2

Basic File I/O

2.1 Introduction to File I/O

In this chapter we'll explore basic I/O on regular files. The I/O story continues in Chapter 3 with more advanced I/O system calls. I/O on special files is in Chapter 4, I/O on pipes in Chapter 6, I/O on named pipes in Chapter 7, and I/O on sockets in Chapter 8.

To get started I'll show a simple example that uses four system calls you may already be familiar with: `open`, `read`, `write`, and `close`. This function copies one file to another (like the `cp` command):

```
#define BUFSIZE 512

void copy(char *from, char *to)  /* has a bug */
{
    int fromfd = -1, tofd = -1;
    ssize_t nread;
    char buf[BUFSIZE];

    ec_neg1( fromfd = open(from, O_RDONLY) )
    ec_neg1( tofd = open(to, O_WRONLY | O_CREAT | O_TRUNC,
      S_IRUSR | S_IWUSR) )
    while ((nread = read(fromfd, buf, sizeof(buf))) > 0)
        if (write(tofd, buf, nread) != nread)
            EC_FAIL
    if (nread == -1)
        EC_FAIL
    ec_neg1( close(fromfd) )
    ec_neg1( close(tofd) )
    return;

EC_CLEANUP_BGN
    (void)close(fromfd); /* can't use ec_neg1 here! */
    (void)close(tofd);
EC_CLEANUP_END
}
```

Try to find the bug in this function (there's a clue in Section 1.4.1). If you can't, I'll point it out in Section 2.9.

I'll say just a few quick words about this function now; there will be plenty of time to go into the details later. The first call to `open` opens the input file for reading (as indicated by `O_RDONLY`) and returns a file descriptor for use in subsequent system calls. The second call to `open` creates a new file (because of `O_CREAT`) if none exists, or truncates an existing file (`O_TRUNC`). In either case, the file is opened for writing and a file descriptor is returned. The third argument to `open` is the set of permission bits to use if the file is created (we want read and write permission for only the owner). `read` reads the number of bytes given by its third argument into the buffer pointed to by its second argument. It returns the number of bytes read, zero on end-of-file, or –1 on error. `write` writes the number of bytes given by its third argument from the buffer given by its second argument. It returns the number of bytes written, which we treat as an error if it isn't equal to the number of bytes we asked to be written. Finally, `close` closes the file descriptors.

Rather than use `if` statements, `fprintf` calls, and `goto`s to deal with errors, I used the convenience macros `ec_neg1`, which leaves the function with an error if its argument is –1, and `EC_FAIL`, which always just leaves with an error. (Actually, they jump to the cleanup code, of which there is none in this case, delimited by `EC_CLEANUP_BGN` and `EC_CLEANUP_END`.) These were introduced back in Section 1.4.2.

2.2 File Descriptors and Open File Descriptions

Each UNIX process has a bunch of file descriptors at its disposal, numbered 0 through N, where N is the maximum. N depends on the version of UNIX and its configuration, but it's always at least 16, and much greater on most systems. To find out the actual value of N at run-time, you call `sysconf` (Section 1.5.5) with an argument of `_SC_OPEN_MAX`, like this:

```
printf("_SC_OPEN_MAX = %ld\n", sysconf(_SC_OPEN_MAX));
```

On a Linux 2.4 system we got 1024, on FreeBSD, 957, and on Solaris, 256. There probably aren't any current systems with just 16, except maybe for some embedded systems.

2.2.1 Standard File Descriptors

By convention, the first three file descriptors are already open when the process begins. File descriptor 0 is the *standard input,* file descriptor 1 is the *standard output,* and file descriptor 2 is the *standard error output,* which is usually open to the controlling terminal. Instead of numbers, it's better to use the symbols `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`.

A UNIX filter would read from `STDIN_FILENO` and write to `STDOUT_FILENO`; that way the shell can use it in pipelines. `STDERR_FILENO` should be used for important messages, since anything written to `STDOUT_FILENO` might go off down a pipe or into a file and never be seen should output be redirected, which is very commonly done from the shell.

Any of these standard file descriptors could be open to a file, a pipe, a FIFO, a device, or even a socket. It's best to program in a way that's independent of the type of source or destination, but this isn't always possible. For example, a screen editor probably won't work at all if the standard output isn't a terminal device.

The three standard file descriptors are ready to be used immediately in `read` and `write` calls. The other file descriptors are available for files, pipes, etc., that the process opens for itself. It's possible for a parent process to bequeath more than just the standard three file descriptors to a child process, and we'll see exactly that in Chapter 6 when we connect processes with pipes.

2.2.2 Using File Descriptors

Generally, UNIX uses file descriptors for anything that behaves in some way like a file, in that you can `read` it or `write` it or both. File descriptors aren't used for less-file-like communication mechanisms, such as message queues, which you can't `read` and `write` (there are specialized calls for the purpose).

There are only a few ways to get a fresh open file descriptor. We're not ready to dig into all of them right now, but it's helpful at least to list them:

- `open`, used for most things that have a path name, including regular and special files and named pipes (FIFOs)
- `pipe`, which creates and opens an un-named pipe (Chapter 6)
- `socket`, `accept`, and `connect`, which are used for networking (Chapter 8)

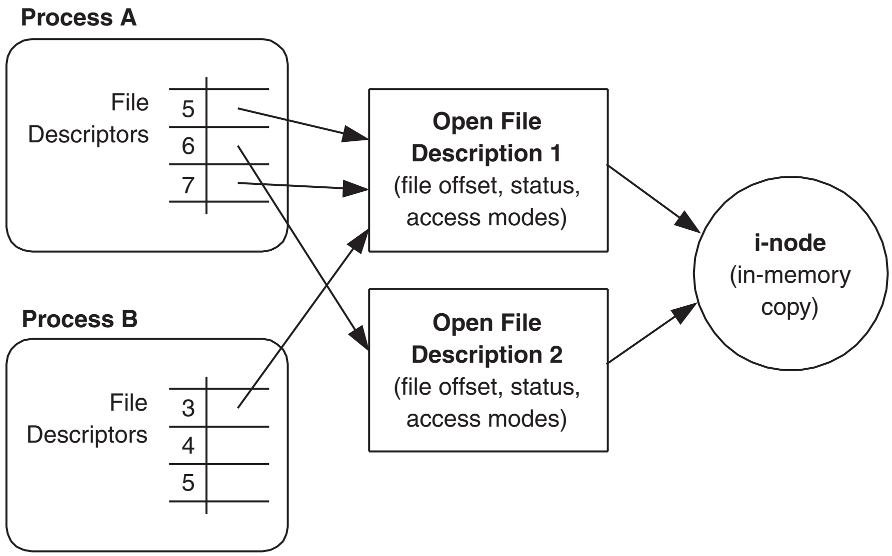

Figure 2.1 File descriptors, open file descriptions, and i-nodes.

There isn't any specific C type for a file descriptor (as there is, say, for a process ID), so we just use a plain `int`.[1]

2.2.3 Open File Descriptions and Sharing

A file descriptor is, as I said in Chapter 1, just an index into a per-process table. Each table entry points to a system-wide open file description (also known as a file-table entry), which in turn points to the file's data (via an in-memory copy of the i-node). Several file descriptors, even from different processes, can point to the same file description, as shown in Figure 2.1.

Each `open` or `pipe` system call creates a new open file description and a new file descriptor. In the figure, Process A opened the same file twice, getting file descriptors 5 and 6, and creating open file descriptions 1 and 2. Then, through a mechanism called file descriptor duplication (the `dup`, `dup2`, and `fork` system

1. I thought about introducing the type `fid_t` just for this book to make the examples a little more readable but then decided not to because in real code you'll see plain `int`, so why not get used to it.

calls do it), Process A got file descriptor 7 as a duplicate of 5, which means it points to the same open file description. Process B, a child of A, also got a duplicate of 5, which is its file descriptor 3.

We'll come back to Figure 2.1 from time to time because it tells us a lot about how things behave. For example, when we explain file offsets in Section 2.8, we'll see that Process A's file descriptors 5 and 7 and Process B's file descriptor 3 all share the same file offset because they share the same open file description.

2.3 Symbols for File Permission Bits

Recall from Section 1.1.5 that a file has 9 permission bits: read, write, and execute for owner, group, and others. We see them all the time in the output of the `ls` command:

```
-rwxr-xr-x    1 marc     users       29808 Aug  4 13:45 hello
```

Everyone thinks of the 9 bits as being together and in a certain order (owner, group, others), but that's not a requirement—only that there be 9 individual bits. So, from POSIX1988 on there have been symbols for the bits that are supposed to be used instead of octal numbers, which had been the traditional way to refer to them. The symbols have the form `S_I`**pwww** where **p** is the permission (`R`, `W`, or `X`) and **www** is for whom (`USR`, `GRP`, or `OTH`). This gives 9 symbols in all.

