4

Files and Directories

4.1 Introduction

In the previous chapter we covered the basic functions that perform I/O. The discussion centered on 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`, `fstatat`, and `lstat` Functions

The discussion in this chapter centers on the four `stat` functions and the information they return.

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

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

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

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

int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);
```

All four 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 *fd*. 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 4.22 when we walk down a directory hierarchy. We describe symbolic links in more detail in Section 4.17.)

The `fstatat` function provides a way to return the file statistics for a pathname relative to an open directory represented by the *fd* argument. The *flag* argument controls whether symbolic links are followed; when the `AT_SYMLINK_NOFOLLOW` flag is set, `fstatat` will not follow symbolic links, but rather returns information about the link itself. Otherwise, the default is to follow symbolic links, returning information about the file to which the symbolic link points. If the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is a relative pathname, then `fstatat` evaluates the *pathname* argument relative to the current directory. If the *pathname* argument is an absolute pathname, then the *fd* argument is ignored. In these two cases, `fstatat` behaves like either `stat` or `lstat`, depending on the value of *flag*.

The *buf* argument is a pointer to a structure that we must supply. The functions fill in the structure. 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 */
  struct timespec st_atim;    /* time of last access */
  struct timespec st_mtim;    /* time of last modification */
  struct timespec st_ctim;    /* 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 part of the XSI option in the Single UNIX Specification.

The `timespec` structure type defines time in terms of seconds and nanoseconds. It includes at least the following fields:

```
time_t tv_sec;
long   tv_nsec;
```

Prior to the 2008 version of the standard, the time fields were named `st_atime`, `st_mtime`, and `st_ctime`, and were of type `time_t` (expressed in seconds). The `timespec` structure enables higher-resolution timestamps. The old names can be defined in terms of the `tv_sec` members for compatibility. For example, `st_atime` can be defined as `st_atim.tv_sec`.

Note that most members of the `stat` structure are 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.

   > Note that FreeBSD no longer supports block special files. All access to devices is through the character special interface.

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 4.17.

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/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: 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 prompted 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. □

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 shown in Section 4.22.

File type	Count	Percentage
regular file	415,803	79.77 %
directory	62,197	11.93
symbolic link	40,018	8.25
character special	155	0.03
block special	47	0.01
socket	45	0.01
FIFO	0	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 real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID 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, respectively, 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 as of the 2001 version of POSIX.1. They were 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. However, we can also 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, we can set another bit 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`, respectively.

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 of only regular files as having access permissions.

There are nine permission bits for each file, divided into three categories. They 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 seven 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 which 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 4.21 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 8.0 and Mac OS X 10.6.8 always copy the new file's group ID from the directory. Several Linux file systems allow the choice between the two options to be selected using a `mount`(1) command option. The default behavior for Linux 3.2.0 and Solaris 10 is to determine the group ID of a new file depending on whether the set-group-ID bit is set for the directory in which the file is created. If this bit is set, the new file's group ID is copied from the directory; otherwise, the new file's group ID is set to the effective group ID of the process.

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

As we mentioned earlier, this option for group ownership is the default for FreeBSD 8.0 and Mac OS X 10.6.8, but an option for Linux and Solaris. Under Solaris 10, and by default under Linux 3.2.0, 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 4.21.)

4.7 `access` and `faccessat` Functions

As we described earlier, when we open a file, the kernel performs its access tests based on the effective user and group IDs. Sometimes, however, 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 might still want to verify that the real user can access a given file. The `access` and `faccessat` functions base their 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);

int faccessat(int fd, const char *pathname, int mode, int flag);
                                                    Both return: 0 if OK, –1 on error
```

The *mode* is either the value `F_OK` to test if a file exists, or the bitwise OR of any of the flags 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

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

The `faccessat` function behaves like `access` when the *pathname* argument is absolute or when the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is relative. Otherwise, `faccessat` evaluates the *pathname* relative to the open directory referenced by the *fd* argument.

