CHAPTER 4
Advanced File I/O

In Chapter 2, we looked at the basic I/O system calls in Linux. These calls form not only the basis of file I/O, but also the foundation of virtually all communication on Linux. In Chapter 3, we looked at how user-space buffering is often needed on top of the basic I/O system calls, and we studied a specific user-space buffering solution, C's standard I/O library. In this chapter, we'll look at the advanced I/O system calls that Linux provides:

Scatter/gather I/O
: Allows a single call to read from or write data to many buffers at once; useful for bunching together fields of different data structures to form one I/O transaction.

Epoll
: Improves on the `poll()` and `select()` system calls described in Chapter 2; useful when hundreds of file descriptors need to be polled from a single thread.

Memory-mapped I/O
: Maps a file into memory, allowing file I/O to occur via simple memory manipulation; useful for certain patterns of I/O.

File advice
: Allows a process to provide hints to the kernel on the process's intended uses for a file; can result in improved I/O performance.

Asynchronous I/O
: Allows a process to issue I/O requests without waiting for them to complete; useful for juggling heavy I/O workloads without the use of threads.

The chapter will conclude with a discussion of performance considerations and the kernel's I/O subsystems.

Scatter/Gather I/O

Scatter/gather I/O is a method of input and output where a single system call writes to a vector of buffers from a single data stream, or, alternatively, reads into a vector of buffers from a single data stream. This type of I/O is so named because the data is *scattered into* or *gathered from* the given vector of buffers. An alternative name for this approach to input and output is *vectored I/O*. In comparison, the standard read and write system calls that we covered in Chapter 2 provide *linear I/O*.

Scatter/gather I/O provides several advantages over linear I/O methods:

More natural coding pattern
: If your data is naturally segmented—say, the fields of a predefined structure—vectored I/O allows for intuitive manipulation.

Efficiency
: A single vectored I/O operation can replace multiple linear I/O operations.

Performance
: In addition to a reduction in the number of issued system calls, a vectored I/O implementation can provide improved performance over a linear I/O implementation via internal optimizations.

Atomicity
: In contrast with multiple linear I/O operations, a process can execute a single vectored I/O operation with no risk of interleaving I/O from another process.

readv() and writev()

POSIX 1003.1-2001 defines, and Linux implements, a pair of system calls that implement scatter/gather I/O. The Linux implementation satisfies all of the goals listed in the previous section.

The `readv()` function reads `count` segments from the file descriptor `fd` into the buffers described by `iov`:

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

ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

The `writev()` function writes at most `count` segments from the buffers described by `iov` into the file descriptor `fd`:

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

ssize_t writev (int fd,
                const struct iovec *iov,
                int count);
```

The `readv()` and `writev()` functions behave the same as `read()` and `write()`, respectively, except that multiple buffers are read from or written to.

Each `iovec` structure describes an independent disjoint buffer, which is called a *segment*:

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

struct iovec {
       void *iov_base;    /* pointer to start of buffer */
       size_t iov_len;    /* size of buffer in bytes */
};
```

A set of segments is called a *vector*. Each segment in the vector describes the address and length of a buffer in memory to or from which data should be written or read. The `readv()` function fills each buffer of `iov_len` bytes completely before proceeding to the next buffer. The `writev()` function always writes out all full `iov_len` bytes before proceeding to the next buffer. Both functions always operate on the segments in order, starting with `iov[0]`, then `iov[1]`, and so on, through `iov[count-1]`.

Return values

On success, `readv()` and `writev()` return the number of bytes read or written, respectively. This number should be the sum of all `count iov_len` values. On error, the system calls return `-1` and set `errno` as appropriate. These system calls can experience any of the errors of the `read()` and `write()` system calls, and will, upon receiving such errors, set the same `errno` codes. In addition, the standards define two other error situations.

First, because the return type is an `ssize_t`, if the sum of all `count iov_len` values is greater than `SSIZE_MAX`, no data will be transferred, `-1` will be returned, and `errno` will be set to `EINVAL`.

Second, POSIX dictates that `count` must be larger than zero and less than or equal to `IOV_MAX`, which is defined in `<limits.h>`. In Linux, `IOV_MAX` is currently `1024`. If `count` is `0`, the system calls return `0`.[1] If `count` is greater than `IOV_MAX`, no data is transferred, the calls return `-1`, and `errno` is set to `EINVAL`.

1. Note that other Unix systems may set errno to `EINVAL` if count is 0. This is explicitly allowed by the standards, which say that `EINVAL` may be set if that value is 0 or that the system can handle the zero case in some other (nonerror) way.

Optimizing the Count

During a vectored I/O operation, the Linux kernel must allocate internal data structures to represent each segment. Normally, this allocation occurs dynamically, based on the size of `count`. As an optimization, however, the Linux kernel creates a small array of segments on the stack that it uses if `count` is sufficiently small, negating the need to dynamically allocate the segments and thereby providing a small boost in performance. This threshold is currently eight, so if `count` is less than or equal to `8`, the vectored I/O operation occurs in a very memory-efficient manner off of the process's kernel stack.

Most likely, you won't have a choice about how many segments you need to transfer at once in a given vectored I/O operation. If you are flexible, however, and are debating over a small value, choosing a value of eight or less definitely improves efficiency.

writev() example

Let's consider a simple example that writes out a vector of three segments, each containing a string of a different size. This self-contained program is complete enough to demonstrate `writev()`, yet simple enough to serve as a useful code snippet:

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

int main ()
{
        struct iovec iov[3];
        ssize_t nr;
        int fd, i;

        char *buf[] = {
                "The term buccaneer comes from the word boucan.\n",
                "A boucan is a wooden frame used for cooking meat.\n",
                "Buccaneer is the West Indies name for a pirate.\n" };

        fd = open ("buccaneer.txt", O_WRONLY | O_CREAT | O_TRUNC);
        if (fd == -1) {
                perror ("open");
                return 1;
        }

        /* fill out three iovec structures */
        for (i = 0; i < 3; i++) {
                iov[i].iov_base = buf[i];
```

```
                iov[i].iov_len = strlen(buf[i]) + 1;
        }
        /* with a single call, write them all out */
        nr = writev (fd, iov, 3);
        if (nr == -1) {
                perror ("writev");
                return 1;
        }
        printf ("wrote %d bytes\n", nr);

        if (close (fd)) {
                perror ("close");
                return 1;
        }

        return 0;
}
```

Running the program produces the desired result:

```
$ ./writev
wrote 148 bytes
```

As does reading the file:

```
$ cat buccaneer.txt
The term buccaneer comes from the word boucan.
A boucan is a wooden frame used for cooking meat.
Buccaneer is the West Indies name for a pirate.
```

readv() example

Now, let's consider an example program that uses the `readv()` system call to read from the previously generated text file using vectored I/O. This self-contained example is likewise simple yet complete:

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

int main ()
{
        char foo[48], bar[51], baz[49];
        struct iovec iov[3];
        ssize_t nr;
        int fd, i;

        fd = open ("buccaneer.txt", O_RDONLY);
        if (fd == -1) {
                perror ("open");
                return 1;
```

```
        }

        /* set up our iovec structures */
        iov[0].iov_base = foo;
        iov[0].iov_len = sizeof (foo);
        iov[1].iov_base = bar;
        iov[1].iov_len = sizeof (bar);
        iov[2].iov_base = baz;
        iov[2].iov_len = sizeof (baz);

        /* read into the structures with a single call */
        nr = readv (fd, iov, 3);
        if (nr == -1) {
                perror ("readv");
                return 1;
        }

        for (i = 0; i < 3; i++)
                printf ("%d: %s", i, (char *) iov[i].iov_base);

        if (close (fd)) {
                perror ("close");
                return 1;
        }

        return 0;
}
```

Running this program after running the previous program produces the following results:

```
$ ./readv
0: The term buccaneer comes from the word boucan.
1: A boucan is a wooden frame used for cooking meat.
2: Buccaneer is the West Indies name for a pirate.
```

Implementation

A naïve implementation of `readv()` and `writev()` could be implemented in user space as a simple loop, something similar to the following:

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

ssize_t naive_writev (int fd, const struct iovec *iov, int count)
{
        ssize_t ret = 0;
        int i;

        for (i = 0; i < count; i++) {
                ssize_t nr;

                errno = 0;
```

```
                nr = write (fd, iov[i].iov_base, iov[i].iov_len);
                if (nr == -1) {
                        if (errno == EINTR)
                                continue;
                        ret = -1;
                        break;
                }
                ret += nr;
        }

        return ret;
}
```

Thankfully, this is *not* the Linux implementation: Linux implements `readv()` and `writev()` as system calls and internally performs scatter/gather I/O. In fact, all I/O inside the Linux kernel is vectored; `read()` and `write()` are implemented as vectored I/O with a vector of only one segment.

Event Poll

Recognizing the limitations of both `poll()` and `select()`, the 2.6 Linux kernel[2] introduced the *event poll* (epoll) facility. While more complex than the two earlier interfaces, epoll solves the fundamental performance problem shared by both of them and adds several new features.

Both `poll()` and `select()` (discussed in Chapter 2) require the full list of file descriptors to watch on each invocation. The kernel must then walk the list of each file descriptor to be monitored. When this list grows large—it may contain hundreds or even thousands of file descriptors—walking the list on each invocation becomes a scalability bottleneck.

Epoll circumvents this problem by decoupling the monitor registration from the actual monitoring. One system call initializes an epoll context, another adds monitored file descriptors to or removes them from the context, and a third performs the actual event wait.

Creating a New Epoll Instance

An epoll context is created via `epoll_create1()`:

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

int epoll_create1 (int flags);
```

2. Epoll was introduced in the 2.5.44 development kernel and the interface was finalized as of 2.5.66. It is Linux-specific.

```
/* deprecated. use epoll_create1() in new code. */
int epoll_create (int size);
```

A successful call to `epoll_create1()` instantiates a new epoll instance and returns a file descriptor associated with the instance. This file descriptor has no relationship to a real file; it is just a handle to be used with subsequent calls using the epoll facility. The `flags` parameter allows the modification of epoll behavior. Currently, only `EPOLL_CLOEXEC` is a valid flag. It enables close-on-exec behavior.