For instance, for the previous file, instead of octal 755, we write

```
S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

There are separate symbols for when a USR, GRP, or OTH has all three permissions, which are of the form `S_IRWX`**w**. This time **w** is just the first letter of the "whom," either `U`, `G`, or `O`. So the permissions could instead be written like this:

```
S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

The symbols are necessary to give implementors the freedom to do what they will with the bit positions, even if they're less readable and much more error-prone than just octal.[2] As any application you write is likely to use only a few combinations (e.g., one or two for data files it creates, and maybe another if it creates directories), it's a good idea to define macros for them just once, rather than using

2. One of the technical reviewers pointed out that even if octal were used, the kernel could map it to whatever the file system used internally.

long sequences of the `S_I*` symbols all over the place. For this book, we'll use just these, which are defined in defs.h (Section 1.6):

```
#define PERM_DIRECTORY   S_IRWXU
#define PERM_FILE        (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

Notice that we named the macros according to how we're going to use them, not according to the bits. Therefore, it's `PERM_FILE`, not something like `PERM_RWURGO`, because the whole point is being able to change the application's permissions policy by changing only one macro.

2.4 `open` and `creat` System Calls

open—open or create file

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

int open(
      const char *path,      /* pathname */
      int flags,             /* flags */
      mode_t perms           /* permissions (when creating) */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

You use `open` to open an existing file (regular, special, or named pipe) or to create a new file, in which case it can only be a regular one. Special files are created with `mknod` (Section 3.8.2) and named pipes with `mkfifo` (Section 7.2.1). Once the file is opened, you can use the file descriptor you get back with `read`, `write`, `lseek`, `close`, and a bunch of other calls that I'll discuss in this and the next chapter.

2.4.1 Opening an Existing File

Let's talk first about opening an existing file, specified by `path`. If `flags` is `O_RDONLY`, it is opened for reading; if `O_WRONLY`, for writing; and if `O_RDWR`, for both reading and writing.[3]

3. Why three flags? Can't we scrap `O_RDWR` and just use `O_RDONLY | O_WRONLY`? No, because implementations have always defined `O_RDONLY` as zero, rather than as some bit.

The process needs read, or write, or both, kinds of permission to open the file, using the algorithm that was explained in Section 1.1.5. For example, if the effective user-ID of the process matches the owner of the file, and file's owner read and/or write permission bits have to be set.

For an existing file, the `perms` argument isn't used and is normally omitted entirely, so `open` is called with only two arguments.

The file offset (where reads and writes will occur) is positioned at the first byte of the file. More about this in Section 2.8.

Here's some code that opens an existing file:

```
int fd;

ec_neg1( fd = open("/home/marc/oldfile", O_RDONLY) )
```

There are lots of reasons why `open` can fail, and for many of them it pays to tell the user what the specific problem is. A path to a nonexistent file (`ENOENT`) requires a different solution from wrong permissions (`EACCES`). Our normal error-checking and reporting handles this very well (the "ec" macros were explained in Section 1.4.2).

The file descriptor returned by a successful `open` is the lowest-numbered one available, but normally you don't care what the number is. This fact is sometimes useful, however, when you want to redirect one of the standard file descriptors, 0, 1, or 2 (Section 2.2.1): You close the one you want to redirect and then open a file, which will then use the number (1, say) that you just made available.

2.4.2 Creating a New File

If the file doesn't exist, `open` will create it for you if you've ORed the `O_CREAT` flag into the flags. You can certainly create a new file opened only for reading, although that makes no sense, as there would be nothing to read. Normally, therefore, either `O_WRONLY` or `O_RDWR` are combined with `O_CREAT`. Now you do need the `perms` argument, as in this example:

```
ec_neg1( fd = open("/home/marc/newfile", O_RDWR | O_CREAT, PERM_FILE) )
```

The `perms` argument is only used if the file gets created. It has no effect on anything if the file already exists.

The permission bits that end up in a newly created file are formed by ANDing the ones in the system call with the complement of process's file mode creation mask, typically set at login time (with the `umask` command), or with the `umask` system call (Section 2.5). The "ANDing the complement" business just means that if a bit is set in the mask, it's cleared in the permissions. Thus, a mask of 002 would cause the `S_IWOTH` bit (write permission for others) to be cleared, even if the flag `S_IWOTH` appeared in the call to `open`. As a programmer, however, you don't usually think about the mask, since it's something users do to *restrict* permissions.

What if you create a file with the `O_WRONLY` or `O_RDWR` flags but the permission bits don't allow writing? As the file is brand new, it is *still* opened for writing. However, the next time it's opened it will already exist, so then the permission bits will control access as described in the previous section.

Sometimes you always want a fresh file, with no data in it. That is, if the file exists you want its data to be thrown away, with the file offset set at zero. The `O_TRUNC` flag does that:

```
ec_neg1( fd = open("/home/marc/newfile", O_WRONLY | O_CREAT | O_TRUNC,
  PERM_FILE) )
```

Since `O_TRUNC` destroys the data, it's allowed on an existing file only if the process has write permission, as it is a form of writing. Similarly, it can't be used if the `O_RDONLY` flag is also set.

For a *new* file (i.e., `O_CREAT` doing its thing), you need write permission in the parent directory, since a new link will be added to it. For an *existing* file, the permissions on the directory don't matter; it's the file's permissions that count. The way to think of this is to ask yourself, "What needs to be written to complete this operation?"

I might also mention that you need search (execute) permission on the intermediate directories in the path (i.e., `home` and `marc`). That, however, universally applies to any use of a path, and we won't say it every time.

`O_TRUNC` doesn't have to be used with `O_CREAT`. By itself it means "truncate the file to zero if it exists, and just fail if it doesn't exist." (Maybe there's a logging feature that gets turned on by creating a log file; no log file means no logging.)

The combination `O_WRONLY | O_CREAT | O_TRUNC` is so common ("create or truncate a file for writing") that there's a special system call just for it:

creat—create or truncate file for writing

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

int creat(
    const char *path,       /* pathname */
    mode_t perms            /* permissions */
);
/* Returns file descriptor  or -1 on error (sets errno) */
```

`open` for an existing file used only the first and second arguments (`path` and `flags`); `creat` uses just the first and third. In fact, `creat` could be just a macro:

```
#define creat(path, perms) open(path, O_WRONLY | O_CREAT | O_TRUNC, perms)
```

Why not just forget about `creat` and use `open` all the time—one system call to remember instead of two, and the flags always stated explicitly? Sounds like a great idea, and that's what we'll do in this book.[4]

We've skipped one important part of creating a new file: Who owns it? Recall from Section 1.1.5 that every file has an owner user-ID and an owner group-ID, which we just call owner and group for short. Here's how they're set for a new file:

- The owner is set from the effective user-ID of the process.
- The group is set to either the group-ID of the parent directory or the effective group-ID of the process.

Your application can't assume which method has been used for the group-ID, although it could find out with the `stat` system call (Section 3.5.1). Or, it could use the `chown` system call (Section 3.7.2) to force the group-ID to what it wants. But it's a rare application that cares about the group-ID at all.

There's another flag, `O_EXCL`, that's used with `O_CREAT` to cause failure if the file already exists. If `open` without the `O_CREAT` flag is "open if existent, fail if nonexistent," then `open` with `O_CREAT | O_EXCL` is the exact opposite, "create if nonexistent, fail if existent."

There's one interesting use for `O_EXCL`, using a file as a lock, which I'll show in the next section. Another use might be for a temporary file that's supposed to be deleted when the application exits. If the application finds it already existing, it

4. If you're interested in the history, `creat` is actually a very old system call that was important when an earlier form of `open` had only two arguments.

means the previous invocation terminated abnormally, so there's some cleanup or salvaging to do. You want the creation to fail in this case, and that's exactly what the `O_EXCL` flag will do for you:

```
int fd;

while ((fd = open("/tmp/apptemp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL,
  PERM_FILE)) == -1) {
    if (errno == EEXIST) {
        if (cleanup_previous_run())
            continue;
        errno = EEXIST; /* may have been reset */
    }
    EC_FAIL /* some other error or can't cleanup */
}
/* file is open; go ahead with rest of app */
```

We don't want to use the `ec_neg1` macro on `open` because we want to investigate `errno` for ourselves. The value `EEXIST` is specifically for the `O_EXCL` case. We call some function named `cleanup_previous_run` (not shown) and try again, which is why the `while` loop is there. If cleanup doesn't work, notice that we reset `errno`, as it's pretty volatile, and we have no idea what `cleanup_previous_run` may have done—it could be thousands of lines of code. (We could have used a `for` loop with only two iterations instead of the `while`, to catch that case when `cleanup_previous_run` returns `true`, keeping us in the loop, but has failed to unlink the file. But you get the idea.)

Our example messes up if the application is being run concurrently, maybe by two different users. In that case the temporary being around makes sense, and unlinking it would be a crime. If concurrent execution is allowed, we really need to redesign things so that each execution has a unique temporary file, and I'll show how to do that in Section 2.7. If we want to prevent concurrent execution entirely, we need some sort of locking mechanism, and that answer is in the next section.

2.4.3 Using a File as a Lock

Processes that want exclusive access to a resource can follow this protocol: Before accessing the resource, they try to create a file (with an agreed-upon name) using `O_EXCL`. Only one of them will succeed; the other processes' `open`s will fail. They can either wait and try later or just give up. When the successful process finishes with the resource, it unlinks the file. One of the unsuccessful processes' `opens` will then work, and it can safely proceed.

For this to work, the checking to see if the file exists (with `access`, say, which is in Section 3.8.1) and the creating of it have to be atomic (indivisible)—no other process can be allowed to execute in the interim, or it might create that very file *after* the first process has already checked. So don't do this:

```
if (access( … ) == 0)        /* file does not exist */
    … open(… ) …             /* create it */
```

We need a more reliable method that guarantees atomicity.

A simple exclusivity mechanism like this is called a *mutex* (short for mutual exclusion), a *binary semaphore* (can count only up to 1), or a *lock*. We'll hit these things several times throughout this book; see Section 1.1.7 for a quick rundown. In the UNIX world, the word "mutex" is more often used in the context of threads and the word "semaphore" suggests something that uses the UNIX semaphore system calls, so we'll just call it a "lock" in this section.

The protocol is best encapsulated into two functions, `lock` and `unlock`, to be used like this:

```
if  (lock("accounts")) {
    … manipulate accounts …
    unlock ("accounts");
}
else
    … couldn't obtain lock …
```

The lock name "accounts" is abstract; it doesn't necessarily have anything to do with an actual file. If two or more processes are concurrently executing this code, the lock will prevent them from simultaneously executing the protected section ("manipulate accounts," whatever that means). Remember that if a process doesn't call `lock`, though, then there is no protection. They're *advisory* locks, not *mandatory*. (See Section 7.11.5 for more on the distinction between the two.)

