned I/O after the first request. Using any integer multiple or divisor of the block size, however, prevents unaligned requests. So long as your chosen size keeps everything block-aligned, performance will be good. Larger multiples will simply result in fewer system calls.

Therefore, the easiest choice is to perform I/O using a large buffer size that is a common multiple of the typical block sizes. Both 4,096 and 8,192 bytes work great.

So perform all your I/O in 4 or 8KB chunks and everything is great? Not so fast. The problem, of course, is that programs rarely deal in terms of blocks. Programs work with fields, lines, and single characters, not abstractions such as blocks. User-buffered I/O closes the gap between the filesystem, which speaks in blocks, and the application, which talks in its own abstractions. How it works is simple, yet very powerful: as data is written, it is stored in a buffer inside the program's address space. When the buffer reaches a set size, called the *buffer size*, the entire buffer is written out in a single write operation. Likewise, data is read using buffer-sized, block-aligned chunks. The application's various-sized read requests are served not directly from the filesystem, but via the chunks of the buffer. As the application reads more and more, data is handed out from the buffer piece-by-piece. Ultimately, when the buffer is empty, another large block-aligned chunk is read in. In this manner, although the application reads and writes in whatever odd sizes it chooses, data is buffered such that only large, block-aligned reads and writes are actually issued to the filesystem. The end result is fewer system calls for larger amounts of data, all aligned on block boundaries. Huge performance benefits ensue.

It is possible to implement user buffering by hand in your own programs. Indeed, many mission-critical applications do just that. The vast majority of programs, however, make use of the popular *standard I/O library* (part of the *standard C library*) or the *iostream library* (part of the *standard C++ library*), which provides a robust and capable user-buffering solution.

Standard I/O

The standard C library provides the standard I/O library (often simply called *stdio*), which in turn provides a platform-independent, user-buffering solution. The standard I/O library is simple to use, yet powerful.

Unlike programming languages such as FORTRAN, the C language does not include any built-in support or keywords providing any functionality more advanced than flow control, arithmetic, and so on—there's certainly no inherent support for I/O. As the C programming language progressed, users developed standard sets of routines to provide core functionality, such as string manipulation, mathematical routines, time and date functionality, and I/O. Over time, these routines matured, and with the ratification of the ANSI C standard in 1989 (C89) they were eventually formalized as the standard C library. Although C95, C99, and C11 added several new interfaces, the standard I/O library has remained relatively untouched since its creation in 1989.

The remainder of this chapter discusses user-buffered I/O as it pertains to file I/O, and is implemented in the standard C library—that is, opening, closing, reading, and writing files via the standard C library. Whether an application will use standard I/O, a home-rolled user-buffering solution, or straight system calls is a decision that developers must make carefully after weighing their application's needs and behavior.

The C standards always leave some details up to each implementation, and implementations often add additional features. This chapter, as with the rest of this book, documents the interfaces and behavior as they are implemented in *glibc* on a modern Linux system. Where Linux deviates from the standard, this is noted.

File Pointers

StandardI/O routines do not operate directly on file descriptors. Instead, they use their own unique identifier, known as the *file pointer*. Inside the C library, the file pointer maps to a file descriptor. The file pointer is represented by a pointer to the `FILE` typedef, which is defined in `<stdio.h>`.

`FILE`: Why the Caps?

The `FILE` name is often derided for its ugly use of all-caps, which is particularly egregious given that the C standard (and consequently most application's own coding styles) use all-lowercase names for functions and types. The peculiarity lies in history: standard I/O was originally written as macros. Not only `FILE` but all of the methods in the library were implemented as a set of macros. The style at the time, which remains common to this day, is to give macros all-caps names. As the C language progressed and standard I/O was ratified as an official part, most of the methods were reimplemented as proper functions and `FILE` became a typedef. But the uppercase remains.

In standard I/O parlance, an open file is called a *stream*. Streams may be opened for reading (*input streams*), writing (*output streams*), or both (*input/output streams*).