On error, the call returns `-1` and sets `errno` to one of the following:

`EINVAL`
: Invalid `flags` parameter.

`EMFILE`
: The user has reached their limit on the total number of open files.

`ENFILE`
: The system has reached its limit on the total number of open files.

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

`epoll_create()` is a deprecated, older variant of `epoll_create1()`. It does not accept any flags. Instead, it takes a `size` argument, which is unused. `size` used to provide a hint about the number of file descriptors to be watched; nowadays the kernel dynamically sizes the required data structures and this parameter just needs to be greater than zero. If it is not, `EINVAL` is returned. New applications should only use this variant if they need to target systems running before `epoll_create1()` was introduced in Linux kernel 2.6.27 and *glibc* 2.9.

A typical call is:

```
int epfd;

epfd = epoll_create1 (0);
if (epfd < 0)
        perror ("epoll_create1");
```

The file descriptor returned from `epoll_create1()` should be destroyed via a call to `close()` after polling is finished.

Controlling Epoll

The `epoll_ctl()` system call can be used to add file descriptors to and remove file descriptors from a given epoll context:

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

int epoll_ctl (int epfd,
```

```
                int op,
                int fd,
                struct epoll_event *event);
```

The header <sys/epoll.h> defines the epoll_event structure as:

```
struct epoll_event {
        __u32 events;  /* events */
        union {
                void *ptr;
                int fd;
                __u32 u32;
                __u64 u64;
        } data;
};
```

A successful call to epoll_ctl() controls the epoll instance associated with the file descriptor epfd. The parameter op specifies the operation to be taken against the file associated with fd. The event parameter further describes the behavior of the operation.

Here are valid values for the op parameter:

EPOLL_CTL_ADD
: Add a monitor on the file associated with the file descriptor fd to the epoll instance associated with epfd, per the events defined in event.

EPOLL_CTL_DEL
: Remove a monitor on the file associated with the file descriptor fd from the epoll instance associated with epfd.

EPOLL_CTL_MOD
: Modify an existing monitor of fd with the updated events specified by event.

The events field in the epoll_event structure lists which events to monitor on the given file descriptor. Multiple events can be bitwise-ORed together. Here are valid values:

EPOLLERR
: An error condition occurred on the file. This event is always monitored, even if it's not specified.

EPOLLET
: Enables edge-triggered behavior for the monitor of the file (see "Edge- Versus Level-Triggered Events" on page 103). The default behavior is level-triggered.

EPOLLHUP
: A hangup occurred on the file. This event is always monitored, even if it's not specified.

EPOLLIN
: The file is available to be read from without blocking.

EPOLLONESHOT
: After an event is generated and read, the file is automatically no longer monitored. A new event mask must be specified via EPOLL_CTL_MOD to reenable the watch.

EPOLLOUT
: The file is available to be written to without blocking.

EPOLLPRI
: There is urgent out-of-band data available to read.

The data field inside the event_poll structure is for the user's private use. The contents are returned to the user upon receipt of the requested event. The common practice is to set event.data.fd to fd, which makes it easy to look up which file descriptor caused the event.

Upon success, epoll_ctl() returns 0. On failure, the call returns -1 and sets errno to one of the following values:

EBADF
: epfd is not a valid epoll instance, or fd is not a valid file descriptor.

EEXIST
: op was EPOLL_CTL_ADD, but fd is already associated with epfd.

EINVAL
: epfd is not an epoll instance, epfd is the same as fd, or op is invalid.

ENOENT
: op was EPOLL_CTL_MOD, or EPOLL_CTL_DEL, but fd is not associated with epfd.

ENOMEM
: There was insufficient memory to process the request.

EPERM
: fd does not support epoll.

As an example, to add a new watch on the file associated with fd to the epoll instance epfd, you would write:

```
struct epoll_event event;
int ret;

event.data.fd = fd; /* return the fd to us later (from epoll_wait) */
event.events = EPOLLIN | EPOLLOUT;

ret = epoll_ctl (epfd, EPOLL_CTL_ADD, fd, &event);
if (ret)
        perror ("epoll_ctl");
```

To modify an existing event on the file associated with `fd` on the epoll instance `epfd`, you would write:

```
struct epoll_event event;
int ret;

event.data.fd = fd; /* return the fd to us later */
event.events = EPOLLIN;

ret = epoll_ctl (epfd, EPOLL_CTL_MOD, fd, &event);
if (ret)
        perror ("epoll_ctl");
```

Conversely, to remove an existing event on the file associated with `fd` from the epoll instance `epfd`, you would write:

```
struct epoll_event event;
int ret;

ret = epoll_ctl (epfd, EPOLL_CTL_DEL, fd, &event);
if (ret)
        perror ("epoll_ctl");
```

Note that the `event` parameter can be `NULL` when `op` is `EPOLL_CTL_DEL`, as there is no event mask to provide. Kernel versions before 2.6.9, however, erroneously check for this parameter to be non-`NULL`. For portability to these older kernels, you should pass in a valid non-`NULL` pointer; it will not be touched. Kernel 2.6.9 fixed this bug.

Waiting for Events with Epoll

The system call `epoll_wait()` waits for events on the file descriptors associated with the given epoll instance:

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

int epoll_wait (int epfd,
                struct epoll_event *events,
                int maxevents,
                int timeout);
```

A call to `epoll_wait()` waits up to `timeout` milliseconds for events on the files associated with the epoll instance `epfd`. Upon success, `events` points to memory containing `epoll_event` structures describing each event—such as file ready to be written to or read from—up to a maximum of `maxevents` events. The return value is the number of events, or `-1` on error, in which case `errno` is set to one of the following:

`EBADF`
: `epfd` is not a valid file descriptor.

`EFAULT`
: The process does not have write access to the memory pointed at by `events`.

`EINTR`
: The system call was interrupted by a signal before it could complete or the timeout expired.

`EINVAL`
: `epfd` is not a valid epoll instance, or `maxevents` is equal to or less than `0`.

If `timeout` is `0`, the call returns immediately, even if no events are available, in which case the call will return `0`. If the `timeout` is `-1`, the call will not return until an event is available.

When the call returns, the `events` field of the `epoll_event` structure describes the events that occurred. The `data` field contains whatever the user set it to before invocation of `epoll_ctl()`.

A full `epoll_wait()` example looks like this:

```
#define MAX_EVENTS    64

struct epoll_event *events;
int nr_events, i, epfd;

events = malloc (sizeof (struct epoll_event) * MAX_EVENTS);
if (!events) {
        perror ("malloc");
        return 1;
}

nr_events = epoll_wait (epfd, events, MAX_EVENTS, -1);
if (nr_events < 0) {
        perror ("epoll_wait");
        free (events);
        return 1;
}

for (i = 0; i < nr_events; i++) {
        printf ("event=%ld on fd=%d\n",
                events[i].events,
                events[i].data.fd);
```

```
            /*
             * We now can, per events[i].events, operate on
             * events[i].data.fd without blocking.
             */
    }

    free (events);
```

We will cover the functions `malloc()` and `free()` in Chapter 9.

Edge- Versus Level-Triggered Events

If the `EPOLLET` value is set in the `events` field of the `event` parameter passed to `epoll_ctl()`, the watch on `fd` is *edge-triggered*, as opposed to *level-triggered*.

Consider the following events between a producer and a consumer communicating over a Unix pipe:

1. The producer writes 1 KB of data onto a pipe.
2. The consumer performs an `epoll_wait()` on the pipe, waiting for the pipe to contain data and thus be readable.

With a level-triggered watch, the call to `epoll_wait()` in step 2 will return immediately, showing that the pipe is ready to read. With an edge-triggered watch, this call will not return until after step 1 occurs. That is, even if the pipe is readable at the invocation of `epoll_wait()`, the call will not return until the data is written onto the pipe.

Level-triggered is the default behavior. It is how `poll()` and `select()` behave, and it is what most developers expect. Edge-triggered behavior requires a different approach to programming, commonly utilizing nonblocking I/O, and careful checking for `EAGAIN`.

Edge Triggered

The terminology "edge-triggered" is borrowed from electrical engineering. A level-triggered interrupt is issued whenever a line is asserted. An edge-triggered interrupt is caused only during the rising or falling edge of the change in assertion. Level-triggered interrupts are useful when the state of the event (the asserted line) is of interest. Edge-triggered interrupts are useful when the event itself (the line being asserted) is of interest.

