```
#include <unistd.h>

ssize_t pread(int filedes, 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 filedes, 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 using `pread`.
- The file pointer 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(pathname, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(pathname, 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, or none are performed. It must not be possible for 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 filedes);

int dup2(int filedes, int filedes2);
```

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 *filedes2* argument. If *filedes2* is already open, it is first closed. If *filedes* equals *filedes2*, then dup2 returns *filedes2* without closing it.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the *filedes* argument. We show this in Figure 3.8.

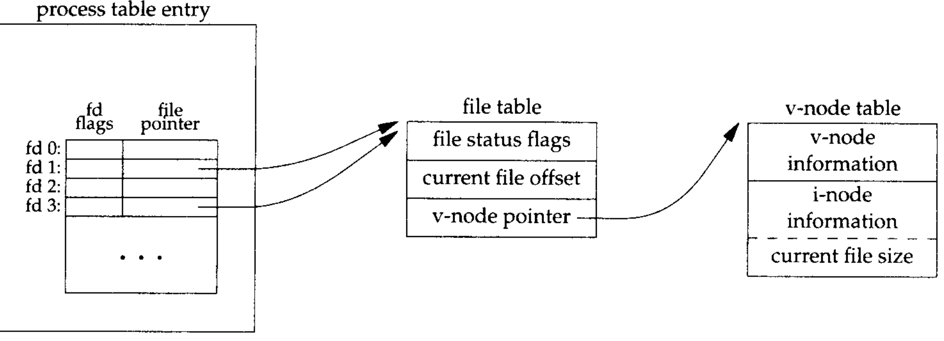


**Figure 3.8** Kernel data structures after dup(1)

In this figure, we're assuming 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 the next section, 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(filedes);
```

is equivalent to

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

Similarly, the call

```
dup2(filedes, filedes2);
```

is equivalent to

```
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

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.)
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 filedes);

int fdatasync(int filedes);

                                        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 *filedes*, and waits for the disk writes to complete before returning. The intended use of `fsync` is for an application, such as a database, that 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 5.2.1 and Mac OS X 10.3 do 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 filedes, 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. But when we describe record locking in Section 14.3, 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`)
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 seven of these ten *cmd* values. (We'll wait until Section 14.3 to describe the last three, which deal with record locking.) Refer to Figure 3.6, since we'll be referring to 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 *filedes*. The new file descriptor is returned as the value of the function. It is the lowest-numbered descriptor that is not already open, that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as *filedes*. (Refer to Figure 3.8.) 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_GETFD` Return the file descriptor flags for *filedes* 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 *filedes*. 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, the 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 *filedes* as the value of the function. We described the file status flags when we described the open function. They are listed in Figure 3.9.

| File status flag | Description |
|---|---|
| O_RDONLY | open for reading only |
| O_WRONLY | open for writing only |
| O_RDWR | open for reading and writing |
| 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 3.9** File status flags for fcntl

Unfortunately, the three access-mode flags—O_RDONLY, O_WRONLY, and O_RDWR—are not separate bits that can be tested. (As we mentioned earlier, these three often have the values 0, 1, and 2, respectively, for historical reasons. Also, these three values are mutually exclusive; a file can have only one of the three 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 three 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 14.6.2.

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 –[illegible] on an error or some other value if OK. The following four commands have s[illegible] return values: F_DUPFD, F_GETFD, F_GETFL, and F_GETOWN. The first retur[illegible] file descriptor, the next two return the corresponding flags, and the fin[illegible] positive process ID or a negative process group ID.

### Example

The program in Figure 3.10 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 defined(O_SYNC)
    if (val & O_SYNC)
        printf(", synchronous writes");
#endif
#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

**Figure 3.10** 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 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 do an `F_SETFD` or an `F_SETFL`, as this could turn off flag bits that were previously set.

Figure 3.11 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 3.11** 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 call set_fl from Figure 3.4 by adding the line

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

at the beginning of the program, 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 clock time when the program runs. To test this, we can run the program in Figure 3.4, copying 98.5 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 ext2 file system are shown in Figure 3.12.

| Operation | User CPU (seconds) | System CPU (seconds) | Clock time (seconds) |
|---|---|---|---|
| read time from Figure 3.5 for BUFFSIZE = 4,096 | 0.03 | 0.16 | 6.86 |
| normal write to disk file | 0.02 | 0.30 | 6.87 |
| write to disk file with O_SYNC set | 0.03 | 0.30 | 6.83 |
| write to disk followed by fdatasync | 0.03 | 0.42 | 18.28 |
| write to disk followed by fsync | 0.03 | 0.37 | 17.95 |
| write to disk with O_SYNC set followed by fsync | 0.05 | 0.44 | 17.95 |

**Figure 3.12** Linux ext2 timing results using various synchronization mechanisms

The six rows in Figure 3.12 were all measured with a BUFFSIZE of 4,096. The results in Figure 3.5 were measured reading a disk file and writing to /dev/null, so there was no disk output. The second row in Figure 3.12 corresponds to reading a disk file and writing to another disk file. This is why the first and second rows in Figure 3.12 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 also when we write to a disk file, but it doesn't increase significantly for this test, which indicates that our writes go to the system cache, and we don't measure the cost to actually write the data to disk.

When we enable synchronous writes, the system time and the clock time should increase significantly. As the third row shows, the time for writing synchronously is about the same as when we used delayed writes. This implies that the Linux ext2 file system isn't honoring the O_SYNC flag. This suspicion is supported by the sixth line, which shows that the time to do synchronous writes followed by a call to fsync is just as large as calling fsync after writing the file without synchronous writes (line 5). After writing a file synchronously, we expect that a call to fsync will have no effect.

Figure 3.13 shows timing results for the same tests on Mac OS X 10.3. Note that the times match our expectations: synchronous writes are far more expensive than delayed writes, and using fsync with synchronous writes makes no measurable difference. Note also that adding a call to fsync at the end of the delayed writes makes no

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 | 0.06 | 0.79 | 4.33 |
| normal write to disk file | 0.05 | 3.56 | 14.40 |
| write to disk file with O_FSYNC set | 0.13 | 9.53 | 22.48 |
| write to disk followed by fsync | 0.11 | 3.31 | 14.12 |
| write to disk with O_FSYNC set followed by fsync | 0.17 | 9.14 | 22.12 |

**Figure 3.13** Mac OS X timing results using various synchronization mechanisms

Compare fsync and fdatasync, 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. □

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 by the shell 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 */
#include <stropts.h>     /* XSI STREAMS */

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

Returns: −1 on error, something else if OK

The ioctl function is included in the Single UNIX Specification only as an extension for dealing with STREAMS devices [Rago 1993]. UNIX System implementations, however, use it 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 5.2.1 and Mac OS X 10.3 declare the second argument as an `unsigned long`. This detail doesn't matter, since the second argument is always a `#define`d 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 3.14.

| Category | Constant names | Header | Number of ioctls |
|---|---|---|---|
| disk labels | DIOxxx | <sys/disklabel.h> | 6 |
| file I/O | FIOxxx | <sys/filio.h> | 9 |
| mag tape I/O | MTIOxxx | <sys/mtio.h> | 11 |
| socket I/O | SIOxxx | <sys/sockio.h> | 60 |
| terminal I/O | TIOxxx | <sys/ttycom.h> | 44 |

**Figure 3.14** 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 14.4 when we describe the STREAMS system, 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 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9. 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 3.8). 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.

We can also call creat with a /dev/fd pathname argument, as well as specifying 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.

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.

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 this to mean standard input. The command

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

is an example. First, cat reads file1, next its standard input (the output of the filter program on file2), 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 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 in Chapter 14, where we'll use `ioctl` with the STREAMS I/O system, and `fcntl` for record locking.

## 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 performs the same service 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(pathname, oflags);
fd2 = dup(fd1);
fd3 = open(pathname, oflags);
```

Draw the resulting picture, similar to Figure 3.8. 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

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

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

(Hint: the shells process their command lines from left to right.)

**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.

# 4

# *Files and Directories*

## 4.1 Introduction

In the previous chapter we covered the basic functions that perform I/O. The discussion centered around I/O for regular files—opening a file, and reading or writing a file. We'll now look at additional features of the file system and the properties of a file. We'll start with the `stat` functions and go through each member of the `stat` structure, looking at all the attributes of a file. In this process, we'll also describe each of the functions that modify these attributes: change the owner, change the permissions, and so on. We'll also look in more detail at the structure of a UNIX file system and symbolic links. We finish this chapter with the functions that operate on directories, and we develop a function that descends through a directory hierarchy.

## 4.2 `stat`, `fstat`, and `lstat` Functions

The discussion in this chapter centers around the three `stat` functions and the information they return.

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

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *restrict pathname, struct stat *restrict buf);
```

All three return: 0 if OK, –1 on error

Given a *pathname*, the `stat` function returns a structure of information about the named file. The `fstat` function obtains information about the file that is already open on the descriptor *filedes*. The `lstat` function is similar to `stat`, but when the named file is a

symbolic link, `lstat` returns information about the symbolic link, not the file referenced by the symbolic link. (We'll need `lstat` in Section 4.21 when we walk down a directory hierarchy. We describe symbolic links in more detail in Section 4.16.)

The second argument is a pointer to a structure that we must supply. The function fills in the structure pointed to by *buf*. The definition of the structure can differ among implementations, but it could look like

```
struct stat {
  mode_t    st_mode;    /* file type & mode (permissions) */
  ino_t     st_ino;     /* i-node number (serial number) */
  dev_t     st_dev;     /* device number (file system) */
  dev_t     st_rdev;    /* device number for special files */
  nlink_t   st_nlink;   /* number of links */
  uid_t     st_uid;     /* user ID of owner */
  gid_t     st_gid;     /* group ID of owner */
  off_t     st_size;    /* size in bytes, for regular files */
  time_t    st_atime;   /* time of last access */
  time_t    st_mtime;   /* time of last modification */
  time_t    st_ctime;   /* time of last file status change */
  blksize_t st_blksize; /* best I/O block size */
  blkcnt_t  st_blocks;  /* number of disk blocks allocated */
};
```

The `st_rdev`, `st_blksize`, and `st_blocks` fields are not required by POSIX.1. They are defined as XSI extensions in the Single UNIX Specification.

Note that each member is specified by a primitive system data type (see Section 2.8). We'll go through each member of this structure to examine the attributes of a file.

The biggest user of the `stat` functions is probably the `ls -l` command, to learn all the information about a file.

## 4.3 File Types

We've talked about two different types of files so far: regular files and directories. Most files on a UNIX system are either regular files or directories, but there are additional types of files. The types are:

1. Regular file. The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file.

   One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format. All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data.

2. Directory file. A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write directly

to a directory file. Processes must use the functions described in this chapter to make changes to a directory.

3. Block special file. A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.
4. Character special file. A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.
5. FIFO. A type of file used for communication between processes. It's sometimes called a named pipe. We describe FIFOs in Section 15.5.
6. Socket. A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host. We use sockets for interprocess communication in Chapter 16.
7. Symbolic link. A type of file that points to another file. We talk more about symbolic links in Section 4.16.

The type of a file is encoded in the `st_mode` member of the `stat` structure. We can determine the file type with the macros shown in Figure 4.1. The argument to each of these macros is the `st_mode` member from the `stat` structure.

| Macro | Type of file |
|---|---|
| `S_ISREG()` | regular file |
| `S_ISDIR()` | directory file |
| `S_ISCHR()` | character special file |
| `S_ISBLK()` | block special file |
| `S_ISFIFO()` | pipe or FIFO |
| `S_ISLNK()` | symbolic link |
| `S_ISSOCK()` | socket |

**Figure 4.1** File type macros in `<sys/stat.h>`

POSIX.1 allows implementations to represent interprocess communication (IPC) objects, such as message queues and semaphores, as files. The macros shown in Figure 4.2 allow us to determine the type of IPC object from the `stat` structure. Instead of taking the `st_mode` member as an argument, these macros differ from those in Figure 4.1 in that their argument is a pointer to the `stat` structure.

| Macro | Type of object |
|---|---|
| `S_TYPEISMQ()` | message queue |
| `S_TYPEISSEM()` | semaphore |
| `S_TYPEISSHM()` | shared memory object |

**Figure 4.2** IPC type macros in `<sys/stat.h>`

Message queues, semaphores, and shared memory objects are discussed in Chapter 15. However, none of the various implementations of the UNIX System discussed in this book represent these objects as files.

## Example

The program in Figure 4.3 prints the type of file for each command-line argument.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

**Figure 4.3** Print type of file for each command-line argument

Sample output from Figure 4.3 is

```
$ ./a.out /etc/passwd /etc /dev/initctl /dev/log /dev/tty \
> /dev/scsi/host0/bus0/target0/lun0/cd /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/initctl: fifo
/dev/log: socket
/dev/tty: character special
/dev/scsi/host0/bus0/target0/lun0/cd: block special
/dev/cdrom: symbolic link
```

(Here, we have explicitly entered a backslash at the end of the first command line, telling the shell that we want to continue entering the command on another line. The shell then prompts us with its secondary prompt, >, on the next line.) We have specifically used the `lstat` function instead of the `stat` function to detect symbolic links. If we used the `stat` function, we would never see symbolic links.

To compile this program on a Linux system, we must define `_GNU_SOURCE` to include the definition of the `S_ISSOCK` macro. □

Historically, early versions of the UNIX System didn't provide the `S_ISxxx` macros. Instead, we had to logically AND the `st_mode` value with the mask `S_IFMT` and then compare the result with the constants whose names are `S_IFxxx`. Most systems define this mask and the related constants in the file `<sys/stat.h>`. If we examine this file, we'll find the `S_ISDIR` macro defined something like

```
#define  S_ISDIR(mode)   (((mode) & S_IFMT) == S_IFDIR)
```

We've said that regular files are predominant, but it is interesting to see what percentage of the files on a given system are of each file type. Figure 4.4 shows the counts and percentages for a Linux system that is used as a single-user workstation. This data was obtained from the program that we show in Section 4.21.

| File type | Count | Percentage |
|---|---|---|
| regular file | 226,856 | 88.22 % |
| directory | 23,017 | 8.95 |
| symbolic link | 6,442 | 2.51 |
| character special | 447 | 0.17 |
| block special | 312 | 0.12 |
| socket | 69 | 0.03 |
| FIFO | 1 | 0.00 |

**Figure 4.4** Counts and percentages of different file types

## 4.4 Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it. These are shown in Figure 4.5.

| | |
|---|---|
| real user ID<br>real group ID | who we really are |
| effective user ID<br>effective group ID<br>supplementary group IDs | used for file access permission checks |
| saved set-user-ID<br>saved set-group-ID | saved by `exec` functions |

**Figure 4.5** User IDs and group IDs associated with each process

- The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally, these

values don't change during a login session, although there are ways for a superuser process to change them, which we describe in Section 8.11.

- The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions, as we describe in the next section. (We defined supplementary group IDs in Section 1.8.)
- The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID when a program is executed. We describe the function of these two saved values when we describe the `setuid` function in Section 8.11.

> The saved IDs are required with the 2001 version of POSIX.1. They used to be optional in older versions of POSIX. An application can test for the constant `_POSIX_SAVED_IDS` at compile time or can call `sysconf` with the `_SC_SAVED_IDS` argument at runtime, to see whether the implementation supports this feature.

Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

Every file has an owner and a group owner. The owner is specified by the `st_uid` member of the `stat` structure; the group owner, by the `st_gid` member.

When we execute a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. But the capability exists to set a special flag in the file's mode word (`st_mode`) that says "when this file is executed, set the effective user ID of the process to be the owner of the file (`st_uid`)." Similarly, another bit can be set in the file's mode word that causes the effective group ID to be the group owner of the file (`st_gid`). These two bits in the file's mode word are called the *set-user-ID* bit and the *set-group-ID* bit.

For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the UNIX System program that allows anyone to change his or her password, `passwd`(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either `/etc/passwd` or `/etc/shadow`, files that should be writable only by the superuser. Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully. We'll discuss these types of programs in more detail in Chapter 8.

Returning to the `stat` function, the set-user-ID bit and the set-group-ID bit are contained in the file's `st_mode` value. These two bits can be tested against the constants `S_ISUID` and `S_ISGID`.

## 4.5 File Access Permissions

The `st_mode` value also encodes the access permission bits for the file. When we say *file,* we mean any of the file types that we described earlier. All the file types—directories, character special files, and so on—have permissions. Many people think only of regular files as having access permissions.

There are nine permission bits for each file, divided into three categories. These are shown in Figure 4.6.

| st_mode mask | Meaning |
| --- | --- |
| S_IRUSR | user-read |
| S_IWUSR | user-write |
| S_IXUSR | user-execute |
| S_IRGRP | group-read |
| S_IWGRP | group-write |
| S_IXGRP | group-execute |
| S_IROTH | other-read |
| S_IWOTH | other-write |
| S_IXOTH | other-execute |

**Figure 4.6** The nine file access permission bits, from <sys/stat.h>

The term *user* in the first three rows in Figure 4.6 refers to the owner of the file. The chmod(1) command, which is typically used to modify these nine permission bits, allows us to specify u for user (owner), g for group, and o for other. Some books refer to these three as owner, group, and world; this is confusing, as the chmod command uses o to mean other, not owner. We'll use the terms *user*, *group*, and *other*, to be consistent with the chmod command.

The three categories in Figure 4.6—read, write, and execute—are used in various ways by different functions. We'll summarize them here, and return to them when we describe the actual functions.

- The first rule is that *whenever* we want to open any type of file by name, we must have execute permission in each directory mentioned in the name, including the current directory, if it is implied. This is why the execute permission bit for a directory is often called the search bit.

  For example, to open the file /usr/include/stdio.h, we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/include. We then need appropriate permission for the file itself, depending on how we're trying to open it: read-only, read–write, and so on.

  If the current directory is /usr/include, then we need execute permission in the current directory to open the file stdio.h. This is an example of the current directory being implied, not specifically mentioned. It is identical to our opening the file ./stdio.h.

  Note that read permission for a directory and execute permission for a directory mean different things. Read permission lets us read the directory, obtaining a list of all the filenames in the directory. Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access. (We need to search the directory to look for a specific filename.)

  Another example of an implicit directory reference is if the PATH environment variable, described in Section 8.10, specifies a directory that does not have execute permission enabled. In this case, the shell will never find executable files in that directory.

- The read permission for a file determines whether we can open an existing file for reading: the `O_RDONLY` and `O_RDWR` flags for the `open` function.
- The write permission for a file determines whether we can open an existing file for writing: the `O_WRONLY` and `O_RDWR` flags for the `open` function.
- We must have write permission for a file to specify the `O_TRUNC` flag in the `open` function.
- We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
- To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.
- Execute permission for a file must be on if we want to execute the file using any of the six `exec` functions (Section 8.10). The file also has to be a regular file.

The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the owners of the file (`st_uid` and `st_gid`), the effective IDs of the process (effective user ID and effective group ID), and the supplementary group IDs of the process, if supported. The two owner IDs are properties of the file, whereas the two effective IDs and the supplementary group IDs are properties of the process. The tests performed by the kernel are as follows.

1. If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system.
2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied. By *appropriate access permission bit*, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.
4. If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied.

These four steps are tried in sequence. Note that if the process owns the file (step 2), access is granted or denied based only on the user access permissions; the group permissions are never looked at. Similarly, if the process does not own the file, but belongs to an appropriate group, access is granted or denied based only on the group access permissions; the other permissions are not looked at.

## 4.6 Ownership of New Files and Directories

When we described the creation of a new file in Chapter 3, using either `open` or `creat`, we never said what values were assigned to the user ID and group ID of the new file. We'll see how to create a new directory in Section 4.20 when we describe the `mkdir` function. The rules for the ownership of a new directory are identical to the rules in this section for the ownership of a new file.

The user ID of a new file is set to the effective user ID of the process. POSIX.1 allows an implementation to choose one of the following options to determine the group ID of a new file.

1. The group ID of a new file can be the effective group ID of the process.
2. The group ID of a new file can be the group ID of the directory in which the file is being created.

> FreeBSD 5.2.1 and Mac OS X 10.3 always uses the group ID of the directory as the group ID of the new file.
>
> The Linux `ext2` and `ext3` file systems allow the choice between these two POSIX.1 options to be made on a file system basis, using a special flag to the `mount(1)` command. On Linux 2.4.22 (with the proper mount option) and Solaris 9, the group ID of a new file depends on whether the set-group-ID bit is set for the directory in which the file is being created. If this bit is set for the directory, the group ID of the new file is set to the group ID of the directory; otherwise, the group ID of the new file is set to the effective group ID of the process.

