3

File I/O

530128

3.1 Introduction

We'll start our discussion of the UNIX System by describing the functions available for file I/O—open a file, read a file, write a file, and so on. Most file I/O on a UNIX system can be performed using only five functions: `open`, `read`, `write`, `lseek`, and `close`. We then examine the effect of various buffer sizes on the `read` and `write` functions.

The functions described in this chapter are often referred to as *unbuffered I/O*, in contrast to the standard I/O routines, which we describe in Chapter 5. The term *unbuffered* means that each `read` or `write` invokes a system call in the kernel. These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1 and the Single UNIX Specification.

Whenever we describe the sharing of resources among multiple processes, the concept of an atomic operation becomes important. We examine this concept with regard to file I/O and the arguments to the `open` function. This leads to a discussion of how files are shared among multiple processes and which kernel data structures are involved. After describing these features, we describe the `dup`, `fcntl`, `sync`, `fsync`, and `ioctl` functions.

3.2 File Descriptors

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by `open` or `creat` as an argument to either `read` or `write`.

By convention, UNIX System shells associate file descriptor 0 with the standard input of a process, file descriptor 1 with the standard output, and file descriptor 2 with the standard error. This convention is used by the shells and many applications; it is not a feature of the UNIX kernel. Nevertheless, many applications would break if these associations weren't followed.

Although their values are standardized by POSIX.1, the magic numbers 0, 1, and 2 should be replaced in POSIX-compliant applications with the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` to improve readability. These constants are defined in the `<unistd.h>` header.

File descriptors range from 0 through `OPEN_MAX-1`. (Recall Figure 2.11.) Early historical implementations of the UNIX System had an upper limit of 19, allowing a maximum of 20 open files per process, but many systems subsequently increased this limit to 63.

> With FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, the limit is essentially infinite, bounded by the amount of memory on the system, the size of an integer, and any hard and soft limits configured by the system administrator.

3.3 `open` and `openat` Functions

A file is opened or created by calling either the `open` function or the `openat` function.

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, –1 on error

We show the last argument as `...`, which is the ISO C way to specify that the number and types of the remaining arguments may vary. For these functions, the last argument is used only when a new file is being created, as we describe later. We show this argument as a comment in the prototype.

The *path* parameter is the name of the file to open or create. This function has a multitude of options, which are specified by the *oflag* argument. This argument is formed by ORing together one or more of the following constants from the `<fcntl.h>` header:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing.

> Most implementations define `O_RDONLY` as 0, `O_WRONLY` as 1, and `O_RDWR` as 2, for compatibility with older programs.

`O_EXEC` Open for execute only.

`O_SEARCH` Open for search only (applies to directories).

> The purpose of the O_SEARCH constant is to evaluate search permissions at the time a directory is opened. Further operations using the directory's file descriptor will not reevaluate permission to search the directory. None of the versions of the operating systems covered in this book support O_SEARCH yet.

One and only one of the previous five constants must be specified. The following constants are optional:

O_APPEND	Append to the end of file on each write. We describe this option in detail in Section 3.11.
O_CLOEXEC	Set the FD_CLOEXEC file descriptor flag. We discuss file descriptor flags in Section 3.14.
O_CREAT	Create the file if it doesn't exist. This option requires a third argument to the open function (a fourth argument to the openat function)—the *mode*, which specifies the access permission bits of the new file. (When we describe a file's access permission bits in Section 4.5, we'll see how to specify the *mode* and how it can be modified by the umask value of a process.)
O_DIRECTORY	Generate an error if *path* doesn't refer to a directory.
O_EXCL	Generate an error if O_CREAT is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation. We describe atomic operations in more detail in Section 3.11.
O_NOCTTY	If *path* refers to a terminal device, do not allocate the device as the controlling terminal for this process. We talk about controlling terminals in Section 9.6.
O_NOFOLLOW	Generate an error if *path* refers to a symbolic link. We discuss symbolic links in Section 4.17.
O_NONBLOCK	If *path* refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O. We describe this mode in Section 14.2.

> In earlier releases of System V, the O_NDELAY (no delay) flag was introduced. This option is similar to the O_NONBLOCK (nonblocking) option, but an ambiguity was introduced in the return value from a read operation. The no-delay option causes a read operation to return 0 if there is no data to be read from a pipe, FIFO, or device, but this conflicts with a return value of 0, indicating an end of file. SVR4-based systems still support the no-delay option, with the old semantics, but new applications should use the nonblocking option instead.

O_SYNC	Have each write wait for physical I/O to complete, including I/O necessary to update file attributes modified as a result of the write. We use this option in Section 3.14.
O_TRUNC	If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0.

`O_TTY_INIT` When opening a terminal device that is not already open, set the nonstandard `termios` parameters to values that result in behavior that conforms to the Single UNIX Specification. We discuss the `termios` structure when we discuss terminal I/O in Chapter 18.

The following two flags are also optional. They are part of the synchronized input and output option of the Single UNIX Specification (and thus POSIX.1).

`O_DSYNC` Have each `write` wait for physical I/O to complete, but don't wait for file attributes to be updated if they don't affect the ability to read the data just written.

> The `O_DSYNC` and `O_SYNC` flags are similar, but subtly different. The `O_DSYNC` flag affects a file's attributes only when they need to be updated to reflect a change in the file's data (for example, update the file's size to reflect more data). With the `O_SYNC` flag, data and attributes are always updated synchronously. When overwriting an existing part of a file opened with the `O_DSYNC` flag, the file times wouldn't be updated synchronously. In contrast, if we had opened the file with the `O_SYNC` flag, every `write` to the file would update the file's times before the `write` returns, regardless of whether we were writing over existing bytes or appending to the file.

`O_RSYNC` Have each `read` operation on the file descriptor wait until any pending writes for the same portion of the file are complete.

> Solaris 10 supports all three synchronization flags. Historically, FreeBSD (and thus Mac OS X) have used the `O_FSYNC` flag, which has the same behavior as `O_SYNC`. Because the two flags are equivalent, they define the flags to have the same value. FreeBSD 8.0 doesn't support the `O_DSYNC` or `O_RSYNC` flags. Mac OS X doesn't support the `O_RSYNC` flag, but defines the `O_DSYNC` flag, treating it the same as the `O_SYNC` flag. Linux 3.2.0 supports the `O_DSYNC` flag, but treats the `O_RSYNC` flag the same as `O_SYNC`.

The file descriptor returned by `open` and `openat` is guaranteed to be the lowest-numbered unused descriptor. This fact is used by some applications to open a new file on standard input, standard output, or standard error. For example, an application might close standard output—normally, file descriptor 1—and then open another file, knowing that it will be opened on file descriptor 1. We'll see a better way to guarantee that a file is open on a given descriptor in Section 3.12, when we explore the `dup2` function.

The *fd* parameter distinguishes the `openat` function from the `open` function. There are three possibilities:

1. The *path* parameter specifies an absolute pathname. In this case, the *fd* parameter is ignored and the `openat` function behaves like the `open` function.
2. The *path* parameter specifies a relative pathname and the *fd* parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated. The *fd* parameter is obtained by opening the directory where the relative pathname is to be evaluated.