Here is the code for `lock`, `unlock`, and a little function `lockpath`:

```
#define LOCKDIR "/tmp/"
#define MAXTRIES 10
#define NAPLENGTH 2

static char *lockpath(char *name)
{
    static char path[100];
```

```
    if (snprintf(path, sizeof(path), "%s%s", LOCKDIR, name) > sizeof(path))
        return NULL;
    return path;
}

bool lock(char *name)
{
    char *path;
    int fd, tries;

    ec_null( path = lockpath(name) )
    tries = 0;
    while ((fd = open(path, O_WRONLY | O_CREAT | O_EXCL, 0)) == -1 &&
      errno == EEXIST) {
        if (++tries >= MAXTRIES) {
            errno = EAGAIN;
            EC_FAIL
        }
        sleep(NAPLENGTH);
    }
    if (fd == -1)
        EC_FAIL
    ec_neg1( close(fd) )
    return(true);

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool unlock(char *name)
{
    char *path;

    ec_null( path = lockpath(name) )
    ec_neg1( unlink(path) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The function `lockpath` generates an actual file name to use as the lock. We put it in the directory `/tmp` because that directory is on every UNIX system and it's writable by everyone. Note that if `snprintf` returns a number too big, it doesn't overrun the buffer passed to it; the number represents what *might* have occurred.

lock places a lock by trying to create the file with O_EXCL, as explained in the previous section, and we distinguish the EEXIST case from the other error cases, just as we did there.

We try to create the file up to MAXTRIES times, sleeping for NAPLENGTH seconds between tries. (sleep is in Standard C, and also in Section 9.7.2.) We close (Section 2.11) the file descriptor we got from open because we aren't interested in actually writing anything to the file—its existence or nonexistence is all we care about. We even created it with no permissions. As an enhancement, we could write our process number and the time on the file so other processes waiting on it could see who they're waiting for and how long it's been busy. If we did this, we'd want the permissions to allow reading.

All unlock has to do is remove the file. The next attempt to create it will succeed. I'll explain the system call unlink in Section 2.6.

The little test program is also interesting:

```
void testlock(void)
{
    int i;

    for (i = 1; i <= 4; i++) {
        if (lock("accounts")) {
            printf("Process %ld got the lock\n", (long)getpid());
            sleep(rand() % 5 + 1); /* work on the accounts */
            ec_false( unlock("accounts") )
        }
        else {
            if (errno == EAGAIN) {
                printf("Process %ld tired of waiting\n", (long)getpid());
                ec_reinit(); /* forget this error */
            }
            else
                EC_FAIL /* something serious */
        }
        sleep(rand() % 5 + 5); /* work on something else */
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("testlock")
EC_CLEANUP_END
}
```

It cycles four times through an acquire/work/release pattern, with the "work" just sleeping for some random number of seconds between 1 and 5. If it doesn't get the lock, it prints a complaint and keeps going. Then it does something else not involving the accounts for between 5 and 9 seconds, and cycles again. The `printf` calls use a system call to get the process ID that I didn't explain yet, `getpid` (Section 5.13). I ran three of these little guys at once:

```
$ tst & tst & tst &
```

and this was the output:

```
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9234 got the lock
```

It started off in a predictable way but then got more interesting on line 6. At the end, process 9234 had to stay late after the others had gone home.

Let's talk about the pros and cons of using files as locks. The pros are that they're easy to code (we're still early in Chapter 2 of this book), being files they can contain some data, and they stay around as long as files do, which is useful when a long-duration lock is wanted. That last point also appears in the list of cons: If a process terminates without clearing its lock, even rebooting won't clear it unless the `/tmp` directory is cleared. Also, they're really slow because creating a file, even if it fails, is a giant operation—maybe OK for a few times per application-execution, but not for the kind of fast locking that a database or a real-time program might need.

I haven't mentioned the worst problem, however: If a process can't get the lock, it keeps sleeping and trying until it can, which is called *polling*. This is a lot of work for the CPU to do just to answer a simple question: "Is it my turn yet?" And, the lock might become available while the process is still asleep, but it won't find out until it wakes up, which is a waste.

Fortunately, all the built-in UNIX facilities for locking use what's called *blocking,* which means that the process sleeps until the event it's waiting for occurs. I'll get to that in Section 7.11.

2.4.4 Summary of `open` Flags

There are more `open` flags besides the ones I've explained so far, but it makes more sense to talk about them later when I've had a chance to place them in the appropriate context. For example, `O_NOCTTY` has to do with terminals, so I'll talk about it in Chapter 4. Table 2.1 shows all the flags defined by SUS3[5] with a quick description of each and a cross reference to the sections where they're discussed in detail.

Table 2.1 `open` Flags

Flag	Comment
`O_RDONLY*`	Open for reading only (Section 2.4.1).
`O_WRONLY`	Open for writing only (Section 2.4.1).
`O_RDWR`	Open for both (Section 2.4.1).
`O_APPEND`	All writes occur at the end of the file (Section 2.8).
`O_CREAT`	Create if nonexistent (Section 2.4.2).
`O_DSYNC`	Set synchronized I/O behavior (Section 2.16.3).
`O_EXCL`	Fail if exists; must be used with `O_CREAT` (Section 2.4.2).
`O_NOCTTY`	Don't make device the controlling terminal (Section 4.10.1).
`O_NONBLOCK†`	Don't wait for named pipe or special file to become available (Sections 4.2.2 and 7.2).
`O_RSYNC`	Set synchronized I/O behavior (Section 2.16.3).
`O_SYNC`	Set synchronized I/O behavior (Section 2.16.3).
`O_TRUNC`	Truncate to zero bytes (Section 2.4.2).

* One of the first three is required.
† Formerly called `O_NDELAY`, with somewhat different semantics.

5. Single UNIX Specification, Version 3; see Section 1.5.1.

2.5 `umask` System Call

We mentioned a process's file mode creation mask in Section 2.4.2. It's set by the `umask` system call, which is seldom used by anything other than the `umask` command:

umask—set and get file mode creation mask

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

mode_t umask(
    mode_t cmask          /* new mask */
);
/* Returns previous mask (no error return) */
```

Since every process has a mask, and since every combination of nine bits is legal, `umask` can never give an error return. It always returns the old mask. To find out what the old mask is without changing it requires two calls to `umask`: one to get the old value, with an argument of anything at all, and a second call to restore the mask to the way it was.

2.6 `unlink` System Call

unlink—remove directory entry

```
#include <unistd.h>

int unlink(
    const char *path      /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `unlink` system call removes a link from a directory, reducing the link count in the i-node by one. If the resulting link count is zero, the file system will discard the file: All disk space that it used will be made available for reuse (added to the "free list"). The i-node will become available for reuse, too. The process must have write permission in the directory containing the link.

Any kind of file (regular, socket, named pipe, special, etc.) can be unlinked, but only a superuser can unlink a directory, and on some systems even the superuser can't. In any case, the `rmdir` system call (Section 3.6.3) should be used for unlinking a directory, not `unlink`.

If the link count goes to zero while some process still has the file open, the file system will delay discarding the file until it's closed, to avoid disrupting a running process. This feature is frequently used to make temporary files that are needed only while a program is running, like this:

```
ec_neg1( fd = open("temp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1( unlink("temp") )
```

There are two advantages to this technique: First, if the process terminates for any reason, the file will be discarded. There's no need to register a function with `atexit` (Section 1.3.4), for example, to make sure the file is unlinked. Second, since the link is removed from the current directory right away with `unlink`, there's less danger of a second process accidentally using the same temporary file and failing in the `open` because of the `O_EXCL` flag. It could still happen, though, if the second process executes its `open` between the first process's `open` and `unlink`.

One way to fix the problem is with a lock (Section 2.4.3):

```
ec_false( lock("opentemp") )
ec_neg1(  fd = open("temp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1(  unlink("temp") )
ec_false( unlock("opentemp") )
```

Our fix is pretty good, but still has a defect: The lock is only advisory, and the temporary-file name is rather unimaginative. Thus, it's possible for another application (not using the lock) to use the same name, and one of the processes would then fail to get its temporary file (failing in `open`, because of `O_EXCL`). A better fix is making the temporary file name unique, but that's a bit tricky, as we'll see in Section 2.7, where I talk more about temporary files.

You might think there is another problem: Since the file name is always `temp`, do two processes both running the code read and write the same temporary file, making a mess of things? No. If you made a file named `myfile` today, removed it, and made another file tomorrow with the same name, you wouldn't expect them to be the same file. That the first process still has the file open (with its data intact) is irrelevant because the i-node it's using is completely different from the i-node that the second process will use. Think of it this way: `unlink` makes the directory entry go away even if the file is still open, and the i-node then becomes anonymous and therefore completely isolated from access by a new process.

2.7 Creating Temporary Files

The approach we used to create a temporary file in the previous section, using a fixed name (`temp`) and a lock to prevent two processes from executing the same code at the same time, is cumbersome, and so it's more typical for a UNIX program to avoid any possible clash by using a name that's guaranteed to be unique. The Standard C function `tmpnam` seems to do what we want:

```
char *pathname;

ec_null( pathname = tmpnam(NULL) )
ec_neg1( fd = open(pathname, O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1( unlink(pathname) )
```

It's guaranteed that what `tmpnam` returns is unique because it makes sure no file by that name exists. But the file doesn't get created until the `open`, so there's a small possibility that another process executing `tmpnam` at the same time could get the same name. (We have to keep reminding ourselves that the time between two lines of code can be arbitrarily long, and other processes can execute.) Because of the `O_EXCL` flag, one would fail to actually create and open it, so there's no danger of an I/O mix-up; however, we're only a little better off than we were with a fixed name. The probability of a clash is reduced, but still not zero. Not good enough.

Here's the answer:

mkstemp—create and open file with unique name

