CHAPTER 2
File I/O

This and the subsequent three chapters cover files. Because so much of a Unix system is represented as files, these chapters discuss the crux of a Unix system. This chapter covers the basics of file I/O, detailing the system calls that comprise the simplest and most common ways to interact with files. The next chapter covers standard I/O from the standard C library; Chapter 4 continues the coverage with a treatment of the more advanced and specialized file I/O interfaces. Chapter 8 rounds out the discussion by addressing the topic of file and directory manipulation.

Before a file can be read from or written to, it must be opened. The kernel maintains a per-process list of open files, called the *file table*. This table is indexed via nonnegative integers known as *file descriptors* (often abbreviated *fds*). Each entry in the list contains information about an open file, including a pointer to an in-memory copy of the file's backing inode and associated metadata, such as the file position and access modes. Both user space and kernel space use file descriptors as unique cookies: opening a file returns a file descriptor, while subsequent operations (reading, writing, and so on) take the file descriptor as their primary argument.

File descriptors are represented by the C `int` type. Not using a special type is often considered odd, but is, historically, the Unix way. Each Linux process has a maximum number of files that it may open. File descriptors start at 0 and go up to one less than this maximum value. By default, the maximum is 1,024, but it can be configured as high as 1,048,576. Because negative values are not legal file descriptors, `-1` is often used to indicate an error from a function that would otherwise return a valid file descriptor.

Unless the process explicitly closes them, every process by convention has at least three file descriptors open: 0, 1, and 2. File descriptor 0 is *standard in* (*stdin*), file descriptor 1 is *standard out* (*stdout*), and file descriptor 2 is *standard error* (*stderr*). Instead of referencing these integers directly, the C library provides the preprocessor defines `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`, respectively. Normally, stdin is connected to the terminal's input device (usually the user's keyboard) and stdout and

stderr are connected to the terminal's display. Users can *redirect* these standard file descriptors and even pipe the output of one program into the input of another. This is how the shell implements redirections and pipes.

File descriptors can reference more than just regular files. Indeed, file descriptors are used for accessing device files and pipes, directories and futexes, FIFOs and sockets—following the everything-is-a-file philosophy, just about anything you can read or write is accessible via a file descriptor.

By default, a child process receives a copy of its parent's file table. The list of open files and their access modes, current file positions, and other metadata are the same, but a change in one process—say, the child closing a file—does not affect the other process's file table. While this is the typical behavior, as you'll see in Chapter 5, it is possible for the child and parent to share the parent's file table (as threads do).

Opening Files

The most basic method of accessing a file is via the `read()` and `write()` system calls. Before a file can be accessed, however, it must be opened via an `open()` or `creat()` call. Once done using the file, it should be closed using the system call `close()`.

The open() System Call

A file is opened and a file descriptor is obtained with the `open()` system call:

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

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

The `open()` system call maps the file given by the pathname `name` to a file descriptor, which it returns on success. The file position is to the start of the file (zero) and the file is opened for access according to the flags given by `flags`.

Flags for open()

The `flags` argument is the bitwise-OR of one or more flags. It must contain an access mode, which is one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. Respectively, these arguments request that the file be opened only for reading, only for writing, or for both reading and writing.

For example, the following code opens */home/kidd/madagascar* for reading:

```
int fd;

fd = open ("/home/kidd/madagascar", O_RDONLY);
```

```
if (fd == -1)
        /* error */
```

A file opened only for reading *cannot* be written to, and vice versa. The process issuing the `open()` system call must have sufficient permissions to obtain the access requested. For example, if a file is read-only to a given user, then a process owned by that user can open that file `O_RDONLY` but not `O_WRONLY` or `O_RDWR`.

On top of the access mode, the `flags` argument may be bitwise-ORed with zero or more of the following values, modifying the behavior of the open request:

`O_APPEND`
: The file will be opened in *append mode*. That is, before each write, the file position will be updated to point to the end of the file. This occurs even if another process has written to the file after the issuing process's last write, thereby changing the file position (see "Append Mode" on page 38).

`O_ASYNC`
: A signal (`SIGIO` by default) will be generated when the specified file becomes readable or writable. This flag is available only for FIFOs, pipes, sockets, and terminals, not for regular files.

`O_CLOEXEC`
: Sets the close-on-exec flag on the opened file. Upon executing a new process, the file will automatically be closed. This obviates needing to call `fcntl()` to set the flag and eliminates a race condition. This flag is only available in Linux kernel 2.6.23 and later.

`O_CREAT`
: If the file denoted by `name` does not exist, the kernel will create it. If the file already exists, this flag has no effect unless `O_EXCL` is also given.

`O_DIRECT`
: The file will be opened for direct I/O (see "Direct I/O" on page 45).

`O_DIRECTORY`
: If `name` is not a directory, the call to `open()` will fail. This flag is used internally by the `opendir()` library call.

`O_EXCL`
: When given with `O_CREAT`, this flag will cause the call to `open()` to fail if the file given by `name` already exists. This is used to prevent race conditions on file creation. If `O_CREAT` is not also provided, this flag has no meaning.

`O_LARGEFILE`
: The given file will be opened using 64-bit offsets, allowing the manipulation of files larger than two gigabytes. This is implied on 64-bit architectures.

`O_NOATIME+`
: The given file's last access time is not updated upon read. This flag is useful for backup, indexing, and similar programs that read every file on a system, to prevent significant write activity resulting from updating the inodes of each read file. This flag is only available in Linux kernel 2.6.8 and later.

`O_NOCTTY`
: If the given `name` refers to a terminal device (say, */dev/tty*), it will not become the process's controlling terminal, even if the process does not currently have a controlling terminal. This flag is not frequently used.

`O_NOFOLLOW`
: If `name` is a symbolic link, the call to `open()` will fail. Normally, the link is resolved, and the target file is opened. If other components in the given path are links, the call will still succeed. For example, if `name` is */etc/ship/plank.txt*, the call will fail if *plank.txt* is a symbolic link. It will succeed, however, if *etc* or *ship* is a symbolic link, so long as *plank.txt* is not.

`O_NONBLOCK`
: If possible, the file will be opened in nonblocking mode. Neither the `open()` call, nor any other operation will cause the process to block (sleep) on the I/O. This behavior may be defined only for FIFOs.

`O_SYNC`
: The file will be opened for synchronous I/O. No write operation will complete until the data has been physically written to disk; normal read operations are already synchronous, so this flag has no effect on reads. POSIX additionally defines `O_DSYNC` and `O_RSYNC`; on Linux, these flags are synonymous with `O_SYNC` (see "The O_SYNC Flag" on page 43).

`O_TRUNC`
: If the file exists, it is a regular file, and the given flags allow for writing, the file will be truncated to zero length. Use of `O_TRUNC` on a FIFO or terminal device is ignored. Use on other file types is undefined. Specifying `O_TRUNC` with `O_RDONLY` is also undefined, as you need write access to the file in order to truncate it.

For example, the following code opens for writing the file */home/teach/pearl*. If the file already exists, it will be truncated to a length of zero. Because the `O_CREAT` flag is not specified, if the file does not exist, the call will fail:

```
int fd;

fd = open ("/home/teach/pearl", O_WRONLY | O_TRUNC);
if (fd == -1)
        /* error */
```

Owners of New Files

Determining which user owns a new file is straightforward: the uid of the file's owner is the effective uid of the process creating the file.

Determining the owning group is more complicated. The default behavior is to set the file's group to the effective gid of the process creating the file. This is the System V behavior, which is the behavioral model for much of Linux, and thus the standard Linux *modus operandi*.

To be difficult, however, BSD defined its own behavior: the file's group is set to the gid of the parent directory. This behavior is available on Linux via a mount-time option[1]—it is also the behavior that will occur on Linux by default if the file's parent directory has the set group ID (setgid) bit set. Although most Linux systems will use the System V behavior (where new files receive the gid of the creating process), the possibility of the BSD behavior (where new files receive the gid of the parent directory) implies that code that strongly cares about a newly created file's owning group needs to manually set said group via the `fchown()` system call (see Chapter 8).

Thankfully, you often need not care what group owns a file.

Permissions of New Files

Both of the previously given forms of the `open()` system call are valid. The `mode` argument is ignored unless a file is created; it is required if `O_CREAT` is given. If you forget to provide the `mode` argument when using `O_CREAT`, the results are undefined and often quite ugly—so don't forget!

When a file is created, the `mode` argument provides the permissions of the newly created file. The mode is not checked on this particular open of the file, so you can perform operations that contradict the assigned permissions, such as opening the file for writing but assigning the file read-only permissions.

The `mode` argument is the familiar Unix permission bitset, such as octal `0644` (owner can read and write, everyone else can only read). Technically speaking, POSIX says the exact values are implementation-specific, allowing different Unix systems to lay out the permission bits however they desired. Every Unix system, however, has implemented the permission bits in the same way. Thus, while technically nonportable, specifying `0644` or `0700` will have the same effect on any system you are likely to come across.