Opening Files

Files are opened for reading or writing via `fopen()`:

```
#include <stdio.h>

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

This function opens the file `path` with the behavior given by `mode` and associates a new stream with it.

Modes

The `mode` argument describes how to open the given file. It is one of the following strings:

`r`
: Open the file for reading. The stream is positioned at the start of the file.

`r+`
: Open the file for both reading and writing. The stream is positioned at the start of the file.

`w`
: Open the file for writing. If the file exists, it is truncated to zero length. If the file does not exist, it is created. The stream is positioned at the start of the file.

`w+`
: Open the file for both reading and writing. If the file exists, it is truncated to zero length. If the file does not exist, it is created. The stream is positioned at the start of the file.

`a`

Open the file for writing in append mode. The file is created if it does not exist. The stream is positioned at the end of the file. All writes will append to the file.

`a+`

Open the file for both reading and writing in append mode. The file is created if it does not exist. The stream is positioned at the end of the file. All writes will append to the file.

The given mode may also contain the character `b`, although this value is always ignored on Linux. Some operating systems treat text and binary files differently, and the `b` mode instructs the file to be opened in binary mode. Linux, as with all POSIX-conforming systems, treats text and binary files identically.

Upon success, `fopen()` returns a valid `FILE` pointer. On failure, it returns `NULL` and sets `errno` appropriately.

For example, the following code opens */etc/manifest* for reading and associates it with `stream`:

```
FILE *stream;

stream = fopen ("/etc/manifest", "r");
if (!stream)
        /* error */
```

Opening a Stream via File Descriptor

The function `fdopen()` converts an already open file descriptor (`fd`) to a stream:

```
#include <stdio.h>

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

The possible modes are the same as for `fopen()` and must be compatible with the modes originally used to open the file descriptor. The modes `w` and `w+` may be specified, but they will not cause truncation. The stream is positioned at the file position associated with the file descriptor.

Once a file descriptor is converted to a stream, I/O should no longer be directly performed on the file descriptor. It is, however, legal to do so. Note that the file descriptor is not duplicated, but is merely associated with a new stream. Closing the stream will close the file descriptor as well.

On success, `fdopen()` returns a valid file pointer; on failure, it returns `NULL` and sets `errno` appropriately.

For example, the following code opens the file */home/kidd/map.txt* via the `open()` system call and then uses the backing file descriptor to create an associated stream:

```
FILE *stream;
int fd;

fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == -1)
        /* error */

stream = fdopen (fd, "r");
if (!stream)
        /* error */
```

Closing Streams

The `fclose()` function closes a given stream:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Any buffered and not-yet-written data is first flushed. On success, `fclose()` returns `0`. On failure, it returns `EOF` and sets `errno` appropriately.

Closing All Streams

The `fcloseall()` function closes all streams associated with the current process, including standard in, standard out, and standard error:

```
#define _GNU_SOURCE

#include <stdio.h>

int fcloseall (void);
```

Before closing, all streams are flushed. The function always returns `0`; it is Linux-specific.

Reading from a Stream

Now that we know how to open and close streams, let's look at doing something useful: first reading from, and then writing to, them.

The standard C library implements multiple functions for reading from an open stream, ranging from the common to the esoteric. This section will look at three of the most popular approaches to reading: reading one character at a time, reading an entire line

at a time, and reading binary data. To read from a stream, it must have been opened as an input stream with the appropriate mode; that is, any valid mode except `w` or `a`.

Reading a Character at a Time

Often, the ideal I/O pattern is simply reading one character at a time. The function `fgetc()` is used to read a single character from a stream:

```
#include <stdio.h>

int fgetc (FILE *stream);
```

This function reads the next character from `stream` and returns it as an `unsigned char` cast to an `int`. The casting is done to have a sufficient range for notification of end-of-file or error: `EOF` is returned in such conditions. The return value of `fgetc()` must be stored in an `int`. Storing it in a `char` is a common but dangerous mistake, as you'll lose the ability to detect errors.

The following example reads a single character from `stream`, checks for errors, and then prints the result as a `char`:

```
int c;

c = fgetc (stream);
if (c == EOF)
        /* error */
else
        printf ("c=%c\n", (char) c);
```

The stream pointed at by `stream` must be open for reading.

Putting the character back

Standard I/O provides a function for pushing a character back onto a stream, allowing you to "peek" at the stream and return the character if it turns out that you don't want it:

```
#include <stdio.h>

int ungetc (int c, FILE *stream);
```

Each call pushes back `c`, cast to an `unsigned char`, onto `stream`. On success, `c` is returned; on failure, `EOF` is returned. A subsequent read from `stream` will return `c`. If multiple characters are pushed back, they are returned in the reverse order, *last in/first out* like a stack—that is, the more recently pushed character is returned first. The C standard dictates that only one pushback is guaranteed to succeed without intervening read requests. Some implementations, in turn, allow only a single pushback; Linux allows an infinite number of pushbacks, so long as memory is available. One pushback, of course, always succeeds.

