```
    length = getLong(argv[2], GN_ANY_BASE, "length");
    offset = (argc > 3) ? getLong(argv[3], GN_ANY_BASE, "offset") : 0;
    alignment = (argc > 4) ? getLong(argv[4], GN_ANY_BASE, "alignment") : 4096;

    fd = open(argv[1], O_RDONLY | O_DIRECT);
    if (fd == -1)
        errExit("open");

    /* memalign() allocates a block of memory aligned on an address that
       is a multiple of its first argument. The following expression
       ensures that 'buf' is aligned on a non-power-of-two multiple of
       'alignment'. We do this to ensure that if, for example, we ask
       for a 256-byte aligned buffer, then we don't accidentally get
       a buffer that is also aligned on a 512-byte boundary.

       The '(char *)' cast is needed to allow pointer arithmetic (which
       is not possible on the 'void *' returned by memalign()). */

    buf = (char *) memalign(alignment * 2, length + alignment) + alignment;
    if (buf == NULL)
        errExit("memalign");

    if (lseek(fd, offset, SEEK_SET) == -1)
        errExit("lseek");

    numRead = read(fd, buf, length);
    if (numRead == -1)
        errExit("read");
    printf("Read %ld bytes\n", (long) numRead);

    exit(EXIT_SUCCESS);
}
```

filebuff/direct_read.c

13.7 Mixing Library Functions and System Calls for File I/O

It is possible to mix the use of system calls and the standard C library functions to perform I/O on the same file. The *fileno()* and *fdopen()* functions assist us with this task.

```
#include <stdio.h>

int fileno(FILE *stream);
```

Returns file descriptor on success, or −1 on error

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

Returns (new) file pointer on success, or NULL on error

Given a stream, *fileno()* returns the corresponding file descriptor (i.e., the one that the *stdio* library has opened for this stream). This file descriptor can then be used in the usual way with I/O system calls such as *read()*, *write()*, *dup()*, and *fcntl()*.

The *fdopen()* function is the converse of *fileno()*. Given a file descriptor, it creates a corresponding stream that uses this descriptor for its I/O. The *mode* argument is the same as for *fopen()*; for example, *r* for read, *w* for write, or *a* for append. If this argument is not consistent with the access mode of the file descriptor *fd*, then *fdopen()* fails.

The *fdopen()* function is especially useful for descriptors referring to files other than regular files. As we'll see in later chapters, the system calls for creating sockets and pipes always return file descriptors. To use the *stdio* library with these file types, we must use *fdopen()* to create a corresponding file stream.

When using the *stdio* library functions in conjunction with I/O system calls to perform I/O on disk files, we must keep buffering issues in mind. I/O system calls transfer data directly to the kernel buffer cache, while the *stdio* library waits until the stream's user-space buffer is full before calling *write()* to transfer that buffer to the kernel buffer cache. Consider the following code used to write to standard output:

```
printf("To man the world is twofold, ");
write(STDOUT_FILENO, "in accordance with his twofold attitude.\n", 41);
```

In the usual case, the output of the *printf()* will typically appear *after* the output of the *write()*, so that this code yields the following output:

```
in accordance with his twofold attitude.
To man the world is twofold,
```

When intermingling I/O system calls and *stdio* functions, judicious use of *fflush()* may be required to avoid this problem. We could also use *setvbuf()* or *setbuf()* to disable buffering, but doing so might impact I/O performance for the application, since each output operation would then result in the execution of a *write()* system call.

> SUSv3 goes to some length specifying the requirements for an application to be able to mix the use of I/O system calls and *stdio* functions. See the section headed *Interaction of File Descriptors and Standard I/O Streams* under the chapter *General Information* in the *System Interfaces* (XSH) volume for details.

13.8 Summary

Buffering of input and output data is performed by the kernel, and also by the *stdio* library. In some cases, we may wish to prevent buffering, but we need to be aware of the impact this has on application performance. Various system calls and library functions can be used to control kernel and *stdio* buffering and to perform one-off buffer flushes.

A process can use *posix_fadvise()* to advise the kernel of its likely pattern for accessing data from a specified file. The kernel may use this information to optimize the use of the buffer cache, thus improving I/O performance.

The Linux-specific *open()* `O_DIRECT` flag allows specialized applications to bypass the buffer cache.

The *fileno()* and *fdopen()* functions assist us with the task of mixing system calls and standard C library functions to perform I/O on the same file. Given a stream, *fileno()* returns the corresponding file descriptor; *fdopen()* performs the converse operation, creating a new stream that employs a specified open file descriptor.

Further information

[Bach, 1986] describes the implementation and advantages of the buffer cache on System V. [Goodheart & Cox, 1994] and [Vahalia, 1996] also describe the rationale and implementation of the System V buffer cache. Further relevant information specific to Linux can be found in [Bovet & Cesati, 2005] and [Love, 2010].

13.9 Exercises

13-1. Using the *time* built-in command of the shell, try timing the operation of the program in Listing 4-1 (`copy.c`) on your system.

a) Experiment with different file and buffer sizes. You can set the buffer size using the *−DBUF_SIZE=nbytes* option when compiling the program.

b) Modify the *open()* system call to include the `O_SYNC` flag. How much difference does this make to the speed for various buffer sizes?

c) Try performing these timing tests on a range of file systems (e.g., *ext3*, *XFS*, *Btrfs*, and *JFS*). Are the results similar? Are the trends the same when going from small to large buffer sizes?

13-2. Time the operation of the `filebuff/write_bytes.c` program (provided in the source code distribution for this book) for various buffer sizes and file systems.

13-3. What is the effect of the following statements?

```
fflush(fp);
fsync(fileno(fp));
```

13-4. Explain why the output of the following code differs depending on whether standard output is redirected to a terminal or to a disk file.

```
printf("If I had more time, \n");
write(STDOUT_FILENO, "I would have written you a shorter letter.\n", 43);
```

13-5. The command *tail [−n num] file* prints the last *num* lines (ten by default) of the named file. Implement this command using I/O system calls (*lseek()*, *read()*, *write()*, and so on). Keep in mind the buffering issues described in this chapter, in order to make the implementation efficient.