```
#include <stdlib.h>

int mkstemp(
    char *template         /* template for file name */
);
/* Returns open file descriptor or -1 on error (may not set errno) */
```

`mkstemp` absolutely guarantees that the file will be created with a unique name; there's no race-condition problem. You give it a template to use for the name that ends in six `X`s, which it replaces with whatever it takes to make the name unique. It then goes further than `tmpnam`: It actually creates and opens the file for reading and writing. You can't assume what permissions have been used, as the standard (SUS3) doesn't say, although most implementations will probably restrict reading and writing to just the owner (`S_IRUSR | S_IWUSR`).

For a portable program, it's not clear what to do about errno if mkstemp returns –1. The standard doesn't define any error codes, but it's a pretty good bet that all implementations will return a valid errno. So we choose to use the ec_neg1 macro (which records errno, recall), even though the synopsis says errno may not be set. To try to prevent a misleading error message if errno is not set on an error, we try to remember to set it to zero prior to the call.

mkstemp is in SUS1, Linux, FreeBSD, and Darwin (it originated with BSD), so you can count on its being available just about everywhere.

Here's an example; just for fun we'll print the file name:

```
char pathname[] = "/tmp/dataXXXXXX";

errno = 0; /* mkstemp may not set it on error */
ec_neg1( fd = mkstemp(pathname) )
ec_neg1( unlink(pathname) )
printf("%s\n", pathname);
```

Output:

```
/tmp/dataKdBy0u
```

You don't have to unlink the file right away, but if you don't you'll have to arrange to unlink it later (in an atexit-registered function, say), or it will be left around. Of course, you can't unlink it immediately if you need to pass the pathname to another part of the program or an external program that needs a name, as in this example:

```
int status;
char cmd[100];

ec_neg1( fd = mkstemp(pathname) )
/* code to write text lines to fd (not shown) */
snprintf(cmd, sizeof(cmd), "sort %s", pathname);
ec_neg1( status = system(cmd) )
```

The comment ("code to write…") substitutes for code that writes the file. system is a Standard C function that invokes an external program; we'll see how it works in Chapter 5.

By the way, there's a Standard C function that you might want to look up named tmpfile that works just as well as mkstemp, but it returns a FILE pointer instead of a file descriptor.

There's one more function you may have heard of named `mktemp` that's closer to `tmpnam` than to `mkstemp`, in that it returns a name but doesn't create the file. It has all the problems of `tmpnam` and isn't in Standard C, so don't use it.

A quick summary:

- Good: `mkstemp` and `tmpfile`
- Bad: `mktemp` and `tmpnam`

2.8 File Offsets and O_APPEND

This section explains the `O_APPEND` flag, which we first saw in Section 2.4.4, and then introduces some properties of the `read`, `write`, `lseek`, `pread`, and `pwrite` system calls, which are explained more fully in the rest of this chapter.

A *file offset* is a position in a regular file that marks where the next `read` or `write` will occur. That's its only purpose. Other types of files—directories, sockets, named pipes, and symbolic links—don't have file offsets. Special files may or may not have them, depending on their implementation (Section 3.2).

Before UNIX came along (hint: the Beatles hadn't broken up yet), most operating systems had both "sequential" and "random" data files. UNIX had just one type of data file, with a movable file offset to handle random access. It was an important innovation in its day, even if it seems normal and obvious today.

You get an independent file offset each time you open a file, because, as Figure 2.1 showed, you get a new open file description. This means that in this example

```
int fd1, fd2, fd3;

ec_neg1( fd1 = open("myfile", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
ec_neg1( fd2 = open("myfile", O_RDONLY) )
ec_neg1( fd3 = open("yourfile", O_RDWR | O_CREAT | O_TRUNC, PERM_FILE) )
```

`fd1` and `fd2` each have their own file offset so writes on `fd1` and reads on `fd2` are independent, but there is only one file offset for both reads and writes on `fd3`.

Absent the `O_APPEND` flag, the file offset starts out at zero on a freshly opened file and is automatically bumped by `read` and `write` by the amount they read or wrote. So, unless something is done to deliberately change the file offset, `reads` and `writes` are sequential. You read some, and then the next `read` reads some more, and so on. Ditto for `writes`.

Assuming the file offsets are starting at zero, if we write 100 bytes on `fd1` and then read 100 bytes on `fd2`, we get the 100 bytes we just wrote. But if we write on `fd3` and then read on `fd3`, we get whatever is *after* the written data, and an end-of-file if there wasn't anything because a file description has only a single offset, used by both `read` and `write`.

You can find out where the file offset is and/or set it to a new value with `lseek` (Section 2.13) on a file descriptor. The new value then affects the next `read` or `write` on that file descriptor.

Later, in Chapter 6, we're going to see how to duplicate an open file descriptor. By "duplicate" I don't mean copying, like

```
fd1 = fd2;
```

I mean using a system call that duplicates, such as `dup`. Anyway, for now the important point is that if a file descriptor is a duplicate of another, they share the same file offset because they share the same open file description.

If the file is opened with the `O_APPEND` flag set, all writes with the `write` system call are preceded by an implicit `lseek` to the end of the file, so writing occurs at the end, atomically. Even if several processes with the file opened with `O_APPEND` are writing at once, each of their writes will go at the end as it is that instant, and they won't overwrite each other or intermingle their data. You can't do the equivalent thing with `lseek` followed by a `write` (`O_APPEND` not set) because as we've seen in other situations, there's a gap between the two system calls that could result in this:

1. Process A seeks its file offset to the end (position 1000, say).
2. Process B seeks its file offset to the end (also position 1000).
3. Process B writes 200 bytes (at position 1000).
4. Process A writes 200 bytes (at position 1000—overwriting B). Ouch!

We could use a lock (Section 2.4.3) to fix this, but there's a much better way: If `O_APPEND` was set when process A and B opened the file, you are guaranteed to get this:

1. Process A seeks its file offset to the end (position 1000, say) and writes.
2. Process B seeks its file offset to the end (position 1200) and writes.

So `O_APPEND` is perfect for log files or other situations when you want to accumulate output from several processes.

You can also read and write by just specifying the position in the system call itself, without first calling `lseek`; that's what `pread` and `pwrite` do (Section 2.14). They don't use the file offset, and they don't change it, either.

Since you probably already know pretty much how `read` and `write` work, if you like you can skip to Section 2.13 to read about `lseek` and see some interesting examples and then loop back to Section 2.9 to continue in sequence.

2.9 `write` System Call

write—write to file descriptor

```
#include <unistd.h>

ssize_t write(
     int fd,             /* file descriptor */
     const void *buf,    /* data to write */
     size_t nbytes       /* amount to write */
);
/* Returns number of bytes written or -1 on error (sets errno) */
```

We've talked so much about `write`, don't you think it's time we got properly introduced?

`write` writes the `nbytes` bytes pointed to by `buf` to the open file represented by `fd`. The write starts at the current position of the file offset, and, after the write, the file offset is incremented by the number of bytes written. The number of bytes written, or –1 if there was an error, is returned.

Recall that if the `O_APPEND` flag is set, the file offset is set automatically to the end of the file prior to the write.

`write` is used to write to pipes, special files, and sockets, too, but its semantics are somewhat different in these cases. One important difference is that such writes can block, which means that they're waiting for some unpredictable event, such as data being available. If a `write` is blocked, it can be interrupted by the arrival of a signal (Section 9.1.4), in which case it returns –1 with `errno` set to `EINTR`. I'll postpone the rest of the discussion of writes to other than regular files until Chapters 4, 6, and 8, when I talk about other types of files.

`write` is deceptively simple. It seems that it writes the data and then returns, but a little experimentation will convince you that this is impossible—it's too fast. It must be cheating!

Indeed, it does cheat. When you issue a `write` system call, it does not perform the write and then return. It just transfers the data to a buffer cache in the kernel and then returns, claiming nothing more than this:

> I've taken note of your request, and rest assured that your file descriptor is OK. I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll try to put your data on the disk where it belongs. If I discover an error then I'll try to print something on the console, but I won 't tell you about it (indeed, you may have terminated by then). If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request. You may ask no further questions. Trust me, and thank me for the speedy reply—I figured that's all you really cared about.

If all does go well, delayed writing is fantastic. The semantics are the same as if the writing actually took place, but it's much faster. However, if there is a disk error, or if the kernel stops for any reason, then the game is up. We discover that the data we "wrote" isn't on the disk at all.

In addition to the uncertainty about when the physical write occurs, there are two other problems with delayed writes. First, a process initiating a write cannot be informed of write errors. Indeed, file system buffers aren't owned by any single process; if several processes write to the same block of the same file at the same time, their data will be transferred to the same buffer. Of course, one could conceive of a scheme in which a "write error" signal would be sent to every process that wrote a particular buffer, but what is a process supposed to do about it at that late date? And how does the kernel notify processes that have already terminated?

The second problem is that the *order* of physical writes can't be controlled. Order often matters. For example, in updating a linked-list structure on a file, it is better to write a new record and then update the pointer to it, rather than the reverse. This is because a record not pointed to is usually less of a problem than a pointer that points nowhere. Even if the `write` system calls are issued in a particular order, however, that doesn't mean that the buffers will be physically written to disk in that order. So *careful replacement* techniques, of which this is but one example, are not as advantageous as they might be. They guard against the process itself terminating at an inopportune time, but not against disk errors or kernel crashes.

Fortunately, you can force synchronized writes, and I'll explain how in Section 2.16.

These problems with `write` should not be overemphasized. Considering how reliable computers are today, and how reliable UNIX implementations usually are, kernel crashes are quite rare. Most users are pleased to benefit from the quick response provided by the buffer cache and never find out that the kernel is cheating.

Now let's look once again at the file copy example at the beginning of this chapter. The bug is in the check for a write error:

```
if (write(tofd, buf, nread) != nread)
    EC_FAIL
```

It is *not* an error if the count returned by `write` is less than the requested count. Maybe the count is short because the write is to a pipe that's momentarily full, or maybe it's a regular file that's reached its size limit. It will be the *next* call to write that will produce the error.[6]

So the bug is that the `EC_FAIL` macro will record a meaningless value for `errno`, which is set *only* when the return value is –1. We could recode the function to withstand partial writes and to keep trying until a real error occurs:

```
#define BUFSIZE 512

void copy2(char *from, char *to)
{
    int fromfd = -1, tofd = -1;
    ssize_t nread, nwrite, n;
    char buf[BUFSIZE];

    ec_neg1( fromfd = open(from, O_RDONLY) )
    ec_neg1( tofd = open(to, O_WRONLY | O_CREAT | O_TRUNC,
      S_IRUSR | S_IWUSR) )
    while ((nread = read(fromfd, buf, sizeof(buf))) > 0) {
        nwrite = 0;
        do {
            ec_neg1( n = write(tofd, &buf[nwrite], nread - nwrite) )
            nwrite += n;
        } while (nwrite < nread);
    }
```

6. Although, even then, it's possible that whatever caused the short count, such as being out of space, resolved itself before the next call, so it won't return an error. That is, there's no 100% reliable way of finding out why a partial `write` or `read` was short.