If you make an intervening call to a seeking function (see "Seeking a Stream" on page 80) after calling `ungetc()` but before issuing a read request, it will cause all pushed-back characters to be discarded. This is true among threads in a single process, as threads share the buffer.

Reading an Entire Line

The function `fgets()` reads a string from a given stream:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

This function reads up to *one less* than `size` bytes from `stream` and stores the results in `str`. A null character (`\0`) is stored in the buffer after the last byte read in. Reading stops after an `EOF` or a newline character is reached. If a newline is read, the `\n` is stored in `str`.

On success, `str` is returned; on failure, `NULL` is returned.

For example:

```
char buf[LINE_MAX];

if (!fgets (buf, LINE_MAX, stream))
        /* error */
```

POSIX defines `LINE_MAX` in `<limits.h>`: it is the maximum size of input line that POSIX line-manipulating interfaces can handle. Linux's C library has no such limitation—lines may be of any size—but there is no way to communicate that with the `LINE_MAX` definition. Portable programs can use `LINE_MAX` to remain safe; it is set relatively high on Linux. Linux-specific programs need not worry about limits on the sizes of lines.

Reading arbitrary strings

Often, the line-based reading of `fgets()` is useful. Nearly as often, it is annoying. Sometimes, developers want to use a delimiter other than the newline. Other times, developers do not want a delimiter at all—and rarely do developers want the delimiter stored in the buffer! In retrospect, the decision to store the newline in the returned buffer rarely appears correct.

It is not hard to write an `fgets()` replacement that uses `fgetc()`. For example, this snippet reads the `n` – 1 bytes from `stream` into `str`, and then appends a `\0` character:

```
char *s;
int c;

s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF)
```

```
        *s++ = c;
*s = '\0';
```

The snippet can be expanded to also stop reading at a delimiter, given by the integer `d` (which cannot be the null character in this example):

```
char *s;
int c = 0;

s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF && (*s++ = c) != d)
        ;

if (c == d)
        *--s = '\0';
else
        *s = '\0';
```

Setting `d` to `\n` would provide behavior similar to `fgets()`, minus storing the newline in the buffer.

Depending on the implementation of `fgets()`, this variant is probably slower, as it issues repeated function calls to `fgetc()`. This is not the same problem exhibited by our original *dd* example, however! Although this snippet incurs additional function call overhead, it does not incur the system call overhead and unaligned I/O penalty burdened on *dd* with `bs=1`. The latter are much larger problems.

Reading Binary Data

For some applications, reading individual characters or lines is insufficient. Sometimes, developers want to read and write complex binary data, such as C structures. For this, the standard I/O library provides `fread()`:

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

A call to `fread()` will read up to `nr` elements of data, each of `size` bytes, from `stream` into the buffer pointed at by `buf`. The file pointer is advanced by the number of bytes read.

The number of elements read (not the number of bytes read!) is returned. The function indicates failure or `EOF` via a return value less than `nr`. Unfortunately, it is impossible to know which of the two conditions occurred without using `ferror()` and `feof()` (see "Errors and End-of-File" on page 83).

Because of differences in variable sizes, alignment, padding, and byte order, binary data written with one application may not be readable by a different application, or even by the same application on a different machine.

Issues of Alignment

All machine architectures have *data alignment* requirements. Programmers tend to think of memory as simply an array of bytes. Our processors, however, do not read and write from memory in byte-sized chunks. Instead, processors access memory with a specific granularity, such as 2, 4, 8, or 16 bytes. Since each process's address space starts at address 0, processes must initiate access from an address that is an integer multiple of the granularity.

Consequently, C variables must be stored at and accessed from aligned addresses. In general, variables are *naturally aligned*, which refers to the alignment that corresponds to the size of the C data type. For example, a 32-bit integer is aligned on a 4-byte boundary. In other words, on most architectures, an `int` is be stored at a memory address that is evenly divisible by four.

Accessing misaligned data has various penalties, which depend on the machine architecture. Some processors can access misaligned data, but with a performance penalty. Other processors cannot access misaligned data at all, and attempting to do so causes a hardware exception. Worse, some processors silently drop the low-order bits in order to force the address to be aligned, almost certainly resulting in unintended behavior.

