4

FILE I/O: THE UNIVERSAL I/O MODEL

We now start to look in earnest at the system call API. Files are a good place to start, since they are central to the UNIX philosophy. The focus of this chapter is the system calls used for performing file input and output.

We introduce the concept of a file descriptor, and then look at the system calls that constitute the so-called universal I/O model. These are the system calls that open and close a file, and read and write data.

We focus on I/O on disk files. However, much of the material covered here is relevant for later chapters, since the same system calls are used for performing I/O on all types of files, such as pipes and terminals.

Chapter 5 extends the discussion in this chapter with further details on file I/O. One other aspect of file I/O, buffering, is complex enough to deserve its own chapter. Chapter 13 covers I/O buffering in the kernel and in the *stdio* library.

4.1 Overview

All system calls for performing I/O refer to open files using a *file descriptor*, a (usually small) nonnegative integer. File descriptors are used to refer to all types of open files, including pipes, FIFOs, sockets, terminals, devices, and regular files. Each process has its own set of file descriptors.

By convention, most programs expect to be able to use the three standard file descriptors listed in Table 4-1. These three descriptors are opened on the program's

behalf by the shell, before the program is started. Or, more precisely, the program inherits copies of the shell's file descriptors, and the shell normally operates with these three file descriptors always open. (In an interactive shell, these three file descriptors normally refer to the terminal under which the shell is running.) If I/O redirections are specified on a command line, then the shell ensures that the file descriptors are suitably modified before starting the program.

Table 4-1: Standard file descriptors

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	`STDIN_FILENO`	*stdin*
1	standard output	`STDOUT_FILENO`	*stdout*
2	standard error	`STDERR_FILENO`	*stderr*

When referring to these file descriptors in a program, we can use either the numbers (0, 1, or 2) or, preferably, the POSIX standard names defined in `<unistd.h>`.

> Although the variables *stdin*, *stdout*, and *stderr* initially refer to the process's standard input, output, and error, they can be changed to refer to any file by using the *freopen()* library function. As part of its operation, *freopen()* may change the file descriptor underlying the reopened stream. In other words, after an *freopen()* on *stdout*, for example, it is no longer safe to assume that the underlying file descriptor is still 1.

The following are the four key system calls for performing file I/O (programming languages and software packages typically employ these calls only indirectly, via I/O libraries):

- *fd = open(pathname, flags, mode)* opens the file identified by *pathname*, returning a file descriptor used to refer to the open file in subsequent calls. If the file doesn't exist, *open()* may create it, depending on the settings of the *flags* bit-mask argument. The *flags* argument also specifies whether the file is to be opened for reading, writing, or both. The *mode* argument specifies the permissions to be placed on the file if it is created by this call. If the *open()* call is not being used to create a file, this argument is ignored and can be omitted.
- *numread = read(fd, buffer, count)* reads at most *count* bytes from the open file referred to by *fd* and stores them in *buffer*. The *read()* call returns the number of bytes actually read. If no further bytes could be read (i.e., end-of-file was encountered), *read()* returns 0.
- *numwritten = write(fd, buffer, count)* writes up to *count* bytes from *buffer* to the open file referred to by *fd*. The *write()* call returns the number of bytes actually written, which may be less than *count*.
- *status = close(fd)* is called after all I/O has been completed, in order to release the file descriptor *fd* and its associated kernel resources.

Before we launch into the details of these system calls, we provide a short demonstration of their use in Listing 4-1. This program is a simple version of the *cp(1)* command. It copies the contents of the existing file named in its first command-line argument to the new file named in its second command-line argument.

We can use the program in Listing 4-1 as follows:

```
$ ./copy oldfile newfile
```

Listing 4-1: Using I/O system calls

fileio/copy.c

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

#ifndef BUF_SIZE        /* Allow "cc -D" to override definition */
#define BUF_SIZE 1024
#endif

int
main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);

    /* Open input and output files */

    inputFd = open(argv[1], O_RDONLY);
    if (inputFd == -1)
        errExit("opening file %s", argv[1]);

    openFlags = O_CREAT | O_WRONLY | O_TRUNC;
    filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH;      /* rw-rw-rw- */
    outputFd = open(argv[2], openFlags, filePerms);
    if (outputFd == -1)
        errExit("opening file %s", argv[2]);

