Table of Contents

Chapter 1: File I/O Essentials 1

Regular file I/O with the library layer APIs 2

The stream 2

The stdio file I/O API set 3

Opening and closing a stream 3

The fopen API 4

Closing the deal 5

Other means to open a stream 5

Seeking upon a stream 5

Simple seek – a quick code example 8

Atomic appends 9

I/O buffering 11

The stdio APIs 14

Quick testing of stdio buffering with code 16

Flushing streams 22

Debugging via instrumentation 22

Changing the I/O buffering mode and size 25

Modifying the I/O buffer mode and size – code example 26

Steam redirection with the freopen API 30

The powerful popen API 31

Regular file I/O with the read and write system calls 34

Open at the system call layer 34

Flags to the open system call 35

The file mode 38

The file descriptor 41

Mapping streams to descriptors and vice versa 42

I/O – the read/write system calls 43

Read and write – the return value 44

Code example – a simple copy program 44

Correctly using the read/write APIs 49

Non-blocking writes and synchronization 52

Synchronous writes 54

Summary 55

Index 57

1
File I/O Essentials

Files provide a key feature for modern computers—persistence, which is the ability to store data safely so that one can power cycle the system and the data remains available (upon power-up). Without it, databases would not exist (and incidentally, this book wouldn't have been written on a computer). So, we get it: we require files.

In the Unix (and therefore Linux) design, the OS works with various types of files, such as the following:

- Regular files
- Directories
- Symbolic (or soft) links
- Other, more exotic ones, such as the following:
    - Named pipes (FIFOs), socket files, device files (block and character)

In this book, we will focus on some of the interfaces (APIs) provided by C library and the Linux OS, allowing us to operate upon the first of them—regular files. These are your normal ASCII text files, binary data files, special format files (a combination of ASCII and binary data, perhaps), and executable files (more technically, **Executable and Linker Format** (**ELF**) files). Working with the others at the API level is beyond the scope of this book.

In this chapter in particular, we will focus on covering what we feel are the essential aspects of file I/O (reading and writing regular files) that a competent developer should know; we leave some of the more advanced aspects to a later chapter. Here, the reader will broadly learn how to think about and correctly perform file I/O under the following:

- The more abstracted stream model—the C library stdio layer APIs
- The system-level with system calls

Important concepts regarding I/O buffering, redirection (with `freopen(3)` and `popen(3)`), understanding an open stream versus a file descriptor, mapping streams to descriptors and vice versa, and so on, will also be covered. We will close this chapter with an important note on synchronous versus asynchronous writes.

Regular file I/O with the library layer APIs

Let's start by looking at the interfaces provided at the layer above the system call layer—the C library.

The stream

First of all, we need to understand this: the library layer works with files by abstracting them into what it refers to as a **stream**. A stream is, as the English word implies, a sequence of characters. The stream has no implicit format and is worked upon via the library layer file I/O API set (which we will see shortly). The library (glibc) uses a data structure to represent a stream—this is the well-known `FILE` structure. We usually use a pointer to it: `FILE *stream`.

As mentioned in `Chapter 1`, *Linux System Architecture*, every process upon being born auto-inherits (from its parent) three open files (by the way, the abbreviation **fd** stands for **file descriptor** —it's an integer value the kernel maintains on a per process basis to identify an open file; we will cover a lot more on working with file descriptors later in this chapter. In particular chapter, there is a section called *The file descriptor* that will we cover later):

- **Standard input** (`stdin: fd 0`): The keyboard device, by default

- **Standard output** (`stdout: fd 1`): The monitor (or terminal) device, by default
- **Standard error** (`stderr: fd 2`): Again, the monitor (or terminal) device, by default

These files (cleverly abstracting the keyboard and monitor devices for us) are indeed streams in the Unix/Linux terminology. More correctly, we access these files via the three stream objects (of type `FILE *`) assigned to them—`stdin`, `stdout`, and `stderr`.

But how exactly can we use them, or for that matter, other files? Read on to find out.

The stdio file I/O API set

To do something with files, anything at all—read from them, write to them – we must first open the file. We can then perform **input** and **output**—we often simply abbreviate this as (**IO**) – operations upon it. The input implies reading from the underlying file object, while the output implies writing to the underlying file object.

The open operation can be performed via the `fopen(3)` API.

The 3 in parentheses implies that it's a library and not a system call API.

Then, I/O would be performed upon the file, and when done, the file is closed:
`fopen(3) → << perform IO >> → fclose(3)`

How exactly is the I/O performed; what are the available APIs? Well, there are plenty. Some that will be familiar to C programmers will be the `fwrite(3)/fread(3)` pair, the `fprintf(3)/fscanf(3)` pair, and the `fgetc(3)/fputc(3)` pair, among others.

Opening and closing a stream

Before any I/O can be performed on a file, the file is required to be opened; without opening it, nothing can happen. Here, we will discuss three APIs within the stdio API that are meant to open streams. The first one we will focus on is `fopen(3)`.

The fopen API

The `fopen(3)` library API is used to open a stream, as shown in the following code (of course, recall that if the stream in question is one of the standard ones—`stdin`, `stdout`, and `stderr`—they are already open and ready for use by default):

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *mode);
```

This is pretty straightforward: the first parameter is the `pathname` to the file to be opened, and the second parameter is called **mode** and implies what the file is being opened for. On success, a stream is set up and associated with the file; the return value is a pointer to this stream, abstracted as a `FILE` structure (on failure, `NULL` is returned and `errno` is set accordingly).

The mode parameter is a string; its possible values and meanings are summarized in the following table:

Mode	**Meaning**	**File seek position**
r	Open file for reading	Beginning of file
r+	Open file for reading and writing	Beginning of file
w	If the file exists, truncate (to zero bytes) and open for writing; if the file does not exist, create it for writing	Beginning of file
w+	If the file exists, truncate (to zero bytes) and open for reading and writing; if the file does not exist, create it (for reading and writing).	Beginning of file
a	Open the file for appending; if the file does not exist, create it	End of file
a+	Open the file for reading and appending; if the file does not exist, create it; seek position will initially show up as zero (BOF)	Beginning of file for reads; end of file for writes

More detail on this is available within the man page on *fopen(3)*: `https://linux.die.net/man/3/fopen`.

The seek position is, of course, an implicit pointer referring to the byte position within the file where the next piece of I/O will occur; the phrase beginning of file implies that the seek position is set to byte zero. The phrase **end-of-file** (**EOF**) implies that the seek position is set to the last byte (and a write will, in effect, append data to the stream). More on manipulating the seek position follows in the upcoming section.

Closing the deal

When I/O is done, it's very important that the stream (and thus the underlying file object) is closed; the `fclose(3)` library API achieves this:

```
    int fclose(FILE *stream);
```

It's important to realize that closing a file has the underlying OS guarantee that it will synchronize the file's data content, which will ensure that all buffers (more on this in the next section) are flushed.

The astute reader will recognize that, under the hood, the `fopen` and `fclose` library APIs will have the C library (glibc on Linux) invoke the `open(2)` and `close(2)` system calls, respectively.

The developer can explicitly flush, that is, force a write, of a given output stream via the `fflush(3)` API; more on this follows a bit later.

Other means to open a stream

The `fdopen(3)` API is a bit indirect; assuming that a file is already opened and one has the file descriptor to it (a file descriptor is essentially a lower, system level abstraction to a Unix/Linux file; we will cover it later on in this chapter), one can convert it to a stream abstraction via the `fdopen` API:

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

The mode parameter is identical to that of the `fopen(3)` API; the return value on success is a valid file pointer—the higher-level abstraction to a stream object. The seek or file position remains unaltered, but the mode must be compatible with the file descriptor's mode.

Seeking upon a stream

Every open file is associated with a file or seek position. It's quite an intuitive concept: typically, when a file is first opened, the file position is at zero—implying **Beginning Of File** (**BOF**). I/Os performed on the file—reads and writes—implicitly move the file (seek) position.

Don't confuse this abbreviation **Beginning Of File** (**BOF**) with another common abbreviation used in infosec, **Buffer overFlow** (**BoF**)

So, for example, a read of 120 bytes will set the seek position at 120; any subsequent file operation will proceed at this point. Of course, if a file is opened in append mode, the seek position will be at EOF. Interestingly, there are interfaces that are provided by the library to explicitly change (or query) the seek position.

The `fseek(3)` and `fgetpos(3)` APIs can be used to explicitly set (and query) the seek position of a stream.

Using the `fseek(3)` API is straightforward:

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

The first parameter is the *stream* to operate upon, and the second is the offset (in bytes). The third parameter, *whence*, can be given the following values:

- `SEEK_SET`: Sets the seek position of the stream *stream* to BOF + offset
- `SEEK_CURR`: Sets the seek position of the stream *stream* to the current position + offset
- `SEEK_END`: Sets the seek position of the stream *stream* to EOF + offset

The library provides a couple of convenience routines as well:

```
#include <stdio.h>
long ftell(FILE *stream);
void rewind(FILE *stream);
```

It's quite clear: `ftell(3)` returns the current seek position (as a long integer) of the given stream object, and the `rewind(3)` API does precisely as it says—it rewinds the given stream, that is, it sets its seek position to byte zero.

In addition, the library provides the `f[g|s]etpos(3)` APIs too; they are essentially equivalent to `ftell(3)` and `fseek(3)`, and only seem to be relevant on some non-POSIX systems; they will not be delved into here.

TIP

It's important to note that the ANSI C standard requires that the seek position be explicitly set between I/O operations being performed upon a stream; think about it, it's a reasonable ask, as the file's/stream's seek position will change implicitly as I/O is carried out upon it. Explicit seeks will help guarantee that I/O is correctly performed.

Simple seek – a quick code example

Here, we will write a quick and simple C program, (`A_fileio/streamio1.c`). The user must pass two parameters:

- A pathname to a regular file
- Either `b` or `e`:
    - `b`: Open the file with mode `r`
    - `e`: Open the file with mode `a`

We will encode the program to accordingly `fopen(3)` the given file, and then use the `ftell(3)` API to query the current seek position.

A couple of trial runs can be seen in the following code:

```
$ make streamio1
gcc -O2 -Wall -UDEBUG -c ../common.c -o common.o
gcc -O2 -Wall -UDEBUG -c streamio1.c -o streamio1.o
gcc -o streamio1 streamio1.o common.o
$ ./streamio1
Usage: ./streamio1 pathname {mode=b|e}
$ ./streamio1 streamio1.o b
Current seek position: 0
$ ./streamio1 streamio1.o e
Current seek position: 2936
$ ls -l streamio1.o
-rw-r--r-- 1 seawolf seawolf 2936 Jul 30 12:39 streamio1.o
$
```

It's quite clear: when run with the second parameter, `b`, the current seek position shows as zero; when the second parameter is `e`, the current seek position shows up as the number of bytes in the file. In effect, we will be appending data to the stream upon a write.

What if the file (pathname) passed into the first parameter does not exist?