Normally, the compiler naturally aligns all data, and alignment is not a visible issue to the programmer. Dealing with structures, performing memory management by hand, saving binary data to disk, and communicating over a network may bring alignment issues to the forefront. System programmers, therefore, ought to be well versed in these issues.

Chapter 9 addresses alignment in greater depth.

The simplest example of `fread()` is reading a single element of linear bytes from a given stream:

```
char buf[64];
size_t nr;

nr = fread (buf, sizeof(buf), 1, stream);
if (nr == 0)
        /* error */
```

We will look at examples that are more complicated when we study the write counterpart to `fread()`, `fwrite()`.

Writing to a Stream

As with reading, the standard C library defines several functions for writing to an open stream. This section will look at three of the most popular approaches to writing: writing

a single character, writing a string of characters, and writing binary data. Such varied writing approaches are ideally suited to buffered I/O. To write to a stream, it must have been opened as an output stream with the appropriate mode, that is, any valid mode except `r`.

Writing a Single Character

The counterpart of `fgetc()` is `fputc()`:

```
#include <stdio.h>

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

The `fputc()` function writes the byte specified by `c` (cast to an `unsigned char`) to the stream pointed at by `stream`. Upon successful completion, the function returns `c`. Otherwise, it returns `EOF`, and `errno` is set appropriately.

Use is simple:

```
if (fputc ('p', stream) == EOF)
        /* error */
```

This example writes the character `p` to `stream`, which must be open for writing.

Writing a String of Characters

The function `fputs()` is used to write an entire string to a given stream:

```
#include <stdio.h>

int fputs (const char *str, FILE *stream);
```

A call to `fputs()` writes all of the null-terminated string pointed at by `str` to the stream pointed at by `stream`. On success, `fputs()` returns a nonnegative number. On failure, it returns `EOF`.

The following example opens the file for writing in append mode, writes the given string to the associated stream, and then closes the stream:

```
FILE *stream;

stream = fopen ("journal.txt", "a");
if (!stream)
        /* error */

if (fputs ("The ship is made of wood.\n", stream) == EOF)
        /* error */

if (fclose (stream) == EOF)
        /* error */
```

Writing Binary Data

Individual characters and lines will not cut it when programs need to write complex data. To directly store binary data such as C variables, standard I/O provides `fwrite()`:

```
#include <stdio.h>

size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

A call to `fwrite()` will write to `stream` up to `nr` elements, each `size` bytes in length, from the data pointed at by `buf`. The file pointer will be advanced by the total number of bytes written.

The number of elements (not the number of bytes!) successfully written will be returned. A return value less than `nr` denotes error.

Sample Program Using Buffered I/O

Now let's look at an example—a complete program, in fact—that integrates many of the interfaces we have covered thus far in this chapter. This program first defines `struct pirate` and then declares two variables of that type. The program initializes one of the variables and subsequently writes it out to disk via an output stream to the file *data*. Via a different stream, the program reads the data back in from *data* directly to the other instance of `struct pirate`. Finally, the program writes the contents of the structure to standard out:

```
#include <stdio.h>

int main (void)
{
        FILE *in, *out;
        struct pirate {
                char            name[100]; /* real name */
                unsigned long   booty;     /* in pounds sterling */
                unsigned int    beard_len; /* in inches */
        } p, blackbeard = { "Edward Teach", 950, 48 };

        out = fopen ("data", "w");
        if (!out) {
                perror ("fopen");
                return 1;
        }

        if (!fwrite (&blackbeard, sizeof (struct pirate), 1, out)) {
                perror ("fwrite");
                return 1;
```

```
        }

        if (fclose (out)) {
                perror ("fclose");
                return 1;
        }

        in = fopen ("data", "r");
        if (!in) {
                perror ("fopen");
                return 1;
        }

        if (!fread (&p, sizeof (struct pirate), 1, in)) {
                perror ("fread");
                return 1;
        }

        if (fclose (in)) {
                perror ("fclose");
                return 1;
        }

        printf ("name=\"%s\" booty=%lu beard_len=%u\n",
                p.name, p.booty, p.beard_len);

        return 0;
}
```

The output is, of course, the original values:

```
name="Edward Teach" booty=950 beard_len=48
```

Again, it's important to bear in mind that because of differences in variable sizes, alignment, and so on, binary data written with one application may not be readable by other applications. That is, a different application—or even the same application on a different machine—may not be able to correctly read back the data written with `fwrite()`. In our example, consider the ramifications if the size of `unsigned long` changed, or if the amount of padding varied. These things are guaranteed to remain constant only on a particular machine type with a particular ABI.