    /* Transfer data until we encounter end of input or an error */

    while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
        if (write(outputFd, buf, numRead) != numRead)
            fatal("write() returned error or partial write occurred");
    if (numRead == -1)
        errExit("read");

    if (close(inputFd) == -1)
        errExit("close input");
    if (close(outputFd) == -1)
        errExit("close output");

    exit(EXIT_SUCCESS);
}
```

fileio/copy.c

4.2 Universality of I/O

One of the distinguishing features of the UNIX I/O model is the concept of *universality of I/O*. This means that the same four system calls—*open()*, *read()*, *write()*, and *close()*—are used to perform I/O on all types of files, including devices such as terminals. Consequently, if we write a program using only these system calls, that program will work on any type of file. For example, the following are all valid uses of the program in Listing 4-1:

```
$ ./copy test test.old          Copy a regular file
$ ./copy a.txt /dev/tty         Copy a regular file to this terminal
$ ./copy /dev/tty b.txt         Copy input from this terminal to a regular file
$ ./copy /dev/pts/16 /dev/tty   Copy input from another terminal
```

Universality of I/O is achieved by ensuring that each file system and device driver implements the same set of I/O system calls. Because details specific to the file system or device are handled within the kernel, we can generally ignore device-specific factors when writing application programs. When access to specific features of a file system or device is required, a program can use the catchall *ioctl()* system call (Section 4.8), which provides an interface to features that fall outside the universal I/O model.

4.3 Opening a File: *open()*

The *open()* system call either opens an existing file or creates and opens a new file.

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

int open(const char *pathname, int flags, ... /* mode_t mode */);
                                    Returns file descriptor on success, or –1 on error
```

The file to be opened is identified by the *pathname* argument. If *pathname* is a symbolic link, it is dereferenced. On success, *open()* returns a file descriptor that is used to refer to the file in subsequent system calls. If an error occurs, *open()* returns –1 and *errno* is set accordingly.

The *flags* argument is a bit mask that specifies the *access mode* for the file, using one of the constants shown in Table 4-2.

> Early UNIX implementations used the numbers 0, 1, and 2 instead of the names shown in Table 4-2. Most modern UNIX implementations define these constants to have those values. Thus, we can see that `O_RDWR` is not equivalent to `O_RDONLY | O_WRONLY`; the latter combination is a logical error.

When *open()* is used to create a new file, the *mode* bit-mask argument specifies the permissions to be placed on the file. (The *mode_t* data type used to type *mode* is an integer type specified in SUSv3.) If the *open()* call doesn't specify `O_CREAT`, *mode* can be omitted.

Table 4-2: File access modes

Access mode	Description
O_RDONLY	Open the file for reading only
O_WRONLY	Open the file for writing only
O_RDWR	Open the file for both reading and writing

We describe file permissions in detail in Section 15.4. Later, we'll see that the permissions actually placed on a new file depend not just on the *mode* argument, but also on the process umask (Section 15.4.6) and the (optionally present) default access control list (Section 17.6) of the parent directory. In the meantime, we'll just note that the *mode* argument can be specified as a number (typically in octal) or, preferably, by ORing (|) together zero or more of the bit-mask constants listed in Table 15-4, on page 295.

Listing 4-2 shows examples of the use of *open()*, some of which employ additional *flags* bits that we describe shortly.

Listing 4-2: Examples of the use of *open()*

```
/* Open existing file for reading */

fd = open("startup", O_RDONLY);
if (fd == -1)
    errExit("open");

/* Open new or existing file for reading and writing, truncating to zero
   bytes; file permissions read+write for owner, nothing for all others */

fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");

/* Open new or existing file for writing; writes should always
   append to end of file */

fd = open("w.log", O_WRONLY | O_CREAT | O_APPEND,
                   S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
```

File descriptor number returned by *open()*

SUSv3 specifies that if *open()* succeeds, it is guaranteed to use the lowest-numbered unused file descriptor for the process. We can use this feature to ensure that a file is opened using a particular file descriptor. For example, the following sequence ensures that a file is opened using standard input (file descriptor 0).

```
if (close(STDIN_FILENO) == -1)      /* Close file descriptor 0 */
    errExit("close");

fd = open(pathname, O_RDONLY);
if (fd == -1)
    errExit("open");
```

Since file descriptor 0 is unused, *open()* is guaranteed to open the file using that descriptor. In Section 5.5, we look at the use of *dup2()* and *fcntl()* to achieve a similar result, but with more flexible control over the file descriptor used. In that section, we also show an example of why it can be useful to control the file descriptor on which a file is opened.

4.3.1 The *open() flags* Argument

In some of the example *open()* calls shown in Listing 4-2, we included other bits (`O_CREAT`, `O_TRUNC`, and `O_APPEND`) in *flags* in addition to the file access mode. We now consider the *flags* argument in more detail. Table 4-3 summarizes the full set of constants that can be bit-wise ORed (|) in *flags*. The final column indicates which of these constants are standardized in SUSv3 or SUSv4.

