Using *fcntl()* to modify open file status flags is particularly useful in the following cases:

- The file was not opened by the calling program, so that it had no control over the flags used in the *open()* call (e.g., the file may be one of the three standard descriptors that are opened before the program is started).
- The file descriptor was obtained from a system call other than *open()*. Examples of such system calls are *pipe()*, which creates a pipe and returns two file descriptors referring to either end of the pipe, and *socket()*, which creates a socket and returns a file descriptor referring to the socket.

To modify the open file status flags, we use *fcntl()* to retrieve a copy of the existing flags, then modify the bits we wish to change, and finally make a further call to *fcntl()* to update the flags. Thus, to enable the `O_APPEND` flag, we would write the following:

```
int flags;

flags = fcntl(fd, F_GETFL);
if (flags == -1)
    errExit("fcntl");
flags |= O_APPEND;
if (fcntl(fd, F_SETFL, flags) == -1)
    errExit("fcntl");
```

5.4 Relationship Between File Descriptors and Open Files

Up until now, it may have appeared that there is a one-to-one correspondence between a file descriptor and an open file. However, this is not the case. It is possible—and useful—to have multiple descriptors referring to the same open file. These file descriptors may be open in the same process or in different processes.

To understand what is going on, we need to examine three data structures maintained by the kernel:

- the per-process file descriptor table;
- the system-wide table of open file descriptions; and
- the file system i-node table.

For each process, the kernel maintains a table of *open file descriptors*. Each entry in this table records information about a single file descriptor, including:

- a set of flags controlling the operation of the file descriptor (there is just one such flag, the close-on-exec flag, which we describe in Section 27.4); and
- a reference to the open file description.

The kernel maintains a system-wide table of all *open file descriptions*. (This table is sometimes referred to as the *open file table*, and its entries are sometimes called *open file handles*.) An open file description stores all information relating to an open file, including:

- the current file offset (as updated by *read()* and *write()*, or explicitly modified using *lseek()*);

- status flags specified when opening the file (i.e., the *flags* argument to *open()*);
- the file access mode (read-only, write-only, or read-write, as specified in *open()*);
- settings relating to signal-driven I/O (Section 63.3); and
- a reference to the *i-node* object for this file.

Each file system has a table of *i-nodes* for all files residing in the file system. The i-node structure, and file systems in general, are discussed in more detail in Chapter 14. For now, we note that the i-node for each file includes the following information:

- file type (e.g., regular file, socket, or FIFO) and permissions;
- a pointer to a list of locks held on this file; and
- various properties of the file, including its size and timestamps relating to different types of file operations.

> Here, we are overlooking the distinction between on-disk and in-memory representations of an i-node. The on-disk i-node records the persistent attributes of a file, such as its type, permissions, and timestamps. When a file is accessed, an in-memory copy of the i-node is created, and this version of the i-node records a count of the open file descriptions referring to the i-node and the major and minor IDs of the device from which the i-node was copied. The in-memory i-node also records various ephemeral attributes that are associated with a file while it is open, such as file locks.

Figure 5-2 illustrates the relationship between file descriptors, open file descriptions, and i-nodes. In this diagram, two processes have a number of open file descriptors.