The *flag* argument can be used to change the behavior of faccessat. If the AT_EACCESS flag is set, the access checks are made using the effective user and group IDs of the calling process instead of the real user and group IDs.

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 4.21. 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 both before we run the program and after it completes. 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. This 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 umask command support a symbolic mode of operation. 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, fchmod, and fchmodat Functions

The chmod, fchmod, and fchmodat 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 fd, mode_t mode);

int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

All three 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. The fchmodat function behaves like chmod when the *pathname* argument is absolute or when the *fd* argument has the value AT_FDCWD and the *pathname* argument is relative. Otherwise, fchmodat evaluates the *pathname* relative to the open directory referenced by the *fd* argument. The *flag* argument can be used to change the behavior of fchmodat—when the AT_SYMLINK_NOFOLLOW flag is set, fchmodat doesn't follow symbolic links.

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 in the XSI option 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 behavior 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 4.19 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.) To prevent malicious users from setting the sticky bit and adversely affecting system performance, only the superuser can set the sticky bit of a regular file.

  In FreeBSD 8.0 and Solaris 10, only the superuser can set the sticky bit on a regular file. Linux 3.2.0 and Mac OS X 10.6.8 place no such restriction on the setting of the sticky bit, because the bit has no meaning when applied to regular files on these systems. Although the bit also has no meaning when applied to regular files on FreeBSD, everyone except the superuser is prevented from setting it on a regular file.

- The group ID of a newly created file might potentially be 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 8.0 fails an attempt to set the set-group-ID in this case. The other systems silently turn the bit off, but don't fail the attempt to change the file access permissions.

  FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 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.) The program would then load into memory more quickly the next time it was executed, because the swap area was handled as a contiguous file, as 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 meets one of the following criteria:

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

The directories `/tmp` and `/var/tmp` 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 part of the XSI option defined in the Single UNIX Specification, and is supported by FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10.
>
> Solaris 10 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, fchownat, and lchown Functions