Table 4-3: Values for the *flags* argument of *open()*

Flag	Purpose	SUS?
`O_RDONLY`	Open for reading only	v3
`O_WRONLY`	Open for writing only	v3
`O_RDWR`	Open for reading and writing	v3
`O_CLOEXEC`	Set the close-on-exec flag (since Linux 2.6.23)	v4
`O_CREAT`	Create file if it doesn't already exist	v3
`O_DIRECTORY`	Fail if *pathname* is not a directory	v4
`O_EXCL`	With `O_CREAT`: create file exclusively	v3
`O_LARGEFILE`	Used on 32-bit systems to open large files	
`O_NOCTTY`	Don't let *pathname* become the controlling terminal	v3
`O_NOFOLLOW`	Don't dereference symbolic links	v4
`O_TRUNC`	Truncate existing file to zero length	v3
`O_APPEND`	Writes are always appended to end of file	v3
`O_ASYNC`	Generate a signal when I/O is possible	
`O_DIRECT`	File I/O bypasses buffer cache	
`O_DSYNC`	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
`O_NOATIME`	Don't update file last access time on *read()* (since Linux 2.6.8)	
`O_NONBLOCK`	Open in nonblocking mode	v3
`O_SYNC`	Make file writes synchronous	v3

The constants in Table 4-3 are divided into the following groups:

- *File access mode flags*: These are the O_RDONLY, O_WRONLY, and O_RDWR flags described earlier. Only one of these values should be specified in *flags*. The access mode can be retrieved using the *fcntl()* F_GETFL operation (Section 5.3).
- *File creation flags*: These are the flags shown in the second part of Table 4-3. They control various aspects of the behavior of the *open()* call itself. These flags can't be retrieved or changed.
- *Open file status flags*: These are the remaining flags in Table 4-3. They affect the semantics of subsequent I/O system calls and can be retrieved and modified using the *fcntl()* F_GETFL and F_SETFL operations (Section 5.3). These flags are sometimes simply called the *file status flags*.

  Since kernel 2.6.22, the Linux-specific files in the directory /proc/*PID*/fdinfo can be read to obtain information about the file descriptors of any process on the system. There is one file in this directory for each of the process's open file descriptors, with a name that matches the number of the descriptor. The *pos* field in this file shows the current file offset (Section 4.7). The *flags* field is an octal number that shows the file access mode flags and open file status flags. (To decode this number, we need to look at the numeric values of these flags in the C library header files.)

Details for the *flags* constants are as follows:

O_APPEND
: Writes are always appended to the end of the file. We discuss the significance of this flag in Section 5.1.

O_ASYNC
: Generate a signal when I/O becomes possible on the file descriptor returned by *open()*. This feature, termed *signal-driven I/O*, is available only for certain file types, such as terminals, FIFOs, and sockets. (The O_ASYNC flag is not specified in SUSv3; however, it, or the older synonym, FASYNC, is found on most UNIX implementations.) On Linux, specifying the O_ASYNC flag when calling *open()* has no effect. To enable signal-driven I/O, we must instead set this flag using the *fcntl()* F_SETFL operation (Section 5.3). (Several other UNIX implementations behave similarly.) Refer to Section 63.3 for more information about the O_ASYNC flag.

O_CLOEXEC (since Linux 2.6.23)
: Enable the close-on-exec flag (FD_CLOEXEC) for the new file descriptor. We describe the FD_CLOEXEC flag in Section 27.4. Using the O_CLOEXEC flag allows a program to avoid additional *fcntl()* F_GETFD and F_SETFD operations to set the close-on-exec flag. It is also necessary in multithreaded programs to avoid the race conditions that could occur using the latter technique. These races can occur when one thread opens a file descriptor and then tries to mark it close-on-exec at the same time as another thread does a *fork()* and then an *exec()* of an arbitrary program. (Suppose that the second thread manages to both *fork()* and *exec()* between the time the first thread opens the file descriptor and uses *fcntl()* to set the close-on-exec flag.) Such races

could result in open file descriptors being unintentionally passed to unsafe programs. (We say more about race conditions in Section 5.1.)

`O_CREAT`

If the file doesn't already exist, it is created as a new, empty file. This flag is effective even if the file is being opened only for reading. If we specify `O_CREAT`, then we must supply a *mode* argument in the *open()* call; otherwise, the permissions of the new file will be set to some random value from the stack.

`O_DIRECT`

Allow file I/O to bypass the buffer cache. This feature is described in Section 13.6. The `_GNU_SOURCE` feature test macro must be defined in order to make this constant definition available from `<fcntl.h>`.