```
    if (nread == -1)
        EC_FAIL
    ec_neg1( close(fromfd) )
    ec_neg1( close(tofd) )
    return;

EC_CLEANUP_BGN
    (void)close(fromfd); /* can't use ec_neg1 here! */
    (void)close(tofd);
EC_CLEANUP_END
}
```

Realistically, this seems like too much trouble for regular files, although we will use a technique similar to this later on, when we do I/O to terminals and pipes, which sometimes simply require additional attempts. Perhaps this simple solution makes more sense when we know we're writing to a regular file:

```
if ((nwrite = write(tofd, buf, nread)) != nread) {
    if (nwrite != -1)
        errno = 0;
    EC_FAIL
}
```

or you might prefer to code it like this:

```
errno = 0;
ec_false( write(tofd, buf, nread) == nread )
```

We set `errno` to zero so that the error report will show an error (along with the line number), but not what could be a misleading error code. The one thing you definitely don't want to do is ignore the short count completely:

```
ec_neg1( write(tofd, buf, nread) ) /* wrong */
```

Here's a handy function, `writeall`, that encapsulates the "keep trying" approach in the `copy2` example. We'll use it later in this book (Sections 4.10.2 and 8.5) when we want to make sure everything got written. Note that it doesn't use the "ec" macros because it's meant as a direct replacement for `write`:

```
ssize_t writeall(int fd, const void *buf, size_t nbyte)
{
    ssize_t nwritten = 0, n;

    do {
        if ((n = write(fd, &((const char *)buf)[nwritten],
          nbyte - nwritten)) == -1) {
            if (errno == EINTR)
                continue;
```

```
            else
                return -1;
        }
        nwritten += n;
    } while (nwritten < nbyte);
    return nwritten;
}
```

We've treated an `EINTR` error specially, because it's not really an error. It just means that `write` was interrupted by a signal before it got to write anything, so we keep going. Signals and how to deal with interrupted system calls are dealt with more thoroughly in Section 9.1.4. There's an analogous `readall` in the next section.

2.10 `read` System Call

read—read from file descriptor

```
#include <unistd.h>

ssize_t read(
    int fd,            /* file descriptor */
    void *buf,         /* address to receive data */
    size_t nbytes      /* amount to read */
);
/* Returns number of bytes read or -1 on error (sets errno) */
```

The `read` system call is the opposite of `write`. It reads the `nbytes` bytes pointed to by `buf` from the open file represented by `fd`. The `read` starts at the current position of the file offset, and then the file offset is incremented by the number of bytes read. `read` returns the number of bytes read, or 0 on an end-of-file, or –1 on an error. It isn't affected by the `O_APPEND` flag.

Unlike `write`, the `read` system call can't very well cheat by passing along the data and then reading it later. If the data isn't already in the buffer cache (due to previous I/O), the process just has to wait for the kernel to get it from disk. Sometimes, the kernel tries to speed things up by noticing access patterns suggestive of sequential reading of consecutive disk blocks and then reading ahead to anticipate the process's needs. If the system is lightly loaded enough for data to remain in buffers a while, and if reads are sequential, read-ahead is quite effective.

There's the same problem in getting information about partial reads as there was with partial writes: Since a short count isn't an error, `errno` isn't valid, and you have to guess what the problem is. If you really need to read the whole amount, it's best to call `read` in a loop, which is what `readall` does (compare it to `writeall` in the previous section):

```
ssize_t readall(int fd, void *buf, size_t nbyte)
{
    ssize_t nread = 0, n;

    do {
        if ((n = read(fd, &((char *)buf)[nread], nbyte - nread)) == -1) {
            if (errno == EINTR)
                continue;
            else
                return -1;
        }
        if (n == 0)
            return nread;
        nread += n;
    } while (nread < nbyte);
    return nread;
}
```

We'll see `readall` in use in Section 8.5.

As with `write`, a `read` from a pipe, special file, or socket can block, in which case it may be interrupted by a signal (Section 9.1.4), causing it to return –1 with `errno` set to `EINTR`.

2.11 `close` System Call

close—close file descriptor

```
#include <unistd.h>

int close(
    int fd              /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The most important thing to know about the `close` system call is that it does practically nothing. It does *not* flush any kernel buffers; it just makes the file descriptor available for reuse. When the last file descriptor pointing to an open file description (Section 2.2.3) is closed, the open file description can be deleted as well. And, in turn, when the last open file description pointing to an in-memory i-node is deleted, the in-memory i-node can be deleted. There's one more step, too: If all the links to the actual i-node have been removed, the i-node on disk and all its data are deleted (explained in Section 2.6).

Since `close` doesn't flush the buffer cache, or even accelerate flushing, there's no need to close a file that you've written before reading it. It's guaranteed that you'll get the data you wrote. To say it another way, the kernel buffering in no way affects the semantics of `read`, `write`, `lseek`, or any other system call.

In fact, if the file descriptor isn't needed again, there's no requirement to call `close` at all, as the file descriptors will be reclaimed when the process terminates. It's still a good idea, however, to free-up kernel structures and to indicate to readers of your program that you're finished with the file. If you're consistently checking for errors, it also keeps you from accidentally using the file descriptor later on by mistake.

There are some additional side-effects of calling `close` that apply to pipes and other irregular files, but I'll talk about them when we get into the details of those types of files.

2.12 User Buffered I/O

2.12.1 User vs. Kernel Buffering

Recall from Chapter 1 that the UNIX file system is built on top of a block special file, and therefore all kernel I/O operations and all kernel buffering are in units of the block size. Anything is possible, but in practice it's always a multiple of 512, with numbers like 1024, 2048, and 4096 fairly typical. It's not necessarily the same on all devices, and it can even vary depending on how the disk partition was created. We'll get to figuring out what it is shortly.

Reads and writes in chunks equal to the block size that occur on a block-sized boundary are faster than any smaller unit. To demonstrate this we recompiled `copy2` from Section 2.9 with a user buffer size of 1 by making this change:

```
#define BUFSIZE 1
```

We timed the two versions of copy on a 4MB file[7] with the results shown in Table 2.2 (times are in seconds).

Table 2.2 Block-Sized I/O vs. Character-at-a-Time

Method	User	System	Total
512-byte buffer	0.07	0.58	0.65
1-byte buffer	18.43	204.98	223.41

(The times shown were on Linux; on FreeBSD they were about the same.)

User time is the time spent executing instructions in the user process. System time is the time spent executing instructions in the kernel on behalf of the process.

The performance penalty for I/O with regular files in such small chunks is so drastic that one simply never does it, unless the program is just for occasional and casual use, or the situation is quite unusual (reading a file backward, for example; see Section 2.13). Most of the penalty is simply from the larger number of system calls (by a factor of 512). To test the penalty from a poor choice of I/O buffer size, the tests were rerun with `BUFSIZE` set to 1024, a "good" size, and then 1100, a larger, but "bad" size. This time the difference was less on Linux, but still about 75% worse for 1100 (7.4 sec. total time vs. 4.25 sec.). On FreeBSD and Solaris the differences were much closer, only 10–20% off.

But the problem is that rarely is the block size a natural fit for what the program really wants to do. Lines of varying lengths and assorted structures are more typical. The solution is to pack the odd pieces of data into blocks in user space and to write a block only when it's full. On input, one does the reverse: unpacking the data from blocks as they're read. That is, one does *user* buffering in addition to *kernel* buffering. Since a piece of data can span a block boundary, it's somewhat tricky to program, and I'll show how to do it in the next section.

First, one nagging question: How do you know what the block size is? Well, you could use the actual number from the file system that's going to hold the files

7. The 1985 edition of this book used a *4000-byte* file, with similar times!

you're working with,[8] but some experimentation has shown that, within reason, larger numbers are better than smaller ones, and here's why: If the number is less than the actual block size, but divides evenly into it, the kernel can very efficiently pack the writes into a buffer and then schedule the buffer to be written when it's full. If the number is bigger and a multiple of the buffer size, it's still very efficient for the kernel to fit the data into buffers *and* there are fewer `write` system calls, which tends to be the overwhelming factor. A similar argument holds for `read`s.

So, by far the easiest thing to do is to use the macro `BUFSIZ`, defined by Standard C, which is what the standard I/O functions (e.g., `fputs`) use. It's not optimized for the actual file systems, being constant, but the experiments showed that that doesn't matter so much. If space is at a premium, you can even use 512 and you'll be fine.

2.12.2 Functions for User Buffering

It's convenient to use a set of functions that do reads, writes, seeks, and so on, in whatever units the caller wishes. These subroutines handle the buffering automatically and never stray from the block model. An exceptionally fine example of such a package is the so-called "standard I/O library," described in most books on C, such as [Har2002].