```
$ ./streamio1 hey_im_not_there e
Current seek position: 0
$ ls -l hey_im_not_there
-rw-r--r-- 1 seawolf seawolf 0 Jul 30 12:43 hey_im_not_there
$ alias rm; rm hey_im_not_there
alias rm='rm -i'
rm: remove regular empty file 'hey_im_not_there'? y
$
```

Clearly, it gets created, as promised, when we use `fopen(3)` with mode `a` (which is what we do when `e` is passed). When we use `fopen` with mode `r`, no new file is created; the call just fails:

```
$ ./streamio1 hey_im_not_there b
FATAL:streamio1.c:main:43: fopen on hey_im_not_there failed
  kernel says: No such file or directory
$
```

Refer to the codes as follows:

```
[...]
int main(int argc, char **argv)
{
  FILE *fp=NULL;
  if (argc != 3) {
    fprintf(stderr, "Usage: %s pathname {mode=b|e}\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  if (!strncmp(argv[2], "b", 1))
    fp = fopen(argv[1], "r");
  else if (!strncmp(argv[2], "e", 1))
    fp = fopen(argv[1], "a");
[...]
  printf("Current seek position: %ld\n", ftell(fp));
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```

Atomic appends

Let's say we have a requirement wherein a process must open a file, and pretty much immediately, start appending data to it (a typical application log file, perhaps). This sounds easy to do; we might write the (pseudo) code as follows:

```
// Process 'A'
FILE *fp = fopen("thelog", "r+");
[...]
// About to append data
fseek(fp, SEEK_END, 0L);                      <--- time t1
fprintf(fp, "%s:%u:%s\n", [...]);             <--- time t2
```

At first glance, the preceding code snippet might appear to be just fine; the `fseek(3)` API – setting the seek position to EOF – is done at *time t1*, immediately followed by the write (via `fprintf(3)`) at *time t2*. Think carefully, though: always remember that we are running on the Linux OS – it's a multiprocess, multithreaded, heavy duty environment! It is entirely possible that between performing `fseek` (done at *t1*) and `fprintf` (done at *t2*) on process A, the kernel scheduler schedules another process, B, (or thread), to run on the CPU. What if this process, B, also appends data to the same file? Then it's entirely conceivable that by the time process A is back on the CPU and running `fprintf`, the file's size has changed underneath it. Now, the seek position set by `SEEK_END` is no longer valid, and our process, A, will perform a write to somewhere within the file, corrupting it.

This is a good example of a classic *race condition*; as we cannot predict in advance how the OS will allocate CPU bandwidth and perform scheduling, we have to code carefully. How, you ask? Check this out in the following:

```
FILE *fp = fopen("thelog", "a+");
[...]
// About to append data
fprintf(fp, "%s:%u:%s\n", [...]);            <--- time t1
```

This time, we open the stream (corresponding to our log file) in append mode; thus, there is no need for an explicit—and possibly racy—`fseek` followed by an `fprintf`; it's just the `fprintf` which, as we know, will append data to the file. In effect, we have now done the right thing—we have made the code atomic with respect to the data append operation (look out for a lot more on this key concept in the chapters on multithreading).

The preceding *race condition* will make you think: how can two (or more) processes be allowed to simultaneously operate upon the same file object? Should we not synchronize this? Yes, indeed: this is precisely what *file locking* is all about. We do not cover file locking in this book; we refer the reader to the *Further reading* section for a link to an excellent tutorial on the subject..

Before we can use the actual APIs for I/O (input/output for reading/writing), at least a minimal understanding of buffering (and caching) concepts is required; read on.

I/O buffering

With the library, or more correctly, the `stdio` layer APIs, being so vast, the following question must be asked: is there a decent way to categorize them? Yes—as it turns out, one can quite neatly classify them based on their buffering type.

What does buffering mean in this context, exactly? For runtime performance optimization, the library layer will often make use of a memory cache—a memory buffer between the library and, ultimately, the underlying storage (disk or flash) device.

Caches work based on two essential principles: one, the principle of locality of reference— data (and instructions) that are adjacent to each other will more likely be used in the near future, both in space (spatial locality) and in time (temporal locality). Two, a cache is a high-speed memory buffer of some sort – a key point is that accessing the cache memory is much faster than accessing the underlying memory device that it's caching (for example, caching disk data in RAM is beneficial because RAM is much faster compared to disk).

Precisely because the principle of locality of reference turns out to be true most of the time (think about how a file is often read sequentially and not randomly; audio/video are excellent examples), caching recently used memory in a fast-access buffer yields tremendous performance benefits.

This caching principle is used again and again in modern computers—from CPUs caching data/instructions in L1/L2/L3 CPU caches to RAM, to firmware caching, to software caching (both at the OS level as well as at the library/application level).

So, the fact is, when a process calls the, say, `fwrite(3)` API, to write data to a file stream (or object), the write does not go immediately and directly to the file on the physical storage device; no, it is cached in a buffer maintained by the library layer! This actually results in performance enhancement. Think about it – if a process wrote a few bytes at a time, it would be very high overhead to immediately flush a few bytes at a time, every time, down through all the complex layers to the underlying storage device. (Recall from `Chapter 2`, *Virtual Memory*, that the average difference in access speeds between RAM and a SCSI disk is in the region of a million times!) Instead, cache it for a while, ideally until the cache memory is full, and then flush the entire buffer in one shot to the underlying layers.

Let's say we have an application that reads in some data (from the user via `stdin`, another file, or a network socket, a pipe, and so on) into a heap buffer (accessed via the `buf` pointer) of size, say, 512 bytes. The application process then intends to write this buffer to a regular file; the file, or stream, is opened, and we have a `FILE *fp` pointer available. So, the application now issues the I/O – the write – via one of the many available APIs; it can use any one of the `stdio` output APIs, that is, `fwrite(3)`, `fprintf(3)`, or `fputc(3)`, or, it can directly talk to the kernel via the system call `write(2)`. Whichever it chooses, we can visualize that the buffer is somehow passed to the OS (see the following diagram). The OS is, of course, a complex technology; here, we won't pretend to delve into the low-level details. Any file-related system call will end up in a kernel layer called the **Virtual Filesystem Switch** (**VFS**). From here, via a myriad of sophisticated layers within the kernel – which we will just treat as a big black box for now—the OS will ultimately end up passing the data buffer, `buf`, to the block or storage device driver, whose job is to actually write it to the underlying device (typically, a disk or flash memory medium):

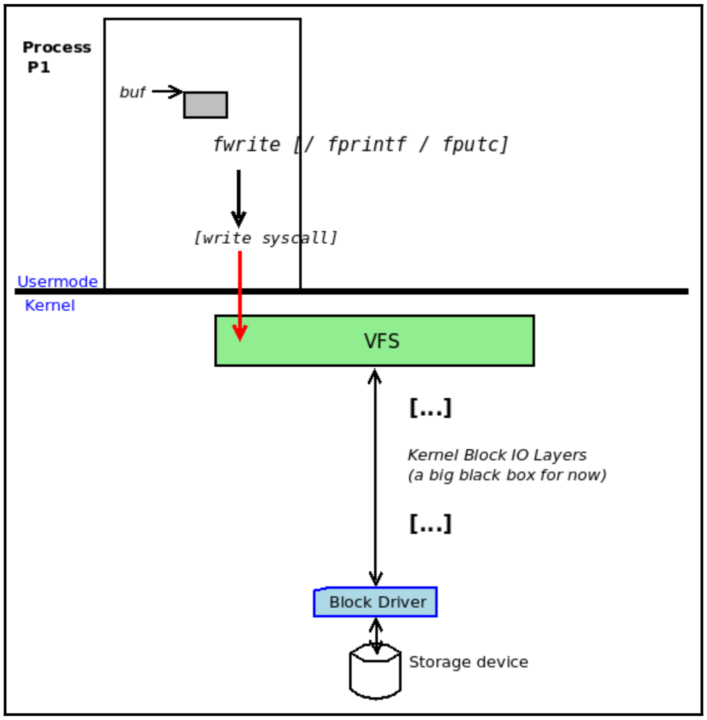


Figure 1: fwrite from an application to disk via the write syscall—a simplistic view

Well, this really is the simplistic view. A more realistic view, deeper under the hood, would reveal a lot more (wait for `Chapter 18`, *Advanced File I/O,* for that); for one, the `stdio` layer APIs will result in the process buffer, `buf`, being first cached in a buffer allocated by the C library upon first use – what we have been referring to as the I/O buffer. So now, it should be more clear: referring to the preceding table (in the *The fopen API* section), if the application calls `fwrite(3)` (let's just assume it does so), the `stdio` I/O buffer will be one page in size (4,096 bytes). `fwrite(3)` would just cause the process buffer (`buf`) of 512 bytes to be written into the `stdio` buffer; nothing more! The following diagram shows us this:

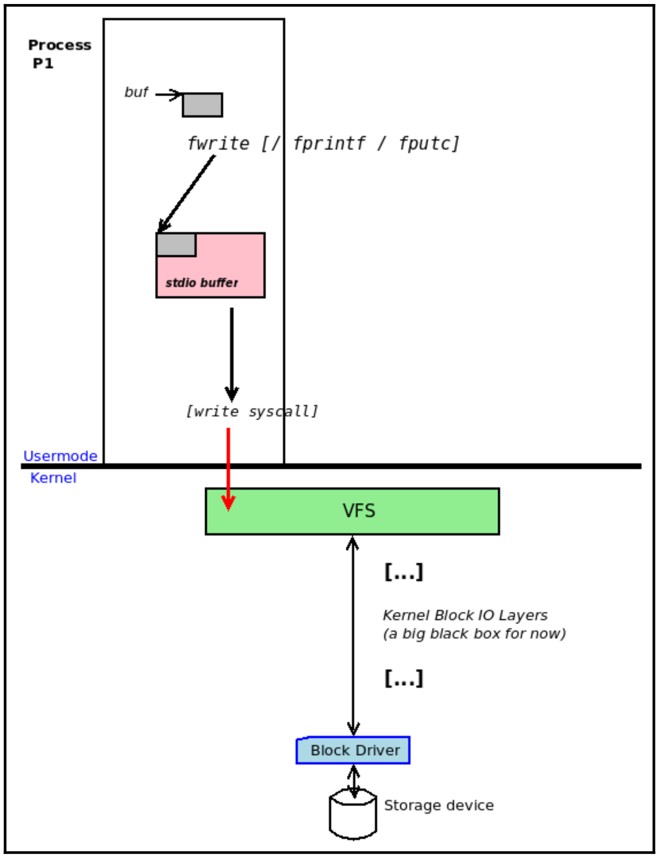


Figure 2: fwrite from the application, via the stdio buffering mechanism—a more realistic view

The reader should realize that once the process buffer, `buf`, is written into the stdio I/O buffer (in the light grey color, and of size 1 page), `fwrite(3)` will return success. As far as it's concerned, the job is done! In reality, the data is still in user space in the stdio I/O buffer, as we can see; subsequently, it will get flushed to the kernel and thus the storage device (read the *Flushing streams* section for more on this).

Another point: the preceding diagrams are really with respect to the `write` path; what happens when reading in data (from storage device to the user space buffer)? Here, stdio I/O buffering is in effect, only this time, the OS uses intelligence to request the underlying layers for sufficient data to fill the I/O buffer in one shot, if possible. Say that the application requests 512 bytes from a (regular) file via `fread(3)`; the OS has the intelligence to issue one `read(2)` system call for one page of data. Thus, the I/O buffer is filled and subsequent uses of `fread(3)` do not require a system call to service them (the data is already in the user space I/O buffer)! This is often called a *read-ahead* strategy; for files that are typically read sequentially (this quite often is the case), it works wonders performance-wise. In fact, there are APIs available for the developer precisely to pass such hints to the OS regarding how to work with data, such as the `posix_madvise(3)` library and the system call `madvise(2)`.

The stdio APIs

As the title of this section implies, the following I/O APIs are all within the stdio C library framework. In fact, as C programmers learn on day one, we must `#include <stdio.h>` in order to work with them.