Seeking a Stream

Often, it is useful to manipulate the current stream position. Perhaps the application is reading a complex record-based file and needs to jump around. Alternatively, perhaps the stream needs to be reset to position zero. Whatever the case, standard I/O provides a family of interfaces equivalent in functionality to the system call `lseek()` (discussed in Chapter 2). The `fseek()` function, the most common of the standard I/O seeking

interfaces, manipulates the file position of `stream` in accordance with `offset` and `whence`:

```
#include <stdio.h>

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

If `whence` is set to `SEEK_SET`, the file position is set to `offset`. If `whence` is set to `SEEK_CUR`, the file position is set to the current position plus `offset`. If `whence` is set to `SEEK_END`, the file position is set to the end of the file plus `offset`.

Upon successful completion, `fseek()` returns `0`, clears the `EOF` indicator, and undoes the effects (if any) of `ungetc()`. On error, it returns `-1`, and `errno` is set appropriately. The most common errors are invalid stream (`EBADF`) and invalid `whence` argument (`EINVAL`).

Alternatively, standard I/O provides `fsetpos()`:

```
#include <stdio.h>

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

This function sets the stream position of `stream` to `pos`. It works the same as `fseek()` with a `whence` argument of `SEEK_SET`. On success, it returns `0`. Otherwise, it returns `-1`, and `errno` is set as appropriate. This function (along with its counterpart `fgetpos()`, which we will cover shortly) is provided solely for other (non-Unix) platforms that have complex types representing the stream position. On those platforms, this function is the only way to set the stream position to an arbitrary value, as the C `long` type is presumably insufficient. Linux-specific applications need not use this interface, although they may, if they want to be portable to all possible platforms.

Standard I/O also provides `rewind()`, as a shortcut:

```
#include <stdio.h>

void rewind (FILE *stream);
```

The invocation:

```
rewind (stream);
```

resets the position back to the start of the stream. It is equivalent to:

```
fseek (stream, 0, SEEK_SET);
```

except that it also clears the error indicator.

Note that `rewind()` has no return value and thus cannot directly communicate error conditions. Callers wishing to ascertain the existence of an error should clear `errno` before invocation, and check to see whether the variable is nonzero afterward. For example:

```
errno = 0;
rewind (stream);
if (errno)
        /* error */
```

Obtaining the Current Stream Position

Unlike `lseek()`, `fseek()` does not return the updated position. A separate interface is provided for this purpose. The `ftell()` function returns the current stream position of `stream`:

```
#include <stdio.h>

long ftell (FILE *stream);
```

On error, it returns `-1` and `errno` is set appropriately.

Alternatively, standard I/O provides `fgetpos()`:

```
#include <stdioh.h>

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

Upon success, `fgetpos()` returns `0`, and places the current stream position of `stream` in `pos`. On failure, it returns `-1` and sets `errno` appropriately. Like `fsetpos()`, `fgetpos()` is provided solely for non-Unix platforms with complex file position types.

Flushing a Stream

The standard I/O library provides an interface for writing out the user buffer to the kernel, ensuring that all data written to a stream is flushed via `write()`. The `fflush()` function provides this functionality:

```
#include <stdio.h>

int fflush (FILE *stream);
```

On invocation, any unwritten data in the stream pointed to by `stream` is flushed to the kernel. If `stream` is `NULL`, *all* open input streams in the process are flushed. On success, `fflush()` returns `0`. On failure, it returns `EOF`, and `errno` is set appropriately.

To understand the effect of `fflush()`, you have to understand the difference between the buffer maintained by the C library and the kernel's own buffering. All of the calls described in this chapter work with a buffer that is maintained by the C library, which resides in user space, not kernel space. That is where the performance improvement comes in—you are staying in user space and therefore running user code, not issuing system calls. A system call is issued only when the disk or some other medium has to be accessed.

`fflush()` merely writes the user-buffered data out to the kernel buffer. The effect is the same as if user buffering was not employed and `write()` was used directly. It does not guarantee that the data is physically committed to any medium—for that need, use something like `fsync()` (see "Synchronized I/O" on page 40). For situations where you are concerned with ensuring that your data is commited to the backing store, you will want to call `fflush()`, followed immediately by `fsync()`: that is, first ensure that the user buffer is written out to the kernel and then ensure that the kernel's buffer is written out to disk.

Errors and End-of-File

Some of the standard I/O interfaces, such as `fread()`, communicate failures back to the caller poorly, as they provide no mechanism for differentiating between error and end-of-file. With these calls, and on other occasions, it can be useful to check the status of a given stream to determine whether it has encountered an error or reached the end of a file. Standard I/O provides two interfaces to this end. The function `ferror()` tests whether the error indicator is set on `stream`:

```
#include <stdio.h>

int ferror (FILE *stream);
```

The error indicator is set by other standard I/O interfaces in response to an error condition. The function returns a nonzero value if the indicator is set, and `0` otherwise.

The function `feof()` tests whether the `EOF` indicator is set on `stream`:

```
#include <stdio.h>

int feof (FILE *stream);
```

The `EOF` indicator is set by other standard I/O interfaces when the end of a file is reached. This function returns a nonzero value if the indicator is set, and `0` otherwise.

The `clearerr()` function clears the error and the `EOF` indicators for `stream`:

```
#include <stdio.h>

void clearerr (FILE *stream);
```

It has no return value, and cannot fail (there is no way to know whether an invalid stream was provided). You should make a call to `clearerr()` only after checking the error and `EOF` indicators, as they will be discarded irretrievably afterward. For example:

```
/* 'f' is a valid stream */

if (ferror (f))
        printf ("Error on f!\n");
```

```
if (feof (f))
        printf ("EOF on f!\n");

clearerr (f);
```

Obtaining the Associated File Descriptor

Sometimes it is advantageous to obtain the file descriptor backing a given stream. For example, it might be useful to perform a system call on a stream, via its file descriptor, when an associated standard I/O function does not exist. To obtain the file descriptor backing a stream, use `fileno()`:

```
#include <stdio.h>

int fileno (FILE *stream);
```

Upon success, `fileno()` returns the file descriptor associated with `stream`. On failure, it returns `-1`. This can only happen when the given stream is invalid, in which case, the function sets `errno` to `EBADF`.

Intermixing standard I/O calls with system calls is not normally advised. Programmers must exercise caution when using `fileno()` to ensure their file descriptor—based actions do not conflict with the user buffering. Particularly, a good practice is to flush the stream before manipulating the backing file descriptor. You should almost never intermix file descriptor and stream-based I/O operations.

Controlling the Buffering

Standard I/O implements three types of user buffering and provides developers with an interface for controlling the type and size of the buffer. The different types of user buffering serve different purposes and are ideal for different situations.

Unbuffered
: No user buffering is performed. Data is submitted directly to the kernel. As this disables user buffering, negating any benefit, this option is not commonly used, with a lone exception: standard error, by default, is unbuffered.

Line-buffered
: Buffering is performed on a per-line basis. With each newline character, the buffer is submitted to the kernel. Line buffering makes sense for streams being output to the screen, since messages printed to the screen are delimited with newlines. Consequently, this is the default buffering used for streams connected to terminals, such as standard out.

Block-buffered
: Buffering is performed on a per-block basis, where a block is a fixed number of bytes. This is the type of buffering discussed at the beginning of this chapter and it is ideal for files. By default, all streams associated with files are block-buffered. Standard I/O uses the term *full buffering* for block buffering.

Most of the time, the default buffering type is correct and optimal. However, standard I/O does provide an interface for controlling the type of buffering employed:

```
#include <stdio.h>

int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

The `setvbuf()` function sets the buffering type of `stream` to `mode`, which must be one of the following:

`_IONBF`
: Unbuffered

`_IOLBF`
: Line-buffered

`_IOFBF`
: Block-buffered

Except with `_IONBF`, in which case `buf` and `size` are ignored, `buf` may point to a buffer of `size` bytes that standard I/O will use as the buffer for the given stream. If `buf` is `NULL`, a buffer of the size you specify is allocated automatically by *glibc*.

The `setvbuf()` function must be called after opening the stream but before any other operations have been performed on it. It returns `0` on success, and a nonzero value otherwise.

The supplied buffer, if any, must exist when the stream is closed. A common mistake is to declare the buffer as an automatic variable in a scope that ends before the stream is closed. Particularly, be careful not to provide a buffer local to `main()` and then fail to explicitly close the streams. For example, the following is a bug:

```
#include <stdio.h>

int main (void)
{
        char buf[BUFSIZ];

        /* set stdin to block-buffered with a BUFSIZ buffer */
        setvbuf (stdout, buf, _IOFBF, BUFSIZ);
        printf ("Arrr!\n");

        return 0;
```

```
        /* 'buf' exits scope and is freed, but stdout isn't closed until later */
}
```

This class of bug can be prevented by explicitly closing the stream before exiting the scope or by making `buf` a global variable.