Using the second option—inheriting the group ID of the directory—assures us that all files and directories created in that directory will have the group ID belonging to the directory. This group ownership of files and directories will then propagate down the hierarchy from that point. This is used, for example, in the `/var/spool/mail` directory on Linux.

> As we mentioned, this option for group ownership is the default for FreeBSD 5.2.1 and Mac OS X 10.3, but an option for Linux and Solaris. Under Linux 2.4.22 and Solaris 9, we have to enable the set-group-ID bit, and the `mkdir` function has to propagate a directory's set-group-ID bit automatically for this to work. (This is described in Section 4.20.)

## 4.7 access Function

As we described earlier, when we open a file, the kernel performs its access tests based on the effective user and group IDs. There are times when a process wants to test accessibility based on the real user and group IDs. This is useful when a process is running as someone else, using either the set-user-ID or the set-group-ID feature. Even though a process might be set-user-ID to root, it could still want to verify that the real user can access a given file. The `access` function bases its tests on the real user and group IDs. (Replace *effective* with *real* in the four steps at the end of Section 4.5.)

```
#include <unistd.h>

int access(const char *pathname, int mode);

                                        Returns: 0 if OK, -1 on error
```

The *mode* is the bitwise OR of any of the constants shown in Figure 4.7.

| *mode* | Description |
|---|---|
| R_OK | test for read permission |
| W_OK | test for write permission |
| X_OK | test for execute permission |
| F_OK | test for existence of file |

**Figure 4.7** The *mode* constants for access function, from <unistd.h>

### Example

Figure 4.8 shows the use of the access function.

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

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

**Figure 4.8** Example of access function

Here is a sample session with this program:

```
$ ls -l a.out
-rwxrwxr-x  1 sar          15945 Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls -l /etc/shadow
-r--------  1 root          1315 Jul 17  2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied
$ su                                  become superuser
Password:                             enter superuser password
# chown root a.out                    change file's user ID to root
# chmod u+s a.out                     and turn on set-user-ID bit
# ls -l a.out                         check owner and SUID bit
-rwsrwxr-x  1 root         15945 Nov 30 12:10 a.out
```

```
# exit                              go back to normal user
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```

In this example, the set-user-ID program can determine that the real user cannot normally read the file, even though the open function will succeed. □

In the preceding example and in Chapter 8, we'll sometimes switch to become the superuser, to demonstrate how something works. If you're on a multiuser system and do not have superuser permission, you won't be able to duplicate these examples completely.

## 4.8 umask Function

Now that we've described the nine permission bits associated with every file, we can describe the file mode creation mask that is associated with every process.

The umask function sets the file mode creation mask for the process and returns the previous value. (This is one of the few functions that doesn't have an error return.)

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

mode_t umask(mode_t cmask);
```

Returns: previous file mode creation mask

The *cmask* argument is formed as the bitwise OR of any of the nine constants from Figure 4.6: S_IRUSR, S_IWUSR, and so on.

The file mode creation mask is used whenever the process creates a new file or a new directory. (Recall from Sections 3.3 and 3.4 our description of the open and creat functions. Both accept a *mode* argument that specifies the new file's access permission bits.) We describe how to create a new directory in Section 4.20. Any bits that are *on* in the file mode creation mask are turned *off* in the file's *mode*.

### Example

The program in Figure 4.9 creates two files, one with a umask of 0 and one with a umask that disables all the group and other permission bits.

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

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
```

```
        err_sys("creat error for bar");
    exit(0);
}
```

Figure 4.9 Example of umask function

If we run this program, we can see how the permission bits have been set.

```
$ umask                          first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw-------  1 sar                0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar                0 Dec  7 21:20 foo
$ umask                          see if the file mode creation mask changed
002
```

□

Most users of UNIX systems never deal with their umask value. It is usually set once, on login, by the shell's start-up file, and never changed. Nevertheless, when writing programs that create new files, if we want to ensure that specific access permission bits are enabled, we must modify the umask value while the process is running. For example, if we want to ensure that anyone can read a file, we should set the umask to 0. Otherwise, the umask value that is in effect when our process is running can cause permission bits to be turned off.

In the preceding example, we use the shell's umask command to print the file mode creation mask before we run the program and after. This shows us that changing the file mode creation mask of a process doesn't affect the mask of its parent (often a shell). All of the shells have a built-in umask command that we can use to set or print the current file mode creation mask.

Users can set the umask value to control the default permissions on the files they create. The value is expressed in octal, with one bit representing one permission to be masked off, as shown in Figure 4.10. Permissions can be denied by setting the corresponding bits. Some common umask values are 002 to prevent others from writing your files, 022 to prevent group members and others from writing your files, and 027 to prevent group members from writing your files and others from reading, writing, or executing your files.

| Mask bit | Meaning |
|---|---|
| 0400 | user-read |
| 0200 | user-write |
| 0100 | user-execute |
| 0040 | group-read |
| 0020 | group-write |
| 0010 | group-execute |
| 0004 | other-read |
| 0002 | other-write |
| 0001 | other-execute |

Figure 4.10 The umask file access permission bits

The Single UNIX Specification requires that the shell support a symbolic form of the umask command. Unlike the octal format, the symbolic format specifies which permissions are to be allowed (i.e., clear in the file creation mask) instead of which ones are to be denied (i.e., set in the file creation mask). Compare both forms of the command, shown below.

```
$ umask                    first print the current file mode creation mask
002
$ umask -S                 print the symbolic form
u=rwx,g=rwx,o=rx
$ umask 027                change the file mode creation mask
$ umask -S                 print the symbolic form
u=rwx,g=rx,o=
```

## 4.9 chmod and fchmod Functions

These two functions allow us to change the file access permissions for an existing file.

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

int chmod(const char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);

                                        Both return: 0 if OK, −1 on error
```

The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened.

To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have superuser permissions.

The *mode* is specified as the bitwise OR of the constants shown in Figure 4.11.

| *mode* | Description |
|---|---|
| S_ISUID | set-user-ID on execution |
| S_ISGID | set-group-ID on execution |
| S_ISVTX | saved-text (sticky bit) |
| S_IRWXU | read, write, and execute by user (owner) |
| S_IRUSR | read by user (owner) |
| S_IWUSR | write by user (owner) |
| S_IXUSR | execute by user (owner) |
| S_IRWXG | read, write, and execute by group |
| S_IRGRP | read by group |
| S_IWGRP | write by group |
| S_IXGRP | execute by group |
| S_IRWXO | read, write, and execute by other (world) |
| S_IROTH | read by other (world) |
| S_IWOTH | write by other (world) |
| S_IXOTH | execute by other (world) |

**Figure 4.11** The *mode* constants for chmod functions, from <sys/stat.h>

Note that nine of the entries in Figure 4.11 are the nine file access permission bits from Figure 4.6. We've added the two set-ID constants (S_ISUID and S_ISGID), the saved-text constant (S_ISVTX), and the three combined constants (S_IRWXU, S_IRWXG, and S_IRWXO).

The saved-text bit (S_ISVTX) is not part of POSIX.1. It is defined as an XSI extension in the Single UNIX Specification. We describe its purpose in the next section.

## Example

Recall the final state of the files foo and bar when we ran the program in Figure 4.9 to demonstrate the umask function:

```
$ ls -l foo bar
-rw-------  1 sar          0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar          0 Dec  7 21:20 foo
```

The program shown in Figure 4.12 modifies the mode of these two files.

```
#include "apue.h"

int
main(void)
{
    struct stat     statbuf;

    /* turn on set-group-ID and turn off group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */

    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");

    exit(0);
}
```

**Figure 4.12** Example of chmod function

After running the program in Figure 4.12, we see that the final state of the two files is

```
$ ls -l foo bar
-rw-r--r--  1 sar          0 Dec  7 21:20 bar
-rw-rwSrw-  1 sar          0 Dec  7 21:20 foo
```

In this example, we have set the permissions of the file bar to an absolute value, regardless of the current permission bits. For the file foo, we set the permissions relative to their current state. To do this, we first call stat to obtain the current permissions and then modify them. We have explicitly turned on the set-group-ID bit and turned off the group-execute bit. Note that the ls command lists the group-execute permission as S to signify that the set-group-ID bit is set without the group-execute bit being set.

On Solaris, the `ls` command displays an `l` instead of an `S` to indicate that mandatory file and record locking has been enabled for this file. This applies only to regular files, but we'll discuss this more in Section 14.3.

Finally, note that the time and date listed by the `ls` command did not change after we ran the program in Figure 4.12. We'll see in Section 4.18 that the `chmod` function updates only the time that the i-node was last changed. By default, the `ls -l` lists the time when the contents of the file were last modified. □

The `chmod` functions automatically clear two of the permission bits under the following conditions:

- On systems, such as Solaris, that place special meaning on the sticky bit when used with regular files, if we try to set the sticky bit (`S_ISVTX`) on a regular file and do not have superuser privileges, the sticky bit in the *mode* is automatically turned off. (We describe the sticky bit in the next section.) This means that only the superuser can set the sticky bit of a regular file. The reason is to prevent malicious users from setting the sticky bit and adversely affecting system performance.

  On FreeBSD 5.2.1, Mac OS X 10.3, and Solaris 9, only the superuser can set the sticky bit on a regular file. Linux 2.4.22 places no such restriction on the setting of the sticky bit, because the bit has no meaning when applied to regular files on Linux. Although the bit also has no meaning when applied to regular files on FreeBSD and Mac OS X, these systems prevent everyone but the superuser from setting it on a regular file.

- It is possible that the group ID of a newly created file is a group that the calling process does not belong to. Recall from Section 4.6 that it's possible for the group ID of the new file to be the group ID of the parent directory. Specifically, if the group ID of the new file does not equal either the effective group ID of the process or one of the process's supplementary group IDs and if the process does not have superuser privileges, then the set-group-ID bit is automatically turned off. This prevents a user from creating a set-group-ID file owned by a group that the user doesn't belong to.

  FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9 add another security feature to try to prevent misuse of some of the protection bits. If a process that does not have superuser privileges writes to a file, the set-user-ID and set-group-ID bits are automatically turned off. If malicious users find a set-group-ID or a set-user-ID file they can write to, even though they can modify the file, they lose the special privileges of the file.

## 4.10 Sticky Bit

The `S_ISVTX` bit has an interesting history. On versions of the UNIX System that predated demand paging, this bit was known as the *sticky bit*. If it was set for an executable program file, then the first time the program was executed, a copy of the program's text was saved in the swap area when the process terminated. (The text portion of a program is the machine instructions.) This caused the program to load into memory more quickly the next time it was executed, because the swap area was handled as a contiguous file, compared to the possibly random location of data blocks

in a normal UNIX file system. The sticky bit was often set for common application programs, such as the text editor and the passes of the C compiler. Naturally, there was a limit to the number of sticky files that could be contained in the swap area before running out of swap space, but it was a useful technique. The name *sticky* came about because the text portion of the file stuck around in the swap area until the system was rebooted. Later versions of the UNIX System referred to this as the *saved-text* bit; hence, the constant `S_ISVTX`. With today's newer UNIX systems, most of which have a virtual memory system and a faster file system, the need for this technique has disappeared.

On contemporary systems, the use of the sticky bit has been extended. The Single UNIX Specification allows the sticky bit to be set for a directory. If the bit is set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory and one of the following:

- Owns the file
- Owns the directory
- Is the superuser

The directories `/tmp` and `/var/spool/uucppublic` are typical candidates for the sticky bit—they are directories in which any user can typically create files. The permissions for these two directories are often read, write, and execute for everyone (user, group, and other). But users should not be able to delete or rename files owned by others.

> The saved-text bit is not part of POSIX.1. It is an XSI extension to the basic POSIX.1 functionality defined in the Single UNIX Specification, and is supported by FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9.

> Solaris 9 places special meaning on the sticky bit if it is set on a regular file. In this case, if none of the execute bits is set, the operating system will not cache the contents of the file.

## 4.11 chown, fchown, and lchown Functions

The chown functions allow us to change the user ID of a file and the group ID of a file.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int filedes, uid_t owner, gid_t group);

int lchown(const char *pathname, uid_t owner, gid_t group);

                                All three return: 0 if OK, -1 on error
```

These three functions operate similarly unless the referenced file is a symbolic link. In that case, `lchown` changes the owners of the symbolic link itself, not the file pointed to by the symbolic link.

> The `lchown` function is an XSI extension to the POSIX.1 functionality defined in the Single UNIX Specification. As such, all UNIX System implementations are expected to provide it.

If either of the arguments *owner* or *group* is −1, the corresponding ID is left unchanged.

Historically, BSD-based systems have enforced the restriction that only the superuser can change the ownership of a file. This is to prevent users from giving away

their files to others, thereby defeating any disk space quota restrictions. System V, however, has allowed any user to change the ownership of any files they own.

> POSIX.1 allows either form of operation, depending on the value of `_POSIX_CHOWN_RESTRICTED`.
>
> With Solaris 9, this functionality is a configuration option, whose default value is to enforce the restriction. FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3 always enforce the `chown` restriction.

Recall from Section 2.6 that the `_POSIX_CHOWN_RESTRICTED` constant can optionally be defined in the header `<unistd.h>`, and can always be queried using either the `pathconf` function or the `fpathconf` function. Also recall that this option can depend on the referenced file; it can be enabled or disabled on a per file system basis. We'll use the phrase, if `_POSIX_CHOWN_RESTRICTED` is in effect, to mean if it applies to the particular file that we're talking about, regardless of whether this actual constant is defined in the header.

If `_POSIX_CHOWN_RESTRICTED` is in effect for the specified file, then

1. Only a superuser process can change the user ID of the file.
2. A nonsuperuser process can change the group ID of the file if the process owns the file (the effective user ID equals the user ID of the file), *owner* is specified as –1 or equals the user ID of the file, and *group* equals either the effective group ID of the process or one of the process's supplementary group IDs.

This means that when `_POSIX_CHOWN_RESTRICTED` is in effect, you can't change the user ID of other users' files. You can change the group ID of files that you own, but only to groups that you belong to.

If these functions are called by a process other than a superuser process, on successful return, both the set-user-ID and the set-group-ID bits are cleared.

## 4.12 File Size

The `st_size` member of the `stat` structure contains the size of the file in bytes. This field is meaningful only for regular files, directories, and symbolic links.

> Solaris also defines the file size for a pipe as the number of bytes that are available for reading from the pipe. We'll discuss pipes in Section 15.2.

For a regular file, a file size of 0 is allowed. We'll get an end-of-file indication on the first read of the file.

For a directory, the file size is usually a multiple of a number, such as 16 or 512. We talk about reading directories in Section 4.21.

For a symbolic link, the file size is the number of bytes in the filename. For example, in the following case, the file size of 7 is the length of the pathname `usr/lib`:

```
lrwxrwxrwx  1 root            7 Sep 25 07:14 lib -> usr/lib
```

(Note that symbolic links do not contain the normal C null byte at the end of the name, as the length is always specified by `st_size`.)

Most contemporary UNIX systems provide the fields `st_blksize` and `st_blocks`. The first is the preferred block size for I/O for the file, and the latter is the actual number of 512-byte blocks that are allocated. Recall from Section 3.9 that we encountered the minimum amount of time required to read a file when we used `st_blksize` for the `read` operations. The standard I/O library, which we describe in Chapter 5, also tries to read or write `st_blksize` bytes at a time, for efficiency.

> Be aware that different versions of the UNIX System use units other than 512-byte blocks for `st_blocks`. Using this value is nonportable.

### Holes in a File

In Section 3.6, we mentioned that a regular file can contain "holes." We showed an example of this in Figure 3.2. Holes are created by seeking past the current end of file and writing some data. As an example, consider the following:

```
$ ls -l core
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
$ du -s core
272     core
```

The size of the file `core` is just over 8 MB, yet the `du` command reports that the amount of disk space used by the file is 272 512-byte blocks (139,264 bytes). (The `du` command on many BSD-derived systems reports the number of 1,024-byte blocks; Solaris reports the number of 512-byte blocks.) Obviously, this file has many holes.

As we mentioned in Section 3.6, the `read` function returns data bytes of 0 for any byte positions that have not been written. If we execute the following, we can see that the normal I/O operations read up through the size of the file:

```
$ wc -c core
 8483248 core
```

> The `wc(1)` command with the `-c` option counts the number of characters (bytes) in the file.

If we make a copy of this file, using a utility such as `cat(1)`, all these holes are written out as actual data bytes of 0:

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
-rw-rw-r--  1 sar   8483248 Nov 18 12:27 core.copy
$ du -s core*
272     core
16592   core.copy
```

Here, the actual number of bytes used by the new file is 8,495,104 ($512 \times 16{,}592$). The difference between this size and the size reported by `ls` is caused by the number of blocks used by the file system to hold pointers to the actual data blocks.

Interested readers should refer to Section 4.2 of Bach [1986], Sections 7.2 and 7.3 of McKusick et al. [1996] (or Sections 8.2 and 8.3 in McKusick and Neville-Neil [2005]), and

Section 14.2 of Mauro and McDougall [2001] for additional details on the physical layout of files.

## 4.13 File Truncation

There are times when we would like to truncate a file by chopping off data at the end of the file. Emptying a file, which we can do with the O_TRUNC flag to open, is a special case of truncation.

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);

int ftruncate(int filedes, off_t length);
```

Both return: 0 if OK, −1 on error

These two functions truncate an existing file to *length* bytes. If the previous size of the file was greater than *length*, the data beyond *length* is no longer accessible. If the previous size was less than *length*, the effect is system dependent, but XSI-conforming systems will increase the file size. If the implementation does extend a file, data between the old end of file and the new end of file will read as 0 (i.e., a hole is probably created in the file).

The ftruncate function is part of POSIX.1. The truncate function is an XSI extension to the POSIX.1 functionality defined in the Single UNIX Specification.

BSD releases prior to 4.4BSD could only make a file smaller with truncate.

Solaris also includes an extension to fcntl (F_FREESP) that allows us to free any part of a file, not just a chunk at the end of the file.

We use ftruncate in the program shown in Figure 13.6 when we need to empty a file after obtaining a lock on the file.

## 4.14 File Systems

To appreciate the concept of links to a file, we need a conceptual understanding of the structure of the UNIX file system. Understanding the difference between an i-node and a directory entry that points to an i-node is also useful.

Various implementations of the UNIX file system are in use today. Solaris, for example, supports several different types of disk file systems: the traditional BSD-derived UNIX file system (called UFS), a file system (called PCFS) to read and write DOS-formatted diskettes, and a file system (called HSFS) to read CD file systems. We saw one difference between file system types in Figure 2.19. UFS is based on the Berkeley fast file system, which we describe in this section.

We can think of a disk drive being divided into one or more partitions. Each partition can contain a file system, as shown in Figure 4.13.

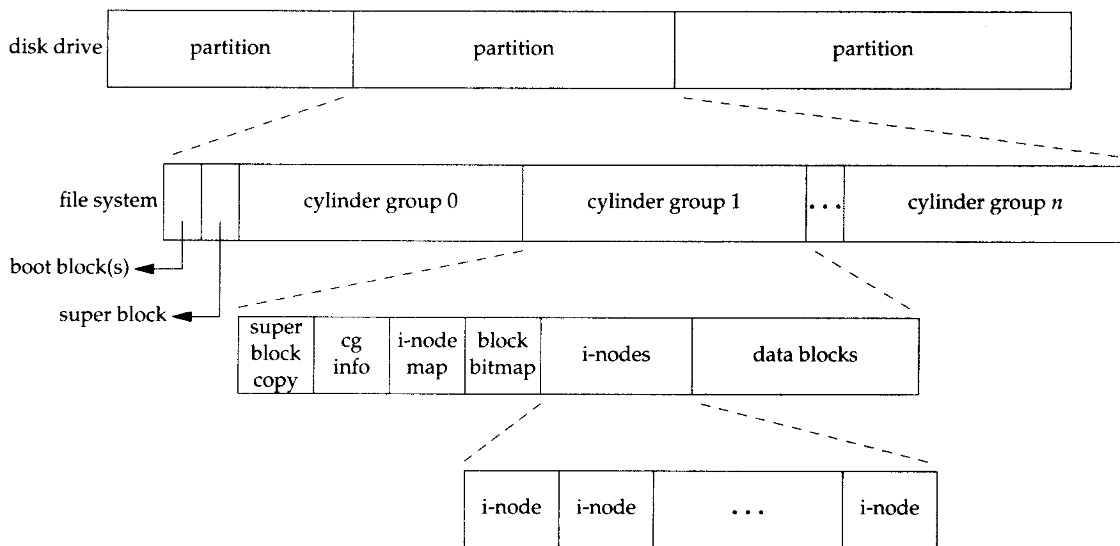


**Figure 4.13** Disk drive, partitions, and a file system

The i-nodes are fixed-length entries that contain most of the information about a file.

If we examine the i-node and data block portion of a cylinder group in more detail, we could have what is shown in Figure 4.14.

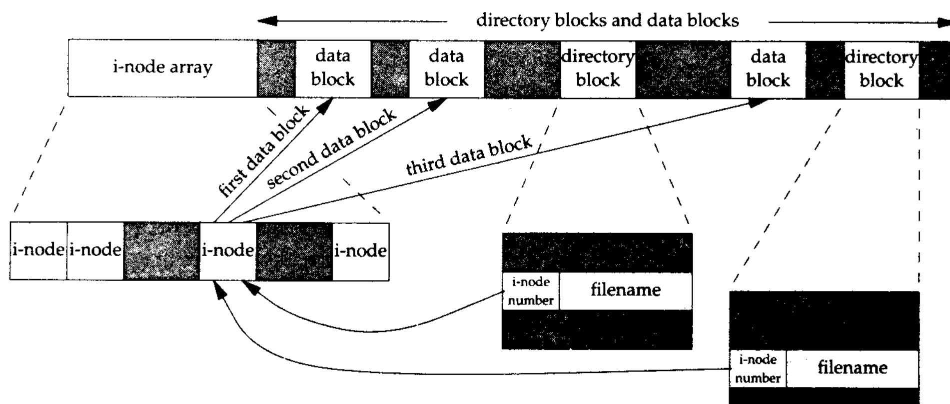


**Figure 4.14** Cylinder group's i-nodes and data blocks in more detail

Note the following points from Figure 4.14.

- We show two directory entries that point to the same i-node entry. Every i-node has a link count that contains the number of directory entries that point to the i-node. Only when the link count goes to 0 can the file be deleted (i.e., can the data blocks associated with the file be released). This is why the operation of "unlinking a file" does not always mean "deleting the blocks associated with the file." This is why the function that removes a directory entry is called `unlink`, not delete. In the `stat` structure, the link count is contained in the `st_nlink` member. Its primitive system data type is `nlink_t`. These types of links are called hard links. Recall from Section 2.5.2 that the POSIX.1 constant `LINK_MAX` specifies the maximum value for a file's link count.
- The other type of link is called a *symbolic link*. With a symbolic link, the actual contents of the file—the data blocks—store the name of the file that the symbolic link points to. In the following example, the filename in the directory entry is the three-character string `lib` and the 7 bytes of data in the file are `usr/lib`:

```
lrwxrwxrwx  1 root      7 Sep 25 07:14 lib -> usr/lib
```

  The file type in the i-node would be `S_IFLNK` so that the system knows that this is a symbolic link.
- The i-node contains all the information about the file: the file type, the file's access permission bits, the size of the file, pointers to the file's data blocks, and so on. Most of the information in the `stat` structure is obtained from the i-node. Only two items of interest are stored in the directory entry: the filename and the i-node number; the other items—the length of the filename and the length of the directory record—are not of interest to this discussion. The data type for the i-node number is `ino_t`.
- Because the i-node number in the directory entry points to an i-node in the same file system, we cannot have a directory entry point to an i-node in a different file system. This is why the `ln`(1) command (make a new directory entry that points to an existing file) can't cross file systems. We describe the `link` function in the next section.
- When renaming a file without changing file systems, the actual contents of the file need not be moved—all that needs to be done is to add a new directory entry that points to the existing i-node, and then unlink the old directory entry. The link count will remain the same. For example, to rename the file `/usr/lib/foo` to `/usr/foo`, the contents of the file `foo` need not be moved if the directories `/usr/lib` and `/usr` are on the same file system. This is how the mv(1) command usually operates.

We've talked about the concept of a link count for a regular file, but what about the link count field for a directory? Assume that we make a new directory in the working directory, as in

```
$ mkdir testdir
```

Figure 4.15 shows the result. Note that in this figure, we explicitly show the entries for dot and dot-dot.

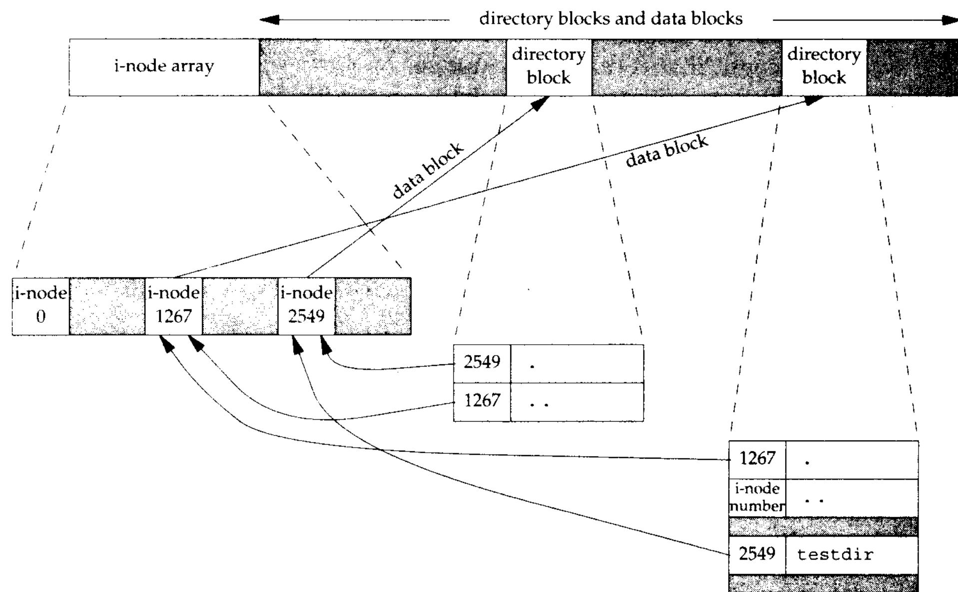


**Figure 4.15** Sample cylinder group after creating the directory `testdir`

The i-node whose number is 2549 has a type field of "directory" and a link count equal to 2. Any leaf directory (a directory that does not contain any other directories) always has a link count of 2. The value of 2 is from the directory entry that names the directory (`testdir`) and from the entry for dot in that directory. The i-node whose number is 1267 has a type field of "directory" and a link count that is greater than or equal to 3. The reason we know that the link count is greater than or equal to 3 is that minimally, it is pointed to from the directory entry that names it (which we don't show in Figure 4.15), from dot, and from dot-dot in the `testdir` directory. Note that every subdirectory in a parent directory causes the parent directory's link count to be increased by 1.

This format is similar to the classic format of the UNIX file system, which is described in detail in Chapter 4 of Bach [1986]. Refer to Chapter 7 of McKusick et al. [1996] or Chapter 8 of McKusick and Neville-Neil [2005] for additional information on the changes made with the Berkeley fast file system. See Chapter 14 of Mauro and McDougall [2001] for details on UFS, the Solaris version of the Berkeley fast file system.

## 4.15 `link`, `unlink`, `remove`, and `rename` Functions

As we saw in the previous section, any file can have multiple directory entries pointing to its i-node. The way we create a link to an existing file is with the `link` function.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
```

Returns: 0 if OK, −1 on error

This function creates a new directory entry, *newpath*, that references the existing file *existingpath*. If the *newpath* already exists, an error is returned. Only the last component of the *newpath* is created. The rest of the path must already exist.

The creation of the new directory entry and the increment of the link count must be an atomic operation. (Recall the discussion of atomic operations in Section 3.11.)

Most implementations require that both pathnames be on the same file system, although POSIX.1 allows an implementation to support linking across file systems. If an implementation supports the creation of hard links to directories, it is restricted to only the superuser. The reason is that doing this can cause loops in the file system, which most utilities that process the file system aren't capable of handling. (We show an example of a loop introduced by a symbolic link in Section 4.16.) Many file system implementations disallow hard links to directories for this reason.

To remove an existing directory entry, we call the `unlink` function.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Returns: 0 if OK, −1 on error

This function removes the directory entry and decrements the link count of the file referenced by *pathname*. If there are other links to the file, the data in the file is still accessible through the other links. The file is not changed if an error occurs.

We've mentioned before that to unlink a file, we must have write permission and execute permission in the directory containing the directory entry, as it is the directory entry that we will be removing. Also, we mentioned in Section 4.10 that if the sticky bit is set in this directory we must have write permission for the directory and one of the following:

- Own the file
- Own the directory
- Have superuser privileges

Only when the link count reaches 0 can the contents of the file be deleted. One other condition prevents the contents of a file from being deleted: as long as some process has the file open, its contents will not be deleted. When a file is closed, the kernel first checks the count of the number of processes that have the file open. If this count has reached 0, the kernel then checks the link count; if it is 0, the file's contents are deleted.

### Example

The program shown in Figure 4.16 opens a file and then unlinks it. The program then goes to sleep for 15 seconds before terminating.

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

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

**Figure 4.16** Open a file and then `unlink` it

Running this program gives us

```
$ ls -l tempfile                  look at how big the file is
-rw-r-----   1 sar      413265408 Jan 21 07:14 tempfile
$ df /home                        check how much free space is available
Filesystem    1K-blocks      Used  Available  Use%  Mounted on
/dev/hda4      11021440   1956332    9065108   18%  /home
$ ./a.out &                       run the program in Figure 4.16 in the background
1364                              the shell prints its process ID
$ file unlinked                   the file is unlinked
ls -l tempfile                    see if the filename is still there
ls: tempfile: No such file or directory    the directory entry is gone
$ df /home                        see if the space is available yet
Filesystem    1K-blocks      Used  Available  Use%  Mounted on
/dev/hda4      11021440   1956332    9065108   18%  /home
$ done                            the program is done, all open files are closed
df /home                          now the disk space should be available
Filesystem    1K-blocks      Used  Available  Use%  Mounted on
/dev/hda4      11021440   1552352    9469088   15%  /home
                                  now the 394.1 MB of disk space are available
```

□

This property of `unlink` is often used by a program to ensure that a temporary file it creates won't be left around in case the program crashes. The process creates a file using either `open` or `creat` and then immediately calls `unlink`. The file is not deleted, however, because it is still open. Only when the process either closes the file or terminates, which causes the kernel to close all its open files, is the file deleted.

If *pathname* is a symbolic link, `unlink` removes the symbolic link, not the file referenced by the link. There is no function to remove the file referenced by a symbolic link given the name of the link.

The superuser can call unlink with *pathname* specifying a directory, but the function rmdir should be used instead to unlink a directory. We describe the rmdir function in Section 4.20.

We can also unlink a file or a directory with the remove function. For a file, remove is identical to unlink. For a directory, remove is identical to rmdir.

```
#include <stdio.h>

int remove(const char *pathname);
```

Returns: 0 if OK, −1 on error

ISO C specifies the remove function to delete a file. The name was changed from the historical UNIX name of unlink because most non-UNIX systems that implement the C standard didn't support the concept of links to a file at the time.

A file or a directory is renamed with the rename function.

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

Returns: 0 if OK, −1 on error

This function is defined by ISO C for files. (The C standard doesn't deal with directories.) POSIX.1 expanded the definition to include directories and symbolic links.

There are several conditions to describe, depending on whether *oldname* refers to a file, a directory, or a symbolic link. We must also describe what happens if *newname* already exists.

1. If *oldname* specifies a file that is not a directory, then we are renaming a file or a symbolic link. In this case, if *newname* exists, it cannot refer to a directory. If *newname* exists and is not a directory, it is removed, and *oldname* is renamed to *newname*. We must have write permission for the directory containing *oldname* and for the directory containing *newname*, since we are changing both directories.

2. If *oldname* specifies a directory, then we are renaming a directory. If *newname* exists, it must refer to a directory, and that directory must be empty. (When we say that a directory is empty, we mean that the only entries in the directory are dot and dot-dot.) If *newname* exists and is an empty directory, it is removed, and *oldname* is renamed to *newname*. Additionally, when we're renaming a directory, *newname* cannot contain a path prefix that names *oldname*. For example, we can't rename /usr/foo to /usr/foo/testdir, since the old name (/usr/foo) is a path prefix of the new name and cannot be removed.

3. If either *oldname* or *newname* refers to a symbolic link, then the link itself is processed, not the file to which it resolves.

4. As a special case, if the *oldname* and *newname* refer to the same file, the function returns successfully without changing anything.

If *newname* already exists, we need permissions as if we were deleting it. Also, because we're removing the directory entry for *oldname* and possibly creating a directory entry for *newname*, we need write permission and execute permission in the directory containing *oldname* and in the directory containing *newname*.

## 4.16 Symbolic Links

A symbolic link is an indirect pointer to a file, unlike the hard links from the previous section, which pointed directly to the i-node of the file. Symbolic links were introduced to get around the limitations of hard links.

- Hard links normally require that the link and the file reside in the same file system
- Only the superuser can create a hard link to a directory

There are no file system limitations on a symbolic link and what it points to, and anyone can create a symbolic link to a directory. Symbolic links are typically used to move a file or an entire directory hierarchy to another location on a system.

Symbolic links were introduced with 4.2BSD and subsequently supported by SVR4.

When using functions that refer to a file by name, we always need to know whether the function follows a symbolic link. If the function follows a symbolic link, a pathname argument to the function refers to the file pointed to by the symbolic link. Otherwise, a pathname argument refers to the link itself, not the file pointed to by the link. Figure 4.17 summarizes whether the functions described in this chapter follow a symbolic link. The functions `mkdir`, `mkfifo`, `mknod`, and `rmdir` are not in this figure, as they return an error when the pathname is a symbolic link. Also, the functions that take a file descriptor argument, such as `fstat` and `fchmod`, are not listed, as the handling of a symbolic link is done by the function that returns the file descriptor (usually `open`). Whether or not `chown` follows a symbolic link depends on the implementation.

In older versions of Linux (those before version 2.1.81), `chown` didn't follow symbolic links. From version 2.1.81 onward, `chown` follows symbolic links. With FreeBSD 5.2.1 and Mac OS X 10.3, `chown` follows symbolic links. (Prior to 4.4BSD, `chown` didn't follow symbolic links, but this was changed in 4.4BSD.) In Solaris 9, `chown` also follows symbolic links. All of these platforms provide implementations of `lchown` to change the ownership of symbolic links themselves.

One exception to Figure 4.17 is when the `open` function is called with both `O_CREAT` and `O_EXCL` set. In this case, if the pathname refers to a symbolic link, `open` will fail with `errno` set to `EEXIST`. This behavior is intended to close a security hole so that privileged processes can't be fooled into writing to the wrong files.

| Function | Does not follow symbolic link | Follows symbolic link |
|---|---|---|
| access | | • |
| chdir | | • |
| chmod | | • |
| chown | • | • |
| creat | | • |
| exec | | • |
| lchown | • | |
| link | | • |
| lstat | • | |
| open | | • |
| opendir | | • |
| pathconf | | • |
| readlink | • | |
| remove | • | |
| rename | • | |
| stat | | • |
| truncate | | • |
| unlink | • | |

**Figure 4.17** Treatment of symbolic links by various functions

## Example

It is possible to introduce loops into the file system by using symbolic links. Most functions that look up a pathname return an errno of ELOOP when this occurs. Consider the following commands:

```
$ mkdir foo                          make a new directory
$ touch foo/a                        create a 0-length file
$ ln -s ../foo foo/testdir           create a symbolic link
$ ls -l foo
total 0
-rw-r-----   1 sar               0 Jan 22 00:16 a
lrwxrwxrwx   1 sar               6 Jan 22 00:16 testdir -> ../foo
```

This creates a directory foo that contains the file a and a symbolic link that points to foo. We show this arrangement in Figure 4.18, drawing a directory as a circle and a file as a square. If we write a simple program that uses the standard function ftw(3) on Solaris to descend through a file hierarchy, printing each pathname encountered, the output is

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
    (many more lines until we encounter an ELOOP error)
```

ID, changing the number of links, and so on. Because all the information in the i-node is stored separately from the actual contents of the file, we need the changed-status time, in addition to the modification time.

Note that the system does not maintain the last-access time for an i-node. This is why the functions `access` and `stat`, for example, don't change any of the three times.

The access time is often used by system administrators to delete files that have not been accessed for a certain amount of time. The classic example is the removal of files named `a.out` or `core` that haven't been accessed in the past week. The `find(1)` command is often used for this type of operation.

The modification time and the changed-status time can be used to archive only those files that have had their contents modified or their i-node modified.

The `ls` command displays or sorts only on one of the three time values. By default, when invoked with either the `-l` or the `-t` option, it uses the modification time of a file. The `-u` option causes it to use the access time, and the `-c` option causes it to use the changed-status time.

Figure 4.20 summarizes the effects of the various functions that we've described on these three times. Recall from Section 4.14 that a directory is simply a file containing directory entries: filenames and associated i-node numbers. Adding, deleting, or modifying these directory entries can affect the three times associated with that directory. This is why Figure 4.20 contains one column for the three times associated with the file or directory and another column for the three times associated with the parent directory of the referenced file or directory. For example, creating a new file affects the directory that contains the new file, and it affects the i-node for the new file. Reading or writing a file, however, affects only the i-node of the file and has no effect on the directory. (The `mkdir` and `rmdir` functions are covered in Section 4.20. The `utime` function is covered in the next section. The six `exec` functions are described in Section 8.10. We describe the `mkfifo` and `pipe` functions in Chapter 15.)

## 4.19 `utime` Function

The access time and the modification time of a file can be changed with the `utime` function.

```
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *times);
```

Returns: 0 if OK, −1 on error

The structure used by this function is

```
struct utimbuf {
  time_t  actime;    /* access time */
  time_t  modtime;   /* modification time */
}
```

| Function | Referenced file or directory | | | Parent directory of referenced file or directory | | | Section | Note |
|---|---|---|---|---|---|---|---|---|
| | a | m | c | a | m | c | | |
| chmod, fchmod | | | • | | | | 4.9 | |
| chown, fchown | | | • | | | | 4.11 | |
| creat | • | • | • | | • | • | 3.4 | O_CREAT new file |
| creat | | • | • | | | | 3.4 | O_TRUNC existing file |
| exec | • | | | | | | 8.10 | |
| lchown | | | • | | | | 4.11 | |
| link | | | • | | • | • | 4.15 | parent of second argument |
| mkdir | • | • | • | | • | • | 4.20 | |
| mkfifo | • | • | • | | • | • | 15.5 | |
| open | • | • | • | | • | • | 3.3 | O_CREAT new file |
| open | | • | • | | | | 3.3 | O_TRUNC existing file |
| pipe | • | • | • | | | | 15.2 | |
| read | • | | | | | | 3.7 | |
| remove | | | • | | • | • | 4.15 | remove file = unlink |
| remove | | | | | • | • | 4.15 | remove directory = rmdir |
| rename | | | • | | • | • | 4.15 | for both arguments |
| rmdir | | | | | • | • | 4.20 | |
| truncate, ftruncate | | • | • | | | | 4.13 | |
| unlink | | | • | | • | • | 4.15 | |
| utime | • | • | • | | | | 4.19 | |
| write | | • | • | | | | 3.8 | |

**Figure 4.20** Effect of various functions on the access, modification, and changed-status times

The two time values in the structure are calendar times, which count seconds since the Epoch, as described in Section 1.10.

The operation of this function, and the privileges required to execute it, depend on whether the *times* argument is NULL.

- If *times* is a null pointer, the access time and the modification time are both set to the current time. To do this, either the effective user ID of the process must equal the owner ID of the file, or the process must have write permission for the file.
- If *times* is a non-null pointer, the access time and the modification time are set to the values in the structure pointed to by *times*. For this case, the effective user ID of the process must equal the owner ID of the file, or the process must be a superuser process. Merely having write permission for the file is not adequate.

Note that we are unable to specify a value for the changed-status time, st_ctime—the time the i-node was last changed—as this field is automatically updated when the utime function is called.

On some versions of the UNIX System, the touch(1) command uses this function. Also, the standard archive programs, tar(1) and cpio(1), optionally call utime to set the times for a file to the time values saved when the file was archived.

### Example

The program shown in Figure 4.21 truncates files to zero length using the `O_TRUNC` option of the open function, but does not change their access time or modification time. To do this, the program first obtains the times with the `stat` function, truncates the file, and then resets the times with the `utime` function.

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

int
main(int argc, char *argv[])
{
    int             i, fd;
    struct stat     statbuf;
    struct utimbuf  timebuf;

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) {  /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;
        }
        close(fd);
        timebuf.actime  = statbuf.st_atime;
        timebuf.modtime = statbuf.st_mtime;
        if (utime(argv[i], &timebuf) < 0) {     /* reset times */
            err_ret("%s: utime error", argv[i]);
            continue;
        }
    }
    exit(0);
}
```

**Figure 4.21** Example of `utime` function

We can demonstrate the program in Figure 4.21 with the following script:

```
$ ls -l changemod times          look at sizes and last-modification times
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ ls -lu changemod times         look at last-access times
-rwxrwxr-x  1 sar   15019 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar   16172 Nov 19 20:05 times
$ date                           print today's date
Thu Jan 22 06:55:17 EST 2004
$ ./a.out changemod times        run the program in Figure 4.21
$ ls -l changemod times          and check the results
```

```
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lu changemod times           check the last-access times also
-rwxrwxr-x  1 sar       0 Nov 18 18:53 changemod
-rwxrwxr-x  1 sar       0 Nov 19 20:05 times
$ ls -lc changemod times           and the changed-status times
-rwxrwxr-x  1 sar       0 Jan 22 06:55 changemod
-rwxrwxr-x  1 sar       0 Jan 22 06:55 times
```

As we expect, the last-modification times and the last-access times are not changed. The changed-status times, however, are changed to the time that the program was run. □

## 4.20 mkdir and rmdir Functions

Directories are created with the mkdir function and deleted with the rmdir function.