The memory size that the C library uses for a memory cache determines the buffering type; the following table helps us understand the different buffering types that the C library layer routinely uses:

Buffering type	**Size of buffer (cache)**	**API set: foo(3)**
Fully (or block) buffered I/O	1 page (4,096 bytes)	`fread/fwrite`
Line buffered I/O	128 bytes	`[v][f|s]scanf, [f]gets[1]/[f]puts, [v][f|d|s|sn]printf` 2
Raw (or unbuffered) IO	0	`read[v]/write[v]`

[1] Please do not use the `gets(3)` API for reasons of security; it's a very well-known and highly exploited API for malicious attackers (usually via the (stack) BoF vector). Using `fgets(3)` is acceptable, though the `getline(3)` API is considered the preferred API for getting a line of input from the user.

[2] What does the [v][f|d|s|sn]printf notation mean?

The [v][f|d|s|sn]printf notation is meant to imply that it refers to any of the following `*printf(3)` APIs:

- fprintf
- dprintf
- sprintf
- snprintf
- vfprintf
- vdprintf
- vsprintf
- vsnprintf

Similarly, [v][f|s]scanf implies that it refers to any of the `*scanf(3)` APIs, that is, [v]scanf, [v]fscanf, and [v]sscanf. Whew.

Interestingly, GNU extensions include the asprintf(3) and vasprintf(3) APIs; they are the same as their more familiar [v]sprintf(3) cousins, except that they will always internally allocate a buffer large enough to contain the output (critical from the lens of secure coding!). The most important thing to mention is that the caller is responsible for invoking free(3) when done (recall our discussion on *Memory leakage bugs* in earlier chapters).

A detailed explanation of each of these APIs is not something we would prefer to perform in this book; we refer the reader to browse through the man pages on them for more information.

Okay, back to the stdio APIs and their buffering. Practically, this is how they're set up to work on the system by default (so that maximum performance is achieved whenever applicable):

- If the stream refers to a regular file (a file on the filesystem), I/O performed on the stream will be fully (block) buffered (a stdio I/O buffer of size 1 page will cache it).
- If the stream refers to a terminal device – a common example being stdout – I/O performed on the stream will be line buffered (a stdio I/O buffer of size 128 bytes will cache it).

- If the stream is stderr, I/O performed on the stream will always be unbuffered (it will not be cached at all at the library layer and writes will proceed immediately via write(2) to the OS layers). This is usually desirable for error output.

Quick testing of stdio buffering with code

Let's get hands-on once again (after all, that's an important objective of this tome: don't assume, try it out) and write a small C program to clearly illustrate how, when using the `fread(3)`/`fwrite(3)` stdio library APIs, the library will automatically buffer the data.

In our program (the source code is here within the book's GitHub repository: `A_fileio/iobuf.c`), we will perform I/O (reading and writing) as follows:

1. First, open/create files as required:
    - We `fopen(3)` a stream (in read mode: r) to our source: an interesting character device file – `/dev/urandom`. Without getting into the details, just understand this: it's part of a memory device driver supplied by the Linux OS; reading one byte from the file retrieves one byte of data – a (pseudo) random value!

There are other ways to get random data from the kernel; one is to read from the (pseudo) device file`/dev/random`; this a good way for applications requiring high security (it's very slow but secure). Another way is to use the (pseudo) device `/dev/urandom`; in fact, the kernel documentation for it states that it's a faster, less secure random number gen. This is fine for us, though, here and now. Also, both `/dev/[u]random` are essentially infinite – they have no EOF.

2. Next, we `fopen(3)` a stream (in append mode: a) to our destination: just a regular file in the `/tmp` directory (it will be created if it doesn't exist).
3. Then, in a loop, for a number of iterations provided by the user as a parameter, we will do the following:
    - Read several (512) bytes from the source stream (they will be random values) via the fread(3) stdio library API
    - Write those values to our destination stream via the fwrite(3) stdio library API (checking for EOF or error conditions)

3. When done, we clean up (`fclose` our streams, free buffers, and so on).

The code (`A_fileio/iobuf.c`) performing these steps is as follows:

```
int main(int argc, char **argv)
{
  char *fname="/tmp/iobuf_tmp";
  FILE *wfp = NULL, *rfp = NULL;
  int numio;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s number-of-times-to-rdwr\n", argv[0]);
    exit(EXIT_FAILURE);
  }

[...]

  unlink(fname);
  wfp = fopen(fname, "a");
  if (!wfp)
    FATAL("fopen on %s failed\n", fname);

  rfp = fopen("/dev/urandom", "r");
  if (!rfp)
    FATAL("fopen on /dev/urandom failed\n");
  testit(wfp, rfp, numio);
  fclose(rfp);
  fclose(wfp);
[...]
}
```

(By the way, we use `unlink(2)` to delete the temporary file, if it exists). The actual I/O is performed in the `testit()` function, as follows:

```
static void testit(FILE * wrstrm, FILE * rdstrm, int numio)
{
  int i, syscalls = NREAD*numio/getpagesize();
  size_t fnr=0;

  if (syscalls <= 0)
      syscalls = 1;
  VPRINT("numio=%d     total rdwr=%u     expected # rw syscalls=%d\n",
                  numio, NREAD*numio, NREAD*numio/getpagesize());

  for (i = 0; i < numio; i++) {
    fnr = fread(gbuf, 1, NREAD, rdstrm);
    if (!fnr)
      FATAL("fread on /dev/urandom failed\n");
    if (!fwrite(gbuf, 1, fnr, wrstrm)) {
      free(gbuf);
      if (feof(wrstrm))
        return;
      if (ferror(wrstrm))
        FATAL("fwrite on our file failed\n");
    }
  }
}
```

Let's try it out:

```
$ ./iobuf
Usage: ./iobuf number-of-times-to-rdwr
$ ./iobuf 10000
./iobuf: using default stdio IO RW buffers of size 4096 bytes; #
IOs=10000
 iobuf.c:testit:37: numio=10000 total rdwr=5120000 expected # rw
syscalls=1250
$
```

Okay, it appears to work; we have it perform I/O (`fread/fwrite` pairs) 10,000 times, reading and writing 512 bytes each time. Thus, the output file is as follows:

```
$ ls -l /tmp/iobuf_tmp
-rw-rw-r--. 1 kai kai 5120000 Aug 1 13:41 /tmp/iobuf_tmp
$
```

But think a moment—recall what we've just learned and in particular, take a quick look again at our earlier diagrams. The first diagram illustrates how we intuitively think it works; we feel that the reads and writes are immediate and that the OS services the requests straight away. But no, the following diagram shows us a more realistic picture: I/O buffering ensures that whenever we perform I/O (an `fread(3)` or an `fwrite(3)`), the corresponding low-level system calls `read(2)` and `write(2)` do not always get invoked immediately, but only when required – when the I/O buffer is full! This is good for performance (frequent system calls are a major performance dampener).

All right, but can we prove this? Yes, indeed; we shall use a powerful tracing utility available on Linux, `ltrace(1)`.

You can use `ltrace(1)` to trace all library layer APIs (as well as all system calls by using the `-S` option switch) issued by a process at runtime.
You can use the `strace(1)` utility to trace all system calls issued by a process at runtime. Do check out their man pages; there are lots of interesting options to be explored.

Okay; here we go:

```
$ ltrace -S ./iobuf 10000
brk@SYS(nil) = 0x6d7000
access@SYS("/etc/ld.so.preload", 04) = 0
openat@SYS(AT_FDCWD, "/etc/ld.so.preload", 0x80000, 00) = 3
fstat@SYS(3, 0x7ffcb77e7360) = 0
close@SYS(3) = 0

[...]
```

First (as shown in the preceding code), we will see the C runtime environment setup (the text, data, shared libraries loading, and so on); we will ignore this. As shown in the following snippet, note that the code has started running; `unlink(2)` and `fopen(3)` have been issued:

```
[...]
printf("%s: using default stdio IO RW bu"..., "./iobuf", 4096, 10000
<unfinished ...>
fstat@SYS(1, 0x7ffcb77e6e70) = 0
brk@SYS(nil) = 0x6d7000
brk@SYS(0x6f8000) = 0x6f8000
brk@SYS(nil) = 0x6f8000
write@SYS(1, "./iobuf: using default stdio IO "..., 75) = 75
<... printf resumed> ) = 75
malloc(512) = 0x6d7670
unlink("/tmp/iobuf_tmp" <unfinished ...>
unlink@SYS("/tmp/iobuf_tmp") = 0
<... unlink resumed> ) = 0
fopen("/tmp/iobuf_tmp", "a" <unfinished ...>
openat@SYS(AT_FDCWD, "/dev/urandom", 0, 00) = 4
lseek@SYS(3, 0, 2) = 0
<... fopen resumed> ) = 0x6d7880                    <-- FILE * wfp
fopen("/dev/urandom", "r" <unfinished ...>
openat@SYS(AT_FDCWD, "/dev/urandom", 0, 00) = 4
<... fopen resumed> ) = 0x6d7ab0                    <-- FILE * rfp

[...]
```

The APIs in the ltrace output suffixed by `@SYS` indicates that it's a system call (and not a library layer API):

```
[...]
fread(0x6d7670, 1, 512, 0x6d7ab0 <unfinished ...>
fstat@SYS(4, 0x7ffcb77e74b0) = 0
ioctl@SYS(4, 0x5401, 0x7ffcb77e7410, 0) = -22
read@SYS(4,
"\vO\356\277\354\313\001\245\267\312EV\006e\343\025\315\372\235\315\\\
3359\313|\320\237q\365f\212\312"..., 4096) = 4096
<... fread resumed> ) = 512
[...]
```

The I/O loop (in our `testit()` routine) is really the key portion. As we can see in the preceding code, it's very interesting that we can literally see how an fread(3) of 512 bytes translates into a read(2) system call of size 4,096 bytes! We have literally seen the read-ahead strategy that we mentioned previously at work here!

In the following code, we can see the loop in execution (we are using newline-separated pairs of `fread/fwrite` for readability):

```
[...]
fread(0x6d7670, 1, 512, 0x6d7ab0) = 512
fwrite("\326\203.\201\312\253Qu=\337\020\342\273\375\201\302\220\002\f
\343\300\361l\260\361\350\340\235$\220n4"..., 1, 512, 0x6d7880) = 512

fread(0x6d7670, 1, 512, 0x6d7ab0) = 512
fwrite("\335\002;\372\313\336\3256q\316\201\241\237\3560\256\025\231\3
21S\\5\373\v\326\303\362'\b\031\254\323"..., 1, 512, 0x6d7880) = 512

fread(0x6d7670, 1, 512, 0x6d7ab0) = 512
fwrite("\266\343\200\025K1o\312'\301\234\020\310\336E\236l\025\361\222
\b\363\200\3340\366YMW\311\347\035"..., 1, 512, 0x6d7880) = 512

fread(0x6d7670, 1, 512, 0x6d7ab0) = 512
fwrite("n\213wd\016\2335f\223\304\240\231-
\374\354U\256\031\3725\376\323\240\240\244e\362\032H\347\314\373"...,
1, 512, 0x6d7880) = 512
[...]
```

However, by far the most interesting analysis is this: let's grep (and count with wc -l) the ltrace output for occurrences of the key I/O APIs – for us, it's the fread(3), read(2), the fwrite(3), and the write(2). In fact, we have rigged up a simple shell script (`A_fileio/trc_iobuf.sh`) to help automate the process. We will run it as follows:

```
$ ./trc_iobuf.sh
------------------------------
ltrace -S ./iobuf 10000 2>ltrc_vanilla.txt
------------------------------
./iobuf: using default stdio IO RW buffers of size 4096 bytes; #
IOs=10000
 iobuf.c:testit:37: numio=10000      total rdwr=5120000     expected #
rw syscalls=1250
------------ ltrace stats -----------------
# calls to fread    :   10000
# calls to read     :    1251
# calls to fwrite   :   10000
# calls to write    :    1252

[...]
```

For now, we've truncated the script's output to just the first part (more to follow). The really interesting thing is that in the last four lines of output, we can clearly see that (for this sample, run with 10,000 I/O loop iterations) the number of fread(3) APIs issues is 10,000, but the number of corresponding read(2) system calls is a fraction of that: 1,251. Similarly, the number of fwrite(3) API issues is 10,000, but the number of corresponding write(2) system calls is 1,252.