`O_DIRECTORY`

Return an error (*errno* equals `ENOTDIR`) if *pathname* is not a directory. This flag is an extension designed specifically for implementing *opendir()* (Section 18.8). The `_GNU_SOURCE` feature test macro must be defined in order to make this constant definition available from `<fcntl.h>`.

`O_DSYNC` (since Linux 2.6.33)

Perform file writes according to the requirements of synchronized I/O data integrity completion. See the discussion of kernel I/O buffering in Section 13.3.

`O_EXCL`

This flag is used in conjunction with `O_CREAT` to indicate that if the file already exists, it should not be opened; instead, *open()* should fail, with *errno* set to `EEXIST`. In other words, this flag allows the caller to ensure that it is the process creating the file. The check for existence and the creation of the file are performed *atomically*. We discuss the concept of atomicity in Section 5.1. When both `O_CREAT` and `O_EXCL` are specified in *flags*, *open()* fails (with the error `EEXIST`) if *pathname* is a symbolic link. SUSv3 requires this behavior so that a privileged application can create a file in a known location without there being a possibility that a symbolic link would cause the file to be created in a different location (e.g., a system directory), which would have security implications.

`O_LARGEFILE`

Open the file with large file support. This flag is used on 32-bit systems in order to work with large files. Although it is not specified in SUSv3, the `O_LARGEFILE` flag is found on several other UNIX implementations. On 64-bit Linux implementations such as Alpha and IA-64, this flag has no effect. See Section 5.10 for more information.

`O_NOATIME` (since Linux 2.6.8)

Don't update the file last access time (the *st_atime* field described in Section 15.1) when reading from this file. To use this flag, the effective user ID of the calling process must match the owner of the file, or the process must be privileged (`CAP_FOWNER`); otherwise, *open()* fails with the error `EPERM`.

(In reality, for an unprivileged process, it is the process's file-system user ID, rather than its effective user ID, that must match the user ID of the file when opening a file with the O_NOATIME flag, as described in Section 9.5.) This flag is a nonstandard Linux extension. To expose its definition from <fcntl.h>, we must define the _GNU_SOURCE feature test macro. The O_NOATIME flag is intended for use by indexing and backup programs. Its use can significantly reduce the amount of disk activity, because repeated disk seeks back and forth across the disk are not required to read the contents of a file and to update the last access time in the file's i-node (Section 14.4). Functionality similar to O_NOATIME is available using the MS_NOATIME *mount()* flag (Section 14.8.1) and the FS_NOATIME_FL flag (Section 15.5).

O_NOCTTY
: If the file being opened is a terminal device, prevent it from becoming the controlling terminal. Controlling terminals are discussed in Section 34.4. If the file being opened is not a terminal, this flag has no effect.

O_NOFOLLOW
: Normally, *open()* dereferences *pathname* if it is a symbolic link. However, if the O_NOFOLLOW flag is specified, then *open()* fails (with *errno* set to ELOOP) if *pathname* is a symbolic link. This flag is useful, especially in privileged programs, for ensuring that *open()* doesn't dereference a symbolic link. To expose the definition of this flag from <fcntl.h>, we must define the _GNU_SOURCE feature test macro.

O_NONBLOCK
: Open the file in nonblocking mode. See Section 5.9.

O_SYNC
: Open the file for synchronous I/O. See the discussion of kernel I/O buffering in Section 13.3.

O_TRUNC
: If the file already exists and is a regular file, then truncate it to zero length, destroying any existing data. On Linux, truncation occurs whether the file is being opened for reading or writing (in both cases, we must have write permission on the file). SUSv3 leaves the combination of O_RDONLY and O_TRUNC unspecified, but most other UNIX implementations behave in the same way as Linux.

4.3.2 Errors from *open()*

If an error occurs while trying to open the file, *open()* returns −1, and *errno* identifies the cause of the error. The following are some possible errors that can occur (in addition to those already noted when describing the *flags* argument above):

EACCES
: The file permissions don't allow the calling process to open the file in the mode specified by *flags*. Alternatively, because of directory permissions, the file could not be accessed, or the file did not exist and could not be created.

EISDIR
: The specified file is a directory, and the caller attempted to open it for writing. This isn't allowed. (On the other hand, there are occasions when it can be useful to open a directory for reading. We consider an example in Section 18.11.)

EMFILE
: The process resource limit on the number of open file descriptors has been reached (RLIMIT_NOFILE, described in Section 36.3).

ENFILE
: The system-wide limit on the number of open files has been reached.