The `chown` functions allow us to change a file's user ID and group ID, but if either of the arguments *owner* or *group* is –1, the corresponding ID is left unchanged.

```
#include <unistd.h>

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

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

int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,
             int flag);

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

All four return: 0 if OK, –1 on error

These four functions operate similarly unless the referenced file is a symbolic link. In that case, `lchown` and `fchownat` (with the `AT_SYMLINK_NOFOLLOW` flag set) change the owners of the symbolic link itself, not the file pointed to by the symbolic link.

The `fchown` function changes the ownership of the open file referenced by the *fd* argument. Since it operates on a file that is already open, it can't be used to change the ownership of a symbolic link.

The `fchownat` function behaves like either `chown` or `lchown` when the *pathname* argument is absolute or when the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is relative. In these cases, `fchownat` acts like `lchown` if the `AT_SYMLINK_NOFOLLOW` flag is set in the *flag* argument, or it acts like `chown` if the `AT_SYMLINK_NOFOLLOW` flag is clear. When the *fd* argument is set to the file descriptor of an open directory and the *pathname* argument is a relative pathname, `fchownat` evaluates the *pathname* relative to the open directory.

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 all users 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 10, this functionality is a configuration option, whose default value is to enforce the restriction. FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 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 your 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.

> FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 also define 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 4.22.

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`. Use of 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 slightly more than 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). Obviously, this file has many holes.

> The `du` command on many BSD-derived systems reports the number of 1,024-byte blocks. Solaris reports the number of 512-byte blocks. On Linux, the units reported depend on the whether the `POSIXLY_CORRECT` environment is set. When it is set, the `du` command reports 1,024-byte block units; when it is not set, the command reports 512-byte block units.

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 command, 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 × 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]), Section 15.2 of McDougall and Mauro [2007], and Chapter 12 in Singh [2006] for additional details on the physical layout of files.

4.13 File Truncation