Nonetheless, to compensate for the nonportability of bit positions in the mode, POSIX introduced the following set of constants that may be binary-ORed together and supplied for the `mode` argument:

1. The mount options `bsdgroups` or `sysvgroups`.

`S_IRWXU`
: Owner has read, write, and execute permission.

`S_IRUSR`
: Owner has read permission.

`S_IWUSR`
: Owner has write permission.

`S_IXUSR`
: Owner has execute permission.

`S_IRWXG`
: Group has read, write, and execute permission.

`S_IRGRP`
: Group has read permission.

`S_IWGRP`
: Group has write permission.

`S_IXGRP`
: Group has execute permission.

`S_IRWXO`
: Everyone else has read, write, and execute permission.

`S_IROTH`
: Everyone else has read permission.

`S_IWOTH`
: Everyone else has write permission.

`S_IXOTH`
: Everyone else has execute permission.

The actual permission bits that hit the disk are determined by binary-ANDing the `mode` argument with the complement of the user's *file creation mask* (*umask*). The umask is a process-specific attribute that is usually set via the login shell but is modifiable by the `umask()` call, allowing the user to modify the permissions placed on newly created files and directories. The bits in the umask are turned *off* in the `mode` argument given to `open()`. Thus, the usual umask of `022` would cause a `mode` argument of `0666` to become `0644`. As a system programmer, you normally do not take into consideration the umask when setting permissions—the umask exists to allow users to limit the permissions that their programs set on new files.

As an example, the following code opens the file given by `file` for writing. If the file does not exist, assuming a umask of `022`, it is created with the permissions `0644` (even though the `mode` argument specifies `0664`). If it does exist, it is truncated to zero length:

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC,
           S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IROTH);
if (fd == -1)
        /* error */
```

Trading portability (in theory at least) for readability, we could have written the following, to identical effect :

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, 0664);
if (fd == -1)
        /* error */
```

The creat() Function

The combination of `O_WRONLY | O_CREAT | O_TRUNC` is so common that a system call exists to provide just that behavior:

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

int creat (const char *name, mode_t mode);
```

Yes, this function's name is missing an *e*. Ken Thompson, the creator of Unix, once joked that the missing letter was his largest regret in the design of Unix.

The following typical `creat()` call,

```
int fd;

fd = creat (filename, 0644);
if (fd == -1)
        /* error */
```

is identical to

```
int fd;

fd = open (filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd == -1)
        /* error */
```

On most Linux architectures,[2] `creat()` is a system call, even though it can be implemented in user space as simply:

```
int creat (const char *name, int mode)
{
        return open (name, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

This duplication is a historic relic from when `open()` had only two arguments. Today, `creat()` remains a system call for backward compatibility. New architectures can implement `creat()` as a library call invoking `open()` as shown.

Return Values and Error Codes

Both `open()` and `creat()` return a file descriptor on success. On error, both return `-1`, and set `errno` to an appropriate error value (Chapter 1 discussed `errno` and listed the potential error values). Handling an error on file open is not complicated, as generally there will have been few or no steps performed prior to the open that need to be undone. A typical response would be prompting the user for a different filename or simply terminating the program.

Reading via read()

Now that you know how to open a file, let's look at how to read it. In the following section, we will examine writing.

The most basic—and common—mechanism used for reading is the `read()` system call, defined in POSIX.1:

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t len);
```

Each call reads up to `len` bytes into the memory pointed at by `buf` from the current file offset of the file referenced by `fd`. On success, the number of bytes written into `buf` is returned. On error, the call returns `-1` and sets `errno`. The file position is advanced by the number of bytes read from `fd`. If the object represented by `fd` is not capable of seeking (for example, a character device file), the read always occurs from the "current" position.

Basic usage is simple. This example reads from the file descriptor `fd` into `word`. The number of bytes read is equal to the size of the `unsigned long` type, which (for Linux

2. Recall that system calls are defined on a per-architecture basis. Thus, while x86-64 has a `creat()` system call, Alpha does not. You can use `creat()` on any architecture, of course, but it may be a library function instead of having its own system call.

at least) is 4 bytes on 32-bit Linux systems, and 8 bytes on 64-bit systems. On return, `nr` contains the number of bytes read, or `-1` on error:

```
unsigned long word;
ssize_t nr;

/* read a couple bytes into 'word' from 'fd' */
nr = read (fd, &word, sizeof (unsigned long));
if (nr == -1)
        /* error */
```

There are two problems with this naïve implementation: the call might return without reading all `len` bytes, and it could produce certain actionable errors that this code does not check for and handle. Code such as this, unfortunately, is very common. Let's see how to improve it.

Return Values

It is legal for `read()` to return a positive nonzero value less than `len`. This can happen for a number of reasons: less than `len` bytes may have been available, the system call may have been interrupted by a signal, the pipe may have broken (if `fd` references a pipe), and so on.

The possibility of a return value of `0` is another consideration when using `read()`. The `read()` system call returns `0` to indicate *end-of-file* (*EOF*); in this case, of course, no bytes were read. EOF is not considered an error (and hence is not accompanied by a `-1` return value); it simply indicates that the file position has advanced past the last valid offset in the file and thus there is nothing else to read. If, however, a call is made for `len` bytes, but no bytes are available for reading, the call will *block* (sleep) until the bytes become available (assuming the file descriptor was not opened in nonblocking mode; see "Nonblocking Reads" on page 35). Note that this is different from returning EOF. That is, there is a difference between "no data available" and "end of data." In the EOF case, the end of the file was reached. In the case of blocking, the read is waiting for more data—say, in the case of reading from a socket or a device file.

Some errors are recoverable. For example, if a call to `read()` is interrupted by a signal before any bytes are read, it returns `-1` (a `0` could be confused with EOF), and `errno` is set to `EINTR`. In that case, you can and should resubmit the read.

Indeed, a call to `read()` can result in many possibilities:

- The call returns a value equal to `len`. All `len` read bytes are stored in `buf`. The results are as intended.
- The call returns a value less than `len`, but greater than zero. The read bytes are stored in `buf`. This can occur because a signal interrupted the read midway; an error occurred in the middle of the read; more than zero, but less than `len` bytes' worth

of data was available; or EOF was reached before `len` bytes were read. Reissuing the read (with correspondingly updated `buf` and `len` values) will read the remaining bytes into the rest of the buffer or indicate the cause of the problem.

- The call returns `0`. This indicates EOF. There is nothing to read.
- The call blocks because no data is currently available. This won't happen in nonblocking mode.
- The call returns `-1`, and `errno` is set to `EINTR`. This indicates that a signal was received before any bytes were read. The call can be reissued.
- The call returns `-1`, and `errno` is set to `EAGAIN`. This indicates that the read would block because no data is currently available, and that the request should be reissued later. This happens only in nonblocking mode.
- The call returns `-1`, and `errno` is set to a value other than `EINTR` or `EAGAIN`. This indicates a more serious error. Simply reissuing the read is unlikely to succeed.

Reading All the Bytes

These possibilities imply that the previous trivial, simplistic use of `read()` is not suitable if you want to handle all errors and actually read all `len` bytes (at least up to an EOF). To do that, you need a loop, and a handful of conditional statements:

```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0) {
        if (ret == -1) {
                if (errno == EINTR)
                        continue;
                perror ("read");
                break;
        }

        len -= ret;
        buf += ret;
}
```

This snippet handles all five conditions. The loop reads `len` bytes from the current file position of `fd` into `buf`, which of course must be at least `len` bytes in length. It continues reading until it reads all `len` bytes, or until EOF is reached. If more than zero, but less than `len` bytes are read, `len` is reduced by the amount read, `buf` is increased by the amount read, and the call is reissued. If the call returns `-1`, and `errno` equals `EINTR`, the call is reissued without updating the parameters. If the call returns `-1`, and `errno` is set to anything else, `perror()` is called to print a description to standard error and the loop terminates.

Partial reads are not only legal, but common. Innumerable bugs derive from programmers not properly checking for and handling short read requests. Do not add to the list!

Nonblocking Reads

Sometimes, programmers do not want a call to `read()` to block when there is no data available. Instead, they prefer that the call return immediately, indicating that no data is available. This is called *nonblocking I/O*; it allows applications to perform I/O, potentially on multiple files, without ever blocking, and thus missing data available in another file.

Consequently, an additional `errno` value is worth checking: `EAGAIN`. As discussed previously, if the given file descriptor was opened in nonblocking mode (if `O_NONBLOCK` was given to `open()`; see "Flags for open()" on page 26) and there is no data to read, the `read()` call will return `-1` and set `errno` to `EAGAIN` instead of blocking. When performing nonblocking reads, you must check for `EAGAIN` or risk confusing a serious error with the mere lack of data. For example, you might use code like the following:

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
        if (errno == EINTR)
                goto start; /* oh shush */
        if (errno == EAGAIN)
                /* resubmit later */
        else
                /* error */
}
```

Handling the `EAGAIN` case like we did the `EINTR` case (with a `goto start`) would make little sense. We might as well not have used nonblocking I/O. The point of nonblocking I/O is to catch the `EAGAIN` and do other, useful work.