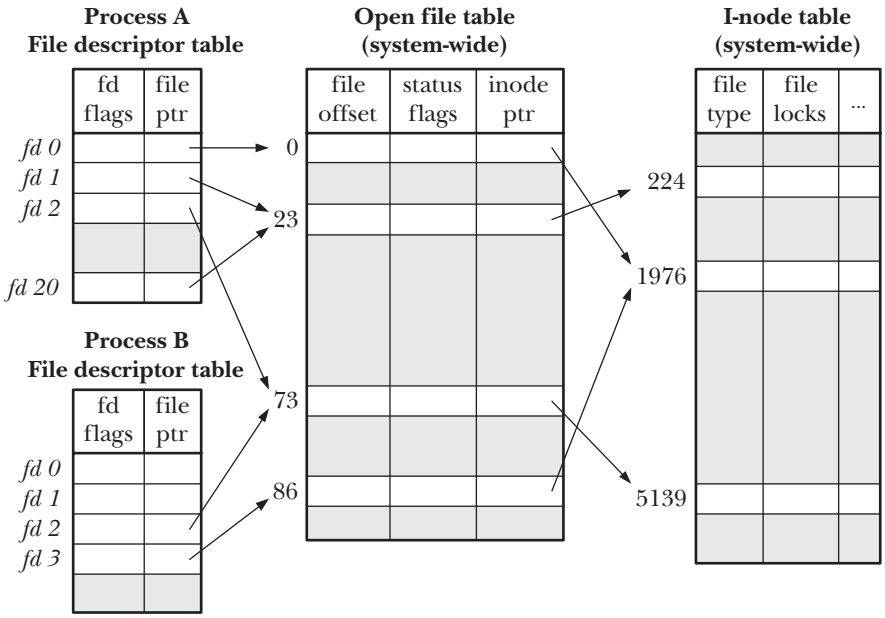

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

In process A, descriptors 1 and 20 both refer to the same open file description (labeled 23). This situation may arise as a result of a call to *dup()*, *dup2()*, or *fcntl()* (see Section 5.5).

Descriptor 2 of process A and descriptor 2 of process B refer to a single open file description (73). This scenario could occur after a call to *fork()* (i.e., process A is the parent of process B, or vice versa), or if one process passed an open descriptor to another process using a UNIX domain socket (Section 61.13.3).

Finally, we see that descriptor 0 of process A and descriptor 3 of process B refer to different open file descriptions, but that these descriptions refer to the same i-node table entry (1976)—in other words, to the same file. This occurs because each process independently called *open()* for the same file. A similar situation could occur if a single process opened the same file twice.

We can draw a number of implications from the preceding discussion:

- Two different file descriptors that refer to the same open file description share a file offset value. Therefore, if the file offset is changed via one file descriptor (as a consequence of calls to *read()*, *write()*, or *lseek()*), this change is visible through the other file descriptor. This applies both when the two file descriptors belong to the same process and when they belong to different processes.
- Similar scope rules apply when retrieving and changing the open file status flags (e.g., `O_APPEND`, `O_NONBLOCK`, and `O_ASYNC`) using the *fcntl()* `F_GETFL` and `F_SETFL` operations.
- By contrast, the file descriptor flags (i.e., the close-on-exec flag) are private to the process and file descriptor. Modifying these flags does not affect other file descriptors in the same process or a different process.

5.5 Duplicating File Descriptors

Using the (Bourne shell) I/O redirection syntax `2>&1` informs the shell that we wish to have standard error (file descriptor 2) redirected to the same place to which standard output (file descriptor 1) is being sent. Thus, the following command would (since the shell evaluates I/O directions from left to right) send both standard output and standard error to the file `results.log`:

```
$ ./myscript > results.log 2>&1
```

The shell achieves the redirection of standard error by duplicating file descriptor 2 so that it refers to the same open file description as file descriptor 1 (in the same way that descriptors 1 and 20 of process A refer to the same open file description in Figure 5-2). This effect can be achieved using the *dup()* and *dup2()* system calls.

Note that it is not sufficient for the shell simply to open the `results.log` file twice: once on descriptor 1 and once on descriptor 2. One reason for this is that the two file descriptors would not share a file offset pointer, and hence could end up overwriting each other's output. Another reason is that the file may not be a disk file. Consider the following command, which sends standard error down the same pipe as standard output:

```
$ ./myscript 2>&1 | less
```

The *dup()* call takes *oldfd*, an open file descriptor, and returns a new descriptor that refers to the same open file description. The new descriptor is guaranteed to be the lowest unused file descriptor.

```
#include <unistd.h>

int dup(int oldfd);
```

Returns (new) file descriptor on success, or −1 on error

Suppose we make the following call:

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

Assuming the normal situation where the shell has opened file descriptors 0, 1, and 2 on the program's behalf, and no other descriptors are in use, *dup()* will create the duplicate of descriptor 1 using file 3.