3. The *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD. In this case, the pathname is evaluated starting in the current working directory and the openat function behaves like the open function.

The openat function is one of a class of functions added to the latest version of POSIX.1 to address two problems. First, it gives threads a way to use relative pathnames to open files in directories other than the current working directory. As we'll see in Chapter 11, all threads in the same process share the same current working directory, so this makes it difficult for multiple threads in the same process to work in different directories at the same time. Second, it provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors.

The basic idea behind TOCTTOU errors is that a program is vulnerable if it makes two file-based function calls where the second call depends on the results of the first call. Because the two calls are not atomic, the file can change between the two calls, thereby invalidating the results of the first call, leading to a program error. TOCTTOU errors in the file system namespace generally deal with attempts to subvert file system permissions by tricking a privileged program into either reducing permissions on a privileged file or modifying a privileged file to open up a security hole. Wei and Pu [2005] discuss TOCTTOU weaknesses in the UNIX file system interface.

Filename and Pathname Truncation

What happens if NAME_MAX is 14 and we try to create a new file in the current directory with a filename containing 15 characters? Traditionally, early releases of System V, such as SVR2, allowed this to happen, silently truncating the filename beyond the 14th character. BSD-derived systems, in contrast, returned an error status, with errno set to ENAMETOOLONG. Silently truncating the filename presents a problem that affects more than simply the creation of new files. If NAME_MAX is 14 and a file exists whose name is exactly 14 characters, any function that accepts a pathname argument, such as open or stat, has no way to determine what the original name of the file was, as the original name might have been truncated.

With POSIX.1, the constant _POSIX_NO_TRUNC determines whether long filenames and long components of pathnames are truncated or an error is returned. As we saw in Chapter 2, this value can vary based on the type of the file system, and we can use fpathconf or pathconf to query a directory to see which behavior is supported.

> Whether an error is returned is largely historical. For example, SVR4-based systems do not generate an error for the traditional System V file system, S5. For the BSD-style file system (known as UFS), however, SVR4-based systems do generate an error. Figure 2.20 illustrates another example: Solaris will return an error for UFS, but not for PCFS, the DOS-compatible file system, as DOS silently truncates filenames that don't fit in an 8.3 format. BSD-derived systems and Linux always return an error.

If _POSIX_NO_TRUNC is in effect, errno is set to ENAMETOOLONG, and an error status is returned if any filename component of the pathname exceeds NAME_MAX.

> Most modern file systems support a maximum of 255 characters for filenames. Because filenames are usually shorter than this limit, this constraint tends to not present problems for most applications.

3.4 creat Function

A new file can also be created by calling the creat function.

```
#include <fcntl.h>
int creat(const char *path, mode_t mode);
                    Returns: file descriptor opened for write-only if OK, –1 on error
```

Note that this function is equivalent to

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

> Historically, in early versions of the UNIX System, the second argument to open could be only 0, 1, or 2. There was no way to open a file that didn't already exist. Therefore, a separate system call, creat, was needed to create new files. With the O_CREAT and O_TRUNC options now provided by open, a separate creat function is no longer needed.

We'll show how to specify *mode* in Section 4.5 when we describe a file's access permissions in detail.

One deficiency with creat is that the file is opened only for writing. Before the new version of open was provided, if we were creating a temporary file that we wanted to write and then read back, we had to call creat, close, and then open. A better way is to use the open function, as in

```
open(path, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5 close Function

An open file is closed by calling the close function.

```
#include <unistd.h>
int close(int fd);
                                    Returns: 0 if OK, –1 on error
```

Closing a file also releases any record locks that the process may have on the file. We'll discuss this point further in Section 14.3.

When a process terminates, all of its open files are closed automatically by the kernel. Many programs take advantage of this fact and don't explicitly close open files. See the program in Figure 1.4, for example.

3.6 lseek Function

Every open file has an associated "current file offset," normally a non-negative integer that measures the number of bytes from the beginning of the file. (We describe some exceptions to the "non-negative" qualifier later in this section.) Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written. By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified.

An open file's offset can be set explicitly by calling lseek.

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
                                    Returns: new file offset if OK, –1 on error
```

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is SEEK_SET, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is SEEK_CUR, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.
- If *whence* is SEEK_END, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Because a successful call to lseek returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

```
off_t    currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

This technique can also be used to determine if a file is capable of seeking. If the file descriptor refers to a pipe, FIFO, or socket, lseek sets errno to ESPIPE and returns –1.

> The three symbolic constants—SEEK_SET, SEEK_CUR, and SEEK_END—were introduced with System V. Prior to this, *whence* was specified as 0 (absolute), 1 (relative to the current offset), or 2 (relative to the end of file). Much software still exists with these numbers hard coded.
>
> The character l in the name lseek means "long integer." Before the introduction of the off_t data type, the *offset* argument and the return value were long integers. lseek was introduced with Version 7 when long integers were added to C. (Similar functionality was provided in Version 6 by the functions seek and tell.)

Example

The program in Figure 3.1 tests its standard input to see whether it is capable of seeking.

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Figure 3.1 Test whether standard input is capable of seeking

If we invoke this program interactively, we get

```
$ ./a.out < /etc/passwd
seek OK
$ cat < /etc/passwd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```

□

Normally, a file's current offset must be a non-negative integer. It is possible, however, that certain devices could allow negative offsets. But for regular files, the offset must be non-negative. Because negative offsets are possible, we should be careful to compare the return value from `lseek` as being equal to or not equal to –1, rather than testing whether it is less than 0.

> The `/dev/kmem` device on FreeBSD for the Intel x86 processor supports negative offsets.
>
> Because the offset (`off_t`) is a signed data type (Figure 2.21), we lose a factor of 2 in the maximum file size. If `off_t` is a 32-bit integer, the maximum file size is 2^{31}–1 bytes.

`lseek` only records the current file offset within the kernel—it does not cause any I/O to take place. This offset is then used by the next read or write operation.

The file's offset can be greater than the file's current size, in which case the next `write` to the file will extend the file. This is referred to as creating a hole in a file and is allowed. Any bytes in a file that have not been written are read back as 0.

A hole in a file isn't required to have storage backing it on disk. Depending on the file system implementation, when you write after seeking past the end of a file, new disk blocks might be allocated to store the data, but there is no need to allocate disk blocks for the data between the old end of file and the location where you start writing.

Example

The program shown in Figure 3.2 creates a file with a hole in it.

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int     fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */
```

```
    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

Figure 3.2 Create a file with a hole in it

Running this program gives us

```
$ ./a.out
$ ls -l file.hole                    check its size
-rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
$ od -c file.hole                    let's look at the actual contents
0000000   a  b  c  d  e  f  g  h  i  j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000   A  B  C  D  E  F  G  H  I  J
0040012
```

We use the od(1) command to look at the contents of the file. The `-c` flag tells it to print the contents as characters. We can see that the unwritten bytes in the middle are read back as zero. The seven-digit number at the beginning of each line is the byte offset in octal.