To show the principles behind a user-buffering package, I'll present a simplified one here that I call `BUFIO`. It supports reads and writes, but not seeks, in units of a single character. First, the header file bufio.h that users of the package must include (prototypes not shown):

```
typedef struct {
    int fd;                     /* file descriptor */
    char dir;                   /* direction: r or w */
    ssize_t total;              /* total chars in buf */
    ssize_t next;               /* next char in buf */
    unsigned char buf[BUFSIZ];  /* buffer */
} BUFIO;
```

Now for the implementation of the package (bufio.c):

8. You use the `stat` system call (Section 3.5.1).

```
BUFIO *Bopen(const char *path, const char *dir)
{
    BUFIO *b = NULL;
    int flags;

    switch (dir[0]) {
    case 'r':
        flags = O_RDONLY;
        break;
    case 'w':
        flags = O_WRONLY | O_CREAT | O_TRUNC;
        break;
    default:
        errno = EINVAL;
        EC_FAIL
    }
    ec_null( b = calloc(1, sizeof(BUFIO)) )
    ec_neg1( b->fd = open(path, flags, PERM_FILE) )
    b->dir = dir[0];
    return b;

EC_CLEANUP_BGN
    free(b);
    return NULL;
EC_CLEANUP_END
}

static bool readbuf(BUFIO *b)
{
    ec_neg1( b->total = read(b->fd, b->buf, sizeof(b->buf)) )
    if (b->total == 0) {
        errno = 0;
        return false;
    }
    b->next = 0;
    return true ;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool writebuf(BUFIO *b)
{
    ssize_t n, total;

    total = 0;
    while (total < b->next) {
        ec_neg1( n = write(b->fd, &b->buf[total], b->next - total) )
        total += n;
    }
```

```
    b->next = 0;
    return true ;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

int Bgetc(BUFIO *b)
{
    if (b->next >= b->total)
        if (!readbuf(b)) {
            if (errno == 0)
                return -1;
            EC_FAIL
        }
    return b->buf[b->next++];

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}

bool Bputc(BUFIO *b, int c)
{
    b->buf[b->next++] = c;
    if (b->next >= sizeof(b->buf))
        ec_false( writebuf(b) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool Bclose(BUFIO *b)
{
    if (b != NULL) {
        if (b->dir == 'w')
            ec_false( writebuf(b) )
        ec_neg1( close(b->fd) )
        free(b);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Finally, we recode our file-copy function to use the new package:

```
#include "bufio.h"

bool copy3(char *from, char *to)
{
    BUFIO *stfrom, *stto;
    int c;

    ec_null( stfrom = Bopen(from, "r") )
    ec_null( stto = Bopen(to, "w") )
    while ((c = Bgetc(stfrom)) != -1)
        ec_false( Bputc(stto, c) )
    if (errno != 0)
        EC_FAIL
    ec_false( Bclose(stfrom) )
    ec_false( Bclose(stto) )
    return true;

EC_CLEANUP_BGN
    (void)Bclose(stfrom);
    (void)Bclose(stto);
    return false;
EC_CLEANUP_END
}
```

You will notice a strong resemblance between BUFIO and a subset of the standard I/O library. Here's a version of copy using library functions:

```
bool copy4(char *from, char *to)
{
    FILE *stfrom, *stto;
    int c;

    ec_null( stfrom = fopen(from, "r") )
    ec_null( stto = fopen(to, "w") )
    while ((c = getc(stfrom)) != EOF)
        ec_eof( putc(c, stto) )
    ec_false( !ferror(stfrom) )
    ec_eof( fclose(stfrom) )
    ec_eof( fclose(stto) )
    return true;

EC_CLEANUP_BGN
    (void)fclose(stfrom);
    (void)fclose(stto);
    return false;
EC_CLEANUP_END
}
```

To see the great benefits of user buffering, Table 2.3 shows the times in seconds for the file copy using BUFIO, the standard I/O Library, and the straight-system-call method (shown in `copy2`).

Table 2.3 Comparison of Buffered and Unbuffered I/O

Method	User *	System	Total
BUFIO			
Solaris	1.00	0.51	1.51
Linux	1.00	0.28	1.28
FreeBSD	1.00	0.45	1.45
Standard I/O			
Solaris	0.57	0.24	0.81
Linux	11.32	0.15	11.48
FreeBSD	1.02	0.20	1.22
BUFSIZ buffer			
Solaris	0.00	0.52	0.52
Linux	0.00	0.23	0.23
FreeBSD	0.01	0.37	0.38

* All times in seconds and normalized within systems so that user BUFIO time on that system is 1.00.

With user buffering we've almost got the best of both worlds: We process the data as we like, even 1 byte at a time, yet we achieve a system time about the same as the `BUFSIZ` buffer method. So user buffering is definitely the right approach. Except for Linux, the standard I/O times are the best; it does what our `BUFIO` functions do, except faster and with more flexibility. The standard I/O user time for Linux (11.32) sticks out. A bit of research showed that while we used the gcc compiler for all the tests, Linux uses the gcc version of stdio.h, FreeBSD uses one based on BSD, and Solaris uses one based on System V. Looks like the gcc version needs some attention.[9]

The wide acceptance of the standard I/O library is ironic. The ability to do I/O in arbitrary units on regular files has always been one of the really notable features of the UNIX kernel, yet in practice, the feature is usually too inefficient to use.

2.13 `lseek` System Call

The `lseek`[10] system call just sets the file offset for use by the next `read`, `write`, or `lseek`. No actual I/O is performed, and no commands are sent to the disk controller (remember, there's normally a buffer cache in the way anyhow).

lseek—set and get file offset

```
#include <unistd.h>

off_t lseek(
    int fd,            /* file descriptor */
    off_t pos,         /* position */
    int whence         /* interpretation */
);
/* Returns new file offset or -1 on error (sets errno) */
```

The argument `whence` can be one of:

`SEEK_SET` The file offset is set to the `pos` argument.

`SEEK_CUR` The file offset is set to its current value plus the `pos` argument, which could be positive, zero, or negative. Zero is a way to find out the current file offset.

`SEEK_END` The file offset is set to the size of the file plus the `pos` argument which could be positive, zero, or negative. Zero is a way to set the file offset to the end of the file.

9. The problem is that it's checking for thread locks on each `putc`, but the other systems were smart enough to realize that we weren't multithreading. This may be fixed by the time you read this.

10. It was called "seek" back in the days before C had a `long` data type (this goes *way* back), and to get to a byte past 65,535 required a seek to the block, and then a second seek to the byte in the block. The extra letter was available for the new system call, as `creat` was one letter short.

The resulting file offset may have any non-negative value at all, even greater than the size of the file. If greater, the next `write` stretches the file to the necessary length, effectively filling the interval with bytes of zero. A `read` with the file offset set at or past the end generates a zero (end-of-file) return. A `read` of the stretched interval caused by a `write` past the end succeeds, and returns bytes of zero, as you would expect.

When a `write` beyond the end of a file occurs, most UNIX systems don't actually store the intervening blocks of zeros. Thus, it is possible for a disk with, say, 3,000,000 available blocks to contain files whose combined lengths are greater than 3,000,000 blocks. This can create a serious problem if files are backed up file-by-file and then restored; as the files have to be read to transfer them to the backup device, more than 3,000,000 blocks will be written and then read back in! Users who create many files with holes in them usually hear about it from their system administrator, unless the backup program is smart enough to recognize the holes.

Of all the possible ways to use `lseek`, three are the most popular. First, `lseek` may be used to seek to an absolute position in the file:

```
ec_neg1( lseek(fd, offset, SEEK_SET) )
```

Second, `lseek` may be used to seek to the end of the file:

```
ec_neg1( lseek(fd, 0, SEEK_END) )
```

Third, `lseek` may be used to find out where the file offset currently is:

```
off_t where;
ec_neg1( where = lseek(fd, 0, SEEK_CUR) )
```

Other ways of using `lseek` are much less common.

As I said in Section 2.8, most seeks done by the kernel are implicit rather than as a result of explicit calls to `lseek`. When `open` is called, the kernel seeks to the first byte. When `read` or `write` is called, the kernel increments the file offset by the number of bytes read or written. When a file is opened with the `O_APPEND` flag set, a seek to the end of the file precedes each write.

To illustrate the use of `lseek`, here is a function `backward` that prints a file backward, a line at a time. For example, if the file contains:

```
dog
bites
man
```

then `backward` will print:

```
man
bites
dog
```

Here is the code:

```
void backward(char *path)
{
    char s[256], c;
    int i, fd;
    off_t where;

    ec_neg1( fd = open(path, O_RDONLY) )
    ec_neg1( where = lseek(fd, 1, SEEK_END) )
    i = sizeof(s) - 1;
    s[i] = '\0';
    do {
        ec_neg1( where = lseek(fd, -2, SEEK_CUR) )
        switch (read(fd, &c, 1)) {
        case 1:
            if (c == '\n') {
                printf("%s", &s[i]);
                i = sizeof(s) - 1;
            }
            if (i <= 0) {
                errno = E2BIG;
                EC_FAIL
            }
            s[--i] = c;
            break;
        case -1:
            EC_FAIL
            break;
        default: /* impossible */
            errno = 0;
            EC_FAIL
        }
    } while (where > 0);
    printf("%s", &s[i]);
    ec_neg1( close(fd) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("backward");
EC_CLEANUP_END
}
```

There are two tricky things to observe in this function: First, since `read` implicitly seeks the file forward, we have to seek backward by *two* bytes to read the previous byte; to get things started we set the file offset to one byte past the end. Second, there's no such thing as a "beginning-of file" return from `read`, as there is an end-of-file return, so we have to watch the file offset (the variable `where`) and stop after we've read the first byte. Alternatively, we could wait until `lseek` tries to make the file pointer negative; however, this is unwise because the error code in this case (`EINVAL`) is also used to indicate other kinds of invalid arguments, and because as a general rule it's just better not to use error returns to implement an algorithm.

2.14 `pread` and `pwrite` System Calls

pread—read from file descriptor at offset

```
#include <unistd.h>

ssize_t pread(
     int fd,             /* file descriptor */
     void *buf,          /* address to receive data */
     size_t nbytes,      /* amount to read */
     off_t offset        /* where to read */
);
/* Returns number of bytes read or -1 on error (sets errno) */
```

pwrite—write to file descriptor at offset