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

int mkdir(const char *pathname, mode_t mode);

                                        Returns: 0 if OK, -1 on error
```

This function creates a new, empty directory. The entries for dot and dot-dot are automatically created. The specified file access permissions, *mode*, are modified by the file mode creation mask of the process.

A common mistake is to specify the same *mode* as for a file: read and write permissions only. But for a directory, we normally want at least one of the execute bits enabled, to allow access to filenames within the directory. (See Exercise 4.16.)

The user ID and group ID of the new directory are established according to the rules we described in Section 4.6.

> Solaris 9 and Linux 2.4.22 also have the new directory inherit the set-group-ID bit from the parent directory. This is so that files created in the new directory will inherit the group ID of that directory. With Linux, the file system implementation determines whether this is supported. For example, the ext2 and ext3 file systems allow this behavior to be controlled by an option to the mount(1) command. With the Linux implementation of the UFS file system, however, the behavior is not selectable; it inherits the set-group-ID bit to mimic the historical BSD implementation, where the group ID of a directory is inherited from the parent directory.

> BSD-based implementations don't propagate the set-group-ID bit; they simply inherit the group ID as a matter of policy. Because FreeBSD 5.2.1 and Mac OS X 10.3 are based on 4.4BSD, they do not require this inheriting of the set-group-ID bit. On these platforms, newly created files and directories always inherit the group ID of the parent directory, regardless of the set-group-ID bit.

> Earlier versions of the UNIX System did not have the mkdir function. It was introduced with 4.2BSD and SVR3. In the earlier versions, a process had to call the mknod function to create a new directory. But use of the mknod function was restricted to superuser processes. To circumvent this, the normal command that created a directory, mkdir(1), had to be owned by root with the set-user-ID bit on. To create a directory from a process, the mkdir(1) command had to be invoked with the system(3) function.

An empty directory is deleted with the `rmdir` function. Recall that an empty directory is one that contains entries only for dot and dot-dot.

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Returns: 0 if OK, −1 on error

If the link count of the directory becomes 0 with this call, and if no other process has the directory open, then the space occupied by the directory is freed. If one or more processes have the directory open when the link count reaches 0, the last link is removed and the dot and dot-dot entries are removed before this function returns. Additionally, no new files can be created in the directory. The directory is not freed, however, until the last process closes it. (Even though some other process has the directory open, it can't be doing much in the directory, as the directory had to be empty for the `rmdir` function to succeed.)

## 4.21 Reading Directories

Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, to preserve file system sanity. Recall from Section 4.5 that the write permission bits and execute permission bits for a directory determine if we can create new files in the directory and remove files from the directory—they don't specify if we can write to the directory itself.

The actual format of a directory depends on the UNIX System implementation and the design of the file system. Earlier systems, such as Version 7, had a simple structure: each directory entry was 16 bytes, with 14 bytes for the filename and 2 bytes for the i-node number. When longer filenames were added to 4.2BSD, each entry became variable length, which means that any program that reads a directory is now system dependent. To simplify this, a set of directory routines were developed and are part of POSIX.1. Many implementations prevent applications from using the `read` function to access the contents of directories, thereby further isolating applications from the implementation-specific details of directory formats.

```
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Returns: pointer if OK, NULL on error

```
struct dirent *readdir(DIR *dp);
```

Returns: pointer if OK, NULL at end of directory or error

```
void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

Returns: 0 if OK, −1 on error

```
long telldir(DIR *dp);
```

Returns: current location in directory associated with *dp*

```
void seekdir(DIR *dp, long loc);
```

The telldir and seekdir functions are not part of the base POSIX.1 standard. They are XSI extensions in the Single UNIX Specifications, so all conforming UNIX System implementations are expected to provide them.

Recall our use of several of these functions in the program shown in Figure 1.3, our bare-bones implementation of the ls command.

The dirent structure defined in the file <dirent.h> is implementation dependent. Implementations define the structure to contain at least the following two members:

```
struct dirent {
  ino_t  d_ino;                  /* i-node number */
  char   d_name[NAME_MAX + 1];   /* null-terminated filename */
}
```

> The d_ino entry is not defined by POSIX.1, since it's an implementation feature, but it is defined in the XSI extension to POSIX.1. POSIX.1 defines only the d_name entry in this structure.

Note that NAME_MAX is not a defined constant with Solaris—its value depends on the file system in which the directory resides, and its value is usually obtained from the fpathconf function. A common value for NAME_MAX is 255. (Recall Figure 2.14.) Since the filename is null terminated, however, it doesn't matter how the array d_name is defined in the header, because the array size doesn't indicate the length of the filename.

The DIR structure is an internal structure used by these six functions to maintain information about the directory being read. The purpose of the DIR structure is similar to that of the FILE structure maintained by the standard I/O library, which we describe in Chapter 5.

The pointer to a DIR structure that is returned by opendir is then used with the other five functions. The opendir function initializes things so that the first readdir reads the first entry in the directory. The ordering of entries within the directory is implementation dependent and is usually not alphabetical.

### Example

We'll use these directory routines to write a program that traverses a file hierarchy. The goal is to produce the count of the various types of files that we show in Figure 4.4. The program shown in Figure 4.22 takes a single argument—the starting pathname—and recursively descends the hierarchy from that point. Solaris provides a function, ftw(3), that performs the actual traversal of the hierarchy, calling a user-defined function for each file. The problem with this function is that it calls the stat function for each file, which causes the program to follow symbolic links. For example, if we start at the root and have a symbolic link named /lib that points to /usr/lib, all the files in the directory /usr/lib are counted twice. To correct this, Solaris provides an additional function, nftw(3), with an option that stops it from following symbolic links. Although we could use nftw, we'll write our own simple file walker to show the use of the directory routines.

In the Single UNIX Specification, both `ftw` and `nftw` are included in the XSI extensions to the base POSIX.1 specification. Implementations are included in Solaris 9 and Linux 2.4.22. BSD-based systems have a different function, `fts`(3), that provides similar functionality. It is available in FreeBSD 5.2.1, Mac OS X 10.3, and Linux 2.4.22.

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* function type that is called for each filename */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc   myfunc;
static int      myftw(char *, Myfunc *);
static int      dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

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

    if (argc != 2)
        err_quit("usage:  ftw  <starting-pathname>");

    ret = myftw(argv[1], myfunc);       /* does it all */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1;       /* avoid divide by 0; print 0 for all counts */
    printf("regular files  = %7ld, %5.2f %%\n", nreg,
      nreg*100.0/ntot);
    printf("directories    = %7ld, %5.2f %%\n", ndir,
      ndir*100.0/ntot);
    printf("block special  = %7ld, %5.2f %%\n", nblk,
      nblk*100.0/ntot);
    printf("char special   = %7ld, %5.2f %%\n", nchr,
      nchr*100.0/ntot);
    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo,
      nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,
      nslink*100.0/ntot);
    printf("sockets        = %7ld, %5.2f %%\n", nsock,
      nsock*100.0/ntot);

    exit(ret);
}

/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */
#define FTW_F   1       /* file other than directory */
```

```
#define FTW_D   2        /* directory */
#define FTW_DNR 3        /* directory that can't be read */
#define FTW_NS  4        /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    int len;
    fullpath = path_alloc(&len);    /* malloc's for PATH_MAX+1 bytes */
                                    /* (Figure 2.15) */
    strncpy(fullpath, pathname, len);   /* protect against */
    fullpath[len-1] = 0;                /* buffer overrun */

    return(dopath(func));
}

/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return.  For a directory, we call ourself
 * recursively for each name in the directory.
 */
static int                  /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat     statbuf;
    struct dirent   *dirp;
    DIR             *dp;
    int             ret;
    char            *ptr;

    if (lstat(fullpath, &statbuf) < 0)  /* stat error */
        return(func(fullpath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0)  /* not a directory */
        return(func(fullpath, &statbuf, FTW_F));

    /*
     * It's a directory.  First call func() for the directory,
     * then process each filename in the directory.
     */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    ptr = fullpath + strlen(fullpath);  /* point to end of fullpath */
    *ptr++ = '/';
    *ptr = 0;

    if ((dp = opendir(fullpath)) == NULL)   /* can't read directory */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0  ||
```

```
            strcmp(dirp->d_name, "..") == 0)
                continue;       /* ignore dot and dot-dot */

        strcpy(ptr, dirp->d_name);  /* append name after slash */

        if ((ret = dopath(func)) != 0)      /* recursive */
            break;  /* time to leave */
    }
    ptr[-1] = 0;    /* erase everything from slash onwards */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);

    return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;     break;
        case S_IFBLK:   nblk++;     break;
        case S_IFCHR:   nchr++;     break;
        case S_IFIFO:   nfifo++;    break;
        case S_IFLNK:   nslink++;   break;
        case S_IFSOCK:  nsock++;    break;
        case S_IFDIR:
            err_dump("for S_IFDIR for %s", pathname);
                    /* directories should have type = FTW_D */
        }
        break;

    case FTW_D:
        ndir++;
        break;

    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
        break;

    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;

    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }

    return(0);
}
```

**Figure 4.22** Recursively descend a directory hierarchy, counting file types

We have provided more generality in this program than needed. This was done to illustrate the ftw function. For example, the function myfunc always returns 0, even though the function that calls it is prepared to handle a nonzero return. □

For additional information on descending through a file system and the use of this technique in many standard UNIX System commands—find, ls, tar, and so on—refer to Fowler, Korn, and Vo [1989].

## 4.22 chdir, fchdir, and getcwd Functions

Every process has a current working directory. This directory is where the search for all relative pathnames starts (all pathnames that do not begin with a slash). When a user logs in to a UNIX system, the current working directory normally starts at the directory specified by the sixth field in the /etc/passwd file—the user's home directory. The current working directory is an attribute of a process; the home directory is an attribute of a login name.

We can change the current working directory of the calling process by calling the chdir or fchdir functions.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int filedes);
```

Both return: 0 if OK, −1 on error

We can specify the new current working directory either as a *pathname* or through an open file descriptor.

The fchdir function is not part of the base POSIX.1 specification. It is an XSI extension in the Single UNIX Specification. All four platforms discussed in this book support fchdir.

### Example

Because it is an attribute of a process, the current working directory cannot affect processes that invoke the process that executes the chdir. (We describe the relationship between processes in more detail in Chapter 8.) This means that the program in Figure 4.23 doesn't do what we might expect.

```
#include "apue.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("chdir failed");
    printf("chdir to /tmp succeeded\n");
    exit(0);
}
```

**Figure 4.23** Example of chdir function

If we compile it and call the executable mycd, we get the following:

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

The current working directory for the shell that executed the mycd program didn't change. This is a side effect of the way that the shell executes programs. Each program is run in a separate process, so the current working directory of the shell is unaffected by the call to chdir in the program. For this reason, the chdir function has to be called directly from the shell, so the cd command is built into the shells. □

Because the kernel must maintain knowledge of the current working directory, we should be able to fetch its current value. Unfortunately, the kernel doesn't maintain the full pathname of the directory. Instead, the kernel keeps information about the directory, such as a pointer to the directory's v-node.

What we need is a function that starts at the current working directory (dot) and works its way up the directory hierarchy, using dot-dot to move up one level. At each directory, the function reads the directory entries until it finds the name that corresponds to the i-node of the directory that it just came from. ,Repeating this procedure until the root is encountered yields the entire absolute pathname of the current working directory. Fortunately, a function is already provided for us that does this task.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);

                                Returns: buf if OK, NULL on error
```

We must pass to this function the address of a buffer, *buf*, and its *size* (in bytes). The buffer must be large enough to accommodate the absolute pathname plus a terminating null byte, or an error is returned. (Recall the discussion of allocating space for a maximum-sized pathname in Section 2.5.5.)

Some older implementations of getcwd allow the first argument *buf* to be NULL. In this case, the function calls malloc to allocate *size* number of bytes dynamically. This is not part of POSIX.1 or the Single UNIX Specification and should be avoided.

## Example

The program in Figure 4.24 changes to a specific directory and then calls getcwd to print the working directory. If we run the program, we get

```
$ ./a.out
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx  1 root  12 Jan 31 07:57 /usr/spool -> ../var/spool
```

```
#include "apue.h"

int
main(void)
{
    char    *ptr;
    int     size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size);    /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

**Figure 4.24** Example of getcwd function

Note that chdir follows the symbolic link—as we expect it to, from Figure 4.17—but when it goes up the directory tree, getcwd has no idea when it hits the /var/spool directory that it is pointed to by the symbolic link /usr/spool. This is a characteristic of symbolic links. □

The getcwd function is useful when we have an application that needs to return to the location in the file system where it started out. We can save the starting location by calling getcwd before we change our working directory. After we complete our processing, we can pass the pathname obtained from getcwd to chdir to return to our starting location in the file system.

The fchdir function provides us with an easy way to accomplish this task. Instead of calling getcwd, we can open the current directory and save the file descriptor before we change to a different location in the file system. When we want to return to where we started, we can simply pass the file descriptor to fchdir.

## 4.23 Device Special Files

The two fields st_dev and st_rdev are often confused. We'll need to use these fields in Section 18.9 when we write the ttyname function. The rules are simple.

- Every file system is known by its major and minor device numbers, which are encoded in the primitive system data type dev_t. The major number identifies the device driver and sometimes encodes which peripheral board to communicate with; the minor number identifies the specific subdevice. Recall from Figure 4.13 that a disk drive often contains several file systems. Each file system on the same disk drive would usually have the same major number, but a different minor number.

- We can usually access the major and minor device numbers through two macros defined by most implementations: `major` and `minor`. This means that we don't care how the two numbers are stored in a `dev_t` object.

  Early systems stored the device number in a 16-bit integer, with 8 bits for the major number and 8 bits for the minor number. FreeBSD 5.2.1 and Mac OS X 10.3 use a 32-bit integer, with 8 bits for the major number and 24 bits for the minor number. On 32-bit systems, Solaris 9 uses a 32-bit integer for `dev_t`, with 14 bits designated as the major number and 18 bits designated as the minor number. On 64-bit systems, Solaris 9 represents `dev_t` as a 64-bit integer, with 32 bits for each number. On Linux 2.4.22, although `dev_t` is a 64-bit integer, currently the major and minor numbers are each only 8 bits.

  POSIX.1 states that the `dev_t` type exists, but doesn't define what it contains or how to get at its contents. The macros `major` and `minor` are defined by most implementations. Which header they are defined in depends on the system. They can be found in `<sys/types.h>` on BSD-based systems. Solaris defines them in `<sys/mkdev.h>`. Linux defines these macros in `<sys/sysmacros.h>`, which is included by `<sys/types.h>`.

- The `st_dev` value for every filename on a system is the device number of the file system containing that filename and its corresponding i-node.
- Only character special files and block special files have an `st_rdev` value. This value contains the device number for the actual device.

### Example

The program in Figure 4.25 prints the device number for each command-line argument. Additionally, if the argument refers to a character special file or a block special file, the `st_rdev` value for the special file is also printed.

```
#include "apue.h"
#ifdef SOLARIS
#include <sys/mkdev.h>
#endif

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("stat error");
            continue;
        }

        printf("dev = %d/%d", major(buf.st_dev),  minor(buf.st_dev));
```

```
        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                    (S_ISCHR(buf.st_mode)) ? "character" : "block",
                    major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }

    exit(0);
}
```

**Figure 4.25** Print st_dev and st_rdev values

Running this program gives us the following output:

```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 3/3
/home/sar: dev = 3/4
/dev/tty0: dev = 0/7 (character) rdev = 4/0
/dev/tty1: dev = 0/7 (character) rdev = 4/1
$ mount                         which directories are mounted on which devices?
/dev/hda3 on / type ext2 (rw,noatime)
/dev/hda4 on /home type ext2 (rw,noatime)
$ ls -lL /dev/tty[01] /dev/hda[34]
brw-------   1 root       3,    3 Dec 31  1969 /dev/hda3
brw-------   1 root       3,    4 Dec 31  1969 /dev/hda4
crw-------   1 root       4,    0 Dec 31  1969 /dev/tty0
crw-------   1 root       4,    1 Jan 18 15:36 /dev/tty1
```

The first two arguments to the program are directories (/ and /home/sar), and the next two are the device names /dev/tty[01]. (We use the shell's regular expression language to shorten the amount of typing we need to do. The shell will expand the string /dev/tty[01] to /dev/tty0 /dev/tty1.)

We expect the devices to be character special files. The output from the program shows that the root directory has a different device number than does the /home/sar directory. This indicates that they are on different file systems. Running the mount(1) command verifies this.

We then use ls to look at the two disk devices reported by mount and the two terminal devices. The two disk devices are block special files, and the two terminal devices are character special files. (Normally, the only types of devices that are block special files are those that can contain random-access file systems: disk drives, floppy disk drives, and CD-ROMs, for example. Some older versions of the UNIX System supported magnetic tapes for file systems, but this was never widely used.)

Note that the filenames and i-nodes for the two terminal devices (st_dev) are on device 0/7—the devfs pseudo file system, which implements the /dev—but that their actual device numbers are 4/0 and 4/1. □

## 4.24 Summary of File Access Permission Bits

We've covered all the file access permission bits, some of which serve multiple purposes. Figure 4.26 summarizes all these permission bits and their interpretation when applied to a regular file and a directory.

| Constant | Description | Effect on regular file | Effect on directory |
|---|---|---|---|
| S_ISUID | set-user-ID | set effective user ID on execution | (not used) |
| S_ISGID | set-group-ID | if group-execute set then set effective group ID on execution; otherwise enable mandatory record locking (if supported) | set group ID of new files created in directory to group ID of directory |
| S_ISVTX | sticky bit | control caching of file contents (if supported) | restrict removal and renaming of files in directory |
| S_IRUSR | user-read | user permission to read file | user permission to read directory entries |
| S_IWUSR | user-write | user permission to write file | user permission to remove and create files in directory |
| S_IXUSR | user-execute | user permission to execute file | user permission to search for given pathname in directory |
| S_IRGRP | group-read | group permission to read file | group permission to read directory entries |
| S_IWGRP | group-write | group permission to write file | group permission to remove and create files in directory |
| S_IXGRP | group-execute | group permission to execute file | group permission to search for given pathname in directory |
| S_IROTH | other-read | other permission to read file | other permission to read directory entries |
| S_IWOTH | other-write | other permission to write file | other permission to remove and create files in directory |
| S_IXOTH | other-execute | other permission to execute file | other permission to search for given pathname in directory |

**Figure 4.26** Summary of file access permission bits

The final nine constants can also be grouped into threes, since

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

## 4.25 Summary

This chapter has centered around the `stat` function. We've gone through each member in the `stat` structure in detail. This in turn led us to examine all the attributes of UNIX files. A thorough understanding of all the properties of a file and all the functions that operate on files is essential to UNIX programming.

## Exercises

4.1 Modify the program in Figure 4.3 to use `stat` instead of `lstat`. What changes if one of the command-line arguments is a symbolic link?

4.2 What happens if the file mode creation mask is set to 777 (octal)? Verify the results using your shell's `umask` command.

4.3 Verify that turning off user-read permission for a file that you own denies your access to the file.

4.4 Run the program in Figure 4.9 *after* creating the files `foo` and `bar`. What happens?

4.5 In Section 4.12, we said that a file size of 0 is valid for a regular file. We also said that the `st_size` field is defined for directories and symbolic links. Should we ever see a file size of 0 for a directory or a symbolic link?

4.6 Write a utility like cp(1) that copies a file containing holes, without writing the bytes of 0 to the output file.

4.7 Note in output from the `ls` command in Section 4.12 that the files `core` and `core.copy` have different access permissions. If the `umask` value didn't change between the creation of the two files, explain how the difference could have occurred.

4.8 When running the program in Figure 4.16, we check the available disk space with the `df(1)` command. Why didn't we use the du(1) command?

4.9 In Figure 4.20, we show the `unlink` function as modifying the changed-status time of the file itself. How can this happen?

4.10 In Section 4.21, how does the system's limit on the number of open files affect the `myftw` function?

4.11 In Section 4.21, our version of `ftw` never changes its directory. Modify this routine so that each time it encounters a directory, it does a `chdir` to that directory, allowing it to use the filename and not the pathname for each call to `lstat`. When all the entries in a directory have been processed, execute `chdir("..")`. Compare the time used by this version and the version in the text.

4.12 Each process also has a root directory that is used for resolution of absolute pathnames. This root directory can be changed with the `chroot` function. Look up the description for this function in your manuals. When might this function be useful?

4.13 How can you set only one of the two time values with the `utime` function?

4.14 Some versions of the `finger(1)` command output "New mail received ..." and "unread since ..." where ... are the corresponding times and dates. How can the program determine these two times and dates?

4.15 Examine the archive formats by the `cpio(1)` and `tar(1)` commands. (These descriptions are usually found in Section 5 of the *UNIX Programmer's Manual*.) How many of the three possible time values are saved for each file? When a file is restored, what value do you think the access time is set to, and why?

4.16 Does the UNIX System have a fundamental limitation on the depth of a directory tree? To find out, write a program that creates a directory and then changes to that directory, in a loop. Make certain that the length of the absolute pathname of the leaf of this directory is

greater than your system's PATH_MAX limit. Can you call getcwd to fetch the directory's pathname? How do the standard UNIX System tools deal with this long pathname? Can you archive the directory using either tar or cpio?

**4.17** In Section 3.16, we described the /dev/fd feature. For any user to be able to access these files, their permissions must be rw-rw-rw-. Some programs that create an output file delete the file first, in case it already exists, ignoring the return code:

```
unlink(path);
if ((fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

What happens if path is /dev/fd/1?