ENOENT
: The specified file doesn't exist, and O_CREAT was not specified, or O_CREAT was specified, and one of the directories in *pathname* doesn't exist or is a symbolic link pointing to a nonexistent pathname (a dangling link).

EROFS
: The specified file is on a read-only file system and the caller tried to open it for writing.

ETXTBSY
: The specified file is an executable file (a program) that is currently executing. It is not permitted to modify (i.e., open for writing) the executable file associated with a running program. (We must first terminate the program in order to be able to modify the executable file.)

When we later describe other system calls or library functions, we generally won't list the range of possible errors that may occur in the above fashion. (Such a list can be found in the corresponding manual page for each system call or library function.) We do so here for two reasons. One of these is that *open()* is the first system call that we describe in detail, and the above list illustrates that a system call or library function may fail for any of a number of reasons. Second, the specific reasons why *open()* may fail make an interesting list in themselves, illustrating a number of factors and checks that come into play when a file is accessed. (The above list is incomplete: see the *open(2)* manual page for more reasons why *open()* may fail.)

4.3.3 The *creat()* System Call

In early UNIX implementations, *open()* had only two arguments and could not be used to create a new file. Instead, the *creat()* system call was used to create and open a new file.

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
                                        Returns file descriptor, or –1 on error
```

The *creat()* system call creates and opens a new file with the given *pathname*, or if the file already exists, opens the file and truncates it to zero length. As its function

result, *creat()* returns a file descriptor that can be used in subsequent system calls. Calling *creat()* is equivalent to the following *open()* call:

```
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Because the *open() flags* argument provides greater control over how the file is opened (e.g., we can specify `O_RDWR` instead of `O_WRONLY`), *creat()* is now obsolete, although it may still be seen in older programs.

4.4 Reading from a File: *read()*

The *read()* system call reads data from the open file referred to by the descriptor *fd*.

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
                        Returns number of bytes read, 0 on EOF, or −1 on error
```

The *count* argument specifies the maximum number of bytes to read. (The *size_t* data type is an unsigned integer type.) The *buffer* argument supplies the address of the memory buffer into which the input data is to be placed. This buffer must be at least *count* bytes long.

> System calls don't allocate memory for buffers that are used to return information to the caller. Instead, we must pass a pointer to a previously allocated memory buffer of the correct size. This contrasts with several library functions that *do* allocate memory buffers in order to return information to the caller.

A successful call to *read()* returns the number of bytes actually read, or 0 if end-of-file is encountered. On error, the usual −1 is returned. The *ssize_t* data type is a signed integer type used to hold a byte count or a −1 error indication.

A call to *read()* may read less than the requested number of bytes. For a regular file, the probable reason for this is that we were close to the end of the file.

When *read()* is applied to other types of files—such as pipes, FIFOs, sockets, or terminals—there are also various circumstances where it may read fewer bytes than requested. For example, by default, a *read()* from a terminal reads characters only up to the next newline (`\n`) character. We consider these cases when we cover other file types in subsequent chapters.

Using *read()* to input a series of characters from, say, a terminal, we might expect the following code to work:

```
#define MAX_READ 20
char buffer[MAX_READ];

if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
    errExit("read");
printf("The input data was: %s\n", buffer);
```

The output from this piece of code is likely to be strange, since it will probably include characters in addition to the string actually entered. This is because *read()*

doesn't place a terminating null byte at the end of the string that *printf()* is being asked to print. A moment's reflection leads us to realize that this must be so, since *read()* can be used to read any sequence of bytes from a file. In some cases, this input might be text, but in other cases, the input might be binary integers or C structures in binary form. There is no way for *read()* to tell the difference, and so it can't attend to the C convention of null terminating character strings. If a terminating null byte is required at the end of the input buffer, we must put it there explicitly:

```
char buffer[MAX_READ + 1];
ssize_t numRead;

numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");

buffer[numRead] = '\0';
printf("The input data was: %s\n", buffer);
```

Because the terminating null byte requires a byte of memory, the size of *buffer* must be at least one greater than the largest string we expect to read.

4.5 Writing to a File: *write()*

The *write()* system call writes data to an open file.

```
#include <unistd.h>

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

Returns number of bytes written, or −1 on error

The arguments to *write()* are similar to those for *read()*: *buffer* is the address of the data to be written; *count* is the number of bytes to write from *buffer*; and *fd* is a file descriptor referring to the file to which data is to be written.

On success, *write()* returns the number of bytes actually written; this may be less than *count*. For a disk file, possible reasons for such a *partial write* are that the disk was filled or that the process resource limit on file sizes was reached. (The relevant limit is `RLIMIT_FSIZE`, described in Section 36.3.)