```
#include <unistd.h>

ssize_t pwrite(
     int fd,             /* file descriptor */
     const void *buf,    /* data to write */
     size_t nbytes,      /* amount to write */
     off_t offset        /* where to write */
);
/* Returns number of bytes written or -1 on error (sets errno) */
```

`pread` and `pwrite` are almost exactly like `read` and `write` preceded with a call to `lseek`, except:

- The file offset isn't used, as the position for reading and writing is supplied explicitly by the `offset` argument.
- The file offset isn't set, either. In fact, it's completely ignored.

The `O_APPEND` flag (Section 2.8) does affect `pwrite`, making it behave exactly like `write` (as I said, the `offset` argument is ignored).

One call instead of two is convenient, but more importantly, `pread` and `pwrite` avoid the problem of the file offset being changed by another process or thread between an `lseek` and the following `read` or `write`. Recall that this could happen, as threads could use the same file descriptor, and processes could have duplicates that share a file offset, as explained in Section 2.2.3. This is the same problem that `O_APPEND` avoids (Section 2.8). As neither `pread` nor `pwrite` even use the file offset, they can't get the position wrong.

To show `pread` in action, here's another version of `backward` (from the previous section). It's more straightforward—easier to write and debug—because the funny business with calling `lseek` to decrement the file offset by –2 is gone:

```
void backward2(char *path)
{
    char s[256], c;
    int i, fd;
    off_t file_size, where;

    ec_neg1( fd = open(path, O_RDONLY) )
    ec_neg1( file_size = lseek(fd, 0, SEEK_END) )
    i = sizeof(s) - 1;
    s[i] = '\0';
    for (where = file_size - 1; where >= 0; where--)
        switch (pread(fd, &c, 1, where)) {
        case 1:
            if (c == '\n') {
                printf("%s", &s[i]);
                i = sizeof(s) - 1;
            }
            if (i <= 0) {
                errno = E2BIG;
                EC_FAIL
            }
            s[--i] = c;
            break;
        case -1:
            EC_FAIL
            break;
        default: /* impossible */
            errno = 0;
            EC_FAIL
        }
    printf("%s", &s[i]);
    ec_neg1( close(fd) )
    return;
```

```
EC_CLEANUP_BGN
    EC_FLUSH("backward2");
EC_CLEANUP_END
}
```

2.15 `readv` and `writev` System Calls

readv—scatter read

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

ssize_t readv(
    int fd,                    /* file descriptor */
    const struct iovec *iov,   /* vector of data buffers */
    int iovcnt                 /* number of elements */
);
/* Returns number of bytes read or -1 on error (sets errno) */
```

writev—gather write

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

ssize_t writev(
    int fd,                    /* file descriptor */
    const struct iovec *iov,   /* vector of data buffers */
    int iovcnt                 /* number of elements */
);
/* Returns number of bytes written or -1 on error (sets errno) */
```

`readv` and `writev` are like `read` and `write`, except that instead of using one data address, they can take data from or put data to several memory addresses at once. They're sometimes called *scatter* read and *gather* write. The data is still contiguous on the file, pipe, socket, or whatever `fd` is open to—it's the process's memory that can be scattered.

In other words, if you have three structures to be written, instead of using three `write`s to do it, you can write all three at once with `writev`; however, looking at the weird second argument, you can readily see that these functions are a pain to use, so they better be worth it, right? I'll get to that question shortly; I'll show how to use them first.

Before you call either function you have to set up the `iov` array (of `iovcnt` elements) so that each element contains a pointer to data and a size—in effect, the second and third arguments to an equivalent call of `read` or `write`. Think of `struct iovec` as being defined like this (it may have additional, nonstandard, members):

struct iovec—structure for readv and writev[11]

```
struct iovec {
    void *iov_base;    /* base address of data */
    size_t iov_len;    /* size of this piece */
};
```

In your program, you can declare an array of `struct iovec`s, or allocate memory with `malloc`, or whatever you like, as long as there's enough of it and it's properly initialized. The maximum number of elements you can have varies with the system, but it's always at least 16. On SUS systems the symbol `IOV_MAX`, if defined, tells you the actual maximum; if it's undefined you can call `sysconf` (Section 1.5.5) with an argument of `_SC_IOV_MAX`. But, really, for what `readv` and `writev` were designed for, 16 is a lot.

Here are some fragments from an example program that uses `writev` to write a header structure followed by two data structures. First, the structure declarations:

```
#define VERSION 506
#define STR_MAX 100

struct header {
    int h_version;
    int h_num_items;
} hdr, *hp;
struct data {
    enum {TYPE_STRING, TYPE_FLOAT} d_type;
    union {
        float d_val;
        char d_str[STR_MAX];
    } d_data;
} d1, d2, *dp;
struct iovec v[3];
```

(The version is just some number to identify this header type.)

Next, we initialize the header, two data structures, and the vector:

```
hdr.h_version = VERSION;
hdr.h_num_items = 2;
d1.d_type = TYPE_STRING;
strcpy(d1.d_data.d_str, "Some data to write");
d2.d_type = TYPE_FLOAT;
d2.d_data.d_val = 123.456;
```

11. And also `sendmsg` and `recvmsg`; see Section 8.6.3.

```
v[0].iov_base = (char *)&hdr; /* iov_base is sometimes char * */
v[0].iov_len = sizeof(hdr);
v[1].iov_base = (char *)&d1;
v[1].iov_len = sizeof(d1);
v[2].iov_base = (char *)&d2;
v[2].iov_len = sizeof(d2);
```

and then write all three structures with a single call to `writev`:

```
ec_neg1( n = writev(fd, v, sizeof(v) / sizeof(v[0])) )
```

Note, in the initialization of `iov_base`, the cast, which we ordinarily wouldn't need, as the SUS declares it as a `void` pointer. But FreeBSD (and probably other systems) define it a `char` pointer. The cast suits both.

To show that the data really is contiguous on the file, we read it back and print it out. We don't read the data into the original structures, but into one big anonymous buffer that we point into with pointers of the appropriate types (`hp` and `dp`). Note the call to `lseek` to rewind the file, which was opened `O_RDWR` (not shown):

```
ec_null( buf = malloc(n) )
ec_neg1( lseek(fd, 0, SEEK_SET) )
ec_neg1( read(fd, buf, n) )
hp = buf;
dp = (struct data *)(hp + 1);
printf("Version = %d\n", hp->h_version);
for (i = 0; i < hp->h_num_items; i++) {
    printf("#%d: ", i);
    switch (dp[i].d_type) {
    case TYPE_STRING:
        printf("%s\n", dp[i].d_data.d_str);
        break;
    case TYPE_FLOAT:
        printf("%.3f\n", dp[i].d_data.d_val);
        break;
    default:
        errno = 0;
        EC_FAIL
    }
}
ec_neg1( close(fd) )
free(buf);
```

This is the output:

```
Version = 506
#0: Some data to write
#1: 123.456
```

Aside from the obvious—being able to do in one system call what would otherwise take several—how much do readv and writev buy you? Table 2.4 shows some timing tests based on using writev to write 16 data items of 200 bytes each 50,000 times vs. substituting 16 calls to write for each call to writev. Thus, each test wrote a 160MB file. As the different UNIX systems were on different hardware, we've normalized the times to show the writev system time as 50 sec., which happens to be about what it was on the Linux machine.[12]

Table 2.4 Speed of writev vs. write

System Call	User	System
Solaris		
writev	1.35	50.00
write	8.45	67.61
Linux		
writev	.17	50.00
write	1.83	27.67
FreeBSD		
writev	.39	50.00
write	4.90	209.89

For Solaris and, especially, FreeBSD, it looks like writev really does win over write. On Linux the system time is actually *worse*. Reading through the Linux code to see why, it turns out that for files, Linux just loops through the vector, calling write for each element! For sockets, which are what readv and writev were designed for, Linux does much better, carrying the vector all the way down to some very low-level code. We don't have timing tests to report, but it's apparent from the code that on sockets writev is indeed worthwhile.

12. As you read the table, remember that you can't conclude anything about the relative speeds of, say, Linux vs. FreeBSD, because each was separately normalized. We're only interested in the degree to which each system provides an advantage of writev over write. Also, Linux may not be as bad as my results show; see www.basepath.com/aup/writer.htm.

2.16 Synchronized I/O

This section explains how to bypass kernel buffering, which was introduced in Section 2.12.1, and, if you don't want to go that far, how to control when buffers are flushed to disk.

2.16.1 Synchronized vs. Synchronous

In English the words "synchronized" and "synchronous" have nearly the same meaning, but in UNIX I/O they mean different things:

- *Synchronized* I/O means that a call to `write` (or its siblings, `pwrite` and `writev`) doesn't return until the data is flushed to the output device (disk, typically). Normally, as I indicated in Section 2.9, `write`, unsynchronized, returns leaving the data in the kernel buffer cache. If the computer crashes in the interim, the data will be lost.
- *Synchronous* I/O means that `read` (and its siblings) doesn't return until the data is available, and `write` (and its siblings) doesn't return until the data has been at least written to the kernel buffer, and all the way to the device if the I/O is also synchronized. The I/O calls we've described so far—`read`, `pread`, `readv`, `write`, `pwrite`, and `writev`—all operate synchronously.

Normally, therefore, UNIX I/O is *unsynchronized* and *synchronous*. Actually, reads and writes are asynchronous to some extent because of the way the buffer cache works; however, as soon as you make writes synchronized (forcing the buffers out on every call), actually waiting for a write to return would slow down the program too much, and that's when you really want writes to operate asynchronously. You want to say, "initiate this write and I'll go off and do something useful while it's writing and ask about what happened later when I feel like it." Reads, especially nonsequential ones, are always somewhat synchronized (the kernel can't fake it if the data's not in the cache); you'd rather not wait for them, either.

Getting `read` and `write` to be synchronized involves setting some `open` flags or executing system calls to flush kernel buffers, and it's the subject of this section. Asynchronous I/O uses a completely different group of system calls (e.g., `aio_write`) that I'll talk about in Section 3.9, where I'll distinguish between synchronized and synchronous even more sharply.

2.16.2 Buffer-Flushing System Calls

The oldest, best known, and least effective I/O-synchronizing call is `sync`:

sync—schedule buffer-cache flushing

```
#include <unistd.h>

void sync(void);
```

All `sync` does is tell the kernel to flush the buffer cache, which the kernel immediately adds to its list of things to do. But `sync` returns right away, so the flushing happens sometime later. You're still not sure when the buffers got flushed. `sync` is also heavy-handed—*all* the buffers that have been written are flushed, not just those associated with the files you care about.

The main use of this system call is to implement the `sync` command, run when UNIX is being shut down or before a removable device is unmounted. There are better choices for applications.

The next call, `fsync`, behaves, at a minimum, like `sync`, but just for those buffers written on behalf of a particular file. It's supported on SUS2 systems and on earlier systems if the option symbol `_POSIX_FSYNC` is defined.

fsync—schedule or force buffer-cache flushing for one file

```
#include <unistd.h>

int fsync(
    int fd            /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

We said "at a minimum" because if the synchronized I/O option (`_POSIX_SYNCHRONIZED_IO`) is supported (Section 1.5.4), the guarantee is much stronger: It doesn't return until the buffers have been physically written to the device controller, or until an error has been detected.[13]

Here's the function `option_sync_io` to check if the option is supported; see Section 1.5.4 for an explanation of what it's doing.