# 5

# *Standard I/O Library*

## 5.1 Introduction

In this chapter, we describe the standard I/O library. This library is specified by the ISO C standard because it has been implemented on many operating systems other than the UNIX System. Additional interfaces are defined as extensions to the ISO C standard by the Single UNIX Specification.

The standard I/O library handles such details as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size (as in Section 3.9). This makes the library easy to use, but at the same time introduces another set of problems if we're not cognizant of what's going on.

> The standard I/O library was written by Dennis Ritchie around 1975. It was a major revision of the Portable I/O library written by Mike Lesk. Surprisingly, little has changed in the standard I/O library after 30 years.

## 5.2 Streams and `FILE` Objects

In Chapter 3, all the I/O routines centered around file descriptors. When a file is opened, a file descriptor is returned, and that descriptor is then used for all subsequent I/O operations. With the standard I/O library, the discussion centers around *streams*. (Do not confuse the standard I/O term *stream* with the STREAMS I/O system that is part of System V and standardized in the XSI STREAMS option in the Single UNIX Specification.) When we open or create a file with the standard I/O library, we say that we have associated a stream with the file.

With the ASCII character set, a single character is represented by a single byte. With international character sets, a character can be represented by more than one byte.

Standard I/O file streams can be used with single-byte and multibyte ("wide") character sets. A stream's orientation determines whether the characters that are read and written are single-byte or multibyte. Initially, when a stream is created, it has no orientation. If a multibyte I/O function (see `<wchar.h>`) is used on a stream without orientation, the stream's orientation is set to wide-oriented. If a byte I/O function is used on a stream without orientation, the stream's orientation is set to byte-oriented. Only two functions can change the orientation once set. The `freopen` function (discussed shortly) will clear a stream's orientation; the `fwide` function can be used to set a stream's orientation.

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *fp, int mode);
```

Returns: positive if stream is wide-oriented,
negative if stream is byte-oriented,
or 0 if stream has no orientation

The `fwide` function performs different tasks, depending on the value of the *mode* argument.

- If the *mode* argument is negative, `fwide` will try to make the specified stream byte-oriented.
- If the *mode* argument is positive, `fwide` will try to make the specified stream wide-oriented.
- If the *mode* argument is zero, `fwide` will not try to set the orientation, but will still return a value identifying the stream's orientation.

Note that `fwide` will not change the orientation of a stream that is already oriented. Also note that there is no error return. Consider what would happen if the stream is invalid. The only recourse we have is to clear `errno` before calling `fwide` and check the value of `errno` when we return. Throughout the rest of this book, we will deal only with byte-oriented streams.

When we open a stream, the standard I/O function `fopen` returns a pointer to a `FILE` object. This object is normally a structure that contains all the information required by the standard I/O library to manage the stream: the file descriptor used for actual I/O, a pointer to a buffer for the stream, the size of the buffer, a count of the number of characters currently in the buffer, an error flag, and the like.

Application software should never need to examine a `FILE` object. To reference the stream, we pass its `FILE` pointer as an argument to each standard I/O function. Throughout this text, we'll refer to a pointer to a `FILE` object, the type `FILE *` as a *file pointer*.

Throughout this chapter, we describe the standard I/O library in the context of a UNIX system. As we mentioned, this library has already been ported to a wide variety of other operating systems. But to provide some insight about how this library can be implemented, we will talk about its typical implementation on a UNIX system.

## 5.3 Standard Input, Standard Output, and Standard Error

Three streams are predefined and automatically available to a process: standard input, standard output, and standard error. These streams refer to the same files as the file descriptors `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`, which we mentioned in Section 3.2.

These three standard I/O streams are referenced through the predefined file pointers `stdin`, `stdout`, and `stderr`. The file pointers are defined in the `<stdio.h>` header.

## 5.4 Buffering

The goal of the buffering provided by the standard I/O library is to use the minimum number of `read` and `write` calls. (Recall Figure 3.5, where we showed the amount of CPU time required to perform I/O using various buffer sizes.) Also, it tries to do its buffering automatically for each I/O stream, obviating the need for the application to worry about it. Unfortunately, the single aspect of the standard I/O library that generates the most confusion is its buffering.

Three types of buffering are provided:

1. Fully buffered. In this case, actual I/O takes place when the standard I/O buffer is filled. Files residing on disk are normally fully buffered by the standard I/O library. The buffer used is usually obtained by one of the standard I/O functions calling `malloc` (Section 7.8) the first time I/O is performed on a stream.

   The term *flush* describes the writing of a standard I/O buffer. A buffer can be flushed automatically by the standard I/O routines, such as when a buffer fills, or we can call the function `fflush` to flush a stream. Unfortunately, in the UNIX environment, *flush* means two different things. In terms of the standard I/O library, it means writing out the contents of a buffer, which may be partially filled. In terms of the terminal driver, such as the `tcflush` function in Chapter 18, it means to discard the data that's already stored in a buffer.

2. Line buffered. In this case, the standard I/O library performs I/O when a newline character is encountered on input or output. This allows us to output a single character at a time (with the standard I/O `fputc` function), knowing that actual I/O will take place only when we finish writing each line. Line buffering is typically used on a stream when it refers to a terminal: standard input and standard output, for example.

   Line buffering comes with two caveats. First, the size of the buffer that the standard I/O library is using to collect each line is fixed, so I/O might take place if we fill this buffer before writing a newline. Second, whenever input is requested through the standard I/O library from either (a) an unbuffered stream

or (b) a line-buffered stream (that requires data to be requested from the kernel), *all* line-buffered output streams are flushed. The reason for the qualifier on (b) is that the requested data may already be in the buffer, which doesn't require data to be read from the kernel. Obviously, any input from an unbuffered stream, item (a), requires data to be obtained from the kernel.

3. Unbuffered. The standard I/O library does not buffer the characters. If we write 15 characters with the standard I/O `fputs` function, for example, we expect these 15 characters to be output as soon as possible, probably with the `write` function from Section 3.8.

   The standard error stream, for example, is normally unbuffered. This is so that any error messages are displayed as quickly as possible, regardless of whether they contain a newline.

ISO C requires the following buffering characteristics.

- Standard input and standard output are fully buffered, if and only if they do not refer to an interactive device.
- Standard error is never fully buffered.

This, however, doesn't tell us whether standard input and standard output can be unbuffered or line buffered if they refer to an interactive device and whether standard error should be unbuffered or line buffered. Most implementations default to the following types of buffering.

- Standard error is always unbuffered.
- All other streams are line buffered if they refer to a terminal device; otherwise, they are fully buffered.

  The four platforms discussed in this book follow these conventions for standard I/O buffering: standard error is unbuffered, streams open to terminal devices are line buffered, and all other streams are fully buffered.

We explore standard I/O buffering in more detail in Section 5.12 and Figure 5.11.

If we don't like these defaults for any given stream, we can change the buffering by calling either of the following two functions.

```
#include <stdio.h>

void setbuf(FILE *restrict fp, char *restrict buf);

int setvbuf(FILE *restrict fp, char *restrict buf, int mode,
            size_t size);

                                Returns: 0 if OK, nonzero on error
```

These functions must be called *after* the stream has been opened (obviously, since each requires a valid file pointer as its first argument) but *before* any other operation is performed on the stream.

With `setbuf`, we can turn buffering on or off. To enable buffering, *buf* must point to a buffer of length `BUFSIZ`, a constant defined in `<stdio.h>`. Normally, the stream is then fully buffered, but some systems may set line buffering if the stream is associated with a terminal device. To disable buffering, we set *buf* to `NULL`.

With `setvbuf`, we specify exactly which type of buffering we want. This is done with the *mode* argument:

| | |
|---|---|
| `_IOFBF` | fully buffered |
| `_IOLBF` | line buffered |
| `_IONBF` | unbuffered |

If we specify an unbuffered stream, the *buf* and *size* arguments are ignored. If we specify fully buffered or line buffered, *buf* and *size* can optionally specify a buffer and its size. If the stream is buffered and *buf* is `NULL`, the standard I/O library will automatically allocate its own buffer of the appropriate size for the stream. By appropriate size, we mean the value specified by the constant `BUFSIZ`.

> Some C library implementations use the value from the `st_blksize` member of the `stat` structure (see Section 4.2) to determine the optimal standard I/O buffer size. As we will see later in this chapter, the GNU C library uses this method.

Figure 5.1 summarizes the actions of these two functions and their various options.

| Function | *mode* | *buf* | Buffer and length | Type of buffering |
|---|---|---|---|---|
| `setbuf` | | non-null | user *buf* of length `BUFSIZ` | fully buffered or line buffered |
| | | `NULL` | (no buffer) | unbuffered |
| `setvbuf` | `_IOFBF` | non-null | user *buf* of length *size* | fully buffered |
| | | `NULL` | system buffer of appropriate length | |
| | `_IOLBF` | non-null | user *buf* of length *size* | line buffered |
| | | `NULL` | system buffer of appropriate length | |
| | `_IONBF` | (ignored) | (no buffer) | unbuffered |

**Figure 5.1** Summary of the `setbuf` and `setvbuf` functions

Be aware that if we allocate a standard I/O buffer as an automatic variable within a function, we have to close the stream before returning from the function. (We'll discuss this more in Section 7.8.) Also, some implementations use part of the buffer for internal bookkeeping, so the actual number of bytes of data that can be stored in the buffer is less than *size*. In general, we should let the system choose the buffer size and automatically allocate the buffer. When we do this, the standard I/O library automatically releases the buffer when we close the stream.

At any time, we can force a stream to be flushed.

```
#include <stdio.h>

int fflush(FILE *fp);
```

Returns: 0 if OK, `EOF` on error

This function causes any unwritten data for the stream to be passed to the kernel. As a special case, if *fp* is `NULL`, this function causes all output streams to be flushed.

## 5.5 Opening a Stream

The following three functions open a standard I/O stream.

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int filedes, const char *type);
```

All three return: file pointer if OK, NULL on error

The differences in these three functions are as follows.

1. The fopen function opens a specified file.
2. The freopen function opens a specified file on a specified stream, closing the stream first if it is already open. If the stream previously had an orientation, freopen clears it. This function is typically used to open a specified file as one of the predefined streams: standard input, standard output, or standard error.
3. The fdopen function takes an existing file descriptor, which we could obtain from the open, dup, dup2, fcntl, pipe, socket, socketpair, or accept functions, and associates a standard I/O stream with the descriptor. This function is often used with descriptors that are returned by the functions that create pipes and network communication channels. Because these special types of files cannot be opened with the standard I/O fopen function, we have to call the device-specific function to obtain a file descriptor, and then associate this descriptor with a standard I/O stream using fdopen.

Both fopen and freopen are part of ISO C; fdopen is part of POSIX.1, since ISO C doesn't deal with file descriptors.

ISO C specifies 15 values for the *type* argument, shown in Figure 5.2.

| *type* | Description |
|---|---|
| r or rb | open for reading |
| w or wb | truncate to 0 length or create for writing |
| a or ab | append; open for writing at end of file, or create for writing |
| r+ or r+b or rb+ | open for reading and writing |
| w+ or w+b or wb+ | truncate to 0 length or create for reading and writing |
| a+ or a+b or ab+ | open or create for reading and writing at end of file |

**Figure 5.2** The *type* argument for opening a standard I/O stream

Using the character b as part of the *type* allows the standard I/O system to differentiate between a text file and a binary file. Since the UNIX kernel doesn't differentiate between these types of files, specifying the character b as part of the *type* has no effect.

With fdopen, the meanings of the *type* argument differ slightly. The descriptor has already been opened, so opening for write does not truncate the file. (If the descriptor was created by the open function, for example, and the file already existed, the O_TRUNC flag would control whether or not the file was truncated. The fdopen function cannot simply truncate any file it opens for writing.) Also, the standard I/O append mode cannot create the file (since the file has to exist if a descriptor refers to it).

When a file is opened with a type of append, each write will take place at the then current end of file. If multiple processes open the same file with the standard I/O append mode, the data from each process will be correctly written to the file.

> Versions of fopen from Berkeley before 4.4BSD and the simple version shown on page 177 of Kernighan and Ritchie [1988] do not handle the append mode correctly. These versions do an lseek to the end of file when the stream is opened. To correctly support the append mode when multiple processes are involved, the file must be opened with the O_APPEND flag, which we discussed in Section 3.3. Doing an lseek before each write won't work either, as we discussed in Section 3.11.

When a file is opened for reading and writing (the plus sign in the *type*), the following restrictions apply.

- Output cannot be directly followed by input without an intervening fflush, fseek, fsetpos, or rewind.
- Input cannot be directly followed by output without an intervening fseek, fsetpos, or rewind, or an input operation that encounters an end of file.

We can summarize the six ways to open a stream from Figure 5.2 in Figure 5.3.

| Restriction | r | w | a | r+ | w+ | a+ |
|---|---|---|---|---|---|---|
| file must already exist | • | | | • | | |
| previous contents of file discarded | | • | | | • | |
| stream can be read | • | | | • | • | • |
| stream can be written | | • | • | • | • | • |
| stream can be written only at end | | | • | | | • |

**Figure 5.3** Six ways to open a standard I/O stream

Note that if a new file is created by specifying a *type* of either w or a, we are not able to specify the file's access permission bits, as we were able to do with the open function and the creat function in Chapter 3.

By default, the stream that is opened is fully buffered, unless it refers to a terminal device, in which case it is line buffered. Once the stream is opened, but before we do any other operation on the stream, we can change the buffering if we want to, with the setbuf or setvbuf functions from the previous section.

An open stream is closed by calling fclose.

```
#include <stdio.h>

int fclose(FILE *fp);
```

Returns: 0 if OK, EOF on error

Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded. If the standard I/O library had automatically allocated a buffer for the stream, that buffer is released.

When a process terminates normally, either by calling the `exit` function directly or by returning from the `main` function, all standard I/O streams with unwritten buffered data are flushed, and all open standard I/O streams are closed.

## 5.6 Reading and Writing a Stream

Once we open a stream, we can choose from among three types of unformatted I/O:

1. Character-at-a-time I/O. We can read or write one character at a time, with the standard I/O functions handling all the buffering, if the stream is buffered.
2. Line-at-a-time I/O. If we want to read or write a line at a time, we use `fgets` and `fputs`. Each line is terminated with a newline character, and we have to specify the maximum line length that we can handle when we call `fgets`. We describe these two functions in Section 5.7.
3. Direct I/O. This type of I/O is supported by the `fread` and `fwrite` functions. For each I/O operation, we read or write some number of objects, where each object is of a specified size. These two functions are often used for binary files where we read or write a structure with each operation. We describe these two functions in Section 5.9.

The term *direct I/O*, from the ISO C standard, is known by many names: binary I/O, object-at-a-time I/O, record-oriented I/O, or structure-oriented I/O.

(We describe the formatted I/O functions, such as `printf` and `scanf`, in Section 5.11.)

### Input Functions

Three functions allow us to read one character at a time.

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);

All three return: next character if OK, EOF on end of file or error
```

The function `getchar` is defined to be equivalent to `getc(stdin)`. The difference between the first two functions is that `getc` can be implemented as a macro, whereas `fgetc` cannot be implemented as a macro. This means three things.

1. The argument to `getc` should not be an expression with side effects.
2. Since `fgetc` is guaranteed to be a function, we can take its address. This allows us to pass the address of `fgetc` as an argument to another function.

3. Calls to fgetc probably take longer than calls to getc, as it usually takes more time to call a function.

These three functions return the next character as an unsigned char converted to an int. The reason for specifying unsigned is so that the high-order bit, if set, doesn't cause the return value to be negative. The reason for requiring an integer return value is so that all possible character values can be returned, along with an indication that either an error occurred or the end of file has been encountered. The constant EOF in <stdio.h> is required to be a negative value. Its value is often −1. This representation also means that we cannot store the return value from these three functions in a character variable and compare this value later against the constant EOF.

Note that these functions return the same value whether an error occurs or the end of file is reached. To distinguish between the two, we must call either ferror or feof.

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);

        Both return: nonzero (true) if condition is true, 0 (false) otherwise

void clearerr(FILE *fp);
```

In most implementations, two flags are maintained for each stream in the FILE object:

- An error flag
- An end-of-file flag

Both flags are cleared by calling clearerr.

After reading from a stream, we can push back characters by calling ungetc.

```
#include <stdio.h>

int ungetc(int c, FILE *fp);

                                        Returns: c if OK, EOF on error
```

The characters that are pushed back are returned by subsequent reads on the stream in reverse order of their pushing. Be aware, however, that although ISO C allows an implementation to support any amount of pushback, an implementation is required to provide only a single character of pushback. We should not count on more than a single character.

The character that we push back does not have to be the same character that was read. We are not able to push back EOF. But when we've reached the end of file, we can push back a character. The next read will return that character, and the read after that will return EOF. This works because a successful call to ungetc clears the end-of-file indication for the stream.

Pushback is often used when we're reading an input stream and breaking the input into words or tokens of some form. Sometimes we need to peek at the next character to determine how to handle the current character. It's then easy to push back the character that we peeked at, for the next call to getc to return. If the standard I/O library didn't

provide this pushback capability, we would have to store the character in a variable of our own, along with a flag telling us to use this character instead of calling `getc` the next time we need a character.

> When we push characters back with `ungetc`, they don't get written back to the underlying file or device. They are kept incore in the standard I/O library's buffer for the stream.

### Output Functions

We'll find an output function that corresponds to each of the input functions that we've already described.

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);

                              All three return: c if OK, EOF on error
```

Like the input functions, `putchar(c)` is equivalent to `putc(c, stdout)`, and `putc` can be implemented as a macro, whereas `fputc` cannot be implemented as a macro.

## 5.7 Line-at-a-Time I/O

Line-at-a-time input is provided by the following two functions.