When performing I/O on a disk file, a successful return from *write()* doesn't guarantee that the data has been transferred to disk, because the kernel performs buffering of disk I/O in order to reduce disk activity and expedite *write()* calls. We consider the details in Chapter 13.

4.6 Closing a File: *close()*

The *close()* system call closes an open file descriptor, freeing it for subsequent reuse by the process. When a process terminates, all of its open file descriptors are automatically closed.

```
#include <unistd.h>

int close(int fd);
```

Returns 0 on success, or –1 on error

It is usually good practice to close unneeded file descriptors explicitly, since this makes our code more readable and reliable in the face of subsequent modifications. Furthermore, file descriptors are a consumable resource, so failure to close a file descriptor could result in a process running out of descriptors. This is a particularly important issue when writing long-lived programs that deal with multiple files, such as shells or network servers.

Just like every other system call, a call to *close()* should be bracketed with error-checking code, such as the following:

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

This catches errors such as attempting to close an unopened file descriptor or close the same file descriptor twice, and catches error conditions that a specific file system may diagnose during a close operation.

> NFS (Network File System) provides an example of an error that is specific to a file system. If an NFS commit failure occurs, meaning that the data did not reach the remote disk, then this error is propagated to the application as a failure in the *close()* call.

4.7 Changing the File Offset: *lseek()*

For each open file, the kernel records a *file offset*, sometimes also called the *read-write offset* or *pointer*. This is the location in the file at which the next *read()* or *write()* will commence. The file offset is expressed as an ordinal byte position relative to the start of the file. The first byte of the file is at offset 0.

The file offset is set to point to the start of the file when the file is opened and is automatically adjusted by each subsequent call to *read()* or *write()* so that it points to the next byte of the file after the byte(s) just read or written. Thus, successive *read()* and *write()* calls progress sequentially through a file.

The *lseek()* system call adjusts the file offset of the open file referred to by the file descriptor *fd*, according to the values specified in *offset* and *whence*.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns new file offset if successful, or –1 on error

The *offset* argument specifies a value in bytes. (The *off_t* data type is a signed integer type specified by SUSv3.) The *whence* argument indicates the base point from which *offset* is to be interpreted, and is one of the following values:

`SEEK_SET`
: The file offset is set *offset* bytes from the beginning of the file.

`SEEK_CUR`
: The file offset is adjusted by *offset* bytes relative to the current file offset.

`SEEK_END`
: The file offset is set to the size of the file plus *offset*. In other words, *offset* is interpreted with respect to the next byte after the last byte of the file.

Figure 4-1 shows how the *whence* argument is interpreted.

> In earlier UNIX implementations, the integers 0, 1, and 2 were used, rather than the `SEEK_*` constants shown in the main text. Older versions of BSD used different names for these values: `L_SET`, `L_INCR`, and `L_XTND`.

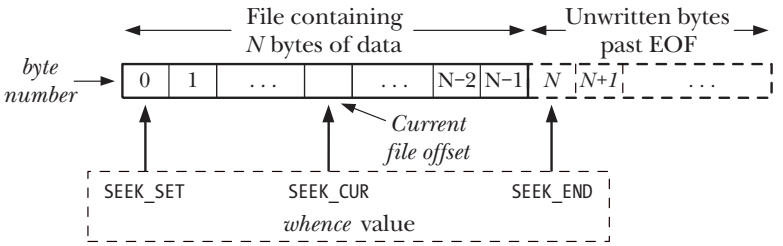

Figure 4-1: Interpreting the *whence* argument of *lseek()*

If *whence* is `SEEK_CUR` or `SEEK_END`, *offset* may be negative or positive; for `SEEK_SET`, *offset* must be nonnegative.

The return value from a successful *lseek()* is the new file offset. The following call retrieves the current location of the file offset without changing it:

```
curr = lseek(fd, 0, SEEK_CUR);
```

> Some UNIX implementations (but not Linux) have the nonstandard *tell(fd)* function, which serves the same purpose as the above *lseek()* call.

Here are some other examples of *lseek()* calls, along with comments indicating where the file offset is moved to:

```
lseek(fd, 0, SEEK_SET);         /* Start of file */
lseek(fd, 0, SEEK_END);         /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END);        /* Last byte of file */
lseek(fd, -10, SEEK_CUR);       /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END);     /* 10001 bytes past last byte of file */
```

Calling *lseek()* simply adjusts the kernel's record of the file offset associated with a file descriptor. It does not cause any physical device access.

We describe some further details of the relationship between file offsets, file descriptors, and open files in Section 5.4.