So, with the default stdio I/O buffering mode set to fully buffered I/O, the I/O buffer is one page in size: in this sample run, we have read and written 512 bytes (of random data) into a loop 10,000 times. So, that's `512*10,000 = 5,120,000` bytes read and written in total (a quick `ls -l /tmp/iobuf_tmp` will verify this). However, we know that the library layer will optimize and use a page-sized I/O buffer, buffering the I/O for performance, so the expected number of read/write system calls will be `(512*10,000)/4,096 = 1250`. The output reveals that it's almost identical (some read/write would have occurred for other things as well). So, there we have it: stdio I/O buffering in action.

Flushing streams

As mentioned earlier, a programmer can always explicitly flush, that is, force a write of an output stream, via the `fflush(3)` API: `int fflush(FILE *stream);` Flushing output streams (ones opened for writing) is intuitive given the way I/O buffering works, but what about input (opened for reading) streams? Well, it is possible: the man page on fflush(3) states that if one is working upon the input stream of a regular file (not a pipe or terminal device), the fflush will discard all fetched and buffered data (that was residing in the buffer or cache) that the process has not yet read (or consumed). Thus, fflush(stdin) has no real meaning, as stdin is (usually) a terminal device.

Calling fflush(NULL) has the interesting effect of flushing all currently open output streams. Check – always check – for the failure case; fflush(3) returns zero on success and EOF on failure (and `errno` is accordingly set).

Debugging via instrumentation

In this section, we do use the fork(2) system call, something that we shall cover in great detail in `Chapter 10`, *Process Creation*. If unfamiliar with the fork, you could skip this section for now and come back to it later.

In the early part of the `Chapter 10`, *Process Creation*, (where the usage of the powerful `fork(2)` system call will be explained in great detail), we will see the following trivial piece of code, with one minor difference: we replace the string to be printed, `Hello, fork.\n`, with `Hello, world`:

```
int main(int argc, char **argv)
{
    fork();
    printf("Hello, world. ");
}
```

(For this simple use case, we don't show error checking code; production-quality apps must do so.) The output is quite obvious: after the fork, there are two processes now alive and executing the next line of code – the printf(3); the original parent and the just-born child process. Thus, the printf output appears twice on the console (on the terminal device, stdout). Now, we will build and run it:

```
$ make fork_printf_1
gcc -O2 -Wall -UDEBUG -c ../common.c -o common.o
gcc -O2 -Wall -UDEBUG -c fork_printf_1.c -o fork_printf_1.o
gcc -o fork_printf_1 fork_printf_1.o common.o
$ ./fork_printf_1
Hello, world. Hello, world. $
```

We have modified the preceding code to this:

```
int main(int argc, char **argv)
{
    printf("Hello, world. ");
    fork();
}
```

A quick quiz: what's the output? Is it:

- <no output>
- Hello, world
- Hello, world. Hello, world.
- Really, I have no idea

Please, dear reader, do try it out before reading on; it's quite interesting!

Okay, we've asked. Now, let's run it and see:

```
$ ./fork_printf_2
Hello, world. Hello, world. $
```

It's the third option how come!?
The answer is interesting: from the preceding table regarding I/O buffering types, we can see that the `printf(3)` API is line buffered, and that the size of the buffer is 128 bytes (by default). This implies that when a process calls `printf(3)`, the output does not go directly to the terminal device (`stdout`) via the OS, but, rather, is buffered in a stdio buffer of 128 bytes in size (this is completely analogous to how, in *Figure 2*, the fwrite data is buffered in the page-size stdio buffer). So, the string `Hello, world.` resides in the library allocated buffer. Next, we call fork(2), which of course creates a new child process – essentially a copy of the parent. The new child's **virtual address space** (**VAS**) includes the stdio library buffer with the same string embedded inside it. After that, both processes terminate when they hit the close brace `}` of main. But, remember—before a process terminates, the kernel ensures that all open files are flushed and then closed. Thus, both the parent's and child's stdio buffers are flushed and then the open files are closed. The flush causes the string `Hello, world.` to be written twice to the terminal device (stdout). Also recall that on fork, our *Fork Rule # 6*: Open files are (loosely) shared across the fork" includes stdout. The parent and child thus share stdin, stdout, and stderr.

So, [f|v|s|sn]printf is line-buffered, but guess what: we could always explicitly flush the buffer (as we saw previously, with fflush(stdout)). Yes, that would work, but there's an easier way: the library also flushes a stream's buffer when the newline character is encountered in the stream! So, just terminate your printf's with a `\n` and it will be flushed. Rewriting the preceding printf to the following:

```
printf("Hello, world.\n");
fork();
```

will cause the message to appear only once. Do confirm this for yourself; empirically testing the results (don't just believe everything "the book" says) is a key best practice! (Importantly, `Chapter 19`, *Troubleshooting and Best Practices*, covers such points.)

A practical case where this knowledge is useful is, well, in any printf(3). Often, though, as I have first-hand seen, a fledgling developer struggles with code they're debugging by introducing printf's – the common and
useful instrumentation technique. But, if the printf is not newline terminated and there is a fork at a later point in the code path, it's quite possible that they will see the output twice, confusing them as to what is even going on now!

Changing the I/O buffering mode and size

The C library provides several APIs, allowing a developer to change the I/O buffering mode and size. These are `setbuf(3)`, `setbuffer(3)`, `setlinebuf(3)`, and `setvbuf(3)`. Internally, the first three all wrap around to the key one, the `setvbuf(3)` API. Its signature is as follows:

```
include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The third parameter, `mode`, specifies the I/O buffering mode, and should be one of the following:

- `_IOFBF`: Fully buffered I/O
- `_IOLBF`: Line buffered I/O
- `_IONBF`: Unbuffered I/O

The second parameter, `buf`, is a pointer to a buffer of size bytes. If used, it will be used as the I/O buffer for the stream. Alternatively, one can leave `buf` as NULL; in this case, the library will allocate a buffer upon first use itself.

Note: One must call `setvbuf(3)` on a stream only after opening it and before any operation is performed on it.

Calling `setvbuf(3)` on a standard stream (`stdin`, `stdout`, or `stderr`) may not yield the desired outcome as these are usually, and by default, pointers to an implementation-dependent terminal window; `setvbuf` only changes the `glibc` stdio buffer type and size, nothing else.

In the opinion of many, given the level of optimization performed by the Linux kernel (both at the block I/O layers and driver), as well as at the firmware layer (modern disks have built-in caches), the `setvbuf` APIs are not particularly fantastic; we suggest that you do not spend too much time attempting to micro-optimize via this route. Nevertheless, it might turn out useful to optimize a particular application's file I/O performance.

Modifying the I/O buffer mode and size – code example

Here, we will present a quick code example on using the `setvbuf(3)` API to modify the size of the standard I/O buffer. Recall that the default size of the fully buffered mode I/O buffer is a page (4,096 bytes; again, we remind you: do not assume anything – use the `getpagesize(2)` API to query the page size).

We will make a copy of our earlier code (the `A_fileio/iobuf.c` program) to `A_fileio/io_setbuf.c`. The majority of the code remains the same; the change, though, is that we use our very own I/O buffers. How, exactly? It's straightforward:

- We allocate two buffers (via the usual `malloc(3)`) for our own read and write I/O buffers; the size of the buffers is specified via the first parameter to this program.
- We issue the `setvbuf(3)` API twice:
    - Once, to set up an I/O buffer for the read stream
    - Again, to set up an I/O buffer for the write stream
- We call our I/O loop code routine—`testit()`—as usual.
- When all I/O is done, we print kernel-maintained statistics about the process I/O (by dumping the pseudofile `/proc/<PID>/io`).

We can always generate the delta between the two programs using the `diff(1)` utility. As you can see in the following code, we are showing a relevant snippet of the resultant patch:

```
$ diff -u iobuf.c io_setbuf.c

[...]

+ // Allocate our own 'write' stdio IO buffer
+ iow_buf = malloc(bufsz);
+ if (!iow_buf)
+     FATAL("malloc %zu failed!\n", bufsz);
+
+ // Allocate our own 'read' stdio IO buffer
+ ior_buf = malloc(bufsz);
+ if (!ior_buf)
+     FATAL("malloc %zu failed!\n", bufsz);

   gbuf = malloc(NREAD);
   if (!gbuf)
@@ -84,9 +112,18 @@
   if (!rfp)
         FATAL("fopen on /dev/urandom failed\n");
+ /* Change stdio buf: to fully buffered IO, and, importantly, with
+ * our specified buffer size 'bufsz'.
+ */
+ if (setvbuf(rfp, ior_buf, _IOFBF, bufsz))
+     FATAL("setvbuf on read buffer failed\n");
+ if (setvbuf(wfp, iow_buf, _IOFBF, bufsz))
+     FATAL("setvbuf on write buffer failed");
[...]
```

The payoff can be seen in the following code snippet—we run our `trc_iobuf.sh` wrapper script (which, recall, internally calls `ltrace` and then greps and counts the relevant APIs). We then run three test cases.

With the default stdio I/O buffer mode set to fully buffered and 4,096 bytes in size (in fact, we saw this case earlier).

1. With our own spec—a large I/O buffer: stdio I/O buffer mode set to fully buffered and 512,000 bytes in size

2. With our own spec—a tiny I/O buffer: stdio I/O buffer mode set to fully buffered and 128 bytes in size
3. The output (focus on the `"# calls to ..."` lines as follows) shows up as expected:

Case 1: Default stdio I/O buffer mode set to fully buffered and 4,096 bytes in size:

```
$ ./trc_iobuf.sh
------------------------------
ltrace -S ./iobuf 10000 2>ltrc_vanilla.txt
------------------------------
./iobuf: using default stdio IO RW buffers of size 4096 bytes; #
IOs=10000
 iobuf.c:testit:37: numio=10000     total rdwr=5120000       expected
# rw syscalls=1250
------------ ltrace stats -----------------
# calls to fread  :  10000
# calls to read   :   1251
# calls to fwrite :  10000
# calls to write  :   1252
```

Case 2: Our own spec—a large I/O buffer: stdio I/O buffer mode set to fully buffered and 512,000 bytes in size:

```
------------------------------
ltrace -S ./io_setbuf 512000 10000 2>ltrc_setvbuf_large.txt
------------------------------
./io_setbuf: setting our IO RW buffers to size 512000 bytes; #
IOs=10000
 io_setbuf.c:testit:41: numio=10000    total rdwr=5120000     expected
# rw syscalls=10
rchar: 5121948
wchar: 5120158
syscr: 16
syscw: 12
read_bytes: 0
write_bytes: 0
cancelled_write_bytes: 0
------------ ltrace stats -----------------
# calls to fread  :  10000
# calls to read   :     11
# calls to fwrite :  10000
# calls to write  :     12
```

Look at that! In spite of 10,000 each for the `fread(3)` and `fwrite(3)` calls, only `11` `read(2)` and `12 write(2)` system calls had to be issued under the hood, as the I/O buffers were very large. Performance goes up.

We can also make performance go down.

Case 3: Our own spec—a tiny I/O buffer: stdio I/O buffer mode set to fully buffered and 128 bytes in size:

```
------------------------------
ltrace -S ./io_setbuf 128 10000 2>ltrc_setvbuf_small.txt
------------------------------
./io_setbuf: setting our IO RW buffers to size 128 bytes; # IOs=10000
 io_setbuf.c:testit:41: numio=10000     total rdwr=5120000
expected # rw syscalls=10000
rchar: 5121948
wchar: 5120158
syscr: 10006
syscw: 20001
read_bytes: 0
write_bytes: 0
cancelled_write_bytes: 0
------------ ltrace stats -----------------
# calls to fread  :  10000
# calls to read   :  10001
# calls to fwrite :  10000
# calls to write  :  20001
$
```

This time, with the really tiny (128 byte) I/O buffer in place, performance actually degrades with over 10,000 calls to read(2) and over 20,000 calls to write(2). As usual, the reader is encouraged to clone this book's GitHub repository and try out these programs for themselves.

The `stdbuf(1)` utility is worth a quick mention: it's a quick fix to some buffering issues faced by annoying default behavior on the default streams. Take a look at its man page for an example.

Steam redirection with the freopen API

A common application requirement is logging. We wish to keep an audit of application activity—the typical way to do so is to append all application messages into a persistent file—a log file.

For the reader's information, there are several projects that implement logging; *syslog (3)* is a well-known traditional system logging facility on Unix/Linux. Modern logging is performed on Linux with *systemd-journald(8)* (with *journalctl(1)* being a useful frontend utility to read the journal). Discussing these in depth goes beyond the scope of this book; readers are advised to look up the man pages if interested. In this chapter, for demonstration purposes, we will show how to perform very simplistic logging—without any framework—just directly with the library stdio APIs and a few helpers.

So, how do we get our application process to send all the messages it usually writes to the standard streams—stdout and stderr—to a log file (here, we do not demo the same with stdin)? The freopen(3) API works well for this very purpose:

```
FILE *freopen(const char *pathname, const char *mode, FILE *stream);
```

The pathname argument is the file we would like to replace the open stream with; the mode is as described in the *The fopen API* section of this chapter. The original open stream (if it exists) is closed. More detail can be found in the man page on freopen(3), though looking at the following code should suffice for understanding.

Let's use freopen(3) to our advantage: we will write a small C program that attempts to redirect stdout and stderr to a log file we specify as the parameter to it. The code is as follows (`A_fileio/redirct.c`):

```
int main(int argc, char **argv)
{
  FILE *fplog=NULL, *pfp=NULL, *fp=NULL;
#define DTMAX 256
  char dt[DTMAX];
  char *junk="abcxyz";

  if (argc != 2) {
          fprintf(stderr, "Usage: %s logfile\n", argv[0]);
          exit(EXIT_FAILURE);
  }

  /* Effectively redirect the stdout stream to our log file stream */
```

```
    fplog = freopen(argv[1], "a", stdout);
    if (!fplog)
        FATAL("freopen stdout on %s failed\n", argv[1]);

    /* Effectively redirect the stderr stream to our log file stream */
    fplog = freopen(argv[1], "a", stderr);
    if (!fplog)
        FATAL("freopen stderr on %s failed\n", argv[1]);
  [...]
```

In the preceding code, note how we have quite easily and effectively redirected `stdout` and `stderr` to the file `argv[1]`. In the following, we continue with a slight but required deviation - the `popen(3)` API.

The powerful popen API

Okay, now we want to log a message. Just performing a simple `printf(3)` will do the trick – it will get appended to the log file! However, we would also like to prefix a human-readable `date-timestamp` to all our log messages; we want a colon `:` delimited log format like this: `program-name:date-timestamp:app-message...`

There are several ways to obtain the timestamp; an interesting and powerful way is to actually use a pipe mechanism—the popen(3) library API. While we do not attempt to delve into the gory details here, the essential working is as follows:

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

`popen(3)` is very powerful; it allows you to perform two things. Either you can run any command that writes to stdout and gain access to its output, or run any command that reads from stdin and gain access to its input.

How, exactly? The first parameter to `popen(3)` is any command you wish to run; the second parameter type can be `r` or `w`:

- `r`: Run the command *command;* the return value `FILE *` is a stream that, when read, will supply the `stdout` output of the command process.

- w: Run the command *command;* the return value FILE * is a stream that, when written to, will write to the stdin of the command process.

For example, to get the output of date into the application, we can use the following:
FILE * fp = popen("date", "r");

Now, reading from the stream, fp will get you the standard output (stdout) of the date process! We exploit exactly this in our program. When done, one must pclose(3) the stream opened via popen. (Many graphical applications use popen(3) to fetch the output of a process (say, ps -ef) and display the same in a scrollable/selectable window.)

(A link to both the popen man page as well as an article on systemd basics is provided in the *Further reading* section on the GitHub repository.)

To continue with our simple redirct.c app code, use the following:

```
[...]
/* Get human-readable timestamp via the popen(3) */
  if (!(pfp = popen("date", "r")))
        FATAL("popen (date) failed\n");
  if (!fgets(dt, DTMAX, pfp)) {
        WARN("fgets (date) failed\n");
        strncpy(dt, "---", 4);
  }
  dt[strlen(dt)-1]='\0';     // rm terminating newline
  pclose(pfp);
[...]
```

A small detail to note: the fgets(3) API, as we understand, reads a single line from the stream provided as its third parameter. It has a peculiarity that we must handle—it appends the newline character into the output stream as well, and so we must get rid of it (see the preceding code):

```
[...]
  /* 1. Write to stdout */
  printf("%s:%s:current seek position: %ld\n", argv[0], dt,
ftell(fplog));

  /* 2. Deliberately cause an error, thus writing to stderr */
  if (!(fp = fopen(junk, "r"))) {
        fprintf(stderr, "%s:%s:fopen on %s failed\n", argv[0], dt,
junk);
        perror(" fopen");
        fclose(fplog);
```

```
            exit(EXIT_SUCCESS);
        }

        fclose(fp);
        fclose(fplog);
        exit(EXIT_SUCCESS);
    }
```

Let's try it out; we will assume that, to begin with, the log file does not exist:

```
$ ./redirct log
$ ls -l log; cat log
-rw-rw-r--. 1 kai kai 160 Aug 7 08:59 log
./redirct:Tue Aug 7 08:59:31 IST 2018:fopen on abcxyz failed
 fopen: No such file or directory
./redirct:Tue Aug 7 08:59:31 IST 2018:current seek position: 0
$ ./redirct log
$ ls -l log; cat log
-rw-rw-r--. 1 kai kai 322 Aug 7 09:00 log
./redirct:Tue Aug 7 08:59:31 IST 2018:fopen on abcxyz failed
 fopen: No such file or directory
./redirct:Tue Aug 7 08:59:31 IST 2018:current seek position: 0
./redirct:Tue Aug 7 09:00:02 IST 2018:fopen on abcxyz failed
 fopen: No such file or directory
./redirct:Tue Aug 7 09:00:02 IST 2018:current seek position: 160
$
```

We run the preceding code twice; as you can see, the output from the second run has been appended (recall that we used `a` as the mode argument to freopen) to the first run's output.

Also, did you notice that although our app first wrote to `stdout` and subsequently to `stderr`, the output seems to be reversed! Why? That—and modifying the preceding program to have the `printf(3)` output appear first in the expected order—is left as an exercise to you, dear reader!

Security note: popen(3) is powerful, but can be easily abused. Imagine an app that asks the user for a command and executes it via popen(3) (with the `r` parameter) so that it can now display the output to the user graphically (or otherwise). This sounds good, but it is very dangerous – what if the end user supplies a not-so-innocent command (`rm -rf *`)? Whoops. The moral, of course, is: do not trust user input; validate it.
(More in a similar vein will be covered in `Chapter 19`, *Troubleshooting and Best Practices*.)
Also note that, popen(3), while powerful, is considered a high-overhead API – it internally creates a pipe object, forks, and execs (well, on modern Linux, it uses clone(2)).

Regular file I/O with the read and write system calls

Until now in this chapter, we have dealt exclusively with file I/O using the higher abstraction layer APIs from the C library (the stdio set). Now, we will go below the hood and learn how to use the APIs invoked by the abstraction layer: the system calls that perform file-based operations, including read/write (I/O). Now, we leave behind the stream abstraction, and work at the lower system layer.

Open at the system call layer

As usual, in order to work upon a file, one must first open it; the system call API to open a file is called—surprise, surprise!—`open(2)`. Its signature is as follows:

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

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

The preceding two signatures imply that one can call `open(2)` with either two or three parameters; in other words, the third parameter is optional. Let's check them out in order: the first parameter is obvious – the `pathname` to the file we would like to open (and subsequently perform I/O upon).

The pathname could refer to something other than a regular file (device files and named pipes (FIFOs) being fairly common examples). This serves to remind us of what we learned in `Chapter 1`, *Linux System Architecture*—"On Unix, if it's not a process, it's a file"!

Flags to the open system call

The second parameter is flags; it's an integer bitmask that communicates quite a lot of detail to the underlying kernel regarding the open semantics and subsequent operation upon the file. Broadly speaking, there are three types of flags: access mode flags, file creation flags, and file status flags. File creation flags affect the open semantics, whereas file status flags affect the operations performed subsequently upon the opened file.

Each type has several possible values; the programmer is allowed to specify zero or more flags by bitwise-ORing them together. An exhaustive list of all possible flags (of the three types mentioned previously) is not required here (if you need to know, or are just curious, we refer you to the in-depth man page on open(2): `https://linux.die.net/man/2/open`); instead, we will summarize the commonly used flags in the following table:

Flag Type	**Flag**	**Meaning**
Access mode	`O_RDONLY`	Open the file in read-only mode.
	`O_WRONLY`	Open the file in write-only mode.
	`O_RDWR`	Open the file in read-write mode.
File creation	`O_CLOEXEC`	Ensure that, upon a possible future execve (or variant), this file is closed. This is important security-wise and in many multithreaded apps (to prevent races).

(Note that all flag values not shown here can be found in man 2 open)	`O_CREAT`	If pathname does not exist, create it (as a regular file); the file's owner and group ID is predicated on the effective user (EUID) and effective group (EGID) ID of the creating process (or thread). (Also, if the set group id bit on the parent directory is set, then the group id of the newly created file would be set to that of the parent directory.) You must specify the mode as well (see the section entitled *The file mode* for details).
	`O_DIRECTORY`	Unless pathname refers to a directory, open(2) will fail (with errno <- ENOTDIR; though the man page specifies a bug here—if `O_CREAT` is specified as well and pathname does not exist, open will create a regular file).
	`O_EXCL`	Useful to perform an exclusive open; if pathname does not exist and `O_CREAT` is specified, create the file; if the file already exists, fail open(2) with errno set to EEXIST.
	`O_TMPFILE`	Creates an unnamed (thus effectively invisible!) temporary file in the directory specified by pathname. If pathname includes a filename, the file is visible and its content remains; if not, the file content is automatically lost when the file is (fully) closed. This is a useful semantic for temporary files. (Available from Linux 3.11 onward, and requires kernel-level filesystem support.) You must specify the mode as well (see the section entitled *The file mode* for details).
	`O_TRUNC`	For an existing and writable regular file, specifying this flag will have it truncated to zero bytes; careful.

File status	`O_APPEND`	Open the file in append mode; this implies that on every write to it, the seek operation to the (current) EOF and the subsequent write will be performed atomically (a caveat: this leads to a race on NFS filesystems which do not support appending to a file).
(Note that all flag values not shown here can be found in man 2 open)	`O_DIRECT`	Bypass the kernel page cache and perform I/O directly to lower kernel layers. It's a (possibly) performance-enhancing feature, but applications have to use it with caution (more on this in a later chapter).
	`O_NONBLOCK` or `O_NDELAY`	When specified, both the open as well as subsequent operations upon the file (such as reading or writing) will not cause the calling process (or thread) to wait: it will be non-blocking. (However, this behavior is actually not guaranteed – the underlying implementation does cause at least some brief waiting.) This is useful for an underlying driver to recognize the semantics required by user space when the device supports non-blocking IO semantics
	`O_SYNC`	This ensures that *writes* to the underlying storage device will be done prior to returning; in other words, it's equivalent to the pseudocode: `[write(); fsync(fd);]`.

The file mode

The third parameter to open(2), the mode, is required only when a file is being created, that is, when the flags (second parameter) include either the `O_CREAT` or the `O_TMPFILE` flags. If neither of these flags is being employed, just don't specify the mode; if you do, it's silently ignored.

The mode specifies the permission bitmask of the file being created (the naming convention is that mode implies permissions; think of the utility/syscall to change permissions— `chmod` that's change mode).

What if the `O_CREAT` flag is specified, but no mode is? Well, let's follow the empirical principle and try it out (`A_fileio/mode_def.c`):

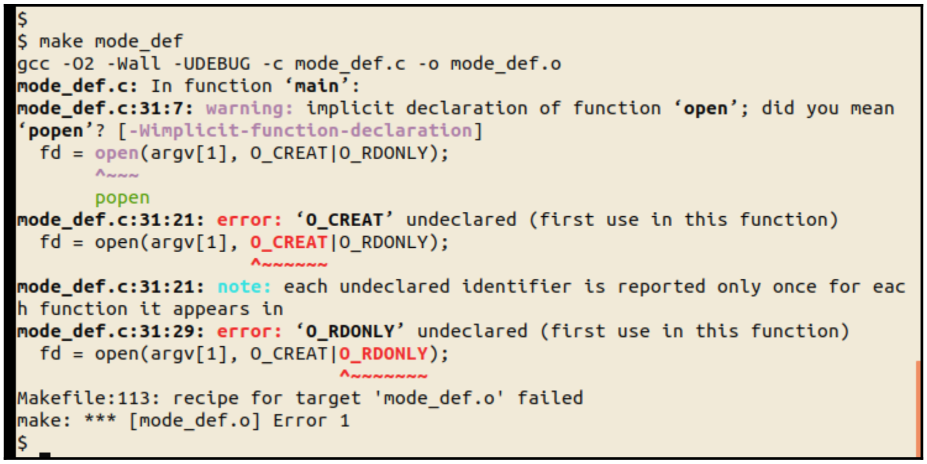
```
[...]
int main(int argc, char **argv)
{
  int fd;
  if (argc != 2) {
    fprintf(stderr, "Usage: %s new-file-pathname\n", argv[0]);
    exit(EXIT_FAILURE);
  }