```
#include <stdio.h>

char *fgets(char *restrict buf, int n, FILE *restrict fp);

char *gets(char *buf);

                    Both return: buf if OK, NULL on end of file or error
```

Both specify the address of the buffer to read the line into. The `gets` function reads from standard input, whereas `fgets` reads from the specified stream.

With `fgets`, we have to specify the size of the buffer, *n*. This function reads up through and including the next newline, but no more than $n-1$ characters, into the buffer. The buffer is terminated with a null byte. If the line, including the terminating newline, is longer than $n-1$, only a partial line is returned, but the buffer is always null terminated. Another call to `fgets` will read what follows on the line.

The `gets` function should never be used. The problem is that it doesn't allow the caller to specify the buffer size. This allows the buffer to overflow, if the line is longer than the buffer, writing over whatever happens to follow the buffer in memory. For a description of how this flaw was used as part of the Internet worm of 1988, see the June 1989 issue (vol. 32, no. 6) of *Communications of the ACM*. An additional difference with `gets` is that it doesn't store the newline in the buffer, as does `fgets`.

This difference in newline handling between the two functions goes way back in the evolution of the UNIX System. Even the Version 7 manual (1979) states "gets deletes a newline, fgets keeps it, all in the name of backward compatibility."

Even though ISO C requires an implementation to provide gets, use fgets instead.

Line-at-a-time output is provided by fputs and puts.

```
#include <stdio.h>

int fputs(const char *restrict str, FILE *restrict fp);

int puts(const char *str);
```

Both return: non-negative value if OK, EOF on error

The function fputs writes the null-terminated string to the specified stream. The null byte at the end is not written. Note that this need not be line-at-a-time output, since the string need not contain a newline as the last non-null character. Usually, this is the case—the last non-null character is a newline—but it's not required.

The puts function writes the null-terminated string to the standard output, without writing the null byte. But puts then writes a newline character to the standard output.

The puts function is not unsafe, like its counterpart gets. Nevertheless, we'll avoid using it, to prevent having to remember whether it appends a newline. If we always use fgets and fputs, we know that we always have to deal with the newline character at the end of each line.

## 5.8 Standard I/O Efficiency

Using the functions from the previous section, we can get an idea of the efficiency of the standard I/O system. The program in Figure 5.4 is like the one in Figure 3.4: it simply copies standard input to standard output, using getc and putc. These two routines can be implemented as macros.

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

**Figure 5.4** Copy standard input to standard output using getc and putc

We can make another version of this program that uses fgetc and fputc, which should be functions, not macros. (We don't show this trivial change to the source code.)

Finally, we have a version that reads and writes lines, shown in Figure 5.5.

```
#include "apue.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

**Figure 5.5** Copy standard input to standard output using fgets and fputs

Note that we do not close the standard I/O streams explicitly in Figure 5.4 or Figure 5.5. Instead, we know that the exit function will flush any unwritten data and then close all open streams. (We'll discuss this in Section 8.5.) It is interesting to compare the timing of these three programs with the timing data from Figure 3.5. We show this data when operating on the same file (98.5 MB with 3 million lines) in Figure 5.6.

| Function | User CPU (seconds) | System CPU (seconds) | Clock time (seconds) | Bytes of program text |
|---|---|---|---|---|
| best time from Figure 3.5 | 0.01 | 0.18 | 6.67 | |
| fgets, fputs | 2.59 | 0.19 | 7.15 | 139 |
| getc, putc | 10.84 | 0.27 | 12.07 | 120 |
| fgetc, fputc | 10.44 | 0.27 | 11.42 | 120 |
| single byte time from Figure 3.5 | 124.89 | 161.65 | 288.64 | |

**Figure 5.6** Timing results using standard I/O routines

For each of the three standard I/O versions, the user CPU time is larger than the best read version from Figure 3.5, because the character-at-a-time standard I/O versions have a loop that is executed 100 million times, and the loop in the line-at-a-time version is executed 3,144,984 times. In the read version, its loop is executed only 12,611 times (for a buffer size of 8,192). This difference in clock times is from the difference in user times and the difference in the times spent waiting for I/O to complete, as the system times are comparable.

The system CPU time is about the same as before, because roughly the same number of kernel requests are being made. Note that an advantage of using the standard I/O routines is that we don't have to worry about buffering or choosing the

optimal I/O size. We do have to determine the maximum line size for the version that uses fgets, but that's easier than trying to choose the optimal I/O size.

The final column in Figure 5.6 is the number of bytes of text space—the machine instructions generated by the C compiler—for each of the main functions. We can see that the version using getc and putc takes the same amount of space as the one using the fgetc and fputc functions. Usually, getc and putc are implemented as macros, but in the GNU C library implementation, the macro simply expands to a function call.

The version using line-at-a-time I/O is almost twice as fast as the version using character-at-a-time I/O. If the fgets and fputs functions are implemented using getc and putc (see Section 7.7 of Kernighan and Ritchie [1988], for example), then we would expect the timing to be similar to the getc version. Actually, we might expect the line-at-a-time version to take longer, since we would be adding the overhead of 200 million extra function calls to the existing 6 million ones. What is happening with this example is that the line-at-a-time functions are implemented using memccpy(3). Often, the memccpy function is implemented in assembler instead of C, for efficiency.

The last point of interest with these timing numbers is that the fgetc version is so much faster than the BUFFSIZE=1 version from Figure 3.5. Both involve the same number of function calls—about 200 million—yet the fgetc version is almost 12 times faster in user CPU time and slightly more than 25 times faster in clock time. The difference is that the version using read executes 200 million function calls, which in turn execute 200 million system calls. With the fgetc version, we still execute 200 million function calls, but this ends up being only 25,222 system calls. System calls are usually much more expensive than ordinary function calls.

As a disclaimer, you should be aware that these timing results are valid only on the single system they were run on. The results depend on many implementation features that aren't the same on every UNIX system. Nevertheless, having a set of numbers such as these, and explaining why the various versions differ, helps us understand the system better. From this section and Section 3.9, we've learned that the standard I/O library is not much slower than calling the read and write functions directly. The approximate cost that we've seen is about 0.11 seconds of CPU time to copy a megabyte of data using getc and putc. For most nontrivial applications, the largest amount of the user CPU time is taken by the application, not by the standard I/O routines.

## 5.9 Binary I/O

The functions from Section 5.6 operated with one character at a time, and the functions from Section 5.7 operated with one line at a time. If we're doing binary I/O, we often would like to read or write an entire structure at a time. To do this using getc or putc, we have to loop through the entire structure, one byte at a time, reading or writing each byte. We can't use the line-at-a-time functions, since fputs stops writing when it hits a null byte, and there might be null bytes within the structure. Similarly, fgets won't work right on input if any of the data bytes are nulls or newlines. Therefore, the following two functions are provided for binary I/O.

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj,
             FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);

                      Both return: number of objects read or written
```

These functions have two common uses:

1. Read or write a binary array. For example, to write elements 2 through 5 of a floating-point array, we could write

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

   Here, we specify *size* as the size of each element of the array and *nobj* as the number of elements.

2. Read or write a structure. For example, we could write

```
struct {
  short  count;
  long   total;
  char   name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

   Here, we specify *size* as the size of structure and *nobj* as one (the number of objects to write).

The obvious generalization of these two cases is to read or write an array of structures. To do this, *size* would be the `sizeof` the structure, and *nobj* would be the number of elements in the array.

Both `fread` and `fwrite` return the number of objects read or written. For the read case, this number can be less than *nobj* if an error occurs or if the end of file is encountered. In this case `ferror` or `feof` must be called. For the write case, if the return value is less than the requested *nobj*, an error has occurred.

A fundamental problem with binary I/O is that it can be used to read only data that has been written on the same system. This was OK many years ago, when all the UNIX systems were PDP-11s, but the norm today is to have heterogeneous systems connected together with networks. It is common to want to write data on one system and process it on another. These two functions won't work, for two reasons.

1. The offset of a member within a structure can differ between compilers and systems, because of different alignment requirements. Indeed, some compilers have an option allowing structures to be packed tightly, to save space with a possible runtime performance penalty, or aligned accurately, to optimize runtime access of each member. This means that even on a single system, the binary layout of a structure can differ, depending on compiler options.

2. The binary formats used to store multibyte integers and floating-point values differ among machine architectures.

We'll touch on some of these issues when we discuss sockets in Chapter 16. The real solution for exchanging binary data among different systems is to use a higher-level protocol. Refer to Section 8.2 of Rago [1993] or Section 5.18 of Stevens, Fenner, & Rudoff [2004] for a description of some techniques various network protocols use to exchange binary data.

We'll return to the `fread` function in Section 8.14 when we'll use it to read a binary structure, the UNIX process accounting records.

## 5.10 Positioning a Stream

There are three ways to position a standard I/O stream:

1. The two functions `ftell` and `fseek`. They have been around since Version 7, but they assume that a file's position can be stored in a long integer.

2. The two functions `ftello` and `fseeko`. They were introduced in the Single UNIX Specification to allow for file offsets that might not fit in a long integer. They replace the long integer with the `off_t` data type.

3. The two functions `fgetpos` and `fsetpos`. They were introduced by ISO C. They use an abstract data type, `fpos_t`, that records a file's position. This data type can be made as big as necessary to record a file's position.

Portable applications that need to move to non-UNIX systems should use `fgetpos` and `fsetpos`.

```
#include <stdio.h>

long ftell(FILE *fp);

                    Returns: current file position indicator if OK, –1L on error

int fseek(FILE *fp, long offset, int whence);

                                           Returns: 0 if OK, nonzero on error

void rewind(FILE *fp);
```

For a binary file, a file's position indicator is measured in bytes from the beginning of the file. The value returned by `ftell` for a binary file is this byte position. To position a binary file using `fseek`, we must specify a byte *offset* and how that offset is interpreted. The values for *whence* are the same as for the `lseek` function from Section 3.6: `SEEK_SET` means from the beginning of the file, `SEEK_CUR` means from the current file position, and `SEEK_END` means from the end of file. ISO C doesn't require an implementation to support the `SEEK_END` specification for a binary file, as some systems require a binary file to be padded at the end with zeros to make the file size a multiple of some magic number. Under the UNIX System, however, `SEEK_END` is supported for binary files.

For text files, the file's current position may not be measurable as a simple byte offset. Again, this is mainly under non-UNIX systems that might store text files in a different format. To position a text file, *whence* has to be `SEEK_SET`, and only two values for *offset* are allowed: 0—meaning rewind the file to its beginning—or a value that was returned by `ftell` for that file. A stream can also be set to the beginning of the file with the `rewind` function.

The `ftello` function is the same as `ftell`, and the `fseeko` function is the same as `fseek`, except that the type of the offset is `off_t` instead of `long`.

```
#include <stdio.h>

off_t ftello(FILE *fp);

                    Returns: current file position indicator if OK, (off_t)-1 on error

int fseeko(FILE *fp, off_t offset, int whence);

                                                     Returns: 0 if OK, nonzero on error
```

Recall the discussion of the `off_t` data type in Section 3.6. Implementations can define the `off_t` type to be larger than 32 bits.

As we mentioned, the `fgetpos` and `fsetpos` functions were introduced by the ISO C standard.

```
#include <stdio.h>

int fgetpos(FILE *restrict fp, fpos_t *restrict pos);

int fsetpos(FILE *fp, const fpos_t *pos);

                                               Both return: 0 if OK, nonzero on error
```

The `fgetpos` function stores the current value of the file's position indicator in the object pointed to by *pos*. This value can be used in a later call to `fsetpos` to reposition the stream to that location.

## 5.11 Formatted I/O

### Formatted Output

Formatted output is handled by the four printf functions.

```
#include <stdio.h>

int printf(const char *restrict format, ...);

int fprintf(FILE *restrict fp, const char *restrict format, ...);
```

Both return: number of characters output if OK, negative value if output error

```
int sprintf(char *restrict buf, const char *restrict format, ...);

int snprintf(char *restrict buf, size_t n,
             const char *restrict format, ...);
```

Both return: number of characters stored in array if OK, negative value if encoding error

The printf function writes to the standard output, fprintf writes to the specified stream, and sprintf places the formatted characters in the array *buf*. The sprintf function automatically appends a null byte at the end of the array, but this null byte is not included in the return value.

Note that it's possible for sprintf to overflow the buffer pointed to by *buf*. It's the caller's responsibility to ensure that the buffer is large enough. Because this can lead to buffer-overflow problems, snprintf was introduced. With it, the size of the buffer is an explicit parameter; any characters that would have been written past the end of the buffer are discarded instead. The snprintf function returns the number of characters that would have been written to the buffer had it been big enough. As with sprintf, the return value doesn't include the terminating null byte. If snprintf returns a positive value less than the buffer size *n*, then the output was not truncated. If an encoding error occurs, snprintf returns a negative value.

The format specification controls how the remainder of the arguments will be encoded and ultimately displayed. Each argument is encoded according to a conversion specification that starts with a percent sign (%). Except for the conversion specifications, other characters in the format are copied unmodified. A conversion specification has four optional components, shown in square brackets below:

```
%[flags][fldwidth][precision][lenmodifier]convtype
```

The flags are summarized in Figure 5.7.

The fldwidth component specifies a minimum field width for the conversion. If the conversion results in fewer characters, it is padded with spaces. The field width is a non-negative decimal integer or an asterisk.

The precision component specifies the minimum number of digits to appear for integer conversions, the minimum number of digits to appear to the right of the decimal point for floating-point conversions, or the maximum number of bytes for string conversions. The precision is a period (.) followed by a optional non-negative decimal integer or an asterisk.

| Flag | Description |
|---|---|
| - | left-justify the output in the field |
| + | always display sign of a signed conversion |
| (space) | prefix by a space if no sign is generated |
| # | convert using alternate form (include 0x prefix for hex format, for example) |
| 0 | prefix with leading zeros instead of padding with spaces |

**Figure 5.7** The flags component of a conversion specification

Both the field width and precision can be an asterisk. In this case, an integer argument specifies the value to be used. The argument appears directly before the argument to converted.

The `lenmodifier` component specifies the size of the argument. Possible values are summarized in Figure 5.8.

| Length modifier | Description |
|---|---|
| `hh` | signed or unsigned `char` |
| `h` | signed or unsigned `short` |
| `l` | signed or unsigned `long` or wide character |
| `ll` | signed or unsigned `long long` |
| `j` | `intmax_t` or `uintmax_t` |
| `z` | `size_t` |
| `t` | `ptrdiff_t` |
| `L` | `long double` |

**Figure 5.8** The length modifier component of a conversion specification

The `convtype` component is not optional. It controls how the argument is interpreted. The various conversion types are summarized in Figure 5.9.

| Conversion type | Description |
|---|---|
| `d,i` | signed decimal |
| `o` | unsigned octal |
| `u` | unsigned decimal |
| `x,X` | unsigned hexadecimal |
| `f,F` | `double` floating-point number |
| `e,E` | `double` floating-point number in exponential format |
| `g,G` | interpreted as f, F, e, or E, depending on value converted |
| `a,A` | `double` floating-point number in hexadecimal exponential format |
| `c` | character (with `l` length modifier, wide character) |
| `s` | string (with `l` length modifier, wide character string) |
| `p` | pointer to a `void` |
| `n` | pointer to a signed integer into which is written the number of characters written so far |
| `%` | a % character |
| `C` | wide character (an XSI extension, equivalent to `lc`) |
| `S` | wide character string (an XSI extension, equivalent to `ls`) |

**Figure 5.9** The conversion type component of a conversion specification

The following four variants of the printf family are similar to the previous four, but the variable argument list (. . .) is replaced with *arg*.

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *restrict format, va_list arg);

int vfprintf(FILE *restrict fp, const char *restrict format,
             va_list arg);
```

Both return: number of characters output if OK, negative value if output error

```
int vsprintf(char *restrict buf, const char *restrict format,
             va_list arg);

int vsnprintf(char *restrict buf, size_t n,
              const char *restrict format, va_list arg);
```

Both return: number of characters stored in array if OK, negative value if encoding error

We use the vsnprintf function in the error routines in Appendix B.

Refer to Section 7.3 of Kernighan and Ritchie [1988] for additional details on handling variable-length argument lists with ISO Standard C. Be aware that the variable-length argument list routines provided with ISO C—the <stdarg.h> header and its associated routines—differ from the <varargs.h> routines that were provided with older UNIX systems.

## Formatted Input

Formatted input is handled by the three scanf functions.

```
#include <stdio.h>

int scanf(const char *restrict format, ...);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

int sscanf(const char *restrict buf, const char *restrict format,
           ...);
```

All three return: number of input items assigned,
EOF if input error or end of file before any conversion

The scanf family is used to parse an input string and convert character sequences into variables of specified types. The arguments following the format contain the addresses of the variables to initialize with the results of the conversions.

The format specification controls how the arguments are converted for assignment. The percent sign (%) indicates the start of a conversion specification. Except for the conversion specifications and white space, other characters in the format have to match the input. If a character doesn't match, processing stops, leaving the remainder of the input unread.

There are three optional components to a conversion specification, shown in square brackets below:

```
%[*][fldwidth][lenmodifier]convtype
```

The optional leading asterisk is used to suppress conversion. Input is converted as specified by the rest of the conversion specification, but the result is not stored in an argument.

The `fldwidth` component specifies the maximum field width in characters. The `lenmodifier` component specifies the size of the argument to be initialized with the result of the conversion. The same length modifiers supported by the `printf` family of functions are supported by the `scanf` family of functions (see Figure 5.8 for a list of the length modifiers).

The `convtype` field is similar to the conversion type field used by the `printf` family, but there are some differences. One difference is that results that are stored in unsigned types can optionally be signed on input. For example, −1 will scan as 4294967295 into an unsigned integer. Figure 5.10 summarizes the conversion types supported by the `scanf` family of functions.

| Conversion type | Description |
|---|---|
| d | signed decimal, base 10 |
| i | signed decimal, base determined by format of input |
| o | unsigned octal (input optionally signed) |
| u | unsigned decimal, base 10 (input optionally signed) |
| x | unsigned hexadecimal (input optionally signed) |
| a,A,e,E,f,F,g,G | floating-point number |
| c | character (with l length modifier, wide character) |
| s | string (with l length modifier, wide character string) |
| [ | matches a sequence of listed characters, ending with ] |
| [^ | matches all characters except the ones listed, ending with ] |
| p | pointer to a void |
| n | pointer to a signed integer into which is written the number of characters read so far |
| % | a % character |
| C | wide character (an XSI extension, equivalent to lc) |
| S | wide character string (an XSI extension, equivalent to ls) |

**Figure 5.10** The conversion type component of a conversion specification

As with the `printf` family, the `scanf` family also supports functions that use variable argument lists as specified by `<stdarg.h>`.

```
#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *restrict format, va_list arg);

int vfscanf(FILE *restrict fp, const char *restrict format,
            va_list arg);

int vsscanf(const char *restrict buf, const char *restrict format,
            va_list arg);
```

All three return: number of input items assigned,
EOF if input error or end of file before any conversion

Refer to your UNIX system manual for additional details on the `scanf` family of functions.

## 5.12 Implementation Details

As we've mentioned, under the UNIX System, the standard I/O library ends up calling the I/O routines that we described in Chapter 3. Each standard I/O stream has an associated file descriptor, and we can obtain the descriptor for a stream by calling `fileno`.

> Note that `fileno` is not part of the ISO C standard, but an extension supported by POSIX.1.

```
#include <stdio.h>

int fileno(FILE *fp);
```

Returns: the file descriptor associated with the stream

We need this function if we want to call the dup or `fcntl` functions, for example.

To look at the implementation of the standard I/O library on your system, start with the header `<stdio.h>`. This will show how the `FILE` object is defined, the definitions of the per-stream flags, and any standard I/O routines, such as `getc`, that are defined as macros. Section 8.5 of Kernighan and Ritchie [1988] has a sample implementation that shows the flavor of many implementations on UNIX systems. Chapter 12 of Plauger [1992] provides the complete source code for an implementation of the standard I/O library. The implementation of the GNU standard I/O library is also publicly available.

### Example

The program in Figure 5.11 prints the buffering for the three standard streams and for a stream that is associated with a regular file.