Assume you have a file descriptor from which you are reading. With level-triggered epoll behavior, you'll receive a notification so long as the file descriptor is ready for reading. It is the *level* of the line that causes notification. With edge-triggered, you'll receive the notification but once, when the data first becomes readable: it is the *edge*, or the change, that causes notification.

Mapping Files into Memory

As an alternative to standard file I/O, the kernel provides an interface that allows an application to map a file into memory, meaning that there is a one-to-one correspondence between a memory address and a word in the file. The programmer can then access the file directly through memory, identically to any other chunk of memory-resident data—it is even possible to allow writes to the memory region to transparently map back to the file on disk.

POSIX.1 standardizes—and Linux implements—the `mmap()` system call for mapping objects into memory. This section will discuss `mmap()` as it pertains to mapping files into memory to perform I/O; in Chapter 9, we will visit other applications of `mmap()`.

mmap()

A call to `mmap()` asks the kernel to map `len` bytes of the object represented by the file descriptor `fd`, starting at `offset` bytes into the file, into memory. If `addr` is included, it indicates a preference to use that starting address in memory. The access permissions are dictated by `prot`, and additional behavior can be given by `flags`:

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

void * mmap (void *addr,
             size_t len,
             int prot,
             int flags,
             int fd,
             off_t offset);
```

The `addr` parameter offers a suggestion to the kernel of where best to map the file. It is only a hint; most users pass `0`. The call returns the actual address in memory where the mapping begins.

The `prot` parameter describes the desired memory protection of the mapping. It may be either `PROT_NONE`, in which case the pages in this mapping may not be accessed (rarely useful!), or a bitwise OR of one or more of the following flags:

`PROT_READ`
: The pages may be read.

`PROT_WRITE`
: The pages may be written.

`PROT_EXEC`
: The pages may be executed.

The desired memory protection must not conflict with the open mode of the file. For example, if the program opens the file read-only, `prot` must not specify `PROT_WRITE`.

Protection Flags, Architectures, and Security

While POSIX defines three protection bits (read, write, and execute), some architectures support only a subset of these. It is common, for example, for a processor to not differentiate between the actions of reading and executing. In that case, the processor may have only a single "read" flag. On those systems, `PROT_READ` implies `PROT_EXEC`. Until recently, the x86 architecture was one such system.

Of course, relying on such behavior is not portable. Portable programs should always set `PROT_EXEC` if they intend to execute code in the mapping.

The reverse situation is one reason for the prevalence of buffer overflow attacks: even if a given mapping does not specify execution permission, the processor may allow execution anyway.

Recent x86 processors have introduced the *NX* (no-execute) bit, which allows for readable, but not executable, mappings. On these newer systems, `PROT_READ` no longer implies `PROT_EXEC`.

The `flags` argument describes the type of mapping and some elements of its behavior. It is a bitwise OR of the following values:

`MAP_FIXED`
: Instructs `mmap()` to treat `addr` as a requirement, not a hint. If the kernel is unable to place the mapping at the given address, the call fails. If the address and length parameters overlap an existing mapping, the overlapped pages are discarded and replaced by the new mapping. As this option requires intimate knowledge of the process address space, it is nonportable, and its use is discouraged.

`MAP_PRIVATE`
: States that the mapping is not shared. The file is mapped copy-on-write, and any changes made in memory by this process are not reflected in the actual file, or in the mappings of other processes.[3]

`MAP_SHARED`
: Shares the mapping with all other processes that map this same file. Writing into the mapping is equivalent to writing to the file. Reads from the mapping will reflect the writes of other processes.

3. Copy-on-write is a concept related to process creation and is described in "Copy-on-write" on page 146.

Either `MAP_SHARED` or `MAP_PRIVATE` must be specified, but not both. Other, more advanced flags are discussed in Chapter 9.

When you map a file descriptor, the file's reference count is incremented. Therefore, you can close the file descriptor after mapping the file, and your process will still have access to it. The corresponding decrement of the file's reference count will occur when you unmap the file, or when the process terminates.

As an example, the following snippet maps the file backed by `fd`, beginning with its first byte, and extending for `len` bytes, into a read-only mapping:

```
void *p;

p = mmap (0, len, PROT_READ, MAP_SHARED, fd, 0);
if (p == MAP_FAILED)
        perror ("mmap");
```

Figure 4-1 shows the effects of parameters supplied with `mmap()` on the mapping between a file and a process's address space.

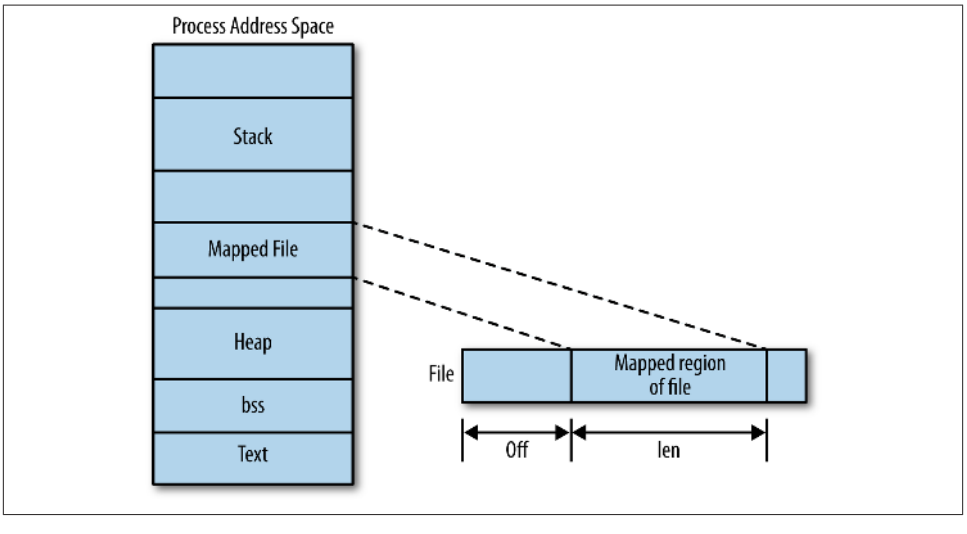

Figure 4-1. Mapping a file into a process's address space

The page size

The *page* is the unit of granularity for the memory management unit (MMU). Consequently it is the smallest unit of memory that can have distinct permissions and behavior. The page is the building block of memory mappings, which in turn are the building blocks of the process address space.

The `mmap()` system call operates on pages. Both the `addr` and `offset` parameters must be aligned on a page-sized boundary. That is, they must be integer multiples of the page size.

Mappings are, therefore, integer multiples of pages. If the `len` parameter provided by the caller is not aligned on a page boundary—perhaps because the underlying file's size is not a multiple of the page size—the mapping is rounded up to the next full page. The bytes inside this added memory, between the last valid byte and the end of the mapping, are zero-filled. Any read from that region will return zeros. Any writes to that memory will not affect the backing file, even if it is mapped as `MAP_SHARED`. Only the original `len` bytes are ever written back to the file.

The standard POSIX method of obtaining the page size is with `sysconf()`, which can retrieve a variety of system-specific information:

```
#include <unistd.h>

long sysconf (int name);
```

A call to `sysconf()` returns the value of the configuration item `name`, or `-1` if `name` is invalid. On error, the call sets `errno` to `EINVAL`. Because `-1` may be a valid value for some items (e.g., limits, where `-1` means no limit), it may be wise to clear `errno` before invocation and check its value after.

POSIX defines `_SC_PAGESIZE` (and a synonym, `_SC_PAGE_SIZE`) to be the size of a page, in bytes. Therefore, obtaining the page size at runtime is simple:

```
long page_size = sysconf (_SC_PAGESIZE);
```

Linux also provides the `getpagesize()` function:

```
#include <unistd.h>

int getpagesize (void);
```

A call to `getpagesize()` will likewise return the size of a page, in bytes. Usage is even simpler than `sysconf()`:

```
int page_size = getpagesize ();
```

Not all Unix systems support this function; it was dropped from the 1003.1-2001 revision of the POSIX standard. It is included here for completeness.

The page size is also statically stored in the macro `PAGE_SIZE`, which is defined in `<sys/user.h>`. Thus, a third way to retrieve the page size is:

```
int page_size = PAGE_SIZE;
```

Unlike the first two options, however, this approach retrieves the system page size at compile time and not runtime. Some architectures support multiple machine types with different page sizes, and some machine types even support multiple page sizes themselves! A single binary should be able to run on all machine types in a given architecture—that is, you should be able to build it once and run it everywhere. Hard-coding the page size would nullify that possibility. Consequently, you should determine the page size at runtime. Because `addr` and `offset` are usually `0`, this requirement is not overly difficult to meet.

Moreover, future kernel versions will likely not export this macro to user space. We cover it in this chapter due to its frequent presence in Unix code, but you should not use it in your own programs. The `sysconf()` approach is your best bet for portability and future compatibility.

Return values and error codes

On success, a call to `mmap()` returns the location of the mapping. On failure, the call returns `MAP_FAILED` and sets `errno` appropriately. A call to `mmap()` never returns `0`.

Possible `errno` values include:

`EACCES`
: The given file descriptor is not a regular file, or the mode with which it was opened conflicts with `prot` or `flags`.

`EAGAIN`
: The file has been locked via a file lock.

`EBADF`
: The given file descriptor is not valid.

`EINVAL`
: One or more of the parameters `addr`, `len`, or `off` are invalid.

`ENFILE`
: The system-wide limit on open files has been reached.

`ENODEV`
: The filesystem on which the file to map resides does not support memory mapping.

`ENOMEM`
: The process does not have enough memory.

`EOVERFLOW`
: The result of `addr+len` exceeds the size of the address space.

`EPERM`
: `PROT_EXEC` was given, but the filesystem is mounted `noexec`.

Associated signals

Two signals are associated with mapped regions:

`SIGBUS`
: This signal is generated when a process attempts to access a region of a mapping that is no longer valid—for example, because the file was truncated after it was mapped.

`SIGSEGV`
: This signal is generated when a process attempts to write to a region that is mapped read-only.

munmap()

Linux provides the `munmap()` system call for removing a mapping created with `mmap()`:

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

int munmap (void *addr, size_t len);
```

A call to `munmap()` removes any mappings that contain pages located anywhere in the process address space starting at `addr`, which must be page-aligned, and continuing for `len` bytes. Once the mapping has been removed, the previously associated memory region is no longer valid, and further access attempts result in a `SIGSEGV` signal.