To prove that there is really a hole in the file, let's compare the file we just created with a file of the same size, but without holes:

```
$ ls -ls file.hole file.nohole     compare sizes
  8 -rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
 20 -rw-r--r--  1 sar       16394 Nov 25 01:03 file.nohole
```

Although both files are the same size, the file without holes consumes 20 disk blocks, whereas the file with holes consumes only 8 blocks.

In this example, we call the `write` function (Section 3.8). We'll have more to say about files with holes in Section 4.12. □

Because the offset address that `lseek` uses is represented by an `off_t`, implementations are allowed to support whatever size is appropriate on their particular platform. Most platforms today provide two sets of interfaces to manipulate file offsets: one set that uses 32-bit file offsets and another set that uses 64-bit file offsets.

The Single UNIX Specification provides a way for applications to determine which environments are supported through the `sysconf` function (Section 2.5.4). Figure 3.3 summarizes the `sysconf` constants that are defined.

Name of option	Description	*name* argument
`_POSIX_V7_ILP32_OFF32`	`int`, `long`, pointer, and `off_t` types are 32 bits.	`_SC_V7_ILP32_OFF32`
`_POSIX_V7_ILP32_OFFBIG`	`int`, `long`, and pointer types are 32 bits; `off_t` types are at least 64 bits.	`_SC_V7_ILP32_OFFBIG`
`_POSIX_V7_LP64_OFF64`	`int` types are 32 bits; `long`, pointer, and `off_t` types are 64 bits.	`_SC_V7_LP64_OFF64`
`_POSIX_V7_LP64_OFFBIG`	`int` types are at least 32 bits; `long`, pointer, and `off_t` types are at least 64 bits.	`_SC_V7_LP64_OFFBIG`

Figure 3.3 Data size options and *name* arguments to `sysconf`

The `c99` compiler requires that we use the `getconf`(1) command to map the desired data size model to the flags necessary to compile and link our programs. Different flags and libraries might be needed, depending on the environments supported by each platform.

Unfortunately, this is one area in which implementations haven't caught up to the standards. If your system does not match the latest version of the standard, the system might support the option names from the previous version of the Single UNIX Specification: `_POSIX_V6_ILP32_OFF32`, `_POSIX_V6_ILP32_OFFBIG`, `_POSIX_V6_LP64_OFF64`, and `_POSIX_V6_LP64_OFFBIG`.

To get around this, applications can set the `_FILE_OFFSET_BITS` constant to 64 to enable 64-bit offsets. Doing so changes the definition of `off_t` to be a 64-bit signed integer. Setting `_FILE_OFFSET_BITS` to 32 enables 32-bit file offsets. Be aware, however, that although all four platforms discussed in this text support both 32-bit and 64-bit file offsets, setting `_FILE_OFFSET_BITS` is not guaranteed to be portable and might not have the desired effect.

Figure 3.4 summarizes the size in bytes of the `off_t` data type for the platforms covered in this book when an application doesn't define `_FILE_OFFSET_BITS`, as well as the size when an application defines `_FILE_OFFSET_BITS` to have a value of either 32 or 64.

Operating system	CPU architecture	`_FILE_OFFSET_BITS` value		
		Undefined	32	64
FreeBSD 8.0	x86 32-bit	8	8	8
Linux 3.2.0	x86 64-bit	8	8	8
Mac OS X 10.6.8	x86 64-bit	8	8	8
Solaris 10	SPARC 64-bit	8	4	8

Figure 3.4 Size in bytes of `off_t` for different platforms

Note that even though you might enable 64-bit file offsets, your ability to create a file larger than 2 GB (2^{31}–1 bytes) depends on the underlying file system type.

3.7 read Function

Data is read from an open file with the read function.

```
#include <unistd.h>

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

Returns: number of bytes read, 0 if end of file, –1 on error

If the read is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned.

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time. (We'll see how to change this default in Chapter 18.)
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
- When reading from a record-oriented device. Some record-oriented devices, such as magnetic tape, can return up to a single record at a time.
- When interrupted by a signal and a partial amount of data has already been read. We discuss this further in Section 10.5.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

POSIX.1 changed the prototype for this function in several ways. The classic definition is

```
int read(int fd, char *buf, unsigned nbytes);
```

- First, the second argument was changed from char * to void * to be consistent with ISO C: the type void * is used for generic pointers.
- Next, the return value was required to be a signed integer (ssize_t) to return a positive byte count, 0 (for end of file), or –1 (for an error).
- Finally, the third argument historically has been an unsigned integer, to allow a 16-bit implementation to read or write up to 65,534 bytes at a time. With the 1990 POSIX.1 standard, the primitive system data type ssize_t was introduced to provide the signed return value, and the unsigned size_t was used for the third argument. (Recall the SSIZE_MAX constant from Section 2.5.2.)

3.8 `write` Function

Data is written to an open file with the `write` function.

```
#include <unistd.h>

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

Returns: number of bytes written if OK, –1 on error

The return value is usually equal to the *nbytes* argument; otherwise, an error has occurred. A common cause for a `write` error is either filling up a disk or exceeding the file size limit for a given process (Section 7.11 and Exercise 10.11).

For a regular file, the write operation starts at the file's current offset. If the `O_APPEND` option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

3.9 I/O Efficiency

The program in Figure 3.5 copies a file, using only the `read` and `write` functions.

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Figure 3.5 Copy standard input to standard output

The following caveats apply to this program.

- It reads from standard input and writes to standard output, assuming that these have been set up by the shell before this program is executed. Indeed, all normal UNIX system shells provide a way to open a file for reading on standard input and to create (or rewrite) a file on standard output. This prevents the program from having to open the input and output files, and allows the user to take advantage of the shell's I/O redirection facilities.

- The program doesn't close the input file or output file. Instead, the program uses the feature of the UNIX kernel that closes all open file descriptors in a process when that process terminates.
- This example works for both text files and binary files, since there is no difference between the two to the UNIX kernel.

One question we haven't answered, however, is how we chose the `BUFFSIZE` value. Before answering that, let's run the program using different values for `BUFFSIZE`. Figure 3.6 shows the results for reading a 516,581,760-byte file, using 20 different buffer sizes.