```
#include "apue.h"

void    pr_stdio(const char *, FILE *);

int
main(void)
{
    FILE    *fp;

    fputs("enter any character\n", stdout);
    if (getchar() == EOF)
        err_sys("getchar error");
    fputs("one line to standard error\n", stderr);

    pr_stdio("stdin",  stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);

    if ((fp = fopen("/etc/motd", "r")) == NULL)
        err_sys("fopen error");
    if (getc(fp) == EOF)
        err_sys("getc error");
    pr_stdio("/etc/motd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);

    /*
     * The following is nonportable.
     */
    if (fp->_IO_file_flags & _IO_UNBUFFERED)
        printf("unbuffered");
    else if (fp->_IO_file_flags & _IO_LINE_BUF)
        printf("line buffered");
    else /* if neither of above */
        printf("fully buffered");
    printf(", buffer size = %d\n", fp->_IO_buf_end - fp->_IO_buf_base);
}
```

**Figure 5.11** Print buffering for various standard I/O streams

Note that we perform I/O on each stream before printing its buffering status, since the first I/O operation usually causes the buffers to be allocated for a stream. The structure members `_IO_file_flags`, `_IO_buf_base`, and `_IO_buf_end` and the constants `_IO_UNBUFFERED` and `_IO_LINE_BUFFERED` are defined by the GNU standard I/O library used on Linux. Be aware that other UNIX systems may have different implementations of the standard I/O library.

If we run the program in Figure 5.11 twice, once with the three standard streams connected to the terminal and once with the three standard streams redirected to files, we get the following result:

```
$ ./a.out                                   stdin, stdout, and stderr connected to terminal
enter any character
                                            we type a newline
one line to standard error
stream = stdin, line buffered, buffer size = 1024
stream = stdout, line buffered, buffer size = 1024
stream = stderr, unbuffered, buffer size = 1
stream = /etc/motd, fully buffered, buffer size = 4096
$ ./a.out < /etc/termcap > std.out 2> std.err
                                            run it again with all three streams redirected
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 4096
stream = stdout, fully buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/motd, fully buffered, buffer size = 4096
```

We can see that the default for this system is to have standard input and standard output line buffered when they're connected to a terminal. The line buffer is 1,024 bytes. Note that this doesn't restrict us to 1,024-byte input and output lines; that's just the size of the buffer. Writing a 2,048-byte line to standard output will require two `write` system calls. When we redirect these two streams to regular files, they become fully buffered, with buffer sizes equal to the preferred I/O size—the `st_blksize` value from the `stat` structure—for the file system. We also see that the standard error is always unbuffered, as it should be, and that a regular file defaults to fully buffered. □

## 5.13 Temporary Files

The ISO C standard defines two functions that are provided by the standard I/O library to assist in creating temporary files.

```
#include <stdio.h>

char *tmpnam(char *ptr);
                                        Returns: pointer to unique pathname

FILE *tmpfile(void);
                                Returns: file pointer if OK, NULL on error
```

The `tmpnam` function generates a string that is a valid pathname and that is not the same name as an existing file. This function generates a different pathname each time it is called, up to `TMP_MAX` times. `TMP_MAX` is defined in `<stdio.h>`.

> Although ISO C defines TMP_MAX, the C standard requires only that its value be at least 25. The Single UNIX Specification, however, requires that XSI-conforming systems support a value of at least 10,000. Although this minimum value allows an implementation to use four digits (0000–9999), most implementations on UNIX systems use lowercase or uppercase characters.

If *ptr* is NULL, the generated pathname is stored in a static area, and a pointer to this area is returned as the value of the function. Subsequent calls to tmpnam can overwrite this static area. (This means that if we call this function more than once and we want to save the pathname, we have to save a copy of the pathname, not a copy of the pointer.) If *ptr* is not NULL, it is assumed that it points to an array of at least L_tmpnam characters. (The constant L_tmpnam is defined in <stdio.h>.) The generated pathname is stored in this array, and *ptr* is also returned as the value of the function.

The tmpfile function creates a temporary binary file (type wb+) that is automatically removed when it is closed or on program termination. Under the UNIX System, it makes no difference that this file is a binary file.

### Example

The program in Figure 5.12 demonstrates these two functions.

```
#include "apue.h"

int
main(void)
{
    char    name[L_tmpnam], line[MAXLINE];
    FILE    *fp;

    printf("%s\n", tmpnam(NULL));       /* first temp name */

    tmpnam(name);                       /* second temp name */
    printf("%s\n", name);

    if ((fp = tmpfile()) == NULL)       /* create temp file */
        err_sys("tmpfile error");
    fputs("one line of output\n", fp);  /* write to temp file */
    rewind(fp);                         /* then read it back */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("fgets error");
    fputs(line, stdout);                /* print the line we wrote */

    exit(0);
}
```

**Figure 5.12** Demonstrate tmpnam and tmpfile functions

If we execute the program in Figure 5.12, we get

```
$ ./a.out
/tmp/fileC1Icwc
/tmp/filemSkHSe
one line of output
```

□

The standard technique often used by the `tmpfile` function is to create a unique pathname by calling `tmpnam`, then create the file, and immediately `unlink` it. Recall from Section 4.15 that unlinking a file does not delete its contents until the file is closed. This way, when the file is closed, either explicitly or on program termination, the contents of the file are deleted.

The Single UNIX Specification defines two additional functions as XSI extensions for dealing with temporary files. The first of these is the `tempnam` function.

```
#include <stdio.h>

char *tempnam(const char *directory, const char *prefix);
```

Returns: pointer to unique pathname

The `tempnam` function is a variation of `tmpnam` that allows the caller to specify both the directory and a prefix for the generated pathname. There are four possible choices for the directory, and the first one that is true is used.

1. If the environment variable `TMPDIR` is defined, it is used as the directory. (We describe environment variables in Section 7.9.)
2. If *directory* is not `NULL`, it is used as the directory.
3. The string `P_tmpdir` in `<stdio.h>` is used as the directory.
4. A local directory, usually `/tmp`, is used as the directory.

If the *prefix* argument is not `NULL`, it should be a string of up to five bytes to be used as the first characters of the filename.

This function calls the `malloc` function to allocate dynamic storage for the constructed pathname. We can free this storage when we're done with the pathname. (We describe the `malloc` and `free` functions in Section 7.8.)

### Example

The program in Figure 5.13 shows the use of `tempnam`.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    if (argc != 3)
        err_quit("usage: a.out <directory> <prefix>");

    printf("%s\n", tempnam(argv[1][0] != ' ' ? argv[1] : NULL,
      argv[2][0] != ' ' ? argv[2] : NULL));

    exit(0);
}
```

**Figure 5.13** Demonstrate `tempnam` function

Note that if either command-line argument—the directory or the prefix—begins with a blank, we pass a null pointer to the function. We can now show the various ways to use it:

```
$ ./a.out /home/sar TEMP                      specify both directory and prefix
/home/sar/TEMPsf00zi
$ ./a.out " " PFX                             use default directory: P_tmpdir
/tmp/PFXfBw7Gi
$ TMPDIR=/var/tmp ./a.out /usr/tmp " "        use environment variable; no prefix
/var/tmp/file8fVYNi                           environment variable overrides directory
$ TMPDIR=/no/such/dir ./a.out /home/sar/tmp QQQ
/home/sar/tmp/QQQ98s8Ui                       invalid environment directory is ignored
```

As the four steps that we listed earlier for specifying the directory name are tried in order, this function also checks whether the corresponding directory name makes sense. If the directory doesn't exist (the `/no/such/dir` example), that case is skipped, and the next choice for the directory name is tried. From this example, we can see that for this implementation, the `P_tmpdir` directory is `/tmp`. The technique that we used to set the environment variable, specifying `TMPDIR=` before the program name, is used by the Bourne shell, the Korn shell, and `bash`. □

The second function that XSI defines is `mkstemp`. It is similar to `tmpfile`, but returns an open file descriptor for the temporary file instead of a file pointer.

```
#include <stdlib.h>

int mkstemp(char *template);
```

Returns: file descriptor if OK, −1 on error

The returned file descriptor is open for reading and writing. The name of the temporary file is selected using the *template* string. This string is a pathname whose last six characters are set to XXXXXX. The function replaces these with different characters to create a unique pathname. If `mkstemp` returns success, it modifies the *template* string to reflect the name of the temporary file.

Unlike `tmpfile`, the temporary file created by `mkstemp` is not removed automatically for us. If we want to remove it from the file system namespace, we need to unlink it ourselves.

There is a drawback to using `tmpnam` and `tempnam`: a window exists between the time that the unique pathname is returned and the time that an application creates a file with that name. During this timing window, another process can create a file of the same name. The `tempfile` and `mkstemp` functions should be used instead, as they don't suffer from this problem.

> The `mktemp` function is similar to `mkstemp`, except that it creates a name suitable only for use as a temporary file. The `mktemp` function doesn't create a file, so it suffers from the same drawback as `tmpnam` and `tempnam`. The `mktemp` function is marked as a legacy interface in the Single UNIX Specification. Legacy interfaces might be withdrawn in future versions of the Single UNIX Specification, and so should be avoided.

## 5.14 Alternatives to Standard I/O

The standard I/O library is not perfect. Korn and Vo [1991] list numerous defects: some in the basic design, but most in the various implementations.

One inefficiency inherent in the standard I/O library is the amount of data copying that takes place. When we use the line-at-a-time functions, `fgets` and `fputs`, the data is usually copied twice: once between the kernel and the standard I/O buffer (when the corresponding `read` or `write` is issued) and again between the standard I/O buffer and our line buffer. The Fast I/O library [`fio`(3) in AT&T 1990a] gets around this by having the function that reads a line return a pointer to the line instead of copying the line into another buffer. Hume [1988] reports a threefold increase in the speed of a version of the `grep`(1) utility, simply by making this change.

Korn and Vo [1991] describe another replacement for the standard I/O library: *sfio*. This package is similar in speed to the *fio* library and normally faster than the standard I/O library. The *sfio* package also provides some new features that aren't in the others: I/O streams generalized to represent both files and regions of memory, processing modules that can be written and stacked on an I/O stream to change the operation of a stream, and better exception handling.

Krieger, Stumm, and Unrau [1992] describe another alternative that uses mapped files—the `mmap` function that we describe in Section 14.9. This new package is called ASI, the Alloc Stream Interface. The programming interface resembles the UNIX System memory allocation functions (`malloc`, `realloc`, and `free`, described in Section 7.8). As with the *sfio* package, ASI attempts to minimize the amount of data copying by using pointers.

Several implementations of the standard I/O library are available in C libraries that were designed for systems with small memory footprints, such as embedded systems. These implementations emphasize modest memory requirements over portability, speed, or functionality. Two such implementations are the uClibc C library (see `http://www.uclibc.org` for more information) and the newlibc C library (`http://sources.redhat.com/newlib`).

## 5.15 Summary

The standard I/O library is used by most UNIX applications. We have looked at all the functions provided by this library, as well as at some implementation details and efficiency considerations. Be aware of the buffering that takes place with this library, as this is the area that generates the most problems and confusion.

## Exercises

**5.1** Implement `setbuf` using `setvbuf`.

**5.2** Type in the program that copies a file using line-at-a-time I/O (`fgets` and `fputs`) from Section 5.8, but use a `MAXLINE` of 4. What happens if you copy lines that exceed this length? Explain what is happening.

**5.3** What does a return value of 0 from `printf` mean?

**5.4** The following code works correctly on some machines, but not on others. What could be the problem?

```
#include    <stdio.h>

int
main(void)
{
    char    c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

**5.5** Why does `tempnam` restrict the *prefix* to five characters?

**5.6** How would you use the `fsync` function (Section 3.13) with a standard I/O stream?

**5.7** In the programs in Figures 1.7 and 1.10, the prompt that is printed does not contain a newline, and we don't call `fflush`. What causes the prompt to be output?

# 6

# System Data Files and Information

## 6.1 Introduction

A UNIX system requires numerous data files for normal operation: the password file `/etc/passwd` and the group file `/etc/group` are two files that are frequently used by various programs. For example, the password file is used every time a user logs in to a UNIX system and every time someone executes an `ls -l` command.

Historically, these data files have been ASCII text files and were read with the standard I/O library. But for larger systems, a sequential scan through the password file becomes time consuming. We want to be able to store these data files in a format other than ASCII text, but still provide an interface for an application program that works with any file format. The portable interfaces to these data files are the subject of this chapter. We also cover the system identification functions and the time and date functions.

## 6.2 Password File

The UNIX System's password file, called the user database by POSIX.1, contains the fields shown in Figure 6.1. These fields are contained in a `passwd` structure that is defined in `<pwd.h>`.

Note that POSIX.1 specifies only five of the ten fields in the `passwd` structure. Most platforms support at least seven of the fields. The BSD-derived platforms support all ten.

| Description | struct passwd member | POSIX.1 | FreeBSD 5.2.1 | Linux 2.4.22 | Mac OS X 10.3 | Solaris 9 |
|---|---|---|---|---|---|---|
| user name | char *pw_name | • | • | • | • | • |
| encrypted password | char *pw_passwd | | • | • | • | • |
| numerical user ID | uid_t pw_uid | • | • | • | • | • |
| numerical group ID | gid_t pw_gid | • | • | • | • | • |
| comment field | char *pw_gecos | | • | • | • | • |
| initial working directory | char *pw_dir | • | • | • | • | • |
| initial shell (user program) | char *pw_shell | • | • | • | • | • |
| user access class | char *pw_class | | • | | • | |
| next time to change password | time_t pw_change | | • | | • | |
| account expiration time | time_t pw_expire | | • | | • | |

**Figure 6.1** Fields in /etc/passwd file

Historically, the password file has been stored in /etc/passwd and has been an ASCII file. Each line contains the fields described in Figure 6.1, separated by colons. For example, four lines from the /etc/passwd file on Linux could be

```
root:x:0:0:root:/root:/bin/bash
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Note the following points about these entries.

- There is usually an entry with the user name root. This entry has a user ID of 0 (the superuser).
- The encrypted password field contains a single character as a placeholder where older versions of the UNIX System used to store the encrypted password. Because it is a security hole to store the encrypted password in a file that is readable by everyone, encrypted passwords are now kept elsewhere. We'll cover this issue in more detail in the next section when we discuss passwords.
- Some fields in a password file entry can be empty. If the encrypted password field is empty, it usually means that the user does not have a password. (This is not recommended.) The entry for squid has one blank field: the comment field. An empty comment field has no effect.
- The shell field contains the name of the executable program to be used as the login shell for the user. The default value for an empty shell field is usually /bin/sh. Note, however, that the entry for squid has /dev/null as the login shell. Obviously, this is a device and cannot be executed, so its use here is to prevent anyone from logging in to our system as user squid.

  Many services have separate user IDs for the daemon processes (Chapter 13) that help implement the service. The squid entry is for the processes implementing the squid proxy cache service.

- There are several alternatives to using /dev/null to prevent a particular user from logging in to a system. It is common to see /bin/false used as the login shell. It simply exits with an unsuccessful (nonzero) status; the shell evaluates the exit status as false. It is also common to see /bin/true used to disable an account. All it does is exit with a successful (zero) status. Some systems provide the nologin command. It prints a customizable error message and exits with a nonzero exit status.
- The nobody user name can be used to allow people to log in to a system, but with a user ID (65534) and group ID (65534) that provide no privileges. The only files that this user ID and group ID can access are those that are readable or writable by the world. (This assumes that there are no files specifically owned by user ID 65534 or group ID 65534, which should be the case.)
- Some systems that provide the finger(1) command support additional information in the comment field. Each of these fields is separated by a comma: the user's name, office location, office phone number, and home phone number. Additionally, an ampersand in the comment field is replaced with the login name (capitalized) by some utilities. For example, we could have

  ```
  sar:x:205:105:Steve Rago, SF 5-121, 555-1111, 555-2222:/home/sar:/bin/sh
  ```

  Then we could use finger to print information about Steve Rago.

  ```
  $ finger -p sar
  Login: sar                          Name: Steve Rago
  Directory: /home/sar                Shell: /bin/sh
  Office:  SF 5-121,  555-1111        Home Phone:  555-2222
  On since Mon Jan 19 03:57 (EST) on ttyv0 (messages off)
  No Mail.
  ```

  Even if your system doesn't support the finger command, these fields can still go into the comment field, since that field is simply a comment and not interpreted by system utilities.

Some systems provide the vipw command to allow administrators to edit the password file. The vipw command serializes changes to the password file and makes sure that any additional files are consistent with the changes made. It is also common for systems to provide similar functionality through graphical user interfaces.

POSIX.1 defines only two functions to fetch entries from the password file. These functions allow us to look up an entry given a user's login name or numerical user ID

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name)
```

Both return: pointer if OK, NULL on error

The getpwuid function is used by the ls(1) program to map the numerical user ID contained in an i-node into a user's login name. The getpwnam function is used by the login(1) program when we enter our login name.

Both functions return a pointer to a passwd structure that the functions fill in. This structure is usually a static variable within the function, so its contents are overwritten each time we call either of these functions.

These two POSIX.1 functions are fine if we want to look up either a login name or a user ID, but some programs need to go through the entire password file. The following three functions can be used for this.

```
#include <pwd.h>

struct passwd *getpwent(void);
```

Returns: pointer if OK, NULL on error or end of file

```
void setpwent(void);

void endpwent(void);
```

These three functions are not part of the base POSIX.1 standard. They are defined as XSI extensions in the Single UNIX Specification. As such, all UNIX systems are expected to provide them.

We call getpwent to return the next entry in the password file. As with the two POSIX.1 functions, getpwent returns a pointer to a structure that it has filled in. This structure is normally overwritten each time we call this function. If this is the first call to this function, it opens whatever files it uses. There is no order implied when we use this function; the entries can be in any order, because some systems use a hashed version of the file /etc/passwd.

The function setpwent rewinds whatever files it uses, and endpwent closes these files. When using getpwent, we must always be sure to close these files by calling endpwent when we're through. Although getpwent is smart enough to know when it has to open its files (the first time we call it), it never knows when we're through.

## Example

Figure 6.2 shows an implementation of the function getpwnam.

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
        if (strcmp(name, ptr->pw_name) == 0)
            break;      /* found a match */
    endpwent();
    return(ptr);    /* ptr is NULL if no match found */
}
```

**Figure 6.2** The getpwnam function

The call to setpwent at the beginning is self-defense: we ensure that the files are rewound, in case the caller has already opened them by calling getpwent. The call to endpwent when we're done is because neither getpwnam nor getpwuid should leave any of the files open. □

## 6.3 Shadow Passwords

The encrypted password is a copy of the user's password that has been put through a one-way encryption algorithm. Because this algorithm is one-way, we can't guess the original password from the encrypted version.

Historically, the algorithm that was used (see Morris and Thompson [1979]) always generated 13 printable characters from the 64-character set [a-zA-Z0-9./]. Some newer systems use an MD5 algorithm to encrypt passwords, generating 31 characters per encrypted password. (The more characters used to store the encrypted password, the more combinations there are, and the harder it will be to guess the password by trying all possible variations.) When we place a single character in the encrypted password field, we ensure that an encrypted password will never match this value.

Given an encrypted password, we can't apply an algorithm that inverts it and returns the plaintext password. (The plaintext password is what we enter at the Password: prompt.) But we could guess a password, run it through the one-way algorithm, and compare the result to the encrypted password. If user passwords were randomly chosen, this brute-force approach wouldn't be too successful. Users, however, tend to choose nonrandom passwords, such as spouse's name, street names, or pet names. A common experiment is for someone to obtain a copy of the password file and try guessing the passwords. (Chapter 4 of Garfinkel et al. [2003] contains additional details and history on passwords and the password encryption scheme used on UNIX systems.)

To make it more difficult to obtain the raw materials (the encrypted passwords), systems now store the encrypted password in another file, often called the *shadow password file*. Minimally, this file has to contain the user name and the encrypted password. Other information relating to the password is also stored here (Figure 6.3).

| Description | struct spwd member |
|---|---|
| user login name | char *sp_namp |
| encrypted password | char *sp_pwdp |
| days since Epoch of last password change | int sp_lstchg |
| days until change allowed | int sp_min |
| days before change required | int sp_max |
| days warning for expiration | int sp_warn |
| days before account inactive | int sp_inact |
| days since Epoch when account expires | int sp_expire |
| reserved | unsigned int sp_flag |

**Figure 6.3** Fields in /etc/shadow file

The only two mandatory fields are the user's login name and encrypted password. The other fields control how often the password is to change—known as "password aging"—and how long an account is allowed to remain active.

The shadow password file should not be readable by the world. Only a few programs need to access encrypted passwords—login(1) and passwd(1), for example—and these programs are often set-user-ID root. With shadow passwords, the regular password file, /etc/passwd, can be left readable by the world.

On Linux 2.4.22 and Solaris 9, a separate set of functions is available to access the shadow password file, similar to the set of functions used to access the password file.

```
#include <shadow.h>

struct spwd *getspnam(const char *name);

struct spwd *getspent(void);

                              Both return: pointer if OK, NULL on error

void setspent(void);

void endspent(void);
```

On FreeBSD 5.2.1 and Mac OS X 10.3, there is no shadow password structure. The additional account information is stored in the password file (refer back to Figure 6.1).