Other Error Values

The other error codes refer to programming errors or (for `EIO`) low-level problems. Possible `errno` values after a failure on `read()` include:

`EBADF`
: The given file descriptor is invalid or is not open for reading.

`EFAULT`
: The pointer provided by `buf` is not inside the calling process's address space.

`EINVAL`
: The file descriptor is mapped to an object that does not allow reading.

`EIO`
: A low-level I/O error occurred.

Size Limits on read()

The `size_t` and `ssize_t` types are mandated by POSIX. The `size_t` type is used for storing values used to measure size in bytes. The `ssize_t` type is a signed version of `size_t` (the negative values are used to connote errors). On 32-bit systems, the backing C types are usually `unsigned int` and `int`, respectively. Because the two types are often used together, the potentially smaller range of `ssize_t` places a limit on the range of `size_t`.

The maximum value of a `size_t` is `SIZE_MAX`; the maximum value of an `ssize_t` is `SSIZE_MAX`. If `len` is larger than `SSIZE_MAX`, the results of the call to `read()` are undefined. On most Linux systems, `SSIZE_MAX` is `LONG_MAX`, which is 2,147,483,647 on a 32-bit machine. That is relatively large for a single read but nonetheless something to keep in mind. If you use the previous read loop as a generic super read, you might want to do something like this:

```
if (len > SSIZE_MAX)
        len = SSIZE_MAX;
```

A call to `read()` with a `len` of zero results in the call returning immediately with a return value of `0`.

Writing with write()

The most basic and common system call used for writing is `write()`. `write()` is the counterpart of `read()` and is also defined in POSIX.1:

```
#include <unistd.h>

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

A call to `write()` writes up to `count` bytes starting at `buf` to the current position of the file referenced by the file descriptor `fd`. Files backed by objects that do not support seeking (for example, character devices) always write starting at the "head."

On success, the number of bytes written is returned, and the file position is updated in kind. On error, `-1` is returned and `errno` is set appropriately. A call to `write()` can return `0`, but this return value does not have any special meaning; it simply implies that zero bytes were written.

As with `read()`, the most basic usage is simple:

```
const char *buf = "My ship is solid!";
ssize_t nr;

/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
        /* error */
```

But again, as with `read()`, this usage is not quite right. Callers also need to check for the possible occurrence of a partial write:

```
unsigned long word = 1720;
size_t count;
ssize_t nr;

count = sizeof (word);
nr = write (fd, &word, count);
if (nr == -1)
        /* error, check errno */
else if (nr != count)
        /* possible error, but 'errno' not set */
```

Partial Writes

The `write()` system call is less likely to return a partial write than the `read()` system call is to return a partial read. Also, there is no EOF condition for a `write()` system call. For regular files, `write()` is guaranteed to perform the entire requested write, unless an error occurs.

Consequently, for regular files, you do not need to perform writes in a loop. However, for other file types—say, sockets—a loop may be required to guarantee that you *really* write out all of the requested bytes. Another benefit of using a loop is that a second call to `write()` may return an error revealing what caused the first call to perform only a partial write (although, again, this situation is not very common). Here is an example:

```
ssize_t ret, nr;

while (len != 0 && (ret = write (fd, buf, len)) != 0) {
        if (ret == -1) {
                if (errno == EINTR)
                        continue;
                perror ("write");
                break;
```

```
        }

        len -= ret;
        buf += ret;
    }
```

Append Mode

When `fd` is opened in append mode (via `O_APPEND`), writes do not occur at the file descriptor's current file position. Instead, they occur at the current end of the file.

For example, assume that two processes intend to write to the end of the same file. This is common: consider a log of events shared among many processes. At the start, their file positions are correctly set to the end of the file. The first process writes to the end of the file. Without append mode, once the second process attempts the same, it will end up writing not to the end of the file, but to the offset that *was* the end of the file, before the data that the first process wrote. This means that multiple processes can never append to the same file without explicit synchronization between them because they will encounter race conditions.

Append mode avoids this problem. It ensures that the file position is always set to the end of the file so all writes always append, even when there are multiple writers. You can think of it as an atomic update to the file position preceding each write request. The file position is then updated to point at the end of the newly written data. This will not matter to the next call to `write()`, as it updates the file position automatically, but it might matter if you next call `read()` for some odd reason.

Append mode makes a lot of sense for certain tasks, such as the aforementioned writing out of log files, but little sense for much else.

Nonblocking Writes

When `fd` is opened in nonblocking mode (via `O_NONBLOCK`), and the write as issued would normally block, the `write()` system call returns `-1` and sets `errno` to `EAGAIN`. The request should be reissued later. Generally, this does not occur with regular files.

Other Error Codes

Other notable `errno` values include:

`EBADF`
: The given file descriptor is not valid or is not open for writing.

`EFAULT`
: The pointer provided by `buf` points outside of the process's address space.

`EFBIG`
: The write would have made the file larger than per-process maximum file, or internal implementation, limits.

`EINVAL`
: The given file descriptor is mapped to an object that is not suitable for writing.

`EIO`
: A low-level I/O error occurred.

`ENOSPC`
: The filesystem backing the given file descriptor does not have sufficient space.

`EPIPE`
: The given file descriptor is associated with a pipe or socket whose reading end is closed. The process will also receive a `SIGPIPE` signal. The default action for the `SIGPIPE` signal is to terminate the receiving process. Therefore, processes receive this `errno` value only if they explicitly ask to ignore, block, or handle this signal.

Size Limits on write()

If `count` is larger than `SSIZE_MAX`, the results of the call to `write()` are undefined.

A call to `write()` with a `count` of zero results in the call returning immediately with a return value of `0`.

Behavior of write()

When a call to `write()` returns, the kernel has copied the data from the supplied buffer into a kernel buffer, but there is no guarantee that the data has been written out to its intended destination. Indeed, `write` calls return much too fast for that to be the case. The disparity in performance between processors and hard disks would make such behavior painfully obvious.

Instead, when a user-space application issues a `write()` system call, the Linux kernel performs a few checks and then simply copies the data into a buffer. Later, in the background, the kernel gathers up all of the *dirty buffers*, which are buffers that contain data newer than what is on disk, sorts them optimally, and writes them out to disk (a process known as *writeback*). This allows `write` calls to occur relatively fast, returning almost immediately. It also allows the kernel to defer writes to more idle periods and batch many writes together.

The delayed writes do not change POSIX semantics. For example, if a read is issued for a piece of just-written data that lives in a buffer and is not yet on disk, the request will be satisfied from the buffer and not cause a read from the "stale" data on disk. This behavior further improves performance, as the read is satisfied from an in-memory

cache without having to go to disk. The read and write requests interleave as intended, and the results are as expected—that is, if the system does not crash before the data makes it to disk! Even though an application may believe that a write has occurred successfully, in the event of a system crash, the data will never make it to disk.

Another issue with delayed writes is the inability to enforce *write ordering*. Although an application may take care to order its write requests in such a way that they hit the disk in a specific order, the kernel will reorder the write requests as it sees fit, primarily for reasons of performance. This is normally a problem only if the system crashes, as eventually all of the buffers are written back and the final state of the file is as intended. The vast majority of applications are not actually concerned with write ordering. Databases are a rare use case concerned with ordering; they want to ensure that write operations are ordered such that the database is never in an inconsistent state.

A final problem with delayed writes is the reporting of certain I/O errors. Any I/O error that occurs during writeback—say, a physical drive failure—cannot be reported back to the process that issued the write request. Indeed, dirty buffers inside the kernel are not associated with processes at all. Multiple processes may have dirtied the data contained in a single buffer, and processes may exit after writing data to a buffer but before that data is written back to disk. Besides, how would you communicate to a process that a write failed *ex post facto*?

Given these potential issues, the kernel attempts to minimize the risks of deferred writes. To ensure that data is written out in a timely manner, the kernel institutes a *maximum buffer age* and writes out all dirty buffers before they mature past the given value. Users can configure this value via */proc/sys/vm/dirty_expire_centisecs*. The value is specified in centiseconds (one hundredths of a second).

It is also possible to force the writeback of a given file's buffer, or even to make all writes synchronous. These topics are discussed in the next section, "Synchronized I/O".

Later in this chapter, "Kernel Internals" on page 62 will cover the Linux kernel's buffer writeback subsystem in depth.

Synchronized I/O

Although synchronizing I/O is an important topic, the issues associated with delayed writes should not be overstated. Buffering writes provides a *significant* performance improvement, and consequently, any operating system even halfway deserving the mark "modern" implements delayed writes via buffers. Nonetheless, there are times when applications want to control when data hits the disk. For those uses, the Linux kernel provides a handful of options that allow performance to be traded for synchronized operations.

fsync() and fdatasync()

The simplest method of ensuring that data has reached the disk is via the `fsync()` system call, standardized by POSIX.1b:

```
#include <unistd.h>

int fsync (int fd);
```

A call to `fsync()` ensures that all dirty data associated with the file mapped by the file descriptor `fd` are written back to disk. The file descriptor `fd` must be open for writing. The call writes back both data and metadata, such as creation timestamps and other attributes contained in the inode. It will not return until the hard drive says that the data and metadata are on the disk.

In the presence of hard disks with write caches, it is not possible for `fsync()` to know whether the data is physically on the disk. The hard drive can report that the data was written, but the data may in fact reside in the drive's write cache. Fortunately, data in a hard disk's cache should be committed to the disk in short order.

Linux also provides the system call `fdatasync()`:

```
#include <unistd.h>

int fdatasync (int fd);
```

This system call does the same thing as `fsync()`, except that it only flushes data and metadata required to properly access the file in the future. For example, a call to `fdatasync()` will flush a file's size, since you need that to read the file correctly. The call does not guarantee that nonessential metadata is synchronized to disk, and is therefore potentially faster. Most use cases do not consider metadata such as the file modification timestamp part of their essential transaction and thus `fdatasync()` is sufficient for their needs and potentially faster.

`fsync()` always results in at least *two* I/O operations: one to write back the modified data and one to update the inode's modification timestamp. Because the inode and the file's data may not be adjacent on disk—and thus require an expensive seek operation—and most concerns over proper transaction ordering do not include metadata that isn't essential to properly access the file in the future (such as the modification timestamp), `fdatasync()` is an easy performance win.

Both functions are used the same way, which is very simple:

```
int ret;

ret = fsync (fd);
```

```
if (ret == -1)
        /* error */
```

or when using `fdatasync()`

```
int ret;

/* same as fsync, but won't flush non-essential metadata */
ret = fdatasync (fd);
if (ret == -1)
        /* error */
```

Neither function guarantees that any updated directory entries containing the file are synchronized to disk. This implies that if a file's link has recently been updated, the file's data may successfully reach the disk but not the associated directory entry, rendering the file unreachable. To ensure that any updates to the directory entry are also committed to disk, `fsync()` must also be called on a file descriptor opened against the file's directory.

Return values and error codes

On success, both calls return `0`. On failure, both calls return `-1` and set `errno` to one of the following three values:

`EBADF`
: The given file descriptor is not a valid file descriptor open for writing.

`EINVAL`
: The given file descriptor is mapped to an object that does not support synchronization.

`EIO`
: A low-level I/O error occurred during synchronization. This represents a real I/O error, and is often the place where such errors are caught.

In some versions of Linux, a call to `fsync()` may fail because `fsync()` is not implemented by the backing filesystem, even when `fdatasync()` is implemented. Paranoid applications may want to try `fdatasync()` if `fsync()` returns `EINVAL`. For example:

```
if (fsync (fd) == -1) {
        /*
         * We prefer fsync(), but let's try fdatasync()
         * if fsync() fails, just in case.
         */
        if (errno == EINVAL) {
                if (fdatasync (fd) == -1)
                        perror ("fdatasync");
        } else
                perror ("fsync");
}
```

Because POSIX requires `fsync()`, but labels `fdatasync()` as optional, the `fsync()` system call should always be implemented for regular files on any of the common Linux filesystems. Atypical file types (perhaps those in which there is no metadata to synchronize) or strange filesystems may implement only `fdatasync()`, however.

sync()

Less optimal, but wider in scope, the old-school `sync()` system call is provided for synchronizing *all* buffers to disk:

```
#include <unistd.h>

void sync (void);
```

The function has no parameters and no return value. It always succeeds and, upon return, all buffers—both data and metadata—are guaranteed to reside on disk.[3]

The standards do not require `sync()` to wait until all buffers are flushed to disk before returning; they require only that the call initiates the process of committing all buffers to disk. For this reason, it is often recommended to invoke `sync()` multiple times to ensure that all data is safely on disk. Linux, however, *does* wait until all buffers are committed. Therefore, a single `sync()` is sufficient.

The only real use for `sync()` is in the implementation of the *sync* utility. Applications should use `fsync()` and `fdatasync()` to commit to disk the data of only the requisite file descriptors. Note that `sync()` may take several minutes or longer to complete on a busy system.

The O_SYNC Flag

The `O_SYNC` flag may be passed to `open()`, indicating that all I/O on the file should be synchronized:

```
int fd;

fd = open (file, O_WRONLY | O_SYNC);
if (fd == -1) {
        perror ("open");
        return -1;
}
```

Read requests are always synchronized. If they weren't, the validity of the read data in the supplied buffer would be unknown. However, as discussed previously, calls to `write()` are normally not synchronized. There is no relation between the call returning

3. With the same caveat as before: the hard drive may lie and inform the kernel that the buffers reside on disk when they actually are still in the disk's cache.

and the data being committed to disk. The `O_SYNC` flag forces the relationship, ensuring that calls to `write()` perform synchronized I/O.

One way of looking at `O_SYNC` is that it forces an implicit `fsync()` after each `write()` operation, before the call returns. These are indeed the semantics provided, although the Linux kernel implements `O_SYNC` a bit more efficiently.

`O_SYNC` results in slightly worse *user* and *kernel times* (time spent in user and kernel space, respectively) for write operations. Moreover, depending on the size of the file being written, `O_SYNC` can cause an increase in total elapsed time of one or two orders of magnitude because all *I/O wait time* (time spent waiting for I/O to complete) is incurred by the process. The increase in cost is huge, so synchronized I/O should be used only after exhausting all possible alternatives.

Normally, applications that need guarantees that write operations have hit the disk use `fsync()` or `fdatasync()`. These tend to incur less cost than `O_SYNC`, as they can be called less often (i.e., only after certain critical operations have completed).

O_DSYNC and O_RSYNC

POSIX defines two other synchronized-I/O-related `open()` flags: `O_DSYNC` and `O_RSYNC`. On Linux, these flags are defined to be synonymous with `O_SYNC`; they provide the same behavior.

The `O_DSYNC` flag specifies that only normal data be synchronized after each write operation, not metadata. Think of it as causing an implicit `fdatasync()` after each write request. Because `O_SYNC` provides stronger guarantees, aliasing `O_DSYNC` to it involves no loss in functionality; there's only a potential performance loss from the stronger requirements provided by `O_SYNC`.

The `O_RSYNC` flag specifies the synchronization of read requests as well as write requests. It must be used with one of `O_SYNC` or `O_DSYNC`. As mentioned earlier, reads are already synchronized—they do not return until they have something to give the user, after all. The `O_RSYNC` flag stipulates that any side effects of a read operation be synchronized, too. This means that metadata updates resulting from a read must be written to disk before the call returns. In practical terms, this requirement most likely means only that the file access time must be updated in the on-disk copy of the inode before the call to `read()` returns. Linux defines `O_RSYNC` to be the same as `O_SYNC`, although this does not make much sense (one is not a subset of the other as with `O_DSYNC` and `O_SYNC`). There is currently no way in Linux to obtain the proper behavior of `O_RSYNC`; the closest a developer can come is invoking `fdatasync()` after each `read()` call. This behavior is rarely needed, though.

Direct I/O

The Linux kernel, like any modern operating system kernel, implements a complex layer of caching, buffering, and I/O management between devices and applications (see "Kernel Internals" on page 62). A high-performance application may wish to bypass this layer of complexity and perform its own I/O management. Rolling your own I/O system is usually not worth the effort, though, and in fact the tools available at the operating-system level are likely to achieve much better performance than those available at the application level. Still, database systems often prefer to perform their own caching and want to minimize the presence of the operating system as much as feasible.

Providing the `O_DIRECT` flag to `open()` instructs the kernel to minimize the presence of I/O management. When this flag is provided, I/O will initiate directly from user-space buffers to the device, bypassing the page cache. All I/O will be synchronous; operations will not return until completed.

When performing direct I/O, the request length, buffer alignment, and file offsets must all be integer multiples of the underlying device's sector size—generally, this is 512 bytes. Before the 2.6 Linux kernel, this requirement was stricter: in 2.4, everything must be aligned on the filesystem's logical block size (often 4 KB). To remain compatible, applications should align to the larger (and potentially less convenient) logical block size.

Closing Files

After a program has finished working with a file descriptor, it can unmap the file descriptor from the associated file via the `close()` system call:

```
#include <unistd.h>

int close (int fd);
```

A call to `close()` unmaps the open file descriptor `fd` and disassociates the file from the process. The given file descriptor is then no longer valid, and the kernel is free to reuse it as the return value to a subsequent `open()` or `creat()` call. A call to `close()` returns `0` on success. On error, it returns `-1` and sets `errno` appropriately. Usage is simple:

```
if (close (fd) == -1)
        perror ("close");
```

Note that closing a file has no bearing on when the file is flushed to disk. To ensure that a file is committed to disk before closing it, an application needs to make use of one of the synchronization options discussed earlier in "Synchronized I/O" on page 40.

Closing a file does have some side effects, though. When the last open file descriptor referring to a file is closed, the data structure representing the file inside the kernel is freed. When this data structure is freed, it unpins the in-memory copy of the inode associated with the file. If nothing else is pinning the inode, it too may be freed from

memory (it may stick around because the kernel caches inodes for performance reasons, but it need not). If a file has been unlinked from the disk but was kept open before it was unlinked, it is not physically removed until it is closed and its inode is removed from memory. Therefore, calling `close()` may also result in an unlinked file finally being physically removed from the disk.

Error Values

It is a common mistake to not check the return value of `close()`. This can result in missing a crucial error condition because errors associated with deferred operations may not manifest until later, and `close()` can report them.

There are a handful of possible `errno` values on failure. Other than `EBADF` (the given file descriptor was invalid), the most important error value is `EIO`, indicating a low-level I/O error probably unrelated to the actual close. Regardless of any reported error, the file descriptor, if valid, is always closed, and the associated data structures are freed.

Although POSIX allows it, `close()` will never return `EINTR`. The Linux kernel developers know better.

Seeking with lseek()

Normally, I/O occurs linearly through a file, and the implicit updates to the file position caused by reads and writes are all the seeking that is needed. Some applications, however, want to jump around in a file, providing random rather than linear access. The `lseek()` system call is provided to set the file position of a file descriptor to a given value. Other than updating the file position, it performs no other action, and initiates no I/O whatsoever:

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int fd, off_t pos, int origin);
```

The behavior of `lseek()` depends on the `origin` argument, which can be one of the following:

`SEEK_CUR`
: The current file position of `fd` is set to its current value plus `pos`, which can be negative, zero, or positive. A `pos` of zero returns the current file position value.

`SEEK_END`
: The current file position of `fd` is set to the current length of the file plus `pos`, which can be negative, zero, or positive. A `pos` of zero sets the offset to the end of the file.

SEEK_SET
: The current file position of `fd` is set to `pos`. A `pos` of zero sets the offset to the beginning of the file.

The call returns the new file position on success. On error, it returns `-1` and `errno` is set as appropriate.

For example, to set the file position of `fd` to 1825:

```
off_t ret;

ret = lseek (fd, (off_t) 1825, SEEK_SET);
if (ret == (off_t) -1)
        /* error */
```

Alternatively, to set the file position of `fd` to the end of the file:

```
off_t ret;

ret = lseek (fd, 0, SEEK_END);
if (ret == (off_t) -1)
        /* error */
```

As `lseek()` returns the updated file position, it can be used to find the current file position via a `SEEK_CUR` to zero:

```
int pos;

pos = lseek (fd, 0, SEEK_CUR);
if (pos == (off_t) -1)
        /* error */
else
        /* 'pos' is the current position of fd */
```

By far, the most common uses of `lseek()` are seeking to the beginning, seeking to the end, or determining the current file position of a file descriptor.

Seeking Past the End of a File

It is possible to instruct `lseek()` to advance the file pointer past the end of a file. For example, this code seeks to 1,688 bytes beyond the end of the file mapped by `fd`:

```
int ret;

ret = lseek (fd, (off_t) 1688, SEEK_END);
if (ret == (off_t) -1)
        /* error */
```

On its own, seeking past the end of a file does nothing—a read request to the newly created file position will return EOF. If a write request is subsequently made to this

position, however, new space will be created between the old length of the file and the new length, and it will be padded with zeros.

This zero padding is called a *hole*. On Unix-style filesystems, holes do not occupy any physical disk space. This implies that the total size of all files on a filesystem can add up to more than the physical size of the disk. Files with holes are called *sparse files*. Sparse files can save considerable space and enhance performance because manipulating the holes does not initiate any physical I/O.

A read request to the part of a file in a hole will return the appropriate number of zeros.

Error Values

On error, `lseek()` returns `-1` and `errno` is set to one of the following four values:

`EBADF`
: The given file descriptor does not refer to an open file descriptor.

`EINVAL`
: The value given for `origin` is not one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, *or* the resulting file position would be negative. The fact that `EINVAL` represents both of these errors is unfortunate. The former is almost assuredly a compile-time programming error, whereas the latter can represent a more insidious runtime logic error.

`EOVERFLOW`
: The resulting file offset cannot be represented in an `off_t`. This can occur only on 32-bit architectures. Currently, the file position *is* updated; this error indicates just that it is impossible to return it.

`ESPIPE`
: The given file descriptor is associated with an unseekable object, such as a pipe, FIFO, or socket.

Limitations

The maximum file positions are limited by the size of the `off_t` type. Most machine architectures define this to be the C `long` type, which on Linux is always the *word size* (usually the size of the machine's general-purpose registers). Internally, however, the kernel stores the offsets in the C `long long` type. This poses no problem on 64-bit machines, but it means that 32-bit machines can generate `EOVERFLOW` errors when performing relative seeks.

Positional Reads and Writes

In lieu of using `lseek()`, Linux provides two variants of the `read()` and `write()` system calls. Both receive the file position from which to read or write. Upon completion, they do *not* update the file position.

The read form is called `pread()`:

```
#define _XOPEN_SOURCE 500

#include <unistd.h>

ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

This call reads up to `count` bytes into `buf` from the file descriptor `fd` at file position `pos`.

The write form is called `pwrite()`:

```
#define _XOPEN_SOURCE 500

#include <unistd.h>

ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

This call writes up to `count` bytes from `buf` to the file descriptor `fd` at file position `pos`.

These calls are almost identical in behavior to their non-`p` brethren, except that they completely ignore the current file position; instead of using the current position, they use the value provided by `pos`. Also, when done, they do not update the file position. In other words, any intermixed `read()` and `write()` calls could potentially corrupt the work done by the positional calls.

Both positional calls can be used only on seekable file descriptors, which include regular files. They provide semantics similar to preceding a `read()` or `write()` call with a call to `lseek()`, with three differences. First, these calls are easier to use, especially when doing a tricky operation such as moving through a file backward or randomly. Second, they do not update the file pointer upon completion. Finally, and most importantly, they avoid any potential races that might occur when using `lseek()`.

Because threads share file tables and the current file position is stored in the shared file table, it is possible for one thread in a program to update the file position after a different thread's call to `lseek()` but before its read or write operation executed. In other words, `lseek()` is inherently racy when you have two or more threads in a process all manipulating the same file descriptor. Such race conditions can be avoided by using the `pread()` and `pwrite()` system calls.

Error Values

On success, both calls return the number of bytes read or written. A return value of `0` from `pread()` indicates EOF; from `pwrite()`, a return value of `0` indicates that the call did not write anything. On error, both calls return `-1` and set `errno` appropriately. For `pread()`, any valid `read()` or `lseek()` `errno` value is possible. For `pwrite()`, any valid `write()` or `lseek()` value is possible.

Truncating Files

Linux provides two system calls for truncating the length of a file, both of which are defined and required (to varying degrees) by various POSIX standards. They are:

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate (int fd, off_t len);
```

and:

```
#include <unistd.h>
#include <sys/types.h>

int truncate (const char *path, off_t len);
```

Both system calls truncate the given file to the length given by `len`. The `ftruncate()` system call operates on the file descriptor given by `fd`, which must be open for writing. The `truncate()` system call operates on the filename given by `path`, which must be writable. Both return `0` on success. On error, they return `-1` and set `errno` as appropriate.

The most common use of these system calls is to truncate a file to a size smaller than its current length. Upon successful return, the file's length is `len`. The data previously existing between `len` and the old length is discarded and no longer accessible via a read request.

The functions can also be used to "truncate" a file to a larger size, similar to the seek plus write combination described earlier in "Seeking Past the End of a File" on page 47. The extended bytes are filled with zeros.

Neither operation updates the current file position.

For example, consider the file *pirate.txt* of length 74 bytes with the following contents:

```
Edward Teach was a notorious English pirate.
He was nicknamed Blackbeard.
```

From the same directory, running the following program:

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

```
int main()
{
        int ret;

        ret = truncate ("./pirate.txt", 45);
        if (ret == -1) {
                perror ("truncate");
                return -1;
        }

        return 0;
}
```

results in a file of length 45 bytes with the following contents:

```
Edward Teach was a notorious English pirate.
```

Multiplexed I/O

Applications often need to block on more than one file descriptor, juggling I/O between keyboard input (*stdin*), interprocess communication, and a handful of files. Modern event-driven graphical user interface (GUI) applications may contend with literally hundreds of pending events via their mainloops.[4]

Without the aid of threads—essentially servicing each file descriptor separately—a single process cannot reasonably block on more than one file descriptor at the same time. Working with multiple file descriptors is fine, so long as they are always ready to be read from or written to. But as soon as one file descriptor that is not yet ready is encountered—say, if a `read()` system call is issued, and there is not yet any data—the process will block, no longer able to service the other file descriptors. It might block for just a few seconds, making the application inefficient and annoying the user. However, if no data becomes available on the file descriptor, it could block forever. Because file descriptors' I/O is often interrelated—think pipes—it is quite possible for one file descriptor not to become ready until another is serviced. Particularly with network applications, which may have many sockets open simultaneously, this is potentially quite a problem.

Imagine blocking on a file descriptor related to interprocess communication while *stdin* has data pending. The application won't know that keyboard input is pending until the blocked IPC file descriptor ultimately returns data—but what if the blocked operation never returns?

4. Mainloops should be familiar to anyone who has written GUI applications—for example, GNOME applications utilize a mainloop provided by *GLib*, their base library. A mainloop allows multiple events to be watched for and responded to from a single blocking point.

Earlier in this chapter, we looked at nonblocking I/O as a solution to this problem. With nonblocking I/O, applications can issue I/O requests that return a special error condition instead of blocking. However, this solution is inefficient for two reasons. First, the process needs to continually issue I/O operations in some arbitrary order, waiting for one of its open file descriptors to be ready for I/O. This is poor program design. Second, it would be much more efficient if the program could sleep, freeing the processor for other tasks, to be woken up only when one or more file descriptors were ready to perform I/O.

Enter *multiplexed I/O*.

Multiplexed I/O allows an application to concurrently block on multiple file descriptors and receive notification when any one of them becomes ready to read or write without blocking. Multiplexed I/O thus becomes the pivot point for the application, designed similarly to the following activity:

1. Multiplexed I/O: Tell me when any of these file descriptors become ready for I/O.
2. Nothing ready? Sleep until one or more file descriptors are ready.
3. Woken up! What is ready?
4. Handle all file descriptors ready for I/O, without blocking.
5. Go back to step 1.

Linux provides three multiplexed I/O solutions: the *select*, *poll*, and *epoll* interfaces. We will cover the first two here, and the last, which is an advanced Linux-specific solution, in Chapter 4.

select()

The `select()` system call provides a mechanism for implementing synchronous multiplexing I/O:

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

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

A call to `select()` blocks until the given file descriptors are ready to perform I/O, or until an optionally specified timeout has elapsed.

The watched file descriptors are broken into three sets, each waiting for a different event. File descriptors listed in the `readfds` set are watched to see if data is available for reading (that is, if a read operation will complete without blocking). File descriptors listed in the `writefds` set are watched to see if a write operation will complete without blocking. Finally, file descriptors in the `exceptfds` set are watched to see if an exception has occurred, or if out-of-band data is available (these states apply only to sockets). A given set may be `NULL`, in which case `select()` does not watch for that event.

On successful return, each set is modified such that it contains only the file descriptors that are ready for I/O of the type delineated by that set. For example, assume two file descriptors, with the values 7 and 9, are placed in the `readfds` set. When the call returns, if 7 is still in the set, that file descriptor is ready to read without blocking. If 9 is no longer in the set, it is probably not readable without blocking. (I say *probably* here because it is possible that data became available after the call completed. In that case, a subsequent call to `select()` will return the file descriptor as ready to read.)[5]

The first parameter, `n`, is equal to the value of the highest-valued file descriptor in any set, plus one. Consequently, a caller to `select()` is responsible for checking which given file descriptor is the highest-valued and passing in that value plus one for the first parameter.

The `timeout` parameter is a pointer to a `timeval` structure, which is defined as follows:

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

struct timeval {
        long tv_sec;         /* seconds */
        long tv_usec;        /* microseconds */
};
```

If this parameter is not `NULL`, the call to `select()` will return after `tv_sec` seconds and `tv_usec` microseconds, even if no file descriptors are ready for I/O. On return, the state of this structure across various Unix systems is undefined, and thus it must be reinitialized (along with the file descriptor sets) before every invocation. Indeed, current versions of Linux modify this parameter automatically, setting the values to the time remaining. Thus, if the timeout was set for 5 seconds, and 3 seconds elapsed before a file descriptor became ready, `tv.tv_sec` would contain 2 upon the call's return.

If both values in the timeout are set to zero, the call will return immediately, reporting any events that were pending at the time of the call, but not waiting for any subsequent events.

5. This is because `select()` and `poll()` are level-triggered and not edge-triggered. `epoll()`, which we'll discuss in Chapter 4, can operate in either mode. Edge-triggered operation is simpler but allows I/O events to be missed if care is not taken.

The sets of file descriptors are not manipulated directly, but are instead managed through helper macros. This allows Unix systems to implement the sets however they want. Most systems, however, implement the sets as simple bit arrays.

`FD_ZERO` removes all file descriptors from the specified set. It should be called before every invocation of `select()`:

```
fd_set writefds;

FD_ZERO(&writefds);
```

`FD_SET` adds a file descriptor to a given set, and `FD_CLR` removes a file descriptor from a given set:

```
FD_SET(fd, &writefds);    /* add 'fd' to the set */
FD_CLR(fd, &writefds);    /* oops, remove 'fd' from the set */
```

Well-designed code should never have to make use of `FD_CLR`, and it is rarely, if ever, used.

`FD_ISSET` tests whether a file descriptor is part of a given set. It returns a nonzero integer if the file descriptor is in the set and `0` if it is not. `FD_ISSET` is used after a call from `select()` returns to test whether a given file descriptor is ready for action:

```
if (FD_ISSET(fd, &readfds))
        /* 'fd' is readable without blocking! */
```

Because the file descriptor sets are statically created, they impose a limit on the maximum number of file descriptors and the largest-valued file descriptor that may be placed inside them, both of which are given by `FD_SETSIZE`. On Linux, this value is 1,024. We will look at the ramifications of this limit later in this chapter.

Return values and error codes

On success, `select()` returns the number of file descriptors ready for I/O, among all three sets. If a timeout was provided, the return value may be `0`. On error, the call returns `-1`, and `errno` is set to one of the following values:

`EBADF`
: An invalid file descriptor was provided in one of the sets.

`EINTR`
: A signal was caught while waiting, and the call can be reissued.

`EINVAL`
: The parameter `n` is negative, or the given timeout is invalid.

`ENOMEM`
: Insufficient memory was available to complete the request.

select() example

Let's consider an example program, trivial but fully functional, to illustrate the use of `select()`. This example blocks waiting for input on *stdin* for up to 5 seconds. Because it watches only a single file descriptor, it is not actually multiplexing I/O, but the usage of the system call is made clear:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define TIMEOUT 5       /* select timeout in seconds */
#define BUF_LEN 1024    /* read buffer in bytes */

int main (void)
{
        struct timeval tv;
        fd_set readfds;
        int ret;

        /* Wait on stdin for input. */
        FD_ZERO(&readfds);
        FD_SET(STDIN_FILENO, &readfds);

        /* Wait up to five seconds. */
        tv.tv_sec = TIMEOUT;
        tv.tv_usec = 0;

        /* All right, now block! */
        ret = select (STDIN_FILENO + 1,
                      &readfds,
                      NULL,
                      NULL,
                      &tv);
        if (ret == -1) {
                perror ("select");
                return 1;
        } else if (!ret) {
                printf ("%d seconds elapsed.\n", TIMEOUT);
                return 0;
        }

        /*
         * Is our file descriptor ready to read?
         * (It must be, as it was the only fd that
         * we provided and the call returned
         * nonzero, but we will humor ourselves.)
         */
        if (FD_ISSET(STDIN_FILENO, &readfds)) {
                char buf[BUF_LEN+1];
                int len;
```

```
                /* guaranteed to not block */
                len = read (STDIN_FILENO, buf, BUF_LEN);
                if (len == -1) {
                        perror ("read");
                        return 1;
                }

                if (len) {
                        buf[len] = '\0';
                        printf ("read: %s\n", buf);
                }

                return 0;
        }

        fprintf (stderr, "This should not happen!\n");
        return 1;
}
```

Portable sleeping with select()

Because `select()` has historically been more readily implemented on various Unix systems than a mechanism for subsecond-resolution sleeping, it is often employed as a portable way to sleep by providing a non-`NULL` timeout but `NULL` for all three sets:

```
struct timeval tv;

tv.tv_sec = 0;
tv.tv_usec = 500;

/* sleep for 500 microseconds */
select (0, NULL, NULL, NULL, &tv);
```

Linux provides interfaces for high-resolution sleeping. We will cover these in Chapter 11.

pselect()

The `select()` system call, first introduced in 4.2BSD, is popular, but POSIX defined its own solution, `pselect()`, in POSIX 1003.1g-2000 and later in POSIX 1003.1-2001:

```
#define _XOPEN_SOURCE 600
#include <sys/select.h>

int pselect (int n,
             fd_set *readfds,
             fd_set *writefds,
             fd_set *exceptfds,
             const struct timespec *timeout,
             const sigset_t *sigmask);
```

```
/* these are the same as those used by select() */
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

There are three differences between `pselect()` and `select()`:

- `pselect()` uses the `timespec` structure, not the `timeval` structure, for its `timeout` parameter. The `timespec` structure uses seconds and nanoseconds, not seconds and microseconds, providing theoretically superior timeout resolution. In practice, however, neither call reliably provides even microsecond resolution.
- A call to `pselect()` does not modify the `timeout` parameter. Consequently, this parameter does not need to be reinitialized on subsequent invocations.
- The `select()` system call does not have the `sigmask` parameter. With respect to signals, when this parameter is set to `NULL`, `pselect()` behaves like `select()`.