`BUFFSIZE`	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	17.54	114.03	131.82	516,581,760
2	9.50	56.60	66.29	258,290,880
4	4.88	32.03	37.24	129,145,440
8	2.58	14.61	17.34	64,572,720
16	1.22	7.71	9.70	32,286,360
32	0.59	4.30	6.57	16,143,180
64	0.33	2.70	6.51	8,071,590
128	0.28	1.82	6.47	4,035,795
256	0.12	1.32	6.47	2,017,898
512	0.07	0.94	6.47	1,008,949
1,024	0.02	0.74	6.48	504,475
2,048	0.02	0.60	6.47	252,238
4,096	0.01	0.54	6.48	126,119
8,192	0.02	0.52	6.47	63,060
16,384	0.01	0.52	6.47	31,530
32,768	0.00	0.54	6.47	15,765
65,536	0.00	0.53	6.47	7,883
131,072	0.01	0.52	6.45	3,942
262,144	0.00	0.53	6.47	1,971
524,288	0.00	0.52	6.47	986

Figure 3.6 Timing results for reading with different buffer sizes on Linux

The file was read using the program shown in Figure 3.5, with standard output redirected to `/dev/null`. The file system used for this test was the Linux `ext4` file system with 4,096-byte blocks. (The `st_blksize` value, which we describe in Section 4.12, is 4,096.) This accounts for the minimum in the system time occurring at the few timing measurements starting around a `BUFFSIZE` of 4,096. Increasing the buffer size beyond this limit has little positive effect.

Most file systems support some kind of read-ahead to improve performance. When sequential reads are detected, the system tries to read in more data than an application requests, assuming that the application will read it shortly. The effect of read-ahead can be seen in Figure 3.6, where the elapsed time for buffer sizes as small as 32 bytes is as good as the elapsed time for larger buffer sizes.

We'll return to this timing example later in the text. In Section 3.14, we show the effect of synchronous writes; in Section 5.8, we compare these unbuffered I/O times with the standard I/O library.

> Beware when trying to measure the performance of programs that read and write files. The operating system will try to cache the file incore, so if you measure the performance of the program repeatedly, the successive timings will likely be better than the first. This improvement occurs because the first run causes the file to be entered into the system's cache, and successive runs access the file from the system's cache instead of from the disk. (The term *incore* means *in main memory*. Back in the day, a computer's main memory was built out of ferrite core. This is where the phrase "core dump" comes from: the main memory image of a program stored in a file on disk for diagnosis.)
>
> In the tests reported in Figure 3.6, each run with a different buffer size was made using a different copy of the file so that the current run didn't find the data in the cache from the previous run. The files are large enough that they all don't remain in the cache (the test system was configured with 6 GB of RAM).

3.10 File Sharing

The UNIX System supports the sharing of open files among different processes. Before describing the dup function, we need to describe this sharing. To do this, we'll examine the data structures used by the kernel for all I/O.

> The following description is conceptual; it may or may not match a particular implementation. Refer to Bach [1986] for a discussion of these structures in System V. McKusick et al. [1996] describe these structures in 4.4BSD. McKusick and Neville-Neil [2005] cover FreeBSD 5.2. For a similar discussion of Solaris, see McDougall and Mauro [2007]. The Linux 2.6 kernel architecture is discussed in Bovet and Cesati [2006].

The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
   (a) The file descriptor flags (close-on-exec; refer to Figure 3.7 and Section 3.14)
   (b) A pointer to a file table entry
2. The kernel maintains a file table for all open files. Each file table entry contains
   (a) The file status flags for the file, such as read, write, append, sync, and nonblocking; more on these in Section 3.14
   (b) The current file offset
   (c) A pointer to the v-node table entry for the file
3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the

v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on. (We talk more about i-nodes in Section 4.14 when we describe the typical UNIX file system in more detail.)

> Linux has no v-node. Instead, a generic i-node structure is used. Although the implementations differ, the v-node is conceptually the same as a generic i-node. Both point to an i-node structure specific to the file system.

We're ignoring some implementation details that don't affect our discussion. For example, the table of open file descriptors can be stored in the user area (a separate per-process structure that can be paged out) instead of the process table. Also, these tables can be implemented in numerous ways—they need not be arrays; one alternate implementation is a linked lists of structures. Regardless of the implementation details, the general concepts remain the same.

Figure 3.7 shows a pictorial arrangement of these three tables for a single process that has two different files open: one file is open on standard input (file descriptor 0), and the other is open on standard output (file descriptor 1).

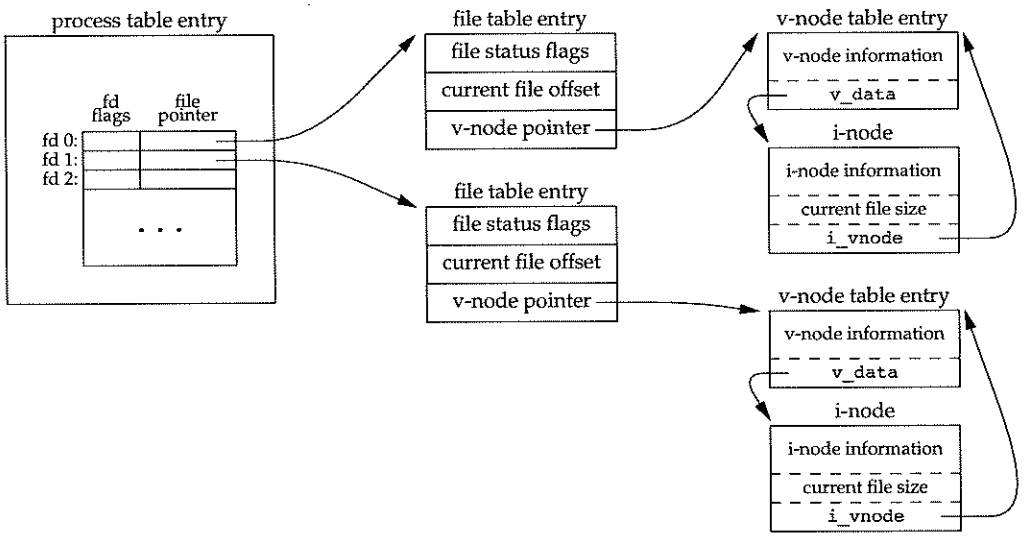

Figure 3.7 Kernel data structures for open files

The arrangement of these three tables has existed since the early versions of the UNIX System [Thompson 1978]. This arrangement is critical to the way files are shared among processes. We'll return to this figure in later chapters, when we describe additional ways that files are shared.

The v-node was invented to provide support for multiple file system types on a single computer system. This work was done independently by Peter Weinberger (Bell Laboratories) and Bill Joy (Sun Microsystems). Sun called this the Virtual File System and called the file system–independent portion of the i-node the v-node [Kleiman 1986]. The v-node propagated through various vendor implementations as support for Sun's Network File System (NFS) was added. The first release from Berkeley to provide v-nodes was the 4.3BSD Reno release, when NFS was added.

In SVR4, the v-node replaced the file system–independent i-node of SVR3. Solaris is derived from SVR4 and, therefore, uses v-nodes.

Instead of splitting the data structures into a v-node and an i-node, Linux uses a file system–independent i-node and a file system–dependent i-node.

If two independent processes have the same file open, we could have the arrangement shown in Figure 3.8.