#if 1
  fd = open(argv[1], O_CREAT|O_RDONLY);
#endif
  if (fd < 0)
    FATAL("open on %s failed\n", argv[1]);

  printf("Current seek position: %ld\n", lseek(fd, 0, SEEK_CUR));
  close(fd);
  exit(EXIT_SUCCESS);
}
```

A quick sidenote.

Sometimes, in the heat of coding, tiny (silly) mistakes can lead the (newbie) developer down the proverbial garden path, that is, astray. For example, with the previous code, we attempt to build it; this is the compiler's output (the following screenshot was taken on an Ubuntu 18.04 LTS desktop Linux system):



```
$
$ make mode_def
gcc -O2 -Wall -UDEBUG -c mode_def.c -o mode_def.o
mode_def.c: In function ‘main’:
mode_def.c:31:7: warning: implicit declaration of function ‘open’; did you mean
‘popen’? [-Wimplicit-function-declaration]
  fd = open(argv[1], O_CREAT|O_RDONLY);
       ^~~~
       popen
mode_def.c:31:21: error: ‘O_CREAT’ undeclared (first use in this function)
  fd = open(argv[1], O_CREAT|O_RDONLY);
                     ^~~~~~~
mode_def.c:31:21: note: each undeclared identifier is reported only once for eac
h function it appears in
mode_def.c:31:29: error: ‘O_RDONLY’ undeclared (first use in this function)
  fd = open(argv[1], O_CREAT|O_RDONLY);
                             ^~~~~~~~
Makefile:113: recipe for target 'mode_def.o' failed
make: *** [mode_def.o] Error 1
$ _
```

Quite confusing, right? Look carefully, though—there are several clues: the warning line **warning: implicit declaration of function ‘open’; did you mean ‘popen’? [-Wimplicit-function-declaration]** is really all we need here. The implicit declaration of function is the giveaway – the compiler requires the prototype of every C function to be present in the source code; it cannot find it for the `open(2)` function. Why not? Because it's in one of the header files for `open(2)`, which we have failed to #include! (In fact, it's really the same situation with the next warning regarding `O_RDONLY`.) So, just look up the man page for `open(2)`; it will clearly indicate all required headers: include them and the warnings will disappear.

Okay, so now we will fix the headers issue:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "../common.h"
[...]
```

(The headers highlighted in bold in the preceding code were the missing ones.)

On our Fedora 28 box, this code compiles just fine. On our Ubuntu 18.04 test system, however, we get an interesting output:

```
$ make mode_def
gcc -O2 -Wall -UDEBUG -c mode_def.c -o mode_def.o
In file included from /usr/include/fcntl.h:290:0,
 from mode_def.c:22:
In function 'open',
 inlined from 'main' at mode_def.c:34:5:
/usr/include/x86_64-linux-gnu/bits/fcntl2.h:50:4: error: call to
'__open_missing_mode' declared with attribute error: open with O_CREAT
or O_TMPFILE in second argument needs 3 arguments
 __open_missing_mode ();
 ^~~~~~~~~~~~~~~~~~~~~~
Makefile:113: recipe for target 'mode_def.o' failed
make: *** [mode_def.o] Error 1
$
```

Cool, the Ubuntu compiler has found a bug! (Modern compilers are getting better and better at this.) As we are using the `O_CREAT` flag, we require the mode argument to be passed. Running without any fix (on our Fedora box) yields weird and random values for the file mode (the permission bitmask):

```
$ ./mode_def
Usage: ./mode_def new-file-pathname
$ ./mode_def a1; ls -l a1
Current seek position: 0
-rw---S--T. 1 kai kai 0 Aug 2 17:55 a1
$ ./mode_def a2; ls -l a2
Current seek position: 0
-rwS-w---T. 1 kai kai 0 Aug 2 17:55 a2
$
```

Here's the fix; include the third argument to `open(2)`, that is, the mode. Also, resetting the default file permission bitmask value to zero – via the `umask(2)` system call – is recommended (the `umask(2)` system call (along with the `pause(2)`), is another peculiar case where it always returns success):

```
[...]
#if 0                              // the buggy way
    fd = open(argv[1], O_CREAT|O_RDONLY);
#else                              // the right way
    umask(0);   // reset the permission bitmask
    fd = open(argv[1], O_CREAT,
S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH);
#endif
```

We will now perform a trial run:

```
$ ./mode_def a1; ls -l a1
Current seek position: 0
-rw-rw-r--. 1 kai kai 0 Aug 2 18:02 a1
$ ./mode_def a2; ls -l a2
Current seek position: 0
-rw-rw-r--. 1 kai kai 0 Aug 2 18:02 a2
$ ./mode_def a2
Current seek position: 0
$
```

All is well!

Yes, but what does
the mode argument, `S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH`, mean? That's the symbolic notation to specify the file mode. It's easy; take the first symbol, `S_IRUSR`: it sets the Read bit for the user access category (we know that in the Unix permission model there are three access categories—three parts to the mode—user, group, others). In general, `S_I[R|W|X][USR|GRP|OTH]` is the syntactic form. As a convenience, we have `S_IRWXU` implying RWX set for the user (this mode in octal representation will be `0700`). Generically, it's `S_IRWX[U|G|O]`.

The file descriptor

The return value from open(2) is a non-negative integer on success, and, as usual for system calls on Linux, `-1` on failure (along with errno being set appropriately to indicate the cause of failure).

The integer returned upon success (value >= 0) is called the file descriptor. We think of it as a handle to the open file. The file descriptor values are per-process and not system-wide values (every process will have its own set of file descriptors). For example, process A might have five files open, with file descriptors numbered from 0 to 4. Process B might have seven open files with file descriptors from 0 to 6, and so on.

In reality, the file descriptor is nothing but an index into a kernel data structure that, in the classic Unix terminology, is called the **open file descriptor table** (**OFDT**)—essentially, it's an array of pointers to open files for a given process (maintained by the OS as part of the process metadata).

In fact, the file descriptors are nothing really new to us; we are familiar with the standard streams— stdin, stdout, and stderr (right from `Chapter 1`, *Linux System Architecture*). We understand that these streams represent the default input, output, and error files—by default, they point to the devices – keyboard, monitor, and respectively. So, guess what? A stream is a higher-level (library layer) abstraction of the open file, as maintained by the OS; its equivalent at the system call layer is the file descriptor. Hence, for every process alive, three open files are minimally inherited from its parent, which are as follows:

- `stdin`; represented by file descriptor `0`; use the macro `STDIN_FILENO`
- `stdout`; represented by file descriptor `1`; use the macro `STDOUT_FILENO`
- `stdin`; represented by file descriptor `2`; use the macro `STDERR_FILENO`

Every open sets up a unique OFDT entry within the OS and returns the index to it to the user space process—the file descriptor. We know that a process will have three files open the moment it comes alive; the file descriptors will be 0, 1, and 2. So, the next open will yield the lowest available file descriptor slot in the process OFDT; 3, in this case. Verify this in the `mode_def` program.

Every file-related system call besides open(2) will require (typically, as the first parameter) the file descriptor to be passed to it.

Mapping streams to descriptors and vice versa

Since streams are the higher-level abstractions of files, it makes sense that at the lower system level, the `FILE *` pointer, will translate to a file descriptor. This is exactly the case; in fact, given a file stream pointer, one can obtain its corresponding file descriptor! This is easily achieved via the fileno(3) API:

```
int fileno(FILE *stream);
```

The reverse semantic—given a file descriptor, associate a corresponding stream pointer with it – can be obtained via the fdopen(3) API: `FILE *fdopen(int fd, const char *mode);`

(Please refer the *fdopen(3)* on *man page* for more details)

`dprintf(3)` is another interesting example; it's identical to *printf(3),* except that it writes into a given file descriptor (well, so does printf(3) – it's just that it always writes to the file descriptor representing stdout—`STDOUT_FILENO` value 1):

```
int dprintf(int fd, const char *format, ...);
```

A word of caution: the preceding APIs give you a feeling that intermixing stream and low-level (system call) I/O is easy to do; it is, but, problems are often encountered. The recommendation is to always avoid mixing stream (library layer) and descriptor-level (system call) I/Os.

It's important to realize – once again – that file descriptors are returned and used by several other APIs—the signalfd, select, [e]poll, mmap, pipe, socket-based APIs—devices (often via the ioctl system call), and several other POSIX IPC technologies (message queues, shared memory, and semaphores). This, of course, is because (as we learned in `Chapter 1`, *Linux System Architecture*) in the Unix paradigm, files are the universal model by which (much of) the world is represented.