Normally, `munmap()` is passed the return value and the `len` parameter from a previous invocation of `mmap()`.

On success, `munmap()` returns `0`; on failure, it returns `-1`, and `errno` is set appropriately. The only standard `errno` value is `EINVAL`, which specifies that one or more parameters were invalid.

As an example, the following snippet unmaps any memory regions with pages contained in the interval `[addr,addr+len]`:

```
if (munmap (addr, len) == -1)
        perror ("munmap");
```

Mapping Example

Let's consider a simple example program that uses `mmap()` to print a file chosen by the user to standard out:

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

```
int main (int argc, char *argv[])
{
        struct stat sb;
        off_t len;
        char *p;
        int fd;

        if (argc < 2) {
                fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                perror ("open");
                return 1;
        }

        if (fstat (fd, &sb) == -1) {
                perror ("fstat");
                return 1;
        }

        if (!S_ISREG (sb.st_mode)) {
                fprintf (stderr, "%s is not a file\n", argv[1]);
                return 1;
        }

        p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
        if (p == MAP_FAILED) {
                perror ("mmap");
                return 1;
        }

        if (close (fd) == -1) {
                perror ("close");
                return 1;
        }

        for (len = 0; len < sb.st_size; len++)
                putchar (p[len]);

        if (munmap (p, sb.st_size) == -1) {
                perror ("munmap");
                return 1;
        }

        return 0;
}
```

The only unfamiliar system call in this example should be `fstat()`, which we will cover in Chapter 8. All you need to know at this point is that `fstat()` returns information about a given file. The `S_ISREG()` macro can check some of this information so that we can ensure that the given file is a regular file (as opposed to a device file or a directory) before we map it. The behavior of nonregular files when mapped depends on the backing device. Some device files are mmap-able; other nonregular files are not mmap-able and will set `errno` to `EACCES`.

The rest of the example should be straightforward. The program is passed a filename as an argument. It opens the file, ensures it is a regular file, maps it, closes it, prints the file byte-by-byte to standard out, and then unmaps the file from memory.

Advantages of mmap()

Manipulating files via `mmap()` has a handful of advantages over the standard `read()` and `write()` system calls. Among them are:

- Reading from and writing to a memory-mapped file avoids the extraneous copy that occurs when using the `read()` or `write()` system calls, where the data must be copied to and from a user-space buffer.
- Aside from any potential page faults, reading from and writing to a memory-mapped file does not incur any system call or context switch overhead. It is as simple as accessing memory.
- When multiple processes map the same object into memory, the data is shared among all the processes. Read-only and shared writable mappings are shared in their entirety; private writable mappings have their not-yet-COW (copy-on-write) pages shared.
- Seeking around the mapping involves trivial pointer manipulations. There is no need for the `lseek()` system call.

For these reasons, `mmap()` is a smart choice for many applications.

Disadvantages of mmap()

There are a few points to keep in mind when using `mmap()`:

- Memory mappings are always an integer number of pages in size. Thus, the difference between the size of the backing file and an integer number of pages is "wasted" as slack space. For small files, a significant percentage of the mapping may be wasted. For example, with 4 KB pages, a 7-byte mapping wastes 4,089 bytes.
- The memory mappings must fit into the process's address space. With a 32-bit address space, a large number of various-sized mappings can result in

fragmentation of the address space, making it hard to find large free contiguous regions. This problem, of course, is much less apparent with a 64-bit address space.

- There is overhead in creating and maintaining the memory mappings and associated data structures inside the kernel. This overhead is generally obviated by the elimination of the double copy mentioned in the previous section, particularly for larger and frequently accessed files.

For these reasons, the benefits of `mmap()` are most greatly realized when the mapped file is large (and thus any wasted space is a small percentage of the total mapping), or when the total size of the mapped file is evenly divisible by the page size (and thus there is no wasted space).

Resizing a Mapping

Linux provides the `mremap()` system call for expanding or shrinking the size of a given mapping. This function is Linux-specific:

```
#define _GNU_SOURCE

#include <sys/mman.h>

void * mremap (void *addr, size_t old_size,
               size_t new_size, unsigned long flags);
```

A call to `mremap()` expands or shrinks mapping in the region `[addr,addr+old_size)` to the new size `new_size`. The kernel can potentially move the mapping at the same time, depending on the availability of space in the process's address space and the value of `flags`.

The opening `[` in `[addr,addr+old_size)` indicates that the region starts with (and includes) the low address, whereas the closing `)` indicates that the region stops just before (does not include) the high address. This convention is known as *interval notation*.

The `flags` parameter can be either `0` or `MREMAP_MAYMOVE`, which specifies that the kernel is free to move the mapping if needed to perform the requested resizing. A large resizing is more likely to succeed if the kernel can move the mapping.

Return values and error codes

On success, `mremap()` returns a pointer to the newly resized memory mapping. On failure, it returns `MAP_FAILED` and sets `errno` to one of the following:

`EAGAIN`
: The memory region is locked and cannot be resized.

`EFAULT`
: Some pages in the given range are not valid pages in the process's address space, or there was a problem remapping the given pages.

`EINVAL`
: An argument was invalid.

`ENOMEM`
: The given range cannot be expanded without moving (and `MREMAP_MAYMOVE` was not given), or there is not enough free space in the process's address space.

Libraries such as *glibc* often use `mremap()` to implement an efficient `realloc()`, which is an interface for resizing a block of memory originally obtained via `malloc()`. For example:

```
void * realloc (void *addr, size_t len)
{
        size_t old_size = look_up_mapping_size (addr);
        void *p;

        p = mremap (addr, old_size, len, MREMAP_MAYMOVE);
        if (p == MAP_FAILED)
                return NULL;
        return p;
}
```

This would work only if all `malloc()` allocations were unique anonymous mappings; nonetheless, it stands as a useful example of the performance gains to be had. The example assumes the libc provides a `look_up_mapping_size()` function.

The GNU C library does use `mmap()` and family for performing some memory allocations. We will look at that topic in depth in Chapter 9.

Changing the Protection of a Mapping

POSIX defines the `mprotect()` interface to allow programs to change the permissions of existing regions of memory:

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

int mprotect (const void *addr,
              size_t len,
              int prot);
```

A call to `mprotect()` will change the protection mode for the memory pages contained in `[addr,addr+len)`, where `addr` is page-aligned. The `prot` parameter accepts the same

values as the prot given to mmap(): PROT_NONE, PROT_READ, PROT_WRITE, and PROT_EXEC. These values are not additive; if a region of memory is readable and prot is set to only PROT_WRITE, the call will make the region only writable.

On some systems, mprotect() may operate only on memory mappings previously created via mmap(). On Linux, mprotect() can operate on any region of memory.

Return values and error codes

On success, mprotect() returns 0. On failure, it returns –1, and sets errno to one of the following:

EACCES
: The memory cannot be given the permissions requested by prot. This can happen, for example, if you attempt to set the mapping of a file opened read-only to writable.

EINVAL
: The parameter addr is invalid or not page-aligned.

ENOMEM
: Insufficient kernel memory is available to satisfy the request, or one or more pages in the given memory region are not a valid part of the process's address space.

Synchronizing a File with a Mapping

POSIX provides a memory-mapped equivalent of the fsync() system call that we discussed in Chapter 2:

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

int msync (void *addr, size_t len, int flags);
```

A call to msync() flushes back to disk any changes made to a file mapped via mmap(), synchronizing the mapped file with the mapping. Specifically, the file or subset of a file associated with the mapping starting at memory address addr and continuing for len bytes is synchronized to disk. The addr argument must be page-aligned; it is generally the return value from a previous mmap() invocation.

Without invocation of msync(), there is no guarantee that a dirty mapping will be written back to disk until the file is unmapped. This is different from the behavior of write(), where a buffer is dirtied as part of the writing process and queued for writeback to disk. When writing into a memory mapping, the process directly modifies the file's pages in the kernel's page cache without kernel involvement. The kernel may not synchronize the page cache and the disk anytime soon.

The flags parameter controls the behavior of the synchronizing operation. It is a bitwise OR of the following values:

`MS_SYNC`
: Specifies that synchronization should occur synchronously. The `msync()` call will not return until all pages are written back to disk.

`MS_ASYNC`
: Specifies that synchronization should occur asynchronously. The update is scheduled, but the `msync()` call returns immediately without waiting for the writes to take place.