If we wanted the duplicate to be descriptor 2, we could use the following technique:

```
close(2);               /* Frees file descriptor 2 */
newfd = dup(1);         /* Should reuse file descriptor 2 */
```

This code works only if descriptor 0 was open. To make the above code simpler, and to ensure we always get the file descriptor we want, we can use *dup2()*.

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

Returns (new) file descriptor on success, or −1 on error

The *dup2()* system call makes a duplicate of the file descriptor given in *oldfd* using the descriptor number supplied in *newfd*. If the file descriptor specified in *newfd* is already open, *dup2()* closes it first. (Any error that occurs during this close is silently ignored; safer programming practice is to explicitly *close() newfd* if it is open before the call to *dup2()*.)

We could simplify the preceding calls to *close()* and *dup()* to the following:

```
dup2(1, 2);
```

A successful *dup2()* call returns the number of the duplicate descriptor (i.e., the value passed in *newfd*).

If *oldfd* is not a valid file descriptor, then *dup2()* fails with the error `EBADF` and *newfd* is not closed. If *oldfd* is a valid file descriptor, and *oldfd* and *newfd* have the same value, then *dup2()* does nothing—*newfd* is not closed, and *dup2()* returns the *newfd* as its function result.

A further interface that provides some extra flexibility for duplicating file descriptors is the *fcntl()* `F_DUPFD` operation:

```
newfd = fcntl(oldfd, F_DUPFD, startfd);
```

This call makes a duplicate of *oldfd* by using the lowest unused file descriptor greater than or equal to *startfd*. This is useful if we want a guarantee that the new descriptor (*newfd*) falls in a certain range of values. Calls to *dup()* and *dup2()* can always be recoded as calls to *close()* and *fcntl()*, although the former calls are more concise. (Note also that some of the *errno* error codes returned by *dup2()* and *fcntl()* differ, as described in the manual pages.)

From Figure 5-2, we can see that duplicate file descriptors share the same file offset value and status flags in their shared open file description. However, the new file descriptor has its own set of file descriptor flags, and its close-on-exec flag (FD_CLOEXEC) is always turned off. The interfaces that we describe next allow explicit control of the new file descriptor's close-on-exec flag.

The *dup3()* system call performs the same task as *dup2()*, but adds an additional argument, *flags*, that is a bit mask that modifies the behavior of the system call.

```
#define _GNU_SOURCE
#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
                        Returns (new) file descriptor on success, or –1 on error
```

Currently, *dup3()* supports one flag, O_CLOEXEC, which causes the kernel to enable the close-on-exec flag (FD_CLOEXEC) for the new file descriptor. This flag is useful for the same reasons as the *open()* O_CLOEXEC flag described in Section 4.3.1.

The *dup3()* system call is new in Linux 2.6.27, and is Linux-specific.

Since Linux 2.6.24, Linux also supports an additional *fcntl()* operation for duplicating file descriptors: F_DUPFD_CLOEXEC. This flag does the same thing as F_DUPFD, but additionally sets the close-on-exec flag (FD_CLOEXEC) for the new file descriptor. Again, this operation is useful for the same reasons as the *open()* O_CLOEXEC flag. F_DUPFD_CLOEXEC is not specified in SUSv3, but is specified in SUSv4.

5.6 File I/O at a Specified Offset: *pread()* and *pwrite()*

The *pread()* and *pwrite()* system calls operate just like *read()* and *write()*, except that the file I/O is performed at the location specified by *offset*, rather than at the current file offset. The file offset is left unchanged by these calls.

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
                        Returns number of bytes read, 0 on EOF, or –1 on error
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
                        Returns number of bytes written, or –1 on error
```