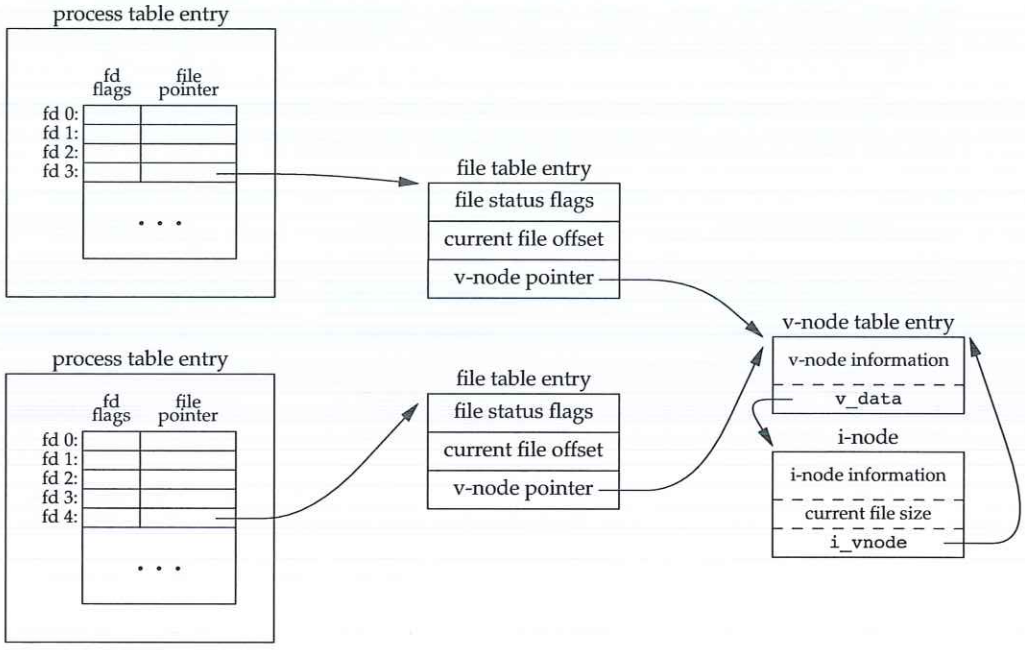

Figure 3.8 Two independent processes with the same file open

We assume here that the first process has the file open on descriptor 3 and that the second process has that same file open on descriptor 4. Each process that opens the file gets its own file table entry, but only a single v-node table entry is required for a given file. One reason each process gets its own file table entry is so that each process has its own current offset for the file.

Given these data structures, we now need to be more specific about what happens with certain operations that we've already described.

- After each `write` is complete, the current file offset in the file table entry is incremented by the number of bytes written. If this causes the current file offset to exceed the current file size, the current file size in the i-node table entry is set to the current file offset (for example, the file is extended).
- If a file is opened with the `O_APPEND` flag, a corresponding flag is set in the file status flags of the file table entry. Each time a `write` is performed for a file with this append flag set, the current file offset in the file table entry is first set to the current file size from the i-node table entry. This forces every `write` to be appended to the current end of file.
- If a file is positioned to its current end of file using `lseek`, all that happens is the current file offset in the file table entry is set to the current file size from the i-node table entry. (Note that this is not the same as if the file was opened with the `O_APPEND` flag, as we will see in Section 3.11.)
- The `lseek` function modifies only the current file offset in the file table entry. No I/O takes place.

It is possible for more than one file descriptor entry to point to the same file table entry, as we'll see when we discuss the dup function in Section 3.12. This also happens after a `fork` when the parent and the child share the same file table entry for each open descriptor (Section 8.3).

Note the difference in scope between the file descriptor flags and the file status flags. The former apply only to a single descriptor in a single process, whereas the latter apply to all descriptors in any process that point to the given file table entry. When we describe the `fcntl` function in Section 3.14, we'll see how to fetch and modify both the file descriptor flags and the file status flags.

Everything that we've described so far in this section works fine for multiple processes that are reading the same file. Each process has its own file table entry with its own current file offset. Unexpected results can arise, however, when multiple processes write to the same file. To see how to avoid some surprises, we need to understand the concept of atomic operations.

3.11 Atomic Operations

Appending to a File

Consider a single process that wants to append to the end of a file. Older versions of the UNIX System didn't support the `O_APPEND` option to open, so the program was coded as follows:

```
if (lseek(fd, 0L, 2) < 0)          /* position to EOF */
   err_sys("lseek error");
if (write(fd, buf, 100) != 100)    /* and write */
   err_sys("write error");
```

This works fine for a single process, but problems arise if multiple processes use this technique to append to the same file. (This scenario can arise if multiple instances of the same program are appending messages to a log file, for example.)

Assume that two independent processes, A and B, are appending to the same file. Each has opened the file but *without* the `O_APPEND` flag. This gives us the same picture as Figure 3.8. Each process has its own file table entry, but they share a single v-node table entry. Assume that process A does the `lseek` and that this sets the current offset for the file for process A to byte offset 1,500 (the current end of file). Then the kernel switches processes, and B continues running. Process B then does the `lseek`, which sets the current offset for the file for process B to byte offset 1,500 also (the current end of file). Then B calls `write`, which increments B's current file offset for the file to 1,600. Because the file's size has been extended, the kernel also updates the current file size in the v-node to 1,600. Then the kernel switches processes and A resumes. When A calls `write`, the data is written starting at the current file offset for A, which is byte offset 1,500. This overwrites the data that B wrote to the file.

The problem here is that our logical operation of "position to the end of file and write" requires two separate function calls (as we've shown it). The solution is to have the positioning to the current end of file and the write be an atomic operation with regard to other processes. Any operation that requires more than one function call cannot be atomic, as there is always the possibility that the kernel might temporarily suspend the process between the two function calls (as we assumed previously).

The UNIX System provides an atomic way to do this operation if we set the `O_APPEND` flag when a file is opened. As we described in the previous section, this causes the kernel to position the file to its current end of file before each `write`. We no longer have to call `lseek` before each `write`.

`pread` and `pwrite` Functions

The Single UNIX Specification includes two functions that allow applications to seek and perform I/O atomically: `pread` and `pwrite`.

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
                                    Returns: number of bytes read, 0 if end of file, -1 on error

ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
                                    Returns: number of bytes written if OK, -1 on error
```

Calling `pread` is equivalent to calling `lseek` followed by a call to `read`, with the following exceptions.

- There is no way to interrupt the two operations that occur when we call `pread`.
- The current file offset is not updated.

Calling pwrite is equivalent to calling lseek followed by a call to write, with similar exceptions.

Creating a File

We saw another example of an atomic operation when we described the O_CREAT and O_EXCL options for the open function. When both of these options are specified, the open will fail if the file already exists. We also said that the check for the existence of the file and the creation of the file was performed as an atomic operation. If we didn't have this atomic operation, we might try

```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

The problem occurs if the file is created by another process between the open and the creat. If the file is created by another process between these two function calls, and if that other process writes something to the file, that data is erased when this creat is executed. Combining the test for existence and the creation into a single atomic operation avoids this problem.

In general, the term *atomic operation* refers to an operation that might be composed of multiple steps. If the operation is performed atomically, either all the steps are performed (on success) or none are performed (on failure). It must not be possible for only a subset of the steps to be performed. We'll return to the topic of atomic operations when we describe the link function (Section 4.15) and record locking (Section 14.3).