`MS_INVALIDATE`
: Specifies that all other cached copies of the mapping be invalidated. Any future access to any mappings of this file will reflect the newly synchronized on-disk contents.

One of `MS_ASYNC` or `MS_SYNC` must be specified, but not both.

Usage is simple:

```
if (msync (addr, len, MS_ASYNC) == -1)
        perror ("msync");
```

This example asynchronously synchronizes (say that 10 times fast) to disk the file mapped in the region `[addr,addr+len)`.

Return values and error codes

On success, `msync()` returns `0`. On failure, the call returns `-1` and sets `errno` appropriately. The following are valid `errno` values:

`EINVAL`
: The `flags` parameter has both `MS_SYNC` and `MS_ASYNC` set, a bit other than one of the three valid flags is set, or `addr` is not page-aligned.

`ENOMEM`
: The given memory region (or part of it) is not mapped. Note that Linux will return `ENOMEM`, as POSIX dictates, when asked to synchronize a region that is only partly unmapped, but it will still synchronize any valid mappings in the region.

Before version 2.4.19 of the Linux kernel, `msync()` returned `EFAULT` in place of `ENOMEM`.

Giving Advice on a Mapping

Linux provides a system call named `madvise()` to let processes give the kernel advice and hints on how they intend to use a mapping. The kernel can then optimize its behavior to take advantage of the mapping's intended use. While the Linux kernel dynamically tunes its behavior and generally provides optimal performance without explicit advice, providing such advice can ensure the desired caching and readahead behavior for some workloads.

A call to `madvise()` advises the kernel on how to behave with respect to the pages in the memory map starting at `addr`, and extending for `len` bytes:

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

int madvise (void *addr,
             size_t len,
             int advice);
```

If `len` is `0`, the kernel will apply the advice to the entire mapping that starts at `addr`. The parameter `advice` delineates the advice, which can be one of:

`MADV_NORMAL`
: The application has no specific advice to give on this range of memory. It should be treated as normal.

`MADV_RANDOM`
: The application intends to access the pages in the specified range in a random (nonsequential) order.

`MADV_SEQUENTIAL`
: The application intends to access the pages in the specified range sequentially, from lower to higher addresses.

`MADV_WILLNEED`
: The application intends to access the pages in the specified range in the near future.

`MADV_DONTNEED`
: The application does not intend to access the pages in the specified range in the near future.

The actual behavior modifications that the kernel takes in response to this advice are implementation-specific: POSIX dictates only the meaning of the advice, not any potential consequences. The Linux kernel from 2.6 onward behaves as follows in response to the `advice` values:

`MADV_NORMAL`
: The kernel behaves as usual, performing a moderate amount of readahead.

`MADV_RANDOM`
: The kernel disables readahead, reading only the minimal amount of data on each physical read operation.

`MADV_SEQUENTIAL`
: The kernel performs aggressive readahead.

`MADV_WILLNEED`
: The kernel initiates readahead, reading the given pages into memory.

`MADV_DONTNEED`
: The kernel frees any resources associated with the given pages and discards any dirty and not-yet-synchronized pages. Subsequent accesses to the mapped data will cause the data to be paged in from the backing file or (for anonymous mappings) zero-fill the requested pages.

`MADV_DONTFORK`
: Do not copy these pages into the child process across a fork. This flag, available only in Linux kernel 2.6.16 and later, is needed when managing DMA pages and rarely otherwise.

`MADV_DOFORK+`
: Undo the behavior of `MADV_DONTFORK`.

Typical usage is:

```
int ret;

ret = madvise (addr, len, MADV_SEQUENTIAL);
if (ret < 0)
        perror ("madvise");
```

This call instructs the kernel that the process intends to access the memory region `[addr,addr+len)` sequentially.

Readahead

When the Linux kernel reads files off the disk, it performs an optimization known as *readahead*: when a request is made for a given chunk of a file, the kernel also reads the following chunk of the file. If a request is subsequently made for that chunk—as is the case when reading a file sequentially—the kernel can return the requested data immediately. Because disks have track buffers (basically, hard disks perform their own readahead internally), and because files are generally laid out sequentially on disk, this optimization is low cost.

Some readahead is usually advantageous, but optimal results depend on the question of how much readahead to perform. A sequentially accessed file may benefit from a larger readahead window, while a randomly accessed file may find readahead to be worthless overhead.

As discussed in "Kernel Internals" on page 62, the kernel dynamically tunes the size of the readahead window in response to the hit rate inside that window. More hits imply that a larger window would be advantageous; fewer hits suggest a smaller window. The `madvise()` system call allows applications to influence the window size right off the bat.

Return values and error codes

On success, `madvise()` returns `0`. On failure, it returns `-1`, and `errno` is set appropriately. The following are valid errors:

`EAGAIN`
: An internal kernel resource (probably memory) was unavailable. The process can try again.

`EBADF`
: The region exists but does not map a file.

`EINVAL`
: The parameter `len` is negative, `addr` is not page-aligned, the `advice` parameter is invalid, or the pages were locked or shared with `MADV_DONTNEED`.

`EIO`
: An internal I/O error occurred with `MADV_WILLNEED`.

`ENOMEM`
: The given region is not a valid mapping in this process's address space, or `MADV_WILLNEED` was given, but there is insufficient memory to page in the given regions.

Advice for Normal File I/O

In the previous subsection, we looked at providing advice on memory mappings. In this section, we will look at providing advice to the kernel on normal file I/O. Linux provides two interfaces for such advice giving: `posix_fadvise()` and `readahead()`.

The posix_fadvise() System Call

The first advice interface, as its name alludes, is standardized by POSIX 1003.1-2003:

```
#include <fcntl.h>

int posix_fadvise (int fd,
                   off_t offset,
                   off_t len,
                   int advice);
```

A call to `posix_fadvise()` provides the kernel with the hint `advice` on the file descriptor `fd` in the interval `[offset,offset+len)`. If `len` is `0`, the advice will apply to the range `[offset,length of file]`. Common usage is to specify `0` for `len` and `offset`, applying the advice to the entire file.

The available `advice` options are similar to those for `madvise()`. Exactly one of the following should be provided for `advice`:

`POSIX_FADV_NORMAL`
: The application has no specific advice to give on this range of the file. It should be treated as normal.

`POSIX_FADV_RANDOM`
: The application intends to access the data in the specified range in a random (non-sequential) order.

`POSIX_FADV_SEQUENTIAL`
: The application intends to access the data in the specified range sequentially, from lower to higher addresses.

`POSIX_FADV_WILLNEED`
: The application intends to access the data in the specified range in the near future.

`POSIX_FADV_NOREUSE`
: The application intends to access the data in the specified range in the near future, but only once.

`POSIX_FADV_DONTNEED`
: The application does not intend to access the pages in the specified range in the near future.

As with `madvise()`, the actual response to the given advice is implementation-specific—even different versions of the Linux kernel may react dissimilarly. The following are the current responses:

`POSIX_FADV_NORMAL`
: The kernel behaves as usual, performing a moderate amount of readahead.

`POSIX_FADV_RANDOM`
: The kernel disables readahead, reading only the minimal amount of data on each physical read operation.

`POSIX_FADV_SEQUENTIAL`
: The kernel performs aggressive readahead, doubling the size of the readahead window.

`POSIX_FADV_WILLNEED`
: The kernel initiates readahead to begin reading into memory the given pages.

`POSIX_FADV_NOREUSE`
: Currently, the behavior is the same as for `POSIX_FADV_WILLNEED`; future kernels may perform an additional optimization to exploit the "use once" behavior. This hint does not have an `madvise()` complement.

`POSIX_FADV_DONTNEED`
: The kernel evicts any cached data in the given range from the page cache. Note that this hint, unlike the others, is different in behavior from its `madvise()` counterpart.

As an example, the following snippet instructs the kernel that the entire file represented by the file descriptor `fd` will be accessed in a random, nonsequential manner:

```
int ret;

ret = posix_fadvise (fd, 0, 0, POSIX_FADV_RANDOM);
if (ret == -1)
        perror ("posix_fadvise");
```

Return values and error codes

On success, `posix_fadvise()` returns `0`. On failure, `-1` is returned, and `errno` is set to one of the following values:

`EBADF`
: The given file descriptor is invalid.

`EINVAL`
: The given advice is invalid, the given file descriptor refers to a pipe, or the specified advice cannot be applied to the given file.

The readahead() System Call

The `posix_fadvise()` system call is new to the 2.6 Linux kernel. The `readahead()` system call was previously available to provide behavior identical to the `POSIX_FADV_WILLNEED` hint. Unlike `posix_fadvise()`, `readahead()` is a Linux-specific interface:

```
#define _GNU_SOURCE

#include <fcntl.h>

ssize_t readahead (int fd,
                   off64_t offset,
                   size_t count);
```

A call to `readahead()` populates the page cache with the region `[offset,offset+count)` from the file descriptor `fd`.

Return values and error codes

On success, `readahead()` returns `0`. On failure, it returns `-1`, and `errno` is set to one of the following values:

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

`EINVAL`
: The given file descriptor does not map to a file that supports readahead.

Advice Is Cheap

A handful of common application workloads can readily benefit from a little well-intentioned advice to the kernel. Such advice can go a long way toward mitigating the burden of I/O. With hard disks being so slow, and modern processors being so fast, every little bit helps, and good advice can go a long way.

Before reading a chunk of a file, a process can provide the `POSIX_FADV_WILLNEED` hint to instruct the kernel to read the file into the page cache. The I/O will occur asynchronously in the background. When the application ultimately accesses the file, the operation can complete without generating blocking I/O.

Conversely, after reading or writing a lot of data—say, while continuously streaming video to disk—a process can provide the `POSIX_FADV_DONTNEED` hint to instruct the kernel to evict the given chunk of the file from the page cache. A large streaming operation can continually fill the page cache. If the application never intends to access the data again, this means the page cache will be filled with superfluous data, at the expense of potentially more useful data. Thus, it makes sense for a streaming video application to periodically request that streamed data be evicted from the cache.

A process that intends to read in an entire file can provide the `POSIX_FADV_SEQUENTIAL` hint, instructing the kernel to perform aggressive readahead. Conversely, a process that knows it is going to access a file randomly, seeking to and fro, can provide the `POSIX_FADV_RANDOM` hint, instructing the kernel that readahead will be nothing but worthless overhead.

Synchronized, Synchronous, and Asynchronous Operations

Unix systems use the terms synchronized, nonsynchronized, synchronous, and asynchronous freely, without much regard to the fact that they are confusing—in English, the differences between "synchronous" and "synchronized" do not amount to much!

A *synchronous* write operation does not return until the written data is—at least—stored in the kernel's buffer cache. A synchronous read operation does not return until the read data is stored in the user-space buffer provided by the application. On the other side of the coin, an *asynchronous* write operation may return before the data even leaves user space; an asynchronous read operation may return before the read data is available. That is, the operations may not actually take place when requested, but only be queued for later. Of course, in this case, some mechanism must exist for determining when the operation has actually completed and with what level of success.

A *synchronized* operation is more restrictive and safer than a merely synchronous operation. A synchronized write operation flushes the data to disk, ensuring that the on-disk data is always synchronized vis-à-vis the corresponding kernel buffers. A synchronized read operation always returns the most up-to-date copy of the data, presumably from the disk.

In sum, the terms synchronous and asynchronous refer to whether I/O operations wait for some event (e.g., storage of the data) before returning. The terms synchronized and nonsynchronized, meanwhile, specify exactly *what* event must occur (e.g., writing the data to disk).

Normally, Unix write operations are synchronous and nonsynchronized; read operations are synchronous and synchronized.[4] For write operations, every combination of these characteristics is possible, as Table 4-1 illustrates.

Table 4-1. Synchronicity of write operations

	Synchronized	Nonsynchronized
Synchronous	Write operations do not return until the data is flushed to disk. This is the behavior if `O_SYNC` is specified during file open.	Write operations do not return until the data is stored in kernel buffers. This is the usual behavior.
Asynchronous	Write operations return as soon as the request is queued. Once the write operation ultimately executes, the data is guaranteed to be on disk.	Write operations return as soon as the request is queued. Once the write operation ultimately executes, the data is guaranteed to at least be stored in kernel buffers.

Read operations are always synchronized, as reading stale data makes little sense. Such operations can be either synchronous or asynchronous, however, as illustrated in Table 4-2.

Table 4-2. Synchronicity of read operations

	Synchronized
Synchronous	Read operations do not return until the data, which is up-to-date, is stored in the provided buffer (this is the usual behavior).
Asynchronous	Read operations return as soon as the request is queued, but when the read operation ultimately executes, the data returned is up-to-date.

In Chapter 2, we discussed how to make writes synchronized (via the `O_SYNC` flag) and how to ensure that all I/O is synchronized as of a given point (via `fsync()` and friends). Now, let's look at what it takes to make reads and writes asynchronous.

4. Read operations are technically also nonsynchronized, like write operations, but the kernel ensures that the page cache contains up-to-date data. That is, the page cache's data is always identical to or newer than the data on disk. In this manner, the behavior in practice is always synchronized. There is little argument for behaving any other way.

Asynchronous I/O

Performing asynchronous I/O requires kernel support at the very lowest layers. POSIX 1003.1-2003 defines the *aio* interfaces, which Linux fortunately implements. The *aio* library provides a family of functions for submitting asynchronous I/O and receiving notification upon its completion:

```
#include <aio.h>

/* asynchronous I/O control block */
struct aiocb {
        int aio_fildes;               /* file descriptor */
        int aio_lio_opcode;           /* operation to perform */
        int aio_reqprio;              /* request priority offset */
        volatile void *aio_buf;       /* pointer to buffer */
        size_t aio_nbytes;            /* length of operation */
        struct sigevent aio_sigevent; /* signal number and value */

        /* internal, private members follow... */
};