## 6.4 Group File

The UNIX System's group file, called the group database by POSIX.1, contains the fields shown in Figure 6.4. These fields are contained in a group structure that is defined in <grp.h>.

| Description | struct group member | POSIX.1 | FreeBSD 5.2.1 | Linux 2.4.22 | Mac OS X 10.3 | Solaris 9 |
|---|---|---|---|---|---|---|
| group name | char *gr_name | • | • | • | • | • |
| encrypted password | char *gr_passwd | | • | • | • | • |
| numerical group ID | int gr_gid | • | • | • | • | • |
| array of pointers to individual user names | char **gr_mem | • | • | • | • | • |

**Figure 6.4** Fields in /etc/group file

The field gr_mem is an array of pointers to the user names that belong to this group. This array is terminated by a null pointer.

We can look up either a group name or a numerical group ID with the following two functions, which are defined by POSIX.1.

```
#include <grp.h>

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);

                              Both return: pointer if OK, NULL on error
```

As with the password file functions, both of these functions normally return pointers to a `static` variable, which is overwritten on each call.

If we want to search the entire group file, we need some additional functions. The following three functions are like their counterparts for the password file.

```
#include <grp.h>

struct group *getgrent(void);

                    Returns: pointer if OK, NULL on error or end of file

void setgrent(void);

void endgrent(void);
```

These three functions are not part of the base POSIX.1 standard. They are defined as XSI extensions in the Single UNIX Specification. All UNIX Systems provide them.

The `setgrent` function opens the group file, if it's not already open, and rewinds it. The `getgrent` function reads the next entry from the group file, opening the file first, if it's not already open. The `endgrent` function closes the group file.

## 6.5 Supplementary Group IDs

The use of groups in the UNIX System has changed over time. With Version 7, each user belonged to a single group at any point in time. When we logged in, we were assigned the real group ID corresponding to the numerical group ID in our password file entry. We could change this at any point by executing newgrp(1). If the `newgrp` command succeeded (refer to the manual page for the permission rules), our real group ID was changed to the new group's ID, and this was used for all subsequent file access permission checks. We could always go back to our original group by executing `newgrp` without any arguments.

This form of group membership persisted until it was changed in 4.2BSD (circa 1983). With 4.2BSD, the concept of supplementary group IDs was introduced. Not only did we belong to the group corresponding to the group ID in our password file entry, but we also could belong to up to 16 additional groups. The file access permission checks were modified so that not only was the effective group ID compared to the file's group ID, but also all the supplementary group IDs were compared to the file's group ID.

Supplementary group IDs are a required feature of POSIX.1. (In older versions of POSIX.1, they were optional.) The constant NGROUPS_MAX (Figure 2.10) specifies the number of supplementary group IDs. A common value is 16 (Figure 2.14).

The advantage in using supplementary group IDs is that we no longer have to change groups explicitly. It is not uncommon to belong to multiple groups (i.e., participate in multiple projects) at the same time.

Three functions are provided to fetch and set the supplementary group IDs.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Returns: number of supplementary group IDs if OK, –1 on error

```
#include <grp.h>     /* on Linux */
#include <unistd.h> /* on FreeBSD, Mac OS X, and Solaris */

int setgroups(int ngroups, const gid_t grouplist[]);

#include <grp.h>     /* on Linux and Solaris */
#include <unistd.h> /* on FreeBSD and Mac OS X */

int initgroups(const char *username, gid_t basegid);
```

Both return: 0 if OK, –1 on error

Of these three functions, only `getgroups` is specified by POSIX.1. Because `setgroups` and `initgroups` are privileged operations, they are not part of POSIX.1. All four platforms covered in this book, however, support all three functions.

On Mac OS X 10.3, *basegid* is declared to be of type `int`.

The `getgroups` function fills in the array *grouplist* with the supplementary group IDs. Up to *gidsetsize* elements are stored in the array. The number of supplementary group IDs stored in the array is returned by the function.

As a special case, if *gidsetsize* is 0, the function returns only the number of supplementary group IDs. The array *grouplist* is not modified. (This allows the caller to determine the size of the *grouplist* array to allocate.)

The `setgroups` function can be called by the superuser to set the supplementary group ID list for the calling process: *grouplist* contains the array of group IDs, and *ngroups* specifies the number of elements in the array. The value of *ngroups* cannot be larger than NGROUPS_MAX.

The only use of `setgroups` is usually from the `initgroups` function, which reads the entire group file—with the functions `getgrent`, `setgrent`, and `endgrent`, which we described earlier—and determines the group membership for *username*. It then calls `setgroups` to initialize the supplementary group ID list for the user. One must be superuser to call `initgroups`, since it calls `setgroups`. In addition to finding all the groups that *username* is a member of in the group file, `initgroups` also includes *basegid* in the supplementary group ID list; *basegid* is the group ID from the password file for *username*.

The `initgroups` function is called by only a few programs: the `login(1)` program, for example, calls it when we log in.

## 6.6 Implementation Differences

We've already discussed the shadow password file supported by Linux and Solaris. FreeBSD and Mac OS X store encrypted passwords differently. Figure 6.5 summarizes how the four platforms covered in this book store user and group information.

| Information | FreeBSD 5.2.1 | Linux 2.4.22 | Mac OS X 10.3 | Solaris 9 |
|---|---|---|---|---|
| Account information | /etc/passwd | /etc/passwd | netinfo | /etc/passwd |
| Encrypted passwords | /etc/master.passwd | /etc/shadow | netinfo | /etc/shadow |
| Hashed password files? | yes | no | no | no |
| Group information | /etc/group | /etc/group | netinfo | /etc/group |

**Figure 6.5** Account implementation differences

On FreeBSD, the shadow password file is /etc/master.passwd. Special commands are used to edit it, which in turn generate a copy of /etc/passwd from the shadow password file. In addition, hashed versions of the files are also generated: /etc/pwd.db is the hashed version of /etc/passwd, and /etc/spwd.db is the hashed version of /etc/master.passwd. These provide better performance for large installations.

On Mac OS X, however, /etc/passwd and /etc/master.passwd are used only in single-user mode (when the system is undergoing maintenance; single-user mode usually means that no system services are enabled). In multiuser mode—during normal operation—the netinfo directory service provides access to account information for users and groups.

Although Linux and Solaris support similar shadow password interfaces, there are some subtle differences. For example, the integer fields shown in Figure 6.3 are defined as type int on Solaris, but as long int on Linux. Another difference is the account-inactive field. Solaris defines it to be the number of days since the user last logged in to the system, whereas Linux defines it to be the number of days after which the maximum password age has been reached.

On many systems, the user and group databases are implemented using the Network Information Service (NIS). This allows administrators to edit a master copy of the databases and distribute them automatically to all servers in an organization. Client systems contact servers to look up information about users and groups. NIS+ and the Lightweight Directory Access Protocol (LDAP) provide similar functionality. Many systems control the method used to administer each type of information through the /etc/nsswitch.conf configuration file.

## 6.7 Other Data Files

We've discussed only two of the system's data files so far: the password file and the group file. Numerous other files are used by UNIX systems in normal day-to-day operation. For example, the BSD networking software has one data file for the services

provided by the various network servers (/etc/services), one for the protocols (/etc/protocols), and one for the networks (/etc/networks). Fortunately, the interfaces to these various files are like the ones we've already described for the password and group files.

The general principle is that every data file has at least three functions:

1. A get function that reads the next record, opening the file if necessary. These functions normally return a pointer to a structure. A null pointer is returned when the end of file is reached. Most of the get functions return a pointer to a static structure, so we always have to copy it if we want to save it.
2. A set function that opens the file, if not already open, and rewinds the file. This function is used when we know we want to start again at the beginning of the file.
3. An end entry that closes the data file. As we mentioned earlier, we always have to call this when we're done, to close all the files.

Additionally, if the data file supports some form of keyed lookup, routines are provided to search for a record with a specific key. For example, two keyed lookup routines are provided for the password file: getpwnam looks for a record with a specific user name, and getpwuid looks for a record with a specific user ID.

Figure 6.6 shows some of these routines, which are common to UNIX systems. In this figure, we show the functions for the password files and group file, which we discussed earlier in this chapter, and some of the networking functions. There are get, set, and end functions for all the data files in this figure.

| Description | Data file | Header | Structure | Additional keyed lookup functions |
|---|---|---|---|---|
| passwords | /etc/passwd | <pwd.h> | passwd | getpwnam, getpwuid |
| groups | /etc/group | <grp.h> | group | getgrnam, getgrgid |
| shadow | /etc/shadow | <shadow.h> | spwd | getspnam |
| hosts | /etc/hosts | <netdb.h> | hostent | gethostbyname, gethostbyaddr |
| networks | /etc/networks | <netdb.h> | netent | getnetbyname, getnetbyaddr |
| protocols | /etc/protocols | <netdb.h> | protoent | getprotobyname, getprotobynumber |
| services | /etc/services | <netdb.h> | servent | getservbyname, getservbyport |

**Figure 6.6** Similar routines for accessing system data files

Under Solaris, the last four data files in Figure 6.6 are symbolic links to files of the same name in the directory /etc/inet. Most UNIX System implementations have additional functions that are like these, but the additional functions tend to deal with system administration files and are specific to each implementation.

## 6.8 Login Accounting

Two data files that have been provided with most UNIX systems are the utmp file, which keeps track of all the users currently logged in, and the wtmp file, which keeps

track of all logins and logouts. With Version 7, one type of record was written to both files, a binary record consisting of the following structure:

```
struct utmp {
  char  ut_line[8]; /* tty line: "ttyh0", "ttyd0", "ttyp0", ... */
  char  ut_name[8]; /* login name */
  long  ut_time;    /* seconds since Epoch */
};
```

On login, one of these structures was filled in and written to the utmp file by the login program, and the same structure was appended to the wtmp file. On logout, the entry in the utmp file was erased—filled with null bytes—by the init process, and a new entry was appended to the wtmp file. This logout entry in the wtmp file had the ut_name field zeroed out. Special entries were appended to the wtmp file to indicate when the system was rebooted and right before and after the system's time and date was changed. The who(1) program read the utmp file and printed its contents in a readable form. Later versions of the UNIX System provided the last(1) command, which read through the wtmp file and printed selected entries.

Most versions of the UNIX System still provide the utmp and wtmp files, but as expected, the amount of information in these files has grown. The 20-byte structure that was written by Version 7 grew to 36 bytes with SVR2, and the extended utmp structure with SVR4 takes over 350 bytes!

> The detailed format of these records in Solaris is given in the utmpx(4) manual page. With Solaris 9, both files are in the /var/adm directory. Solaris provides numerous functions described in getutx(3) to read and write these two files.
>
> On FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3, the utmp(5) manual page gives the format of their versions of these login records. The pathnames of these two files are /var/run/utmp and /var/log/wtmp.

## 6.9 System Identification

POSIX.1 defines the uname function to return information on the current host and operating system.

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

int uname(struct utsname *name);
```

Returns: non-negative value if OK, −1 on error

We pass the address of a utsname structure, and the function fills it in. POSIX.1 defines only the minimum fields in the structure, which are all character arrays, and it's up to each implementation to set the size of each array. Some implementations provide additional fields in the structure.

```
struct utsname {
  char  sysname[];    /* name of the operating system */
  char  nodename[];   /* name of this node */
  char  release[];    /* current release of operating system */
  char  version[];    /* current version of this release */
  char  machine[];    /* name of hardware type */
};
```

Each string is null-terminated. The maximum name lengths supported by the four platforms discussed in this book are listed in Figure 6.7. The information in the `utsname` structure can usually be printed with the `uname(1)` command.

> POSIX.1 warns that the `nodename` element may not be adequate to reference the host on a communications network. This function is from System V, and in older days, the `nodename` element was adequate for referencing the host on a UUCP network.
>
> Realize also that the information in this structure does not give any information on the POSIX.1 level. This should be obtained using `_POSIX_VERSION`, as described in Section 2.6.
>
> Finally, this function gives us a way only to fetch the information in the structure; there is nothing specified by POSIX.1 about initializing this information.

Historically, BSD-derived systems provide the `gethostname` function to return only the name of the host. This name is usually the name of the host on a TCP/IP network.

```
#include <unistd.h>

int gethostname(char *name, int namelen);

                                        Returns: 0 if OK, -1 on error
```

The *namelen* argument specifies the size of the *name* buffer. If enough space is provided, the string returned through *name* is null terminated. If insufficient room is provided, however, it is unspecified whether the string is null terminated.

The `gethostname` function, now defined as part of POSIX.1, specifies that the maximum host name length is `HOST_NAME_MAX`. The maximum name lengths supported by the four implementations covered in this book are summarized in Figure 6.7.

| Interface | Maximum name length | | | |
|---|---|---|---|---|
| | FreeBSD 5.2.1 | Linux 2.4.22 | Mac OS X 10.3 | Solaris 9 |
| `uname` | 256 | 65 | 256 | 257 |
| `gethostname` | 256 | 64 | 256 | 256 |

**Figure 6.7** System identification name limits

If the host is connected to a TCP/IP network, the host name is normally the fully qualified domain name of the host.

There is also a hostname(1) command that can fetch or set the host name. (The host name is set by the superuser using a similar function, sethostname.) The host name is normally set at bootstrap time from one of the start-up files invoked by /etc/rc or init.

## 6.10 Time and Date Routines

The basic time service provided by the UNIX kernel counts the number of seconds that have passed since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). In Section 1.10, we said that these seconds are represented in a time_t data type, and we call them *calendar times*. These calendar times represent both the time and the date. The UNIX System has always differed from other operating systems in (a) keeping time in UTC instead of the local time, (b) automatically handling conversions, such as daylight saving time, and (c) keeping the time and date as a single quantity.

The time function returns the current time and date.

```
#include <time.h>

time_t time(time_t *calptr);
```

Returns: value of time if OK, –1 on error

The time value is always returned as the value of the function. If the argument is non-null, the time value is also stored at the location pointed to by *calptr*.

> We haven't said how the kernel's notion of the current time is initialized. Historically, on implementations derived from System V, the stime(2) function was called, whereas BSD-derived systems used settimeofday(2).
>
> The Single UNIX Specification doesn't specify how a system sets its current time.

The gettimeofday function provides greater resolution (up to a microsecond) than the time function. This is important for some applications.

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

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

Returns: 0 always

This function is defined as an XSI extension in the Single UNIX Specification. The only legal value for *tzp* is NULL; other values result in unspecified behavior. Some platforms support the specification of a time zone through the use of *tzp*, but this is implementation-specific and not defined by the Single UNIX Specification.

The gettimeofday function stores the current time as measured from the Epoch in the memory pointed to by *tp*. This time is represented as a timeval structure, which stores seconds and microseconds:

```
struct timeval {
        time_t tv_sec;     /* seconds */
        long   tv_usec;    /* microseconds */
};
```

Once we have the integer value that counts the number of seconds since the Epoch, we normally call one of the other time functions to convert it to a human-readable time and date. Figure 6.8 shows the relationships between the various time functions.

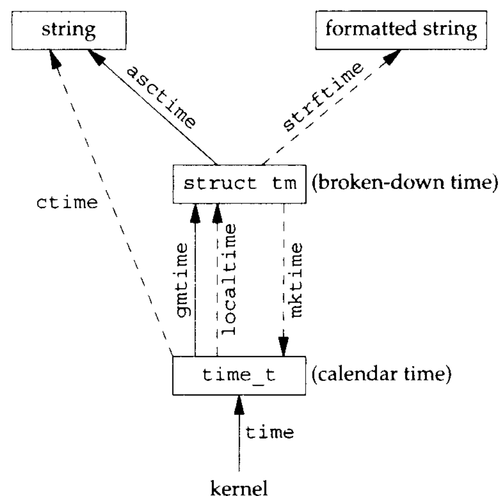


**Figure 6.8** Relationship of the various time functions

(The four functions in this figure that are shown with dashed lines—`localtime`, `mktime`, `ctime`, and `strftime`—are all affected by the `TZ` environment variable, which we describe later in this section.)

The two functions `localtime` and `gmtime` convert a calendar time into what's called a broken-down time, a `tm` structure.

```
struct tm {       /* a broken-down time */
  int  tm_sec;    /* seconds after the minute: [0 - 60] */
  int  tm_min;    /* minutes after the hour: [0 - 59] */
  int  tm_hour;   /* hours after midnight: [0 - 23] */
  int  tm_mday;   /* day of the month: [1 - 31] */
  int  tm_mon;    /* months since January: [0 - 11] */
  int  tm_year;   /* years since 1900 */
  int  tm_wday;   /* days since Sunday: [0 - 6] */
  int  tm_yday;   /* days since January 1: [0 - 365] */
  int  tm_isdst;  /* daylight saving time flag: <0, 0, >0 */
};
```

The reason that the seconds can be greater than 59 is to allow for a leap second. Note that all the fields except the day of the month are 0-based. The daylight saving time flag is positive if daylight saving time is in effect, 0 if it's not in effect, and negative if the information isn't available.

In previous versions of the Single UNIX Specification, double leap seconds were allowed. Thus, the valid range of values for the tm_sec member was 0–61. The formal definition of UTC doesn't allow for double leap seconds, so the valid range for seconds is now defined to be 0–60.

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

Both return: pointer to broken-down time

The difference between localtime and gmtime is that the first converts the calendar time to the local time, taking into account the local time zone and daylight saving time flag, whereas the latter converts the calendar time into a broken-down time expressed as UTC.

The function mktime takes a broken-down time, expressed as a local time, and converts it into a time_t value.

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

Returns: calendar time if OK, –1 on error

The asctime and ctime functions produce the familiar 26-byte string that is similar to the default output of the date(1) command:

```
Tue Feb 10 18:27:38 2004\n\0
```

```
#include <time.h>

char *asctime(const struct tm *tmptr);

char *ctime(const time_t *calptr);
```

Both return: pointer to null-terminated string

The argument to asctime is a pointer to a broken-down string, whereas the argument to ctime is a pointer to a calendar time.

The final time function, strftime, is the most complicated. It is a printf-like function for time values.

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr);
```

Returns: number of characters stored in array if room, 0 otherwise

The final argument is the time value to format, specified by a pointer to a broken-down time value. The formatted result is stored in the array *buf* whose size is *maxsize* characters. If the size of the result, including the terminating null, fits in the buffer, the function returns the number of characters stored in *buf*, excluding the terminating null. Otherwise, the function returns 0.

The *format* argument controls the formatting of the time value. Like the `printf` functions, conversion specifiers are given as a percent followed by a special character. All other characters in the *format* string are copied to the output. Two percents in a row generate a single percent in the output. Unlike the `printf` functions, each conversion specified generates a different fixed-size output string—there are no field widths in the *format* string. Figure 6.9 describes the 37 ISO C conversion specifiers. The third column of this figure is from the output of `strftime` under Linux, corresponding to the time and date `Tue Feb 10 18:27:38 EST 2004`.

The only specifiers that are not self-evident are `%U`, `%V`, and `%W`. The `%U` specifier represents the week number of the year, where the week containing the first Sunday is week 1. The `%W` specifier represents the week number of the year, where the week containing the first Monday is week 1. The `%V` specifier is different. If the week containing the first day in January has four or more days in the new year, then this is treated as week 1. Otherwise, it is treated as the last week of the previous year. In both cases, Monday is treated as the first day of the week.

As with `printf`, `strftime` supports modifiers for some of the conversion specifiers. The `E` and `O` modifiers can be used to generate an alternate format if supported by the locale.

Some systems support additional, nonstandard extensions to the *format* string for `strftime`.

We mentioned that the four functions in Figure 6.8 with dashed lines were affected by the `TZ` environment variable: `localtime`, `mktime`, `ctime`, and `strftime`. If defined, the value of this environment variable is used by these functions instead of the default time zone. If the variable is defined to be a null string, such as `TZ=`, then UTC is normally used. The value of this environment variable is often something like `TZ=EST5EDT`, but POSIX.1 allows a much more detailed specification. Refer to the Environment Variables chapter of the Single UNIX Specification [Open Group 2004] for all the details on the `TZ` variable.

All the time and date functions described in this section, except `gettimeofday`, are defined by the ISO C standard. POSIX.1, however, added the `TZ` environment variable. On FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3, more information on the `TZ` variable can be found in the `tzset`(3) manual page. On Solaris 9, this information is in the `environ`(5) manual page.