3.12 dup and dup2 Functions

An existing file descriptor is duplicated by either of the following functions:

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, –1 on error

The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor. With dup2, we specify the value of the new descriptor with the *fd2* argument. If *fd2* is already open, it is first closed. If *fd* equals *fd2*, then dup2 returns *fd2* without closing it. Otherwise, the FD_CLOEXEC file descriptor flag is cleared for *fd2*, so that *fd2* is left open if the process calls exec.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the *fd* argument. We show this in Figure 3.9.

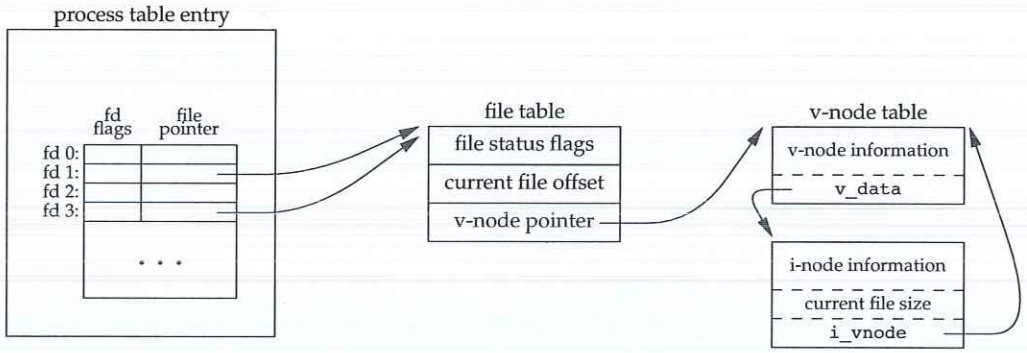

Figure 3.9 Kernel data structures after `dup(1)`

In this figure, we assume that when it's started, the process executes

```
newfd = dup(1);
```

We assume that the next available descriptor is 3 (which it probably is, since 0, 1, and 2 are opened by the shell). Because both descriptors point to the same file table entry, they share the same file status flags—read, write, append, and so on—and the same current file offset.

Each descriptor has its own set of file descriptor flags. As we describe in Section 3.14, the close-on-exec file descriptor flag for the new descriptor is always cleared by the dup functions.

Another way to duplicate a descriptor is with the `fcntl` function, which we describe in Section 3.14. Indeed, the call

```
dup(fd);
```

is equivalent to

```
fcntl(fd, F_DUPFD, 0);
```

Similarly, the call

```
dup2(fd, fd2);
```

is equivalent to

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

In this last case, the dup2 is not exactly the same as a `close` followed by an `fcntl`. The differences are as follows:

1. dup2 is an atomic operation, whereas the alternate form involves two function calls. It is possible in the latter case to have a signal catcher called between the close and the fcntl that could modify the file descriptors. (We describe signals in Chapter 10.) The same problem could occur if a different thread changes the file descriptors. (We describe threads in Chapter 11.)

2. There are some errno differences between dup2 and fcntl.

The dup2 system call originated with Version 7 and propagated through the BSD releases. The fcntl method for duplicating file descriptors appeared with System III and continued with System V. SVR3.2 picked up the dup2 function, and 4.2BSD picked up the fcntl function and the F_DUPFD functionality. POSIX.1 requires both dup2 and the F_DUPFD feature of fcntl.

3.13 sync, fsync, and fdatasync Functions

Traditional implementations of the UNIX System have a buffer cache or page cache in the kernel through which most disk I/O passes. When we write data to a file, the data is normally copied by the kernel into one of its buffers and queued for writing to disk at some later time. This is called *delayed write*. (Chapter 3 of Bach [1986] discusses this buffer cache in detail.)

The kernel eventually writes all the delayed-write blocks to disk, normally when it needs to reuse the buffer for some other disk block. To ensure consistency of the file system on disk with the contents of the buffer cache, the sync, fsync, and fdatasync functions are provided.

```
#include <unistd.h>

int fsync(int fd);

int fdatasync(int fd);
                                                    Returns: 0 if OK, –1 on error

void sync(void);
```

The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.

The function sync is normally called periodically (usually every 30 seconds) from a system daemon, often called update. This guarantees regular flushing of the kernel's block buffers. The command sync(1) also calls the sync function.

The function fsync refers only to a single file, specified by the file descriptor *fd*, and waits for the disk writes to complete before returning. This function is used when an application, such as a database, needs to be sure that the modified blocks have been written to the disk.

The fdatasync function is similar to fsync, but it affects only the data portions of a file. With fsync, the file's attributes are also updated synchronously.

All four of the platforms described in this book support sync and fsync. However, FreeBSD 8.0 does not support fdatasync.

3.14 fcntl Function

The fcntl function can change the properties of a file that is already open.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );
```

Returns: depends on *cmd* if OK (see following), –1 on error

In the examples in this section, the third argument is always an integer, corresponding to the comment in the function prototype just shown. When we describe record locking in Section 14.3, however, the third argument becomes a pointer to a structure.

The fcntl function is used for five different purposes.

1. Duplicate an existing descriptor (*cmd* = F_DUPFD or F_DUPFD_CLOEXEC)
2. Get/set file descriptor flags (*cmd* = F_GETFD or F_SETFD)
3. Get/set file status flags (*cmd* = F_GETFL or F_SETFL)
4. Get/set asynchronous I/O ownership (*cmd* = F_GETOWN or F_SETOWN)
5. Get/set record locks (*cmd* = F_GETLK, F_SETLK, or F_SETLKW)

We'll now describe the first 8 of these 11 *cmd* values. (We'll wait until Section 14.3 to describe the last 3, which deal with record locking.) Refer to Figure 3.7, as we'll discuss both the file descriptor flags associated with each file descriptor in the process table entry and the file status flags associated with each file table entry.

F_DUPFD — Duplicate the file descriptor *fd*. The new file descriptor is returned as the value of the function. It is the lowest-numbered descriptor that is not already open, and that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as *fd*. (Refer to Figure 3.9.) But the new descriptor has its own set of file descriptor flags, and its FD_CLOEXEC file descriptor flag is cleared. (This means that the descriptor is left open across an exec, which we discuss in Chapter 8.)

F_DUPFD_CLOEXEC — Duplicate the file descriptor and set the FD_CLOEXEC file descriptor flag associated with the new descriptor. Returns the new file descriptor.

F_GETFD — Return the file descriptor flags for *fd* as the value of the function. Currently, only one file descriptor flag is defined: the FD_CLOEXEC flag.

F_SETFD — Set the file descriptor flags for *fd*. The new flag value is set from the third argument (taken as an integer).

Be aware that some existing programs that deal with the file descriptor flags don't use the constant FD_CLOEXEC. Instead, these programs set the flag to either 0 (don't close-on-exec, the default) or 1 (do close-on-exec).

F_GETFL Return the file status flags for *fd* as the value of the function. We described the file status flags when we described the open function. They are listed in Figure 3.10.

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXEC	open for execute only
O_SEARCH	open directory for searching only
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	synchronize reads and writes
O_FSYNC	wait for writes to complete (FreeBSD and Mac OS X only)
O_ASYNC	asynchronous I/O (FreeBSD and Mac OS X only)

Figure 3.10 File status flags for fcntl

Unfortunately, the five access-mode flags—O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, and O_SEARCH—are not separate bits that can be tested. (As we mentioned earlier, the first three often have the values 0, 1, and 2, respectively, for historical reasons. Also, these five values are mutually exclusive; a file can have only one of them enabled.) Therefore, we must first use the O_ACCMODE mask to obtain the access-mode bits and then compare the result against any of the five values.

F_SETFL Set the file status flags to the value of the third argument (taken as an integer). The only flags that can be changed are O_APPEND, O_NONBLOCK, O_SYNC, O_DSYNC, O_RSYNC, O_FSYNC, and O_ASYNC.

F_GETOWN Get the process ID or process group ID currently receiving the SIGIO and SIGURG signals. We describe these asynchronous I/O signals in Section 14.5.2.

F_SETOWN Set the process ID or process group ID to receive the SIGIO and SIGURG signals. A positive *arg* specifies a process ID. A negative *arg* implies a process group ID equal to the absolute value of *arg*.

The return value from fcntl depends on the command. All commands return –1 on an error or some other value if OK. The following four commands have special return values: F_DUPFD, F_GETFD, F_GETFL, and F_GETOWN. The first command returns the new file descriptor, the next two return the corresponding flags, and the final command returns a positive process ID or a negative process group ID.

Example

The program in Figure 3.11 takes a single command-line argument that specifies a file descriptor and prints a description of selected file flags for that descriptor.