Generally, developers need not mess with the buffering on a stream. With the exception of standard error, terminals are line-buffered, and that makes sense. Files are block-buffered, and that, too, makes sense. The default buffer size for block buffering is `BUFSIZ`, defined in `<stdio.h>`, and it is usually an optimal choice (a large multiple of a typical block size).

Thread Safety

Threads are the units of execution within a process. Most processes have a single thread. Processes, however, can boast multiple threads, each executing their own code. We call such processes *multithreaded*. One way to conceptualize a multithreaded process is as multiple processes that share an address space. Without explicit coordination, threads can run at any time, interleaving in any way. On a multiprocessor system, two or more threads in the same process may even run concurrently. Threads can overwrite shared data unless care is taken to synchronize access to the data (a practice called *locking*) or make it *thread-local* (a practice called *thread confinement*).

Operating systems that support threads provide locking mechanisms (programming constructs that ensure mutual exclusion) to ensure that threads do not stomp on each other's feet. Standard I/O uses these mechanisms, ensuring that multiple threads in a single process may issue concurrent standard I/O calls—even against the same stream!—without the concurrent operations trampling on each other. Still, they are not always adequate. For example, sometimes you want to lock a group of calls, enlarging the *critical region* (the chunk of code that runs without interference from another thread) from one I/O operation to several. In other situations, you may want to eliminate locking altogether to improve efficiency.[1] In this section, we will discuss how to do both.

The standard I/O functions are inherently *thread-safe*. Internally, they associate a lock, a lock count, and an owning thread with each open stream. Any given thread must acquire the lock and become the owning thread before issuing any I/O requests. Two or more threads operating on the same stream cannot interleave standard I/O operations, and thus, within the context of single function calls, standard I/O operations are atomic.

1. Normally, eliminating locking will lead to an assortment of problems. But some programs might implement their thread safety strategy to delegate all I/O to a single thread, a form of thread confinement. In that case, there is no need for the overhead of locking.

Of course, in practice, many applications require greater atomicity than at the level of individual function calls. For example, imagine multiple threads in a single process issuing write requests. As the standard I/O functions are thread-safe, the individual writes will not interleave and result in garbled output. That is, even if two threads each issue a write request at the same time, locking will ensure that one write request completes before the other. But what if the process wants to issue several write requests in a row, all without the risk of another thread's write requests interleaving not just the individual request, but the set of requests as a whole? To allow for this, standard I/O provides a family of functions for individually manipulating the lock associated with a stream.

Manual File Locking

The function `flockfile()` waits until `stream` is no longer locked, bumps the lock count, and then acquires the lock, becoming the owning thread of the stream, and returns:

```
#include <stdio.h>

void flockfile (FILE *stream);
```

The function `funlockfile()` decrements the lock count associated with `stream`:

```
#include <stdio.h>

void funlockfile (FILE *stream);
```

If the lock count reaches zero, the current thread relinquishes ownership of the stream. Another thread is now able to acquire the lock.

These calls can nest. That is, a single thread can issue multiple `flockfile()` calls, and the stream will not unlock until the process issues a corresponding number of `funlockfile()` calls.

The `ftrylockfile()` function is a nonblocking version of `flockfile()`:

```
#include <stdio.h>

int ftrylockfile (FILE *stream);
```

If `stream` is currently locked, `ftrylockfile()` does nothing and immediately returns a nonzero value. If `stream` is not currently locked, it acquires the lock, bumps the lock count, becomes the owning thread of `stream`, and returns `0`.

Let's consider an example. Let's say we want to write out several lines to a file, ensuring that the lines are written out without interleaving write operations from other threads:

```
flockfile (stream);

fputs ("List of treasure:\n", stream);
fputs ("    (1) 500 gold coins\n", stream);
```

```
    fputs ("    (2) Wonderfully ornate dishware\n", stream);

    funlockfile (stream);
```

Although the individual `fputs()` operations could never race with other I/O—for example, we would never end up with anything interleaving with "List of treasure"—another standard I/O operation from another thread to this same stream could interleave between two `fputs()` calls. Ideally, an application is designed such that multiple threads are not submitting I/O to the same stream. If your application does need to do so, however, and you need an atomic region greater than a single function, `flockfile()` and friends can save the day.