The `timespec` structure is defined as follows:

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

struct timespec {
        long tv_sec;         /* seconds */
        long tv_nsec;        /* nanoseconds */
};
```

The primary motivator behind the addition of `pselect()` to Unix's toolbox was the addition of the `sigmask` parameter, which attempts to solve a race condition between waiting on file descriptors and signals (signals are covered in depth in Chapter 10). Assume that a signal handler sets a global flag (as most do), and the process checks this flag before a call to `select()`. Now, assume that the signal arrives after the check, but before the call. The application may block indefinitely and never respond to the set flag. The `pselect()` call solves this problem by allowing an application to call `pselect()`, providing a set of signals to block. Blocked signals are not handled until they are unblocked. Once `pselect()` returns, the kernel restores the old signal mask.

Until the 2.6.16 kernel, the Linux implementation of `pselect()` was not a system call, but a simple wrapper around `select()`, provided by *glibc*. This wrapper minimized—but did not totally eliminate—the risk of this race condition occurring. With the introduction of a true system call, the race is gone.

Despite the (relatively minor) improvements in `pselect()`, most applications continue to use `select()`, either out of habit or in the name of greater portability.

poll()

The `poll()` system call is System V's multiplexed I/O solution. It solves several deficiencies in `select()`, although `select()` is still often used (most likely out of habit, or in the name of portability):

```
#include <poll.h>

int poll (struct pollfd *fds, nfds_t nfds, int timeout);
```

Unlike `select()`, with its inefficient three bitmask-based sets of file descriptors, `poll()` employs a single array of `nfds` `pollfd` structures, pointed to by `fds`. The structure is defined as follows:

```
#include <poll.h>

struct pollfd {
        int fd;           /* file descriptor */
        short events;     /* requested events to watch */
        short revents;    /* returned events witnessed */
};
```

Each `pollfd` structure specifies a single file descriptor to watch. Multiple structures may be passed, instructing `poll()` to watch multiple file descriptors. The `events` field of each structure is a bitmask of events to watch for on that file descriptor. The user sets this field. The `revents` field is a bitmask of events that were witnessed on the file descriptor. The kernel sets this field on return. All of the events requested in the `events` field may be returned in the `revents` field. Valid events are as follows:

`POLLIN`
: There is data to read.

`POLLRDNORM`
: There is normal data to read.

`POLLRDBAND`
: There is priority data to read.

`POLLPRI`
: There is urgent data to read.

`POLLOUT`
: Writing will not block.

`POLLWRNORM`
: Writing normal data will not block.

`POLLWRBAND`
: Writing priority data will not block.

`POLLMSG`
: A `SIGPOLL` message is available.

In addition, the following events may be returned in the `revents` field:

`POLLER`
: Error on the given file descriptor.

`POLLHUP`
: Hung up event on the given file descriptor.

`POLLNVAL`
: The given file descriptor is invalid.

These events have no meaning in the `events` field and you should not pass them in that field because they are always returned if applicable. With `poll()`, unlike `select()`, you need not explicitly ask for reporting of exceptions.

`POLLIN | POLLPRI` is equivalent to `select()`'s read event, and `POLLOUT | POLLWRBAND` is equivalent to `select()`'s write event. `POLLIN` is equivalent to `POLLRDNORM | POLLRDBAND`, and `POLLOUT` is equivalent to `POLLWRNORM`.

For example, to watch a file descriptor for both readability and writability, we would set `events` to `POLLIN | POLLOUT`. On return, we would check `revents` for these flags in the structure corresponding to the file descriptor in question. If `POLLIN` were set, the file descriptor would be readable without blocking. If `POLLOUT` were set, the file descriptor would be writable without blocking. The flags are not mutually exclusive: both may be set, signifying that both reads and writes will return instead of blocking on that file descriptor.

The `timeout` parameter specifies the length of time to wait, in milliseconds, before returning regardless of any ready I/O. A negative value denotes an infinite timeout. A value of `0` instructs the call to return immediately, listing any file descriptors with pending ready I/O, but not waiting for any further events. In this manner, `poll()` is true to its name, polling once, and immediately returning.

Return values and error codes

On success, `poll()` returns the number of file descriptors whose structures have nonzero `revents` fields. It returns `0` if the timeout occurred before any events occurred. On failure, `-1` is returned, and `errno` is set to one of the following:

`EBADF`
: An invalid file descriptor was given in one or more of the structures.

`EFAULT`
: The pointer to `fds` pointed outside of the process's address space.

`EINTR`
: A signal occurred before any requested event. The call may be reissued.

`EINVAL`
: The `nfds` parameter exceeded the `RLIMIT_NOFILE` value.

`ENOMEM`
: Insufficient memory was available to complete the request.

poll() example

Let's look at an example program that uses `poll()` to simultaneously check whether a read from *stdin* and a write to *stdout* will block:

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

#define TIMEOUT 5       /* poll timeout, in seconds */

int main (void)
{
        struct pollfd fds[2];
        int ret;

        /* watch stdin for input */
        fds[0].fd = STDIN_FILENO;
        fds[0].events = POLLIN;

        /* watch stdout for ability to write (almost always true) */
        fds[1].fd = STDOUT_FILENO;
        fds[1].events = POLLOUT;

        /* All set, block! */
        ret = poll (fds, 2, TIMEOUT * 1000);
        if (ret == -1) {
                perror ("poll");
                return 1;
        }

        if (!ret) {
                printf ("%d seconds elapsed.\n", TIMEOUT);
                return 0;
        }

        if (fds[0].revents & POLLIN)
                printf ("stdin is readable\n");

        if (fds[1].revents & POLLOUT)
                printf ("stdout is writable\n");
```

```
        return 0;
}
```

Running this, we get the following, as expected:

```
$ ./poll
stdout is writable
```

Running it again, but this time redirecting a file into standard in, we see both events:

```
$ ./poll < ode_to_my_parrot.txt
stdin is readable
stdout is writable
```

If we were using `poll()` in a real application, we would not need to reconstruct the `pollfd` structures on each invocation. The same structure may be passed in repeatedly; the kernel will handle zeroing the `revents` field as needed.

ppoll()

Linux provides a `ppoll()` cousin to `poll()`, in the same vein as `pselect()`. Unlike `pselect()`, however, `ppoll()` is a Linux-specific interface:

```
#define _GNU_SOURCE

#include <poll.h>

int ppoll (struct pollfd *fds,
           nfds_t nfds,
           const struct timespec *timeout,
           const sigset_t *sigmask);
```

As with `pselect()`, the `timeout` parameter specifies a timeout value in seconds and nanoseconds, and the `sigmask` parameter provides a set of signals for which to wait.

poll() Versus select()

Although they perform the same basic job, the `poll()` system call is superior to `select()` for a handful of reasons:

- `poll()` does not require that the user calculate and pass in as a parameter the value of the highest-numbered file descriptor plus one.
- `poll()` is more efficient for large-valued file descriptors. Imagine watching a single file descriptor with the value 900 via `select()`—the kernel would have to check each bit of each passed-in set, up to the 900th bit.
- `select()`'s file descriptor sets are statically sized, introducing a trade-off: they are small, limiting the maximum file descriptor that `select()` can watch, or they are inefficient. Operations on large bitmasks are not efficient, especially if it is not

known whether they are sparsely populated.[6] With `poll()`, one can create an array of exactly the right size. Only watching one item? Just pass in a single structure.

- With `select()`, the file descriptor sets are reconstructed on return, so each subsequent call must reinitialize them. The `poll()` system call separates the input (`events` field) from the output (`revents` field), allowing the array to be reused without change.
- The `timeout` parameter to `select()` is undefined on return. Portable code needs to reinitialize it. This is not an issue with `pselect()`, however.

The `select()` system call does have a few things going for it, though:

- `select()` is more portable, as some Unix systems do not support `poll()`.
- `select()` provides better timeout resolution: down to the microsecond wheres `poll()` provides only millisecond resolution. Both `ppoll()` and `pselect()` theoretically provide nanosecond resolution but, in practice, none of these calls reliably provides even microsecond resolution.