I/O – the read/write system calls

The actual transfer of data—from and to a file (or any object abstracted as a file)—is achieved via I/O – the read(2) and write(2) system calls. To perform the I/O, we require the file to be open; in other words, we require a file descriptor—ultimately a handle to the underlying open file object. We obtain the file descriptor to the file by performing open(2) on it (or indirectly, perhaps via the fileno(3) API).

The read(2) and write(2) system call signatures are as follows:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

The read(2) and write(2) system calls are quite similar; the first parameter to both – fd – is the file descriptor, identifying, in effect, which file the OS (the underlying filesystem or device driver) will operate upon.

The second parameter is a pointer to the memory buffer: in the case of read(2), it's a pointer to the destination buffer, buf—the place in user space memory where data from the file will be placed. In the case of write(2), it's a pointer to the user space memory source buffer—the buffer whose data content will be written to the file. (In early versions, the pointers were of type `char *`; now, its type is a generic `void *` pointer, allowing any data type (including binary data) to be read or written.) Notice how, in write(2), the buffer buf is marked as `const`—it is the programmer's job to treat this buffer as read-only and just transfer its content to the file without modifying it in any manner.

The third parameter to both APIs is the number of bytes to be read or written (note its data type, `size_t`; obviously, this is a typedef within the C library, which boils down to an unsigned long integer value).

Read and write – the return value

First, note the return data type—it's once again a typedef – it's a `ssize_t`; checking it will reveal that this is nothing but a signed long integer value.

The return value of both the read(2) and write(2) system calls is very important:

- On failure, `-1` is returned and errno is set appropriately; we once again emphasize—it's a key lesson—always check for the failure case!
- On success, the return value is the number of bytes of data actually read or written.
- There exists a special case for read(2): zero being returned implies that we have hit EOF; the programmer is expected to check for this. (It's also possible that the programmer deliberately asks read(2) to read 0 bytes for us; this is a valid operation, used for detecting errors; the man page on read(2) describes all possible errors that might occur.)

The first and third cases are straightforward; it's actually the second case that has us quite often create defects (bugs); we shall explain this in detail shortly in the *Correctly using the read/write* APIs section.

Code example – a simple copy program

To give the reader a feel for these important I/O APIs, we will construct and try out a very simplistic program whose job is to copy one file to another – in other words, a very simplistic **copy** (**cp**) program of our very own. Let's call it *simpcp1—simple copy ver 1*.

Here's how it should work:

```
simpcp1 <source-file> <destination-file> [1 for verbose-mode]
```

To keep the code easily readable and to just focus on getting across some key points within the domain of regular file I/O, we will deliberately make a number of simplifying assumptions:

- Both the source and destination files are regular files (not directories, device files, pipes, sockets, symbolic links, and so on)
- No fancy options—no copy to directory, recursive copying, following symbolic links, and so on
- If a destination file exists, just overwrite it
- No signal handling
- No performance optimizations (well, a little bit)

In `A_fileio/simpcp1.c`: `main()`, we set up argument processing, the file opens (or creation), invoke the actual worker routine (`docopy()`), close the files, and exit:

```
int main(int argc, char **argv)
{
  int fdr, fdw, ret=0;
  if (argc < 3) {
    fprintf(stderr, "Usage: %s source-file destination-file"
                    " [1 for verbose-mode]\n",
                argv[0]);
    exit(EXIT_FAILURE);
  }
  if (argc == 4 && (!strcmp(argv[3], "1")))
    gVerbose = 1;

  /* Open source as read-only */
  fdr = open(argv[1], O_RDONLY);
  if (fdr < 0)
    FATAL("open on source file \"%s\" failed, aborting.\n", argv[1]);

  /* Create / overwrite destination as write-only for append;
mode=0644 */
  fdw = open(argv[2], O_CREAT|O_WRONLY,
      S_IWUSR|S_IRUSR|S_IRGRP|S_IROTH);
  if (fdw < 0)
    FATAL("open on destination file \"%s\" failed, aborting.\n",
argv[2]);

  if ((ret = docopy(argv[1], fdr, fdw)) < 0) {
    close(fdr);
    close(fdw);
```

```
        FATAL("docopy() failed: (stat=%d)\n", ret);
      }
      if (gVerbose == 1)
        printf("\n");
      close(fdr);
      close(fdw);
      [...]
```

The preceding comments make this code self-explanatory. `docopy()` is where the real action happens; this will be shown in the code that follows. Our strategy is that we will read from the source file into a local (heap) buffer and write that buffer's content into the destination file, in an infinite loop: the stopping condition is when the read returns `0`, implying EOF.

That's fine, but how large should we make the local heap buffer? Ah, that's a good question, as performance will be impacted: use too small a buffer (say, a few bytes in size), and you will have to loop too many times, where issuing too many system calls will slow things down. (A quick example: say the source file is 128,000 bytes in size and we use a heap buffer of size 128 bytes; it will work, but we will have to perform read/write in a loop 128,000/128 = 1,000 times!) On the other hand, trying to allocate too large a buffer (say, a gigabyte in size) is unrealistic on tiny 32-bit embedded Linux boxes, but might actually be just fine on a server class system. So, how do you decide what to do? A tip: the inode of every (regular) file contains a hint—a member called `st_blksize`, which is specified as the block size for filesystem I/O. This is usually considered to be a good enough optimization – we will use this value. In fact, it's typically a page (4,096 bytes). Much more on IO performance optimization will follow in a later chapter.

To query the inode details of the source file, we issue the useful `stat(2)` system call (also, recall that our `VPRINT` macro will only actually output a message when the `gVerbose` global has the value `1`; we control this via the third parameter – the default being `0`):

```
static int docopy(char *src, int fd_src, int fd_dest)
{
  char *buf=NULL;
  struct stat sb;
  size_t n=0;
  ssize_t nr=0, nw=0;
  int loop=1;

  memset(&sb, 0, sizeof(struct stat));
  if (stat(src, &sb) < 0) {
    WARN("stat on %s failed\n", src);
```

```
        return -1;
      }
      VPRINT("%s: size=%ld Block size for filesystem I/O = %ld bytes\n",
          src, sb.st_size, sb.st_blksize);
      n = sb.st_blksize; /* 'Optimal' block size for filesystem I/O */

      buf = malloc(sb.st_blksize);
      if (!buf) {
        WARN("malloc (%ld) failed!\n", sb.st_blksize);
        return -2;
      }
      VPRINT("Expected # loop iterations: %ld [+1 for rem %ld bytes]\n\n",
          sb.st_size/n, sb.st_size%n);
```

Thus, we know how many times we will have to loop around—see the last line of the preceding code. Now, we come to the key part of this function—the loop that performs the actual I/O:

```
      while (1) {
        nr = read(fd_src, buf, n);
        VPRINT("%06d: read %9ld ", loop, nr);
        if (nr < 0) {
          WARN("read on src failed\n");
          free(buf);
          return -3;
        } else if (nr == 0)
          break;

        nw = write(fd_dest, buf, nr);
        VPRINT("wrote %9ld\n", nw);
        if (nw < 0) {
          WARN("write on dest failed\n");
          free(buf);
          return -4;
        }
        loop ++;
      }
      free(buf);
      return 0;
    }
```

To run it, we write a simple wrapper script, `A_fileio/test_simpcp.sh`. Here it is:

```
#!/bin/bash
name=$(basename $0)
SRC=srcfile
DEST=destfile
```

```
[ $# -ne 2 ] && {
  echo "Usage: ${name} PUT(program-under-test) src-filesize-KB
 Eg. ${name} ./simpcp2 21"
  exit 1
}

PUT=$1
[ ! -f ${PUT} ] && {
  echo "${name}: program-under-test \"${PUT}\" not built or missing?"
  exit 1
}
SRC_SIZE_BLK_KB=$2        # srcfile size will be this many kilobytes
rm -f ${SRC} ${DEST}

# Create the source file
dd if=/dev/urandom of=${SRC} bs=1k count=${SRC_SIZE_BLK_KB}
2>/dev/null

echo "${PUT} ${SRC} ${DEST} 1"
${PUT} ${SRC} ${DEST} 1

echo
ls -l ${SRC} ${DEST}
echo "Diff-ing them now ..."
diff ${SRC} ${DEST}
echo "Done."
exit 0
```

Let's run it (in verbose mode by passing the third parameter to simpcp1 as `1`), asking for a source file (and thus destination file) of 23 KB in size:

```
$ ./test_simpcp.sh
Usage: test_simpcp.sh PUT(program-under-test) src-filesize-KB
 Eg. test_simpcp.sh ./simpcp2 21
$ ./test_simpcp.sh ./simpcp1 23
./simpcp1 srcfile destfile 1
 simpcp1.c:docopy:51: srcfile: size=23552 Block size for filesystem
I/O = 4096 bytes
 simpcp1.c:docopy:61: Expected # loop iterations: 5 [+1 for rem 3072
bytes]

simpcp1.c:docopy:65: 000001: read 4096 simpcp1.c:docopy:74: wrote 4096
simpcp1.c:docopy:65: 000002: read 4096 simpcp1.c:docopy:74: wrote 4096
simpcp1.c:docopy:65: 000003: read 4096 simpcp1.c:docopy:74: wrote 4096
simpcp1.c:docopy:65: 000004: read 4096 simpcp1.c:docopy:74: wrote 4096
simpcp1.c:docopy:65: 000005: read 4096 simpcp1.c:docopy:74: wrote 4096
simpcp1.c:docopy:65: 000006: read 3072 simpcp1.c:docopy:74: wrote 3072
simpcp1.c:docopy:65: 000007: read 0
```

```
-rw-r--r--. 1 kai kai 23552 Aug 6 19:01 destfile
-rw-rw-r--. 1 kai kai 23552 Aug 6 19:01 srcfile
Diff-ing them now ...
Done.
$
```

(To prove that it worked, the preceding script even does a quick *diff* of the source and destination files; no output from *diff* implies that the files are identical.) We encourage the reader to build and try it out, tweaking the script parameters (size, verbosity, and so on).

Did you notice? The mode of the destination file is different from that of the source file (it's hard-coded to 0664 octal); we will improve this—and another key concern—in a second version! Read on.

Correctly using the read/write APIs

As we have already learned, the read and write system calls are, under the hood, the underlying routines that perform actual I/O on a file (of course, again, realize that a file on Unix/Linux can mean many things!). For the reader's convenience, we will once again show the signature of these key I/O routines:

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

So, it seems quite intuitive that if we issue a read of count, let's say, 16,000 bytes on an open file, it will occur, and by the time the system call returns the destination buffer, buf will now contain the requested 16,000 bytes from the underlying file. But no! This may not necessarily be the case. You see, as is often the case, we make too many assumptions – here, we have quite happily assumed that if we request some n bytes from a file (via the OS), exactly n bytes will be read and returned to us by the calling process. This may not be the case; why? Think about it:

- One, what if the file size is less than the requested amount (or the program has already read a good part of the file and only a small portion remains to be read in)? Then only the data for that (remaining) size will be read and returned in the user space buffer.

- Two, and even more important to understand, the POSIX standard does not guarantee that IO (reads and writes) via system calls are necessarily atomic. In other words, it's entirely possible that when we request, say, 16,000 bytes to be read from a file, read(2) goes through but returns the value 12,288, indicating that only 12,288 bytes of the 16,000 requested have actually been read. This situation can also occur when reading from a pipe (IPC object) when less than the amount requested for resides in the pipe, or when the read (or write) system calls are interrupted by a signal (Signaling is covered in depth in later `Chapter 11`, *Signaling - Part I*, and `Chapter 12`, *Signaling - Part II*, in this book).

The key point here in terms of code is this: do not assume that the amount requested to be read or written is actually the amount read or written; rather, write the code in a loop that checks the actual I/O performed and continues in the loop until all requested data has actually been read or written.