```
#include "apue.h"
#include <fcntl.h>

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

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    if (val & O_SYNC)
        printf(", synchronous writes");

#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC) && (O_FSYNC != O_SYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif

    putchar('\n');
    exit(0);
}
```

Figure 3.11 Print file flags for specified descriptor

Note that we use the feature test macro `_POSIX_C_SOURCE` and conditionally compile the file access flags that are not part of POSIX.1. The following script shows the

operation of the program, when invoked from bash (the Bourne-again shell). Results will vary, depending on which shell you use.

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write
```

The clause 5<>temp.foo opens the file temp.foo for reading and writing on file descriptor 5. □

Example

When we modify either the file descriptor flags or the file status flags, we must be careful to fetch the existing flag value, modify it as desired, and then set the new flag value. We can't simply issue an F_SETFD or an F_SETFL command, as this could turn off flag bits that were previously set.

Figure 3.12 shows a function that sets one or more of the file status flags for a descriptor.

```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

Figure 3.12 Turn on one or more of the file status flags for a descriptor

If we change the middle statement to

```
val &= ~flags;      /* turn flags off */
```

we have a function named clr_fl, which we'll use in some later examples. This statement logically ANDs the one's complement of flags with the current val.

If we add the line

```
set_fl(STDOUT_FILENO, O_SYNC);
```

to the beginning of the program shown in Figure 3.5, we'll turn on the synchronous-write flag. This causes each `write` to wait for the data to be written to disk before returning. Normally in the UNIX System, a `write` only queues the data for writing; the actual disk write operation can take place sometime later. A database system is a likely candidate for using `O_SYNC`, so that it knows on return from a `write` that the data is actually on the disk, in case of an abnormal system failure.

We expect the `O_SYNC` flag to increase the system and clock times when the program runs. To test this, we can run the program in Figure 3.5, copying 492.6 MB of data from one file on disk to another and compare this with a version that does the same thing with the `O_SYNC` flag set. The results from a Linux system using the `ext4` file system are shown in Figure 3.13.

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
read time from Figure 3.6 for `BUFFSIZE` = 4,096	0.03	0.58	8.62
normal `write` to disk file	0.00	1.05	9.70
`write` to disk file with `O_SYNC` set	0.02	1.09	10.28
`write` to disk followed by `fdatasync`	0.02	1.14	17.93
`write` to disk followed by `fsync`	0.00	1.19	18.17
`write` to disk with `O_SYNC` set followed by `fsync`	0.02	1.15	17.88

Figure 3.13 Linux `ext4` timing results using various synchronization mechanisms

The six rows in Figure 3.13 were all measured with a `BUFFSIZE` of 4,096 bytes. The results in Figure 3.6 were measured while reading a disk file and writing to `/dev/null`, so there was no disk output. The second row in Figure 3.13 corresponds to reading a disk file and writing to another disk file. This is why the first and second rows in Figure 3.13 are different. The system time increases when we write to a disk file, because the kernel now copies the data from our process and queues the data for writing by the disk driver. We expect the clock time to increase as well when we write to a disk file.

When we enable synchronous writes, the system and clock times should increase significantly. As the third row shows, the system time for writing synchronously is not much more expensive than when we used delayed writes. This implies that the Linux operating system is doing the same amount of work for delayed and synchronous writes (which is unlikely), or else the `O_SYNC` flag isn't having the desired effect. In this case, the Linux operating system isn't allowing us to set the `O_SYNC` flag using `fcntl`, instead failing without returning an error (but it would have honored the flag if we were able to specify it when the file was opened).

The clock time in the last three rows reflects the extra time needed to wait for all of the writes to be committed to disk. After writing a file synchronously, we expect that a call to `fsync` will have no effect. This case is supposed to be represented by the last

row in Figure 3.13, but since the O_SYNC flag isn't having the intended effect, the last row behaves the same way as the fifth row.

Figure 3.14 shows timing results for the same tests run on Mac OS X 10.6.8, which uses the HFS file system. Note that the times match our expectations: synchronous writes are far more expensive than delayed writes, and using fsync with synchronous writes makes very little difference. Note also that adding a call to fsync at the end of the delayed writes makes little measurable difference. It is likely that the operating system flushed previously written data to disk as we were writing new data to the file, so by the time that we called fsync, very little work was left to be done.

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
write to /dev/null	0.14	1.02	5.28
normal write to disk file	0.14	3.21	17.04
write to disk file with O_SYNC set	0.39	16.89	60.82
write to disk followed by fsync	0.13	3.07	17.10
write to disk with O_SYNC set followed by fsync	0.39	18.18	62.39

Figure 3.14 Mac OS X HFS timing results using various synchronization mechanisms

Compare fsync and fdatasync, both of which update a file's contents when we say so, with the O_SYNC flag, which updates a file's contents every time we write to the file. The performance of each alternative will depend on many factors, including the underlying operating system implementation, the speed of the disk drive, and the type of the file system. □

With this example, we see the need for fcntl. Our program operates on a descriptor (standard output), never knowing the name of the file that was opened on that descriptor. We can't set the O_SYNC flag when the file is opened, since the shell opened the file. With fcntl, we can modify the properties of a descriptor, knowing only the descriptor for the open file. We'll see another need for fcntl when we describe nonblocking pipes (Section 15.2), since all we have with a pipe is a descriptor.

3.15 ioctl Function

The ioctl function has always been the catchall for I/O operations. Anything that couldn't be expressed using one of the other functions in this chapter usually ended up being specified with an ioctl. Terminal I/O was the biggest user of this function. (When we get to Chapter 18, we'll see that POSIX.1 has replaced the terminal I/O operations with separate functions.)