Superior to both `poll()` and `select()` is the *epoll* interface, a Linux-specific multiplexing I/O solution that we'll look at in Chapter 4.

Kernel Internals

This section looks at how the Linux kernel implements I/O, focusing on three primary subsystems of the kernel: the *virtual filesystem* (VFS), the *page cache*, and *page writeback*. Together, these subsystems help make I/O seamless, efficient, and optimal.

In Chapter 4, we will look at a fourth subsystem, the *I/O scheduler*.

The Virtual Filesystem

The virtual filesystem, occasionally also called a *virtual file switch*, is a mechanism of abstraction that allows the Linux kernel to call filesystem functions and manipulate filesystem data without knowing—or even caring about—the specific type of filesystem being used.

6. If a bitmask is generally sparsely populated, each word composing the mask can be checked against zero; only if that operation returns false need each bit be checked. This work is wasted, however, if the bitmask is densely populated.

The VFS accomplishes this abstraction by providing a *common file model*, which is the basis for all filesystems in Linux. Via function pointers and various object-oriented practices,[7] the common file model provides a framework to which filesystems in the Linux kernel must adhere. This allows the VFS to generically make requests of the filesystem. The framework provides hooks to support reading, creating links, synchronizing, and so on. Each filesystem then registers functions to handle the operations of which it is capable.

This approach forces a certain amount of commonality between filesystems. For example, the VFS talks in terms of inodes, superblocks, and directory entries. A filesystem not of Unix origins, possibly devoid of Unix-like concepts such as inodes, simply has to cope. Indeed, cope they do: Linux supports filesystems such as FAT and NTFS without issues.

The benefits of the VFS are enormous. A single system call can read from *any* filesystem on *any* medium; a single utility can copy from any one filesystem to any other. All filesystems support the same concepts, the same interfaces, and the same calls. Everything just works—and works well.

When an application issues a `read()` system call, it takes an interesting journey. The C library provides definitions of the system call that are converted to the appropriate trap statements at compile time. Once a user-space process is trapped into the kernel, passed through the system call handler, and handed to the `read()` system call, the kernel figures out what object *backs* the given file descriptor. The kernel then invokes the read function associated with the backing object. For filesystems, this function is part of the filesystem code. The function then does its thing—for example, physically reading the data from the filesystem—and returns the data to the user-space `read()` call, which then returns to the system call handler, which copies the data back to user space, where the `read()` system call returns and the process continues to execute.

To system programmers, the ramifications of the VFS are important. Programmers need not worry about the type of filesystem or media on which a file resides. Generic system calls—`read()`, `write()`, and so on—can manipulate files on any supported filesystem and on any supported media.

The Page Cache

The page cache is an in-memory store of recently accessed data from an on-disk filesystem. Disk access is painfully slow, particularly relative to today's processor speeds. Storing requested data in memory allows the kernel to fulfill subsequent requests for the same data from memory, avoiding repeated disk access.

7. Yes, in C.

The page cache exploits the concept of *temporal locality*, a type of *locality of reference*, which says that a resource accessed at one point has a high probability of being accessed again in the near future. The memory consumed to cache data on its first access therefore pays off, as it prevents future expensive disk accesses.

The page cache is the first place that the kernel looks for filesystem data. The kernel invokes the memory subsystem to read data from the disk only when it isn't found in the cache. Thus, the first time any item of data is read, it is transferred from the disk into the page cache, and is returned to the application from the cache. If that data is then read again, it is simply returned from the cache. All operations transparently execute through the page cache, ensuring that its data is relevant and always valid.

The Linux page cache is dynamic in size. As I/O operations bring more and more data into memory, the page cache grows larger and larger, consuming any free memory. If the page cache eventually does consume all free memory and an allocation is committed that requests additional memory, the page cache is *pruned*, releasing its least-used pages to make room for "real" memory usage. This pruning occurs seamlessly and automatically. A dynamically sized cache allows Linux to use all of the memory in the system and cache as much data as possible.

Often, however, it would make more sense to *swap* to disk a seldom-used page of process memory than it would to prune an oft-used piece of the page cache that could well be reread into memory on the next read request (swapping allows the kernel to store data on the disk to allow a larger memory footprint than the machine has RAM). The Linux kernel implements heuristics to balance the swapping of data versus the pruning of the page cache (and other in-memory reserves). These heuristics might decide to swap data out to disk in lieu of pruning the page cache, particularly if the data being swapped out is not in use.

The swap-versus-cache balance is tuned via */proc/sys/vm/swappiness*. This virtual file has a value from 0 to 100, with a default of 60. A higher value implies a stronger preference toward keeping the page cache in memory, and swapping more readily. A lower value implies a stronger preference toward pruning the page cache and not swapping.

Another form of locality of reference is *sequential locality*, which says that data is often referenced sequentially. To take advantage of this principle, the kernel also implements page cache *readahead*. Readahead is the act of reading extra data off the disk and into the page cache following each read request—in effect, reading a little bit ahead. When the kernel reads a chunk of data from the disk, it also reads the following chunk or two. Reading large sequential chunks of data at once is efficient, as the disk usually need not seek. In addition, the kernel can fulfill the readahead request while the process is manipulating the first chunk of read data. If, as often happens, the process goes on to submit a new read request for the subsequent chunk, the kernel can hand over the data from the initial readahead without having to issue a disk I/O request.

As with the page cache, the kernel manages readahead dynamically. If it notices that a process is consistently using the data that was read in via readahead, the kernel grows the readahead window, thereby reading ahead more and more data. The readahead window may be as small as 16 KB, and as large as 128 KB. Conversely, if the kernel notices that readahead is not resulting in any useful hits—that is, that the application is seeking around the file and not reading sequentially—it can disable readahead entirely.

The presence of a page cache is meant to be transparent. System programmers generally cannot optimize their code to better take advantage of the fact that a page cache exists—other than, perhaps, not implementing such a cache in user space themselves. Normally, efficient code is all that is needed to best utilize the page cache. Utilizing readahead, on the other hand, is possible. Sequential file I/O is always preferred to random access, although it's not always feasible.

Page Writeback

As discussed earlier in "Behavior of write()" on page 39, the kernel defers writes via buffers. When a process issues a write request, the data is copied into a buffer, and the buffer is marked *dirty*, denoting that the in-memory copy is newer than the on-disk copy. The write request then simply returns. If another write request is made to the same chunk of a file, the buffer is updated with the new data. Write requests elsewhere in the same file generate new buffers.

Eventually the dirty buffers need to be committed to disk, synchronizing the on-disk files with the data in memory. This is known as writeback. It occurs in two situations:

- When free memory shrinks below a configurable threshold, dirty buffers are written back to disk so that the now-clean buffers may be removed, freeing memory.
- When a dirty buffer ages beyond a configurable threshold, the buffer is written back to disk. This prevents data from remaining dirty indefinitely.

Writebacks are carried out by a gang of kernel threads named *flusher* threads. When one of the previous two conditions is met, the flusher threads wake up and begin committing dirty buffers to disk until neither condition is true.

There may be multiple flusher threads instantiating writebacks at the same time. This is done to capitalize on the benefits of parallelism and to implement *congestion avoidance*. Congestion avoidance attempts to keep writes from getting backed up while waiting to be written to any one block device. If dirty buffers from different block devices exist, the various flusher threads will work to fully use each block device. This fixes a deficiency in earlier kernels: the predecessor to flusher threads (*pdflush* and before that, *bdflush*) could spend all of its time waiting on a single block device, while other block devices sat idle. On a modern machine, the Linux kernel can now keep a very large number of disks saturated.

Buffers are represented in the kernel by the `buffer_head` data structure. This data structure tracks various metadata associated with the buffer, such as whether the buffer is clean or dirty. It also contains a pointer to the actual data. This data resides in the page cache. In this manner, the buffer subsystem and the page cache are unified.

In early versions of the Linux kernel—before 2.4—the buffer subsystem was separate from the page cache, and thus there was both a page and a buffer cache. This implied that data could exist in the buffer cache (as a dirty buffer) and the page cache (as cached data) at the same time. Naturally, synchronizing these two separate caches took some effort. The unified page cache introduced in the 2.4 Linux kernel was a welcomed improvement.

Deferred writes and the buffer subsystem in Linux enable fast writes at the expense of the risk of data loss on power failure. To avoid this risk, paranoid and critical applications can use synchronized I/O (discussed earlier in this chapter).

Conclusion

This chapter discussed the basics of Linux system programming: file I/O. On a system such as Linux, which strives to represent as much as possible as a file, it's very important to know how to open, read, write, and close files. All of these operations are classic Unix and are represented in many standards.

The next chapter tackles buffered I/O and the standard C library's standard I/O interfaces. The standard C library is not just a convenience; buffering I/O in user space provides crucial performance improvements.