Unlocked Stream Operations

There is a second reason for performing manual locking on streams. With the finer-grained and more precise control of locking that only the application programmer can provide, it might be possible to minimize the overhead of locking and to improve performance. To this end, Linux provides a family of functions, cousins to the usual standard I/O interfaces, that do not perform any locking whatsoever. They are, in effect, the unlocked counterparts to standard I/O:

```
#define _GNU_SOURCE

#include <stdio.h>

int fgetc_unlocked (FILE *stream);
char *fgets_unlocked (char *str, int size, FILE *stream);
size_t fread_unlocked (void *buf, size_t size, size_t nr,
                       FILE *stream);
int fputc_unlocked (int c, FILE *stream);
int fputs_unlocked (const char *str, FILE *stream);
size_t fwrite_unlocked (void *buf, size_t size, size_t nr,
                        FILE *stream);
int fflush_unlocked (FILE *stream);
int feof_unlocked (FILE *stream);
int ferror_unlocked (FILE *stream);
int fileno_unlocked (FILE *stream);
void clearerr_unlocked (FILE *stream);
```

These functions all behave identically to their locked cousins, except that they do not check for or acquire the lock associated with the given `stream`. If locking is required, it is the responsibility of the programmer to ensure that the lock is manually acquired and released.

Relegating I/O

There is a sizable performance win from using the unlocked standard I/O functions. Moreover, there's a nontrivial amount of code simplicity in not having to worry about locking complex operations with `flockfile()`. When designing your application, consider relegating all I/O to a single thread (or all I/O to a pool of threads, where each stream is mapped to exactly one thread in the pool).

Although POSIX does define some unlocked variants of the standard I/O functions, none of the above functions are defined by POSIX. They are all Linux-specific, although various other Unix systems support a subset.

We'll discuss threads thoroughly in Chapter 7.

Critiques of Standard I/O

As widely used as standard I/O is, some experts point to flaws in it. Some of the functions, such as `fgets()`, are occasionally inadequate and ill-designed. Other functions, such as `gets()`, are so horrendous that they have been all but evicted from the standards.

The biggest complaint with standard I/O is the performance impact from the double copy. When reading data, standard I/O issues a `read()` system call to the kernel, copying the data from the kernel to the standard I/O buffer. When an application then issues a read request via standard I/O using, say, `fgetc()`, the data is copied again, this time from the standard I/O buffer to the supplied buffer. Write requests work in the opposite fashion: the data is copied once from the supplied buffer to the standard I/O buffer and then later from the standard I/O buffer to the kernel via `write()`.

An alternative implementation could avoid the double copy by having each read request return a pointer into the standard I/O buffer. The data could then be read directly, inside of the standard I/O buffer, without ever needing an extraneous copy. In the event that the application did want the data in its own local buffer—perhaps to write to it—it could always perform the copy manually. This implementation would provide a "free" interface, allowing applications to signal when they are done with a given chunk of the read buffer.

Writes would be a bit more complicated, but the double copy could still be avoided. When issuing a write request, the implementation would record the pointer. Ultimately, when ready to flush the data to the kernel, the implementation could walk its list of stored pointers, writing out the data. This could be done using scatter-gather I/O, via `writev()`, and thus incur only a single system call. (We will discuss scatter-gather I/O in the next chapter.)

Highly optimal user-buffering libraries exist, solving the double copy problem with implementations similar to what we've just discussed. Alternatively, some developers choose to implement their own user-buffering solutions. But despite these alternatives, standard I/O remains popular.

Conclusion

Standard I/O is a user-buffering library provided as part of the standard C library. Modulo a few flaws, it is a powerful and very popular solution. Many C programmers, in fact, know nothing but standard I/O. Certainly, for terminal I/O, where line-based buffering is ideal, standard I/O is the only game in town. You don't often use `write()` to print to standard out!

Standard I/O—and user buffering in general, for that matter—makes sense when any of the following are true:

- You could conceivably issue many system calls, and you want to minimize the overhead by combining many calls into few.
- Performance is crucial, and you want to ensure that all I/O occurs in block-sized chunks on block-aligned boundaries.
- Your access patterns are character- or line-based, and you want interfaces to make such access easy without issuing extraneous system calls.
- You prefer a higher-level interface to the low-level Linux system calls.

The most flexibility, however, exists when you work directly with the Linux system calls. In the next chapter, we will look at advanced forms of I/O and the associated system calls.