int aio_read (struct aiocb *aiocbp);
int aio_write (struct aiocb *aiocbp);
int aio_error (const struct aiocb *aiocbp);
int aio_return (struct aiocb *aiocbp);
int aio_cancel (int fd, struct aiocb *aiocbp);
int aio_fsync (int op, struct aiocb *aiocbp);
int aio_suspend (const struct aiocb * const cblist[],
                 int n,
                 const struct timespec *timeout);
```

I/O Schedulers and I/O Performance

In a modern system, the relative performance gap between disks and the rest of the system is quite large—and widening. The worst component of disk performance is the process of moving the read/write head from one part of the disk to another, an operation known as a *seek*. In a world where many operations are measured in a handful of processor cycles (which might take all of a third of a nanosecond each), a single disk seek can average over 8 milliseconds—still a small number, to be sure, but *25 million times longer than a single processor cycle*!

Given the disparity in performance between disk drives and the rest of the system, it would be incredibly crude and inefficient to send I/O requests to the disk in the order in which they are issued. Therefore, modern operating system kernels implement *I/O schedulers*, which work to minimize the number and size of disk seeks by manipulating the order in which I/O requests are serviced and the times at which they are serviced. I/O schedulers work hard to lessen the performance penalties associated with disk access.

Disk Addressing

To understand the role of an I/O scheduler, some background information is necessary. Hard disks address their data using the familiar geometry-based addressing of cylinders, heads, and sectors, or *CHS addressing*. A hard drive is composed of multiple *platters*, each consisting of a single disk, spindle, and read/write head. You can think of each platter as a CD (or record), and the set of platters in a disk as a stack of CDs. Each platter is divided into ring-like *tracks*, like on a CD. Each track is then divided into an integer number of *sectors*.

To locate a specific unit of data on a disk, the drive's logic requires three pieces of information: the cylinder, head, and sector values. The cylinder value specifies the track on which the data resides. If you lay the platters on top of one another, a given track forms a cylinder through each platter. In other words, a cylinder is represented by a track at the same distance from the center on each disk. The head value identifies the exact read/write head (and thus the exact platter) in question. The search is now narrowed down to a single track on a single platter. The disk then uses the sector value to identify an exact sector on the track. The search is now complete: the hard disk knows what platter, what track, and what sector to look in for the data. It can position the read/write head of the correct platter over the correct track and read from or write to the requisite sector.

Thankfully, modern hard disks do not force computers to communicate with their disks in terms of cylinders, heads, and sectors. Instead, contemporary hard drives map a unique *block number* (also called *physical blocks* or *device blocks*) over each cylinder/head/sector triplet—effectively, a block maps to a specific sector. Modern operating systems can then address hard drives using these block numbers—a process known as *logical block addressing* (LBA)—and the hard drive internally translates the block number into the correct CHS address.[5] Although nothing guarantees it, the block-to-CHS mapping tends to be sequential: logical block `n` tends to be physically adjacent on disk to logical block `n` + 1. This sequential mapping is important, as we shall soon see.

Filesystems, meanwhile, exist only in software. They operate on their own units, known as *logical blocks* (sometimes called *filesystem blocks*, or, confusingly, just *blocks*). The logical block size must be an integer multiple of the physical block size. In other words, a filesystem's logical blocks map to one or more of a disk's physical blocks.

The Life of an I/O Scheduler

I/O schedulers perform two basic operations: merging and sorting. *Merging* is the process of taking two or more adjacent I/O requests and combining them into a single

5. Limits on the absolute size of this block number are largely responsible for the various limits on total drive sizes over the years.

request. Consider two requests, one to read from disk block 5, and another to read from disk blocks 6 through 7. These requests can be merged into a single request to read from disk blocks 5 through 7. The total amount of I/O might be the same, but the number of I/O operations is reduced by half.

Sorting, the more important of the two operations, is the process of arranging pending I/O requests in ascending block order. For example, given I/O operations to blocks 52, 109, and 7, the I/O scheduler would sort these requests into the ordering 7, 52, and 109. If a request was then issued to block 81, it would be inserted between the requests to blocks 52 and 109. The I/O scheduler would then dispatch the requests to the disk in the order that they exist in the queue: 7, then 52, then 81, and finally 109.

In this manner, the disk head's movements are minimized. Instead of potentially haphazard movements—here to there and back, seeking all over the disk—the disk head moves in a smooth, linear fashion. Because seeks are the most expensive part of disk I/O, performance is improved.

Helping Out Reads

Each read request must return up-to-date data. Thus, if the requested data is not in the page cache, the reading process must block until the data can be read from disk—a potentially lengthy operation. We call this performance impact *read latency*.

A typical application might initiate several read I/O requests in a short period. Because each request is individually synchronized, the later requests are *dependent* on the earlier ones' completion. Consider reading every file in a directory. The application opens the first file, reads a chunk of it, waits for data, reads another chunk, and so on, until the entire file is read. Then the application starts again, on the next file. The requests become serialized: a subsequent request cannot be issued until the current request completes.

This is in stark contrast to write requests, which (in their default, nonsynchronized state) need not initiate any disk I/O until some time in the future. Thus, from the perspective of a user-space application, write requests *stream*, unencumbered by the performance of the disk. This streaming behavior only compounds the problem for reads: as writes stream, they can hog the kernel and disk's attention. This phenomenon is known as the *writes-starving-reads* problem.

If an I/O scheduler *always* sorted new requests by the order of insertion, it would be possible to starve requests to far-off blocks indefinitely. Consider our previous example. If new requests were continually issued to blocks in, say, the 50s, the request to block 109 would never be serviced. Because read latency is critical, this behavior would greatly hurt system performance. Thus, I/O schedulers employ a mechanism to prevent starvation.

A simple approach—such as the one taken by the 2.4 Linux kernel's I/O scheduler, the *Linus Elevator*[6]—is to simply stop insertion-sorting if there is a sufficiently old request in the queue. This trades overall performance for per-request fairness and, in the case of reads, improves latency. The problem is that this heuristic is a bit too simplistic. Recognizing this, the 2.6 Linux kernel witnessed the demise of the Linus Elevator and unveiled several new I/O schedulers in its place.

The Deadline I/O Scheduler

The Deadline I/O Scheduler was introduced to solve the problems with the 2.4 I/O scheduler and traditional elevator algorithms in general. The Linus Elevator maintains a sorted list of pending I/O requests. The I/O request at the head of the queue is the next one to be serviced. The Deadline I/O Scheduler keeps this queue but kicks things up a notch by introducing two additional queues: the *read FIFO queue* and the *write FIFO queue*. The items in each of these queues are sorted by submission time (effectively, the first in is the first out). The read FIFO queue, as its name suggests, contains only read requests. The write FIFO queue, likewise, contains only write requests. Each request in the FIFO queues is assigned an expiration value. The read FIFO queue has an expiration time of 500 milliseconds. The write FIFO queue has an expiration time of 5 seconds.

When a new I/O request is submitted, it is insertion-sorted into the standard queue and placed at the tail of its respective (read or write) FIFO queue. Normally, the hard drive is sent I/O requests from the head of the standard sorted queue. This maximizes global throughput by minimizing seeks, as the normal queue is sorted by block number (as with the Linus Elevator).

When the item at the head of one of the FIFO queues grows older than the expiration value associated with its queue, however, the I/O scheduler stops dispatching I/O requests from the standard queue and begins servicing requests from that queue—the request at the head of the FIFO queue is serviced, plus a couple of extras for good measure. The I/O scheduler needs to check and handle only the requests at the head of the queue, as those are the oldest requests.

In this manner, the Deadline I/O Scheduler can enforce a soft deadline on I/O requests. Although it makes no promise that an I/O request will be serviced before its expiration time, the I/O scheduler generally services requests near their expiration times. Thus, the Deadline I/O Scheduler continues to provide good global throughput without starving any one request for an unacceptably long time. Because read requests are given shorter expiration times, the writes-starving-reads problem is minimized.

6. Yes, the man has an I/O scheduler named after him. I/O schedulers are sometimes called elevator algorithms because they solve a problem similar to that of keeping an elevator running smoothly.

The Anticipatory I/O Scheduler

The Deadline I/O Scheduler's behavior is good, but not perfect. Recall our discussion on read dependency. With the Deadline I/O Scheduler, the first read request in a series of reads is serviced in short order, at or before its expiration time, and the I/O scheduler then returns to servicing I/O requests from the sorted queue—so far, so good. But what if the application then swoops in and hits us with another read request? Eventually its expiration time will also approach, and the I/O scheduler will submit it to the disk, which will seek over to promptly handle the request, then seek back to continue handling requests from the sorted queue. This seeking back and forth can continue for some time because many applications exhibit this behavior. While latency is kept to a minimum, global throughput is not very good because the read requests keep coming in, and the disk has to keep seeking back and forth to handle them. Performance would be improved if the disk just took a break to wait for another read and did not move away to service the sorted queue again. But, unfortunately, by the time the application is scheduled and submits its next dependent read request, the I/O scheduler has already shifted gears.

The problem again stems from those darn dependent reads. Each new read request is issued only when the previous one is returned, but by the time the application receives the read data, is scheduled to run, and submits its next read request, the I/O scheduler has moved on and begun servicing other requests. This results in a wasted pair of seeks for each read: the disk seeks to the read, services it, and then seeks back. If only there was some way for the I/O scheduler to know—to anticipate—that another read would soon be submitted to the same part of the disk, instead of seeking back and forth, it could wait in anticipation of the next read. Saving those awful seeks certainly would be worth a few milliseconds of waiting.

This is exactly how the Anticipatory I/O Scheduler operates. It began life as the Deadline I/O Scheduler but was gifted with the addition of an anticipation mechanism. When a read request is submitted, the Anticipatory I/O Scheduler services it within its deadline, as usual. Unlike the Deadline I/O Scheduler, however, the Anticipatory I/O Scheduler then sits and waits, doing nothing, for up to 6 milliseconds. Chances are good that the application will issue another read to the same part of the filesystem during those six milliseconds. If so, that request is serviced immediately, and the Anticipatory I/O Scheduler waits some more. If 6 milliseconds go by without a read request, the Anticipatory I/O Scheduler decides it has guessed wrong and returns to whatever it was doing before (i.e., servicing the standard sorted queue). If even a moderate number of requests are anticipated correctly, a great deal of time—two expensive seeks' worth at each go—is saved. Because most reads are dependent, the anticipation pays off much of the time.

The CFQ I/O Scheduler

The Completely Fair Queuing (CFQ) I/O Scheduler works to achieve similar goals, albeit via a different approach.[7] With CFQ, each process is assigned its own queue, and each queue is assigned a timeslice. The I/O scheduler visits each queue in a round-robin fashion, servicing requests from the queue until the queue's timeslice is exhausted, or until no more requests remain. In the latter case, the CFQ I/O Scheduler will then sit idle for a brief period—by default, 10 ms—waiting for a new request on the queue. If the anticipation pays off, the I/O scheduler avoids seeking. If not, the waiting was in vain, and the scheduler moves on to the next process's queue.

Within each process's queue, synchronized requests (such as reads) are given priority over nonsynchronized requests. In this manner, CFQ favors reads and prevents the writes-starving-reads problem. Due to the per-process queue setup, the CFQ I/O Scheduler is fair to all processes while still providing good global performance.

The CFQ I/O Scheduler is well suited to most workloads, and makes an excellent first choice.

The Noop I/O Scheduler

The Noop I/O Scheduler is the most basic of the available schedulers. It performs no sorting whatsoever, only basic merging. It is used for specialized devices that do not require (or that perform) their own request sorting.

Solid-State Drives

Solid-state drives (SSDs) such as flash drives grow more popular each year. Whole classes of devices, such as mobile phones and tablets, have no rotational storage devices; everything is flash. SSDs such as flash drives have significantly lower seek times than classic hard drives, as there is no rotational cost to finding a given block of data. Instead, SSDs are referenced not unlike random-access memory: while it can be more efficient to read large chunks of contiguous data in one fell swoop, there isn't a penalty for accessing data elsewhere on the drive.

Consequently, the benefits of sorting I/O requests are significantly smaller (if not zero) for solid-state drives and the utility of I/O schedulers is less for such devices. Many systems thus use the Noop I/O Scheduler for solid-state storage, as it provides merging (which is beneficial) but not sorting. Systems, however, that wish to optimize for interactive performance prefer the fairness of the CFQ I/O Scheduler, even for SSDs.

7. The following text discusses the CFQ I/O Scheduler as it is currently implemented. Previous incarnations did not use timeslices or the anticipation heuristic, but operated in a similar fashion.

Selecting and Configuring Your I/O Scheduler

The default I/O scheduler is selectable at boot time via the *iosched* kernel command-line parameter. Valid options are *as*, *cfq*, *deadline*, and *noop*. The I/O scheduler is also runtime-selectable on a per-device basis via */sys/block/[device]/queue/scheduler*, where `device` is the block device in question. Reading this file returns the current I/O scheduler; writing one of the valid options to this file sets the I/O scheduler. For example, to set the device *hda* to the CFQ I/O Scheduler, one would do the following:

```
# echo cfq > /sys/block/hda/queue/scheduler
```

The directory */sys/block/[device]/queue/iosched* contains files that allow the administrator to retrieve and set tunable values related to the I/O scheduler. The exact options depend on the current I/O scheduler. Changing any of these settings requires root privileges.

A good programmer writes programs that are agnostic to the underlying I/O subsystem. Nonetheless, knowledge of this subsystem can surely help one write optimal code.

Optimzing I/O Performance

665.180bBecause disk I/O is so slow relative to the performance of other components in the system, yet I/O is such an important aspect of modern computing, maximizing I/O performance is crucial.

Minimizing I/O operations (by coalescing many smaller operations into fewer larger operations), performing block-size-aligned I/O, or using user buffering (see Chapter 3) are important tools in the system programmer's kit. Similarly, taking advantage of advanced I/O techniques, such as vectored I/O, positional I/O (see Chapter 2), and asynchronous I/O, are important patterns to consider when system programming.

The most demanding mission-critical and I/O-intense applications, however, can employ additional tricks to maximize performance. Although the Linux kernel, as discussed previously, utilizes advanced I/O schedulers to minimize dreaded disk seeks, user-space applications can work toward the same end, in a similar fashion, to further improve performance.

Scheduling I/O in user space

I/O-intensive applications that issue a large number of I/O requests and need to extract every ounce of performance can sort and merge their pending I/O requests, performing the same duties as the Linux I/O scheduler.[8]

8. One should apply the techniques discussed here only to I/O-intensive, mission-critical applications. Sorting the I/O requests—assuming there is even anything to sort—of applications that do not issue many such requests is silly and unneeded.

Why perform the same work twice, if you know the I/O scheduler will sort requests block-wise, minimizing seeks and allowing the disk head to move in a smooth, linear fashion? Consider an application that submits a large number of unsorted I/O requests. These requests arrive in the I/O scheduler's queue in a generally random order. The I/O scheduler does its job, sorting and merging the requests before sending them out to the disk—but the requests start hitting the disk while the application is still generating I/O and submitting requests. The I/O scheduler is able to sort only a subset of requests at a time.

Therefore, if an application is generating many requests—particularly if they are for data all over the disk—it can benefit from sorting the requests before submitting them, ensuring they reach the I/O scheduler in the desired order.

A user-space application is not bestowed with access to the same information as the kernel, however. At the lowest levels inside the I/O scheduler, requests are already specified in terms of physical disk blocks. Sorting them is trivial. But, in user space, requests are specified in terms of files and offsets. User-space applications must probe for information and make educated guesses about the layout of the filesystem.

Given the goal of determining the most seek-friendly ordering given a list of I/O requests to specific files, user-space applications have a couple of options. They can sort based on:

- The full path
- The inode number
- The physical disk block of the file

Each of these options involves a trade-off. Let's look at each briefly.

Sorting by path

Sorting by the pathname is the easiest, yet least effective, way of approximating a block-wise sort. Due to the layout algorithms used by most filesystems, the files in each directory—and thus the directories sharing a parent directory—tend to be adjacent on disk. The probability that files in the same directory were created around the same time only amplifies this characteristic.

Sorting by path, therefore, roughly approximates the physical locations of files on the disk. It is certainly true that two files in the same directory have a better chance of being located near each other than two files in radically different parts of the filesystem. The downside of this approach is that it fails to take into account fragmentation: the more fragmented the filesystem, the less useful is sorting by path. Even ignoring fragmentation, a path-wise sort only approximates the actual block-wise ordering. On the upside, a path-wise sort is at least somewhat applicable to all filesystems. No matter the approach

to file layout, temporal locality suggests a path-wise sort will be at least mildly accurate. It is also an easy sort to perform.

Sorting by inode

Inodes are Unix constructs that contain the metadata associated with individual files. While a file's data may consume multiple physical disk blocks, each file has exactly one inode, which contains information such as the file's size, permissions, owner, and so on. We will discuss inodes in depth in Chapter 8. For now, you need to know two facts: that every file has an inode associated with it and that the inodes are assigned unique numbers.

Sorting by inode is better than sorting by path, assuming that the relation:

```
file i's inode number < file j's inode number
```

implies, *in general*, that:

```
physical blocks of file i < physical blocks of file j
```

This is certainly true for Unix-style filesystems such as *ext3* and *ext4*. Anything is possible for filesystems that do not employ actual inodes, but the inode number (whatever it may map to) is still a good first-order approximation.

Obtaining the inode number is done via the `stat()` system call, also discussed in Chapter 8. Given the inode associated with the file involved in each I/O request, the requests can be sorted in ascending order by inode number.

Here is a simple program that prints out the inode number of a given file:

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

/*
 * get_inode - returns the inode of the file associated
 * with the given file descriptor, or -1 on failure
 */
int get_inode (int fd)
{
        struct stat buf;
        int ret;

        ret = fstat (fd, &buf);
        if (ret < 0) {
                perror ("fstat");
                return -1;
        }

        return buf.st_ino;
```

```
}

int main (int argc, char *argv[])
{
        int fd, inode;

        if (argc < 2) {
                fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd < 0) {
                perror ("open");
                return 1;
        }

        inode = get_inode (fd);
        printf ("%d\n", inode);

        return 0;
}
```

The `get_inode()` function is easily adaptable for use in your programs.

Sorting by inode number has a few upsides: the inode number is easy to obtain, is easy to sort on, and is a good approximation of the physical file layout. The major downsides are that fragmentation degrades the approximation, that the approximation is just a guess, and that the approximation is less accurate for non-Unix filesystems. Nonetheless, this is the most commonly used method for scheduling I/O requests in user space.

Sorting by physical block

The best approach to designing your own elevator algorithm, of course, is to sort by physical disk block. As discussed earlier, each file is broken up into logical blocks, which are the smallest allocation units of a filesystem. The size of a logical block is filesystem-dependent; each logical block maps to a single physical block. We can thus find the number of logical blocks in a file, determine what physical blocks they map to, and sort based on that.

The kernel provides a method for obtaining the physical disk block from the logical block number of a file. This is done via the `ioctl()` system call, discussed in Chapter 8, with the `FIBMAP` command:

```
ret = ioctl (fd, FIBMAP, &block);
if (ret < 0)
        perror ("ioctl");
```

Here, `fd` is the file descriptor of the file in question, and `block` is the logical block whose physical block we want to determine. On successful return, `block` is replaced with the

physical block number. The logical blocks passed in are zero-indexed and file-relative. That is, if a file is made up of eight logical blocks, valid values are 0 through 7.

Finding the logical-to-physical-block mapping is thus a two-step process. First, we must determine the number of blocks in a given file. This is done via the `stat()` system call. Second, for each logical block, we must issue an `ioctl()` request to find the corresponding physical block.

Here is a sample program to do just that for a file passed in on the command line:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <linux/fs.h>

/*
 * get_block - for the file associated with the given fd, returns
 * the physical block mapping to logical_block
 */
int get_block (int fd, int logical_block)
{
        int ret;

        ret = ioctl (fd, FIBMAP, &logical_block);
        if (ret < 0) {
                perror ("ioctl");
                return -1;
        }

        return logical_block;
}

/*
 * get_nr_blocks - returns the number of logical blocks
 * consumed by the file associated with fd
 */
int get_nr_blocks (int fd)
{
        struct stat buf;
        int ret;

        ret = fstat (fd, &buf);
        if (ret < 0) {
                perror ("fstat");
                return -1;
        }
        return buf.st_blocks;
}
```

```
/*
 * print_blocks - for each logical block consumed by the file
 * associated with fd, prints to standard out the tuple
 * "(logical block, physical block)"
 */
void print_blocks (int fd)
{
        int nr_blocks, i;

        nr_blocks = get_nr_blocks (fd);
        if (nr_blocks < 0) {
                fprintf (stderr, "get_nr_blocks failed!\n");
                return;
        }

        if (nr_blocks == 0) {
                printf ("no allocated blocks\n");
                return;
        } else if (nr_blocks == 1)
                printf ("1 block\n\n");
        else
                printf ("%d blocks\n\n", nr_blocks);

        for (i = 0; i < nr_blocks; i++) {
                int phys_block;

                phys_block = get_block (fd, i);
                if (phys_block < 0) {
                        fprintf (stderr, "get_block failed!\n");
                        return;
                }
                if (!phys_block)
                        continue;

                printf ("(%u, %u) ", i, phys_block);
        }

        putchar ('\n');
}

int main (int argc, char *argv[])
{
        int fd;

        if (argc < 2) {
                fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd < 0) {
                perror ("open");
```

```
                return 1;
        }

        print_blocks (fd);

        return 0;
}
```

Because files tend to be contiguous, and it would be difficult (at best) to sort our I/O requests on a per-logical-block basis, it makes sense to sort based on the location of just the first logical block of a given file. Consequently, `get_nr_blocks()` is not needed, and our applications can sort based on the return value from:

```
get_block (fd, 0);
```

The downside of `FIBMAP` is that it requires the `CAP_SYS_RAWIO` capability—effectively, root privileges. Consequently, nonroot applications cannot make use of this approach. Further, while the `FIBMAP` command is standardized, its actual implementation is left up to the filesystems. While common systems such as *ext2* and *ext3* support it, a more esoteric beast may not. The `ioctl()` call will return `EINVAL` if `FIBMAP` is not supported.

Among the pros of this approach, however, is that it returns the *actual* physical disk block at which a file resides, which is exactly what you want to sort on. Even if you sort all I/O to a single file based on the location of just one block (the kernel's I/O scheduler sorts each individual request on a block-wise basis), this approach comes very close to the optimal ordering. The root requirement, however, is a bit of a nonstarter for many.

Conclusion

Over the course of the last three chapters, we have touched on all aspects of file I/O in Linux. In Chapter 2, we looked at the basics of Linux file I/O—really, the basis of Unix programming—with system calls such as `read()`, `write()`, `open()`, and `close()`. In Chapter 3, we discussed user-space buffering and the standard C library's implementation thereof. In this chapter, we discussed various facets of advanced I/O, from the more-powerful-but-more-complex I/O system calls to optimization techniques and the dreaded performance-sucking disk seek.

In the next two chapters, we will look at process management: creating, destroying, and managing processes. Onward!