We can't apply *lseek()* to all types of files. Applying *lseek()* to a pipe, FIFO, socket, or terminal is not permitted; *lseek()* fails, with *errno* set to `ESPIPE`. On the other hand, it is possible to apply *lseek()* to devices where it is sensible to do so. For example, it is possible to seek to a specified location on a disk or tape device.

> The *l* in the name *lseek()* derives from the fact that the *offset* argument and the return value were both originally typed as *long*. Early UNIX implementations provided a *seek()* system call, which typed these values as *int*.

File holes

What happens if a program seeks past the end of a file, and then performs I/O? A call to *read()* will return 0, indicating end-of-file. Somewhat surprisingly, it is possible to write bytes at an arbitrary point past the end of the file.

The space in between the previous end of the file and the newly written bytes is referred to as a *file hole*. From a programming point of view, the bytes in a hole exist, and reading from the hole returns a buffer of bytes containing 0 (null bytes).

File holes don't, however, take up any disk space. The file system doesn't allocate any disk blocks for a hole until, at some later point, data is written into it. The main advantage of file holes is that a sparsely populated file consumes less disk space than would otherwise be required if the null bytes actually needed to be allocated in disk blocks. Core dump files (Section 22.1) are common examples of files that contain large holes.

> The statement that file holes don't consume disk space needs to be qualified slightly. On most file systems, file space is allocated in units of blocks (Section 14.3). The size of a block depends on the file system, but is typically something like 1024, 2048, or 4096 bytes. If the edge of a hole falls within a block, rather than on a block boundary, then a complete block is allocated to store the data in the other part of the block, and the part corresponding to the hole is filled with null bytes.

Most native UNIX file systems support the concept of file holes, but many nonnative file systems (e.g., Microsoft's VFAT) do not. On a file system that doesn't support holes, explicit null bytes are written to the file.

The existence of holes means that a file's nominal size may be larger than the amount of disk storage it utilizes (in some cases, considerably larger). Writing bytes into the middle of the file hole will decrease the amount of free disk space as the kernel allocates blocks to fill the hole, even though the file's size doesn't change. Such a scenario is uncommon, but nevertheless one to be aware of.

> SUSv3 specifies a function, *posix_fallocate(fd, offset, len)*, that ensures that space is allocated on disk for the byte range specified by *offset* and *len* for the disk file referred to by the descriptor *fd*. This allows an application to be sure that a later *write()* to the file won't fail because disk space is exhausted (which could otherwise occur if a hole in the file was filled in, or some other application consumed space on the disk before the full extent of the file was written). Historically, the *glibc* implementation of this function achieved the desired result by writing a 0 byte into each block in the specified range. Since version 2.6.23, Linux provides an *fallocate()* system call, which provides a more efficient way of ensuring that the necessary space is allocated, and the *glibc posix_fallocate()* implementation makes use of this system call when it is available.

Section 14.4 describes how holes are represented in a file, and Section 15.1 describes the *stat()* system call, which can tell us the current size of a file, as well as the number of blocks actually allocated to the file.

Example program

Listing 4-3 demonstrates the use of *lseek()* in conjunction with *read()* and *write()*. The first command-line argument to this program is the name of a file to be opened. The remaining arguments specify I/O operations to be performed on the file. Each of these operations consists of a letter followed by an associated value (with no separating space):

- s*offset*: Seek to byte *offset* from the start of the file.
- r*length*: Read *length* bytes from the file, starting at the current file offset, and display them in text form.
- R*length*: Read *length* bytes from the file, starting at the current file offset, and display them in hexadecimal.
- w*str*: Write the string of characters specified in *str* at the current file offset.

Listing 4-3: Demonstration of *read()*, *write()*, and *lseek()*

fileio/seek_io.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <ctype.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    size_t len;
    off_t offset;
    int fd, ap, j;
    unsigned char *buf;
    ssize_t numRead, numWritten;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file {r<length>|R<length>|w<string>|s<offset>}...\n",
                 argv[0]);

    fd = open(argv[1], O_RDWR | O_CREAT,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
              S_IROTH | S_IWOTH);                    /* rw-rw-rw- */
    if (fd == -1)
        errExit("open");

    for (ap = 2; ap < argc; ap++) {
        switch (argv[ap][0]) {
        case 'r':   /* Display bytes at current offset, as text */
        case 'R':   /* Display bytes at current offset, in hex */
            len = getLong(&argv[ap][1], GN_ANY_BASE, argv[ap]);
```