Sometimes 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 fd, 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. Otherwise, if the previous size was less than *length*, the file size will increase and the 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).

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 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 2.20. `UFS` is based on the Berkeley fast file system, which we describe in this section.

> Each file system type has its own characteristic features—and some of these features can be confusing. For example, most UNIX file systems support case-sensitive filenames. Thus, if you create one file named `file.txt` and another named `file.TXT`, then two distinct files are created. On Mac OS X, however, the `HFS` file system is case-preserving with case-insensitive comparisons. Thus, if you create `file.txt`, when you try to create `file.TXT`, you will overwrite `file.txt`. However, only the name used when the file was created is stored in the file system (the case-preserving aspect). In fact, any permutation of uppercase and lowercase letters in the sequence `f`, `i`, `l`, `e`, `.`, `t`, `x`, `t` will match when searching for the file (the case-insensitive comparison aspect). As a consequence, besides `file.txt` and `file.TXT`, we can access the file with the names `File.txt`, `fILE.tXt`, and `FiLe.TxT`.

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

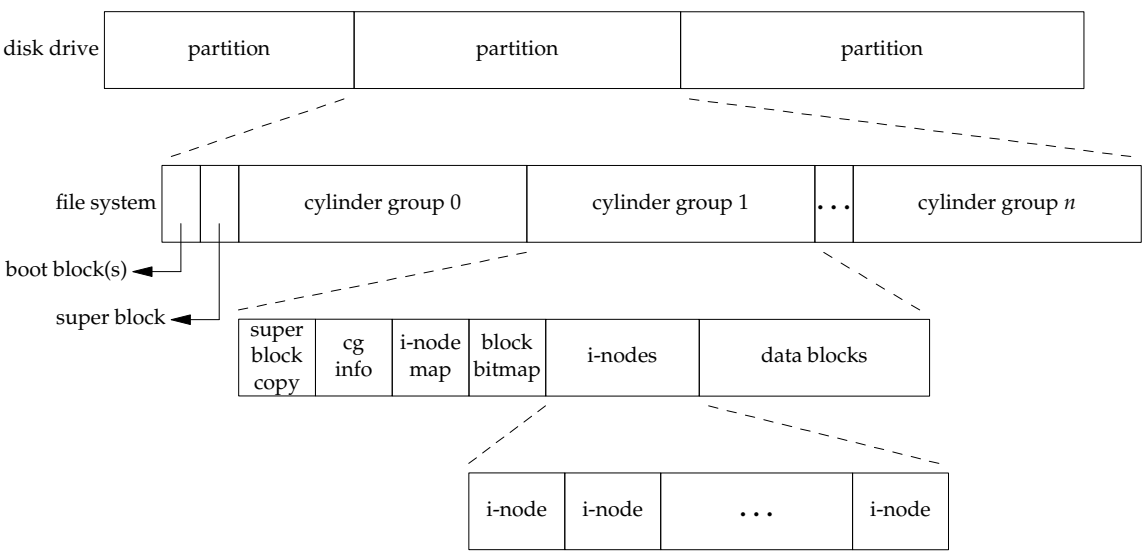

Figure 4.13 Disk drive, partitions, and a file system

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

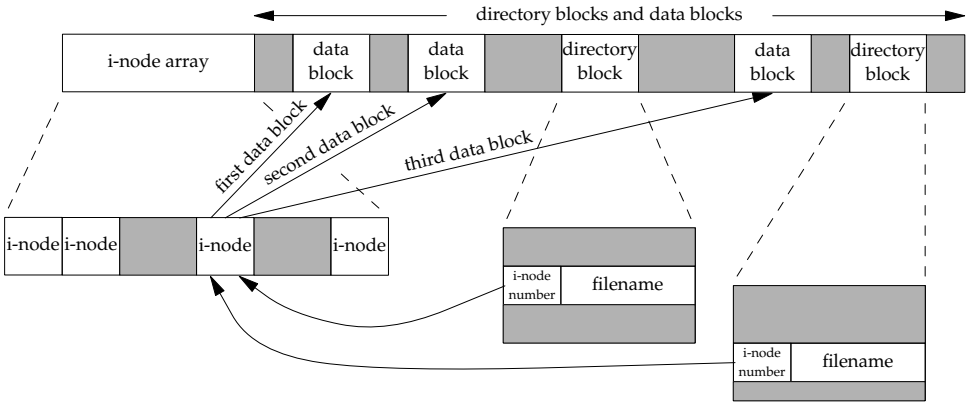

Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

Note the following points from Figure 4.14.

- Two directory entries point to the same i-node entry. Every i-node has a link count that contains the number of directory entries that point to it. Only when the link count goes to 0 can the file be deleted (thereby releasing the data blocks associated with the file). 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, a directory entry can't refer 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.

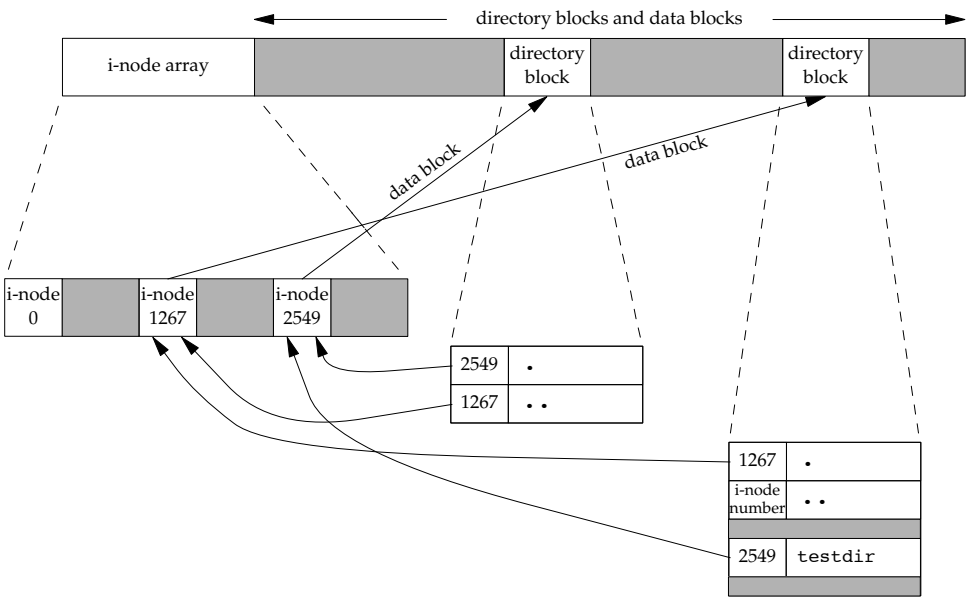

Figure 4.15 Sample cylinder group after creating the directory `testdir`

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 comes 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. We know that this link count is greater than or equal to 3 because, at a minimum, the i-node 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 15 of McDougall and Mauro [2007] for details on `UFS`, the Solaris version of the Berkeley fast file system. For information on the `HFS` file system format used in Mac OS X, see Chapter 12 of Singh [2006].

4.15 `link`, `linkat`, `unlink`, `unlinkat`, and `remove` Functions

As we saw in the previous section, a file can have multiple directory entries pointing to its i-node. We can use either the `link` function or the `linkat` function to create a link to an existing file.