One implementation of the code to do this is shown in the following code—our routines `fd_read` and `fd_write` perform the I/O operations just described:

```
int fd_read(int fd, void *dbuf, size_t n)
{
  ssize_t rd=0;
  size_t tr=0;

  do {
    rd = read(fd, dbuf, n);
    if (rd < 0)
      return -1;
    if (rd == 0)
      break;
    tr += rd;
#ifdef DEBUG
    printf(" requested=%9lu, actualrd=%9zu, totalrd=%9zu\n", n, rd,
tr);
#endif
  } while (tr < n);
  return tr;
}

int fd_write(int fd, void *sbuf, size_t n)
{
  ssize_t wr=0;
  size_t tw=0;

  do {
    wr = write(fd, sbuf, n);
```

```
        if (wr < 0)
            return -1;
        tw += wr;
#ifdef DEBUG
        printf(" requested=%9zu, actualwr=%9zu, totalwr=%9lu\n", n, wr,
tw);
#endif
    } while (tw < n);
    return tw;
}
```

Great, but we must test it. Here, we will make a copy of our previous program, `A_fileio/simpcp1.c`, and call it `A_fileio/simpcp2.c`: the key difference is that in our next version, in the infinite loop of our `docopy()` function, we invoke the superior I/O processing routines described previously (`fd_read()` and `fd_write()`):

```
[...]
    while (1) {
        nr = fd_read(fd_src, buf, n);
        VPRINT("%06d: read %9ld ", loop, nr);
        if (nr < 0) {
            WARN("read on src failed\n");
            free(buf);
            return -3;
        } else if (nr == 0)
            break;

        nw = fd_write(fd_dest, buf, nr);
        VPRINT("wrote %9ld\n", nw);
        if (nw < 0) {
            WARN("write on dest failed\n");
            free(buf);
            return -4;
        }
        loop ++;
    }
[...]
```

There's another small detail to take care of: let's ensure that the mode of the destination file matches that of the source file:

```
[...]
/* Make the destination file mode match the source file's */
    if (stat(argv[1], &sb) < 0)
            FATAL("stat on %s failed\n", argv[1]);
    VPRINT("%s mode=%o (octal)\n", argv[1], sb.st_mode & ACCESSPERMS);
    /* Create / overwrite destination as write-only for append;
     * mode=<srcfile's> */
    fdw = open(argv[2], O_CREAT|O_WRONLY, sb.st_mode & ACCESSPERMS);
[...]
```

A quick test run via the convenience script gives us the following:

```
$ ./test_simpcp.sh ./simpcp2_dbg 5
./simpcp2_dbg srcfile destfile 1
simpcp2.c:main:151: srcfile mode=664 (octal)
simpcp2.c:docopy:93: srcfile: size=5120 Block size for filesystem I/O
= 4096 bytes
simpcp2.c:docopy:103: Expected # loop iterations: 1 [+1 for rem 1024
bytes]

 requested= 4096, actualrd= 4096, totalrd= 4096
simpcp2.c:docopy:107: 000001: read 4096 requested= 4096, actualwr=
4096, totalwr= 4096
simpcp2.c:docopy:116: wrote 4096
 requested= 4096, actualrd= 1024, totalrd= 1024
simpcp2.c:docopy:107: 000002: read 1024 requested= 1024, actualwr=
1024, totalwr= 1024
simpcp2.c:docopy:116: wrote 1024
simpcp2.c:docopy:107: 000003: read 0

-rw-rw-r--. 1 kai kai 5120 Aug 7 06:22 destfile
-rw-rw-r--. 1 kai kai 5120 Aug 7 06:22 srcfile
Diff-ing them now ...
Done.
$
```

There, it's done.

Non-blocking writes and synchronization

At first glance, it seems obvious and intuitive that, once the `write(2)` call returns success, the actual write to the underlying storage device has thus been successfully carried out. However, the fact is that this is by default—not the case! In other words, writing to a file is actually an asynchronous or non-blocking activity; the reality is that data written via an application to the OS via system calls (at least temporarily) resides inside another cache within the OS called the **page cache** (see the following diagram for more details). Of course, the underlying kernel and drivers will ensure that the cached data ultimately gets to persistent storage, but it's important to understand that this is not immediate; it's not synchronous:

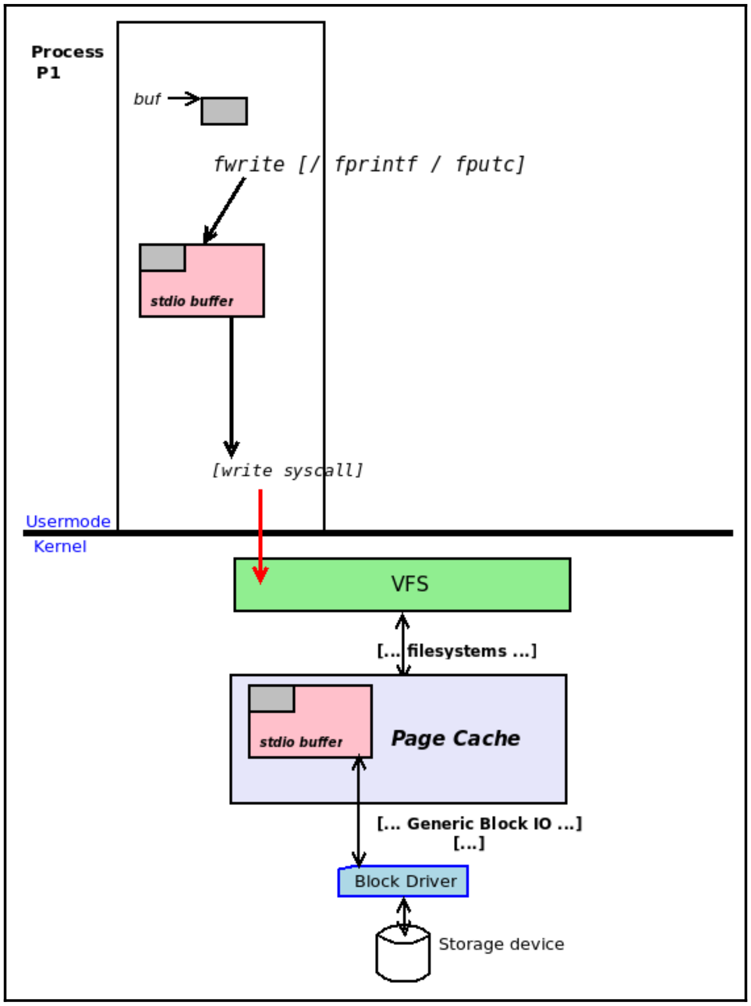


Figure 3: More detail—app to stdio I/O buffer to kernel page cache

Recall from our discussion on I/O buffering earlier that the third type – Raw (or unbuffered) I/O – had a cache size of zero bytes. Now, we understand that this is merely from the abstracted user space viewpoint; from the OS viewpoint, there is a cache – the page cache. The reality is that every single piece of file I/O passes through the page cache. (Okay, there are exceptions and explicit ways to sidestep this cache, but let's postpone this discussion until `Chapter 18`, *Advanced File I/O*.)

Synchronous writes

What if the developer would like to ensure that a write is synchronous? Recall the `O_SYNC` flag for `open(2)` – that's one way. But hang on, don't rush into it; you must realize that asynchronous writes are the default for a very good reason: performance.

Nevertheless, to guarantee that a write is actually (more or less immediately) flushed to the underlying persistent storage device, we have some useful utilities and system calls available to us—`sync(1)`, `sync(2)`, and `syncfs(2)`.

The *sync(1)* utility (no parameters) flushes all cached data to the underlying persistent storage device(s); it's actually just a wrapper over the *sync(2)* system call: `#include <unistd.h>`

```
void sync(void);
```

Thinking on this, the astute reader will realize that, it will work, but what about performance? Yes, indeed, performance takes a blow when cached data is continually synced. Why? Recall that the whole point of a cache is that the speed of the cache memory is much faster than the speed of the underlying memory it is ultimately destined to go to; think about the case where the caches are in the CPU and the destination is RAM, and similarly, our case here: the cache is RAM and the destination is a persistent storage device! So, using a cache greatly helps performance.

However, what if the developer must ensure that a certain file's contents are synchronized with the underlying storage device? Luckily, there is a fairly efficient way to do so – the `fsync` system call:

```
#include <unistd.h>
int fsync(int fd);
```

The key point is that we pass, as a parameter, the file descriptor to the particular file whose content we want synchronized (flushed); the OS will thus flush only that file's data to storage, not the entire system's cached data. This will provide better performance.

The `syncfs(2)` system call performs a similar function:

```
int syncfs(int fd);
```

One might well ask: what if I am using library APIs to perform I/O (say, `fwrite(3)`)? In that case, I will have a stream (`FILE *fp`) and not a file descriptor; how can I use `fsync(2)`? Easy: recall the `fileno(3)` API – given a stream pointer, it returns the corresponding file descriptor:

```
int fileno(FILE *stream);
```

Thus, one could code in this fashion (pseudocode):

```
FILE *fp;
fp = fopen(...);
...
fwrite(..., fp);
fflush(fp);
fsync(fileno(fp));
```

A call to `fflush(fp)` in between `fwrite(3)` and `fsync(2)` can be very handy, otherwise you may synchronize the kernel data on the underlying file descriptor but still lose the user space data held by glibc.

Reminder: check for the failure case: `fileno(3)` returns `-1` on failure.

Summary

In this chapter, we covered several important concepts regarding file I/O (on regular files). We approached this topic at two abstraction layers—one, the C library and the stdio streams model, and two, at the deeper system call layer.

In both, the reader has seen not just the APIs and mechanics of performing IO, but also learned key concepts such as I/O buffering (and flushing), stream redirection, correct usage of the powerful read and write system calls, and so on. Understanding the various layers that are gone through – right from the application buffer to the intermediate C library stdio I/O buffers to the OS-level page cache-has been emphasized.

Mixing I/O at these two levels is discouraged; it has been a known cause of issues. Trying out and working on code is, as usual, highly encouraged.

We leave more advanced materials, especially I/O performance enhancement, to `Chapter 18`, *Advanced File I/O*. Getting a firm grasp on the details of this chapter is recommended before moving on to subsequent ones.

Index

B

Beginning Of File (BOF) 7

C

C library
- about 2
- I/O buffering 11
- stdio APIs 14
- stdio file I/O API set 3
- stream 2

E

end-of-file (EOF) 4
Executable and Linker Format (ELF) 1

F

file descriptor (fd)
- about 2, 41
- streams, mapping to 42

freopen API
- used, stream redirection 30
- using 31

I

I/O buffering mode and size
- about 25
- code example 26

O

open system call
- about 34
- file descriptor 41
- file mode 38
- flags 35

R

read/write system calls
- about 43
- APIs, using 49
- non-blocking write 52
- read/write system calls 44
- simple copy program 44
- synchronization 52

regular file I/O
- using, with read and write system calls 34

S

stdio APIs
- about 14
- I/O buffering mode and size, modifying 25
- stdio buffering, testing with code 16, 18, 21
- steam redirection, with freopen API 30
- streams, flushing 22

stdio file I/O API set
- about 3
- deal, closing 5
- fopen API 4
- I/O buffering 11
- stream, closing 3
- stream, opening 3, 5
- stream, seeking 5

stream
- atomic appends 9
- debugging, via instrumentation 22
- seeking 7
- seeking, example 8

synchronous writes 54
system call layer
- opening at 34

V

virtual address space (VAS) 24
Virtual Filesystem Switch (VFS) 12