```
            buf = malloc(len);
            if (buf == NULL)
                errExit("malloc");

            numRead = read(fd, buf, len);
            if (numRead == -1)
                errExit("read");

            if (numRead == 0) {
                printf("%s: end-of-file\n", argv[ap]);
            } else {
                printf("%s: ", argv[ap]);
                for (j = 0; j < numRead; j++) {
                    if (argv[ap][0] == 'r')
                        printf("%c", isprint(buf[j]) ?  buf[j] : '?');
                    else
                        printf("%02x ", buf[j]);
                }
                printf("\n");
            }

            free(buf);
            break;

        case 'w':   /* Write string at current offset */
            numWritten = write(fd, &argv[ap][1], strlen(&argv[ap][1]));
            if (numWritten == -1)
                errExit("write");
            printf("%s: wrote %ld bytes\n", argv[ap], (long) numWritten);
            break;

        case 's':   /* Change file offset */
            offset = getLong(&argv[ap][1], GN_ANY_BASE, argv[ap]);
            if (lseek(fd, offset, SEEK_SET) == -1)
                errExit("lseek");
            printf("%s: seek succeeded\n", argv[ap]);
            break;

        default:
            cmdLineErr("Argument must start with [rRws]: %s\n", argv[ap]);
        }
    }

    exit(EXIT_SUCCESS);
}
```

fileio/seek_io.c

The following shell session log demonstrates the use of the program in Listing 4-3, showing what happens when we attempt to read bytes from a file hole:

```
$ touch tfile                       Create new, empty file
$ ./seek_io tfile s100000 wabc      Seek to offset 100,000, write "abc"
s100000: seek succeeded
wabc: wrote 3 bytes
```

```
$ ls -l tfile                          Check size of file
-rw-r--r--    1 mtk    users   100003 Feb 10 10:35 tfile
$ ./seek_io tfile s10000 R5            Seek to offset 10,000, read 5 bytes from hole
s10000: seek succeeded
R5: 00 00 00 00 00                     Bytes in the hole contain 0
```

4.8 Operations Outside the Universal I/O Model: *ioctl()*

The *ioctl()* system call is a general-purpose mechanism for performing file and device operations that fall outside the universal I/O model described earlier in this chapter.

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

int ioctl(int fd, int request, ... /* argp */);
                    Value returned on success depends on request, or –1 on error
```

The *fd* argument is an open file descriptor for the device or file upon which the control operation specified by *request* is to be performed. Device-specific header files define constants that can be passed in the *request* argument.

As indicated by the standard C ellipsis (...) notation, the third argument to *ioctl()*, which we label *argp*, can be of any type. The value of the *request* argument enables *ioctl()* to determine what type of value to expect in *argp*. Typically, *argp* is a pointer to either an integer or a structure; in some cases, it is unused.

We'll see a number of uses for *ioctl()* in later chapters (see, for example, Section 15.5).

> The only specification that SUSv3 makes for *ioctl()* is for operations to control STREAMS devices. (The STREAMS facility is a System V feature that is not supported by the mainline Linux kernel, although a few add-on implementations have been developed.) None of the other *ioctl()* operations described in this book is specified in SUSv3. However, the *ioctl()* call has been part of the UNIX system since early versions, and consequently several of the *ioctl()* operations that we describe are provided on many other UNIX implementations. As we describe each *ioctl()* operation, we note portability issues.

4.9 Summary

In order to perform I/O on a regular file, we must first obtain a file descriptor using *open()*. I/O is then performed using *read()* and *write()*. After performing all I/O, we should free the file descriptor and its associated resources using *close()*. These system calls can be used to perform I/O on all types of files.

The fact that all file types and device drivers implement the same I/O interface allows for universality of I/O, meaning that a program can typically be used with any type of file without requiring code that is specific to the file type.

For each open file, the kernel maintains a file offset, which determines the location at which the next read or write will occur. The file offset is implicitly updated by reads and writes. Using *lseek()*, we can explicitly reposition the file offset to any location within the file or past the end of the file. Writing data at a position beyond the previous end of the file creates a hole in the file. Reads from a file hole return bytes containing zeros.

The *ioctl()* system call is a catchall for device and file operations that don't fit into the standard file I/O model.

4.10 Exercises

4-1. The *tee* command reads its standard input until end-of-file, writing a copy of the input to standard output and to the file named in its command-line argument. (We show an example of the use of this command when we discuss FIFOs in Section 44.7.) Implement *tee* using I/O system calls. By default, *tee* overwrites any existing file with the given name. Implement the *−a* command-line option (*tee −a file*), which causes *tee* to append text to the end of a file if it already exists. (Refer to Appendix B for a description of the *getopt()* function, which can be used to parse command-line options.)

4-2. Write a program like *cp* that, when used to copy a regular file that contains holes (sequences of null bytes), also creates corresponding holes in the target file.