13. The data still might be only in the controller's cache, but it will get to the storage medium very rapidly from there, usually even if UNIX crashes or locks up, as long as the hardware is still powered and functional.

```
OPT_RETURN option_sync_io(const char *path)
{
#if _POSIX_SYNCHRONIZED_IO <= 0
    return OPT_NO;
#elif _XOPEN_VERSION >= 500 && !defined(LINUX)
    #if !defined(_POSIX_SYNC_IO)
        errno = 0;
        if (pathconf(path, _PC_SYNC_IO) == -1)
            if (errno == 0)
                return OPT_NO;
            else
                EC_FAIL
        else
            return OPT_YES;

    EC_CLEANUP_BGN
        return OPT_ERROR;
    EC_CLEANUP_END
    #elif _POSIX_SYNC_IO == -1
        return OPT_NO;
    #else
        return OPT_YES;
    #endif /* _POSIX_SYNC_IO */
#elif _POSIX_VERSION >= 199309L
    return OPT_YES;
#else
    errno = EINVAL;
    return OPT_ERROR;
#endif /* _POSIX_SYNCHRONIZED_IO */
}
```

The last syncing call, `fdatasync`, is a slightly faster form of `fsync` because it forces out only the actual data, not control information such as the file's modification time. For most critical applications, this is enough; the normal buffer-cache writing will take care of the control information later. `fdatasync` is only available as part of the synchronized I/O option (i.e., there's no `sync`-like behavior).

fdatasync—force buffer-cache flushing for one file's data

```
#include <unistd.h>

int fdatasync(
    int fd             /* file descriptor */
);
/* Returns 0 or -1 on error (sets errno) */
```

With the synchronized I/O option, both `fsync` and `fdatasync` can return genuine I/O errors, for which `errno` will be set to `EIO`.

One last point on these functions: If synchronized I/O is not supported, the implementation isn't required to do anything when you call `sync` or `fsync` (`fdatasync` won't be present); they might be no-ops. Think of them as mere requests. If the option is supported, however, the implementation is required to provide a high level of data integrity, although what actually happens depends on the device driver and the device.

2.16.3 `open` Flags for Synchronization

Typically you call `fsync` or `fdatasync` when it's necessary for the application to know that the data has been written, such as before reporting to the user that a database transaction has been committed. For even more critical applications, however, you can arrange, via `open` flags, for an implicit `fsync` or `fdatasync` on *every* `write`, `pwrite`, and `writev`.

I first mentioned these flags, `O_SYNC`, `O_DSYNC`, and `O_RSYNC` in the table in Section 2.4.4. They're only available if synchronized I/O is supported. Here's what they do:

`O_SYNC`	causes an implicit `fsync` (full-strength variety) after every `write`.
`O_DSYNC`	causes an implicit `fdatasync` after every `write`.
`O_RSYNC`	causes `read`, as well as `write`, synchronization; must be used with `O_SYNC` or `O_DSYNC`.

(When we say `write`, we also mean `pwrite` and `writev`, and similarly for `read`.)

The only thing that `O_RSYNC` really does is ensure that the access time in the i-node is updated in a synchronized manner, which means that with `O_DSYNC` it probably doesn't do anything at all. Even with `O_SYNC`, the access time is rarely critical enough to require synchronized updating. It's also possible on some systems for `O_RSYNC` to disable read-ahead.

Here's an example program using `O_DSYNC` that compares synchronized and unsynchronized writing:

```
#define SYNCREPS 5000
#define PATHNAME "tmp"
```

```
void synctest(void)
{
    int i, fd = -1;
    char buf[4096];

#if !defined(_POSIX_SYNCHRONIZED_IO) || _POSIX_SYNCHRONIZED_IO == -1
    printf("No synchronized I/O -- comparison skipped\n");
#else
    /* Create the file so it can be checked */
    ec_neg1( fd = open("tmp", O_WRONLY | O_CREAT, PERM_FILE) )
    ec_neg1( close(fd) )
    switch (option_sync_io(PATHNAME)) {
    case OPT_YES:
        break;
    case OPT_NO:
        printf("sync unsupported on %s\n", PATHNAME);
        return;
    case OPT_ERROR:
        EC_FAIL
    }

    memset(buf, 1234, sizeof(buf));

    ec_neg1( fd = open(PATHNAME, O_WRONLY | O_TRUNC | O_DSYNC) )
    timestart();
    for (i = 0; i < SYNCREPS; i++)
        ec_neg1( write(fd, buf, sizeof(buf)) )
    ec_neg1( close(fd) )
    timestop("synchronized");

    ec_neg1( fd = open(PATHNAME, O_WRONLY | O_TRUNC) )
    timestart();
    for (i = 0; i < SYNCREPS; i++)
        ec_neg1( write(fd, buf, sizeof(buf)) )
    ec_neg1( close(fd) )
    timestop("unsynchronized");
#endif
    return;

EC_CLEANUP_BGN
    EC_FLUSH("backward");
    (void)close(fd);
EC_CLEANUP_END
}
```

The functions `timestart` and `timestop` are used to get the timings; we showed them in Section 1.7.2. Note the call to `option_sync_io` to check the pathname, which required us to first create the file to be used. The options in the three open calls are a little unusual: The first uses `O_CREAT` without `O_TRUNC` because we just want to ensure that the file is there (we close it right away); the last two use `O_TRUNC` without `O_CREAT`, since we know it already exists.

On Linux, we got the results in Table 2.5 (Solaris times were similar; FreeBSD doesn't support the option).

Table 2.5 Synchronized vs. Unsynchronized I/O

Test	User*	System	Real
Synchronized	.03	5.57	266.45
Unsynchronized	.02	1.13	1.15

* Times in seconds.

(Real time is total elapsed time, including time waiting for I/O to complete.)

As you can see, synchronization is pretty costly. That's why you want to do it asynchronously, and I'll explain how in Section 3.9.[14]

2.17 `truncate` and `ftruncate` System Calls

truncate—truncate or stretch file by path

```
#include <unistd.h>

int truncate(
    const char *path,  /* pathname */
    off_t length       /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

ftruncate—truncate or stretch file by file descriptor

```
#include <unistd.h>

int ftruncate(
    int fd,            /* file descriptor */
    off_t length       /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

14. If this paragraph makes no sense to you, reread Section 2.16.1.

Stretching a file—making it bigger without actually writing lots of data—is easy, and I already showed how to do it: `lseek` to someplace beyond the end and `write` something. `truncate` and `ftruncate` can do that, but they're most useful in truncating (shrinking) a file, which was impossible on UNIX until they came along. (You used to have to write a completely new file and then rename it to the old name.)

Here's a rather contrived example. Note the unusual error checking for `write`, which I explained at the end of Section 2.9:

```
void ftruncate_test(void)
{
    int fd;
    const char s[] = "Those are my principles.\n"
      "If you don't like them I have others.\n"
      "\t--Groucho Marx\n";

    ec_neg1( fd = open("tmp", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
    errno = 0;
    ec_false( write(fd, s, sizeof(s)) == sizeof(s) )
    (void)system("ls -l tmp; cat tmp");
    ec_neg1( ftruncate(fd, 25) )
    (void)system("ls -l tmp; cat tmp");
    ec_neg1( close(fd) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("ftruncate_test");
EC_CLEANUP_END
}
```

Here's the output:

```
-rw-r--r--   1 marc     sysadmin      80 Oct  2 14:03 tmp
Those are my principles.
If you don't like them I have others.
        --Groucho Marx
-rw-r--r--   1 marc     sysadmin      25 Oct  2 14:03 tmp
Those are my principles.
```

`ftruncate` is also used to size shared memory, as explained in Section 7.14.

Exercises

2.1. Change `lock` in Section 2.4.3 to store the login name in the lock file (use `getlogin`; Section 3.5.2). Add an argument to be used when `lock` returns `false` that provides the login name of the user who has acquired the lock.

2.2. Write a program that opens a file for writing with the `O_APPEND` flag and then writes a line of text on the file. Run several concurrent processes executing this program to convince yourself that the text lines won't get intermixed. Then recode the program without `O_APPEND` and use `lseek` to seek to the end before each `write`. Rerun the concurrent processes to see if the text gets intermixed now.

2.3. Rerun the buffered I/O timing tests in Section 2.12.2 with buffer sizes of 2, 57, 128, 256, 511, 513, and 1024. Try some other interesting numbers if you wish. If you have access to several versions of UNIX, run the experiment on each version and assemble the results into a table.

2.4. Enhance the BUFIO package (Section 2.12.2) to open a file for both reading and writing.

2.5. Add a `Bseek` function to the BUFIO package.

2.6. Write a `cat` command that takes no options. For extra credit, implement as many options as you can. Use the SUS as a specification.

2.7. Same as Exercise 2.6, but for the `tail` command.

This page intentionally left blank