```
#include <unistd.h>     /* System V */
#include <sys/ioctl.h>  /* BSD and Linux */

int ioctl(int fd, int request, ...);
```

Returns: −1 on error, something else if OK

The `ioctl` function was included in the Single UNIX Specification only as an extension for dealing with STREAMS devices [Rago 1993], but it was moved to obsolescent status in SUSv4. UNIX System implementations use `ioctl` for many miscellaneous device operations. Some implementations have even extended it for use with regular files.

The prototype that we show corresponds to POSIX.1. FreeBSD 8.0 and Mac OS X 10.6.8 declare the second argument as an `unsigned long`. This detail doesn't matter, since the second argument is always a `#defined` name from a header.

For the ISO C prototype, an ellipsis is used for the remaining arguments. Normally, however, there is only one more argument, and it's usually a pointer to a variable or a structure.

In this prototype, we show only the headers required for the function itself. Normally, additional device-specific headers are required. For example, the `ioctl` commands for terminal I/O, beyond the basic operations specified by POSIX.1, all require the `<termios.h>` header.

Each device driver can define its own set of `ioctl` commands. The system, however, provides generic `ioctl` commands for different classes of devices. Examples of some of the categories for these generic `ioctl` commands supported in FreeBSD are summarized in Figure 3.15.

Category	Constant names	Header	Number of `ioctl`s
disk labels	`DIOxxx`	`<sys/disklabel.h>`	4
file I/O	`FIOxxx`	`<sys/filio.h>`	14
mag tape I/O	`MTIOxxx`	`<sys/mtio.h>`	11
socket I/O	`SIOxxx`	`<sys/sockio.h>`	73
terminal I/O	`TIOxxx`	`<sys/ttycom.h>`	43

Figure 3.15 Common FreeBSD `ioctl` operations

The mag tape operations allow us to write end-of-file marks on a tape, rewind a tape, space forward over a specified number of files or records, and the like. None of these operations is easily expressed in terms of the other functions in the chapter (`read`, `write`, `lseek`, and so on), so the easiest way to handle these devices has always been to access their operations using `ioctl`.

We use the `ioctl` function in Section 18.12 to fetch and set the size of a terminal's window, and in Section 19.7 when we access the advanced features of pseudo terminals.

3.16 `/dev/fd`

Newer systems provide a directory named `/dev/fd` whose entries are files named `0`, `1`, `2`, and so on. Opening the file `/dev/fd/`*n* is equivalent to duplicating descriptor *n*, assuming that descriptor *n* is open.

The `/dev/fd` feature was developed by Tom Duff and appeared in the 8th Edition of the Research UNIX System. It is supported by all of the systems described in this book: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. It is not part of POSIX.1.

In the function call

```
fd = open("/dev/fd/0", mode);
```

most systems ignore the specified mode, whereas others require that it be a subset of the mode used when the referenced file (standard input, in this case) was originally opened. Because the previous open is equivalent to

```
fd = dup(0);
```

the descriptors 0 and fd share the same file table entry (Figure 3.9). For example, if descriptor 0 was opened read-only, we can only read on fd. Even if the system ignores the open mode and the call

```
fd = open("/dev/fd/0", O_RDWR);
```

succeeds, we still can't write to fd.

> The Linux implementation of /dev/fd is an exception. It maps file descriptors into symbolic links pointing to the underlying physical files. When you open /dev/fd/0, for example, you are really opening the file associated with your standard input. Thus the mode of the new file descriptor returned is unrelated to the mode of the /dev/fd file descriptor.

We can also call creat with a /dev/fd pathname argument as well as specify O_CREAT in a call to open. This allows a program that calls creat to still work if the pathname argument is /dev/fd/1, for example.

> Beware of doing this on Linux. Because the Linux implementation uses symbolic links to the real files, using creat on a /dev/fd file will result in the underlying file being truncated.

Some systems provide the pathnames /dev/stdin, /dev/stdout, and /dev/stderr. These pathnames are equivalent to /dev/fd/0, /dev/fd/1, and /dev/fd/2, respectively.

The main use of the /dev/fd files is from the shell. It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames. For example, the cat(1) program specifically looks for an input filename of - and uses it to mean standard input. The command

```
filter file2 | cat file1 - file3 | lpr
```

is an example. First, cat reads file1, then its standard input (the output of the filter program on file2), and then file3. If /dev/fd is supported, the special handling of - can be removed from cat, and we can enter

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

The special meaning of - as a command-line argument to refer to the standard input or the standard output is a kludge that has crept into many programs. There are also problems if we specify - as the first file, as it looks like the start of another command-line option. Using /dev/fd is a step toward uniformity and cleanliness.

3.17 Summary

This chapter has described the basic I/O functions provided by the UNIX System. These are often called the unbuffered I/O functions because each `read` or `write` invokes a system call into the kernel. Using only `read` and `write`, we looked at the effect of various I/O sizes on the amount of time required to read a file. We also looked at several ways to flush written data to disk and their effect on application performance.

Atomic operations were introduced when multiple processes append to the same file and when multiple processes create the same file. We also looked at the data structures used by the kernel to share information about open files. We'll return to these data structures later in the text.

We also described the `ioctl` and `fcntl` functions. We return to both of these functions later in the book. In Chapter 14, we'll use `fcntl` for record locking. In Chapter 18 and Chapter 19, we'll use `ioctl` when we deal with terminal devices.

Exercises

3.1 When reading or writing a disk file, are the functions described in this chapter really unbuffered? Explain.

3.2 Write your own `dup2` function that behaves the same way as the `dup2` function described in Section 3.12, without calling the `fcntl` function. Be sure to handle errors correctly.

3.3 Assume that a process executes the following three function calls:

```
fd1 = open(path, oflags);
fd2 = dup(fd1);
fd3 = open(path, oflags);
```

Draw the resulting picture, similar to Figure 3.9. Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFD`? Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFL`?

3.4 The following sequence of code has been observed in various programs:

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
    close(fd);
```

To see why the `if` test is needed, assume that `fd` is 1 and draw a picture of what happens to the three descriptor entries and the corresponding file table entry with each call to `dup2`. Then assume that `fd` is 3 and draw the same picture.

3.5 The Bourne shell, Bourne-again shell, and Korn shell notation

digit1`>&`*digit2*

says to redirect descriptor *digit1* to the same file as descriptor *digit2*. What is the difference between the two commands shown below? (Hint: The shells process their command lines from left to right.)

```
./a.out > outfile 2>&1
./a.out 2>&1 > outfile
```

3.6 If you open a file for read–write with the append flag, can you still `read` from anywhere in the file using `lseek`? Can you use `lseek` to replace existing data in the file? Write a program to verify this.

This page intentionally left blank