```
#include <unistd.h>

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

int linkat(int efd, const char *existingpath, int nfd, const char *newpath,
           int flag);
```

Both return: 0 if OK, –1 on error

These functions create 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.

With the `linkat` function, the existing file is specified by both the *efd* and *existingpath* arguments, and the new pathname is specified by both the *nfd* and *newpath* arguments. By default, if either pathname is relative, it is evaluated relative to the corresponding file descriptor. If either file descriptor is set to `AT_FDCWD`, then the corresponding pathname, if it is a relative pathname, is evaluated relative to the current directory. If either pathname is absolute, then the corresponding file descriptor argument is ignored.

When the existing file is a symbolic link, the *flag* argument controls whether the `linkat` function creates a link to the symbolic link or to the file to which the symbolic link points. If the `AT_SYMLINK_FOLLOW` flag is set in the *flag* argument, then a link is created to the target of the symbolic link. If this flag is clear, then a link is created to the symbolic link itself.

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. This constraint exists because such hard links 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 4.17.) 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);

int unlinkat(int fd, const char *pathname, int flag);
```

Both return: 0 if OK, –1 on error

These functions remove the directory entry and decrement 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.

As mentioned earlier, 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, as mentioned in Section 4.10, if the sticky bit is set in this directory we must have write permission for the directory and meet one of the following criteria:

- 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.

If the *pathname* argument is a relative pathname, then the `unlinkat` function evaluates the pathname relative to the directory represented by the *fd* file descriptor argument. If the *fd* argument is set to the value `AT_FDCWD`, then the pathname is evaluated relative to the current working directory of the calling process. If the *pathname* argument is an absolute pathname, then the *fd* argument is ignored.

The *flag* argument gives callers a way to change the default behavior of the `unlinkat` function. When the `AT_REMOVEDIR` flag is set, then the `unlinkat` function can be used to remove a directory, similar to using `rmdir`. If this flag is clear, then `unlinkat` operates like `unlink`.

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 if the file system supports it, but the function rmdir should be used instead to unlink a directory. We describe the rmdir function in Section 4.21.

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.

4.16 rename and renameat Functions

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

```
#include <stdio.h>

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

int renameat(int oldfd, const char *oldname, int newfd,
             const char *newname);
```

Both return: 0 if OK, –1 on error

The rename 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 for these functions, 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 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, because 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. We can't rename dot or dot-dot. More precisely, neither dot nor dot-dot can appear as the last component of *oldname* or *newname*.
5. As a special case, if *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*.

The `renameat` function provides the same functionality as the `rename` function, except when either *oldname* or *newname* refers to a relative pathname. If *oldname* specifies a relative pathname, it is evaluated relative to the directory referenced by *oldfd*. Similarly, *newname* is evaluated relative to the directory referenced by *newfd* if *newname* specifies a relative pathname. Either the *oldfd* or *newfd* arguments (or both) can be set to `AT_FDCWD` to evaluate the corresponding pathname relative to the current directory.

4.17 Symbolic Links

A symbolic link is an indirect pointer to a file, unlike the hard links described in 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 (when supported by the underlying file system).

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.

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` do not appear 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 function that returns the file descriptor (usually `open`) handles the symbolic link. Historically, implementations have differed in whether `chown` follows symbolic links. In all modern systems, however, `chown` does follow symbolic links.

Symbolic links were introduced with 4.2BSD. Initially, `chown` didn't follow symbolic links, but this behavior was changed in 4.4BSD. System V included support for symbolic links in SVR4, but diverged from the original BSD behavior by implementing `chown` to follow symbolic links. 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 8.0, Mac OS X 10.6.8, and Solaris 10, `chown` follows symbolic links. All of these platforms provide implementations of `lchown` to change the ownership of symbolic links themselves.

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

One exception to the behavior summarized in Figure 4.17 occurs 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.

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.

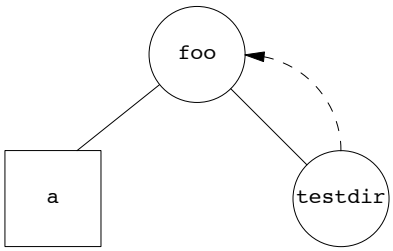

Figure 4.18 Symbolic link `testdir` that creates a loop

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)
```

In Section 4.22, we provide our own version of the `ftw` function that uses `lstat` instead of `stat`, to prevent it from following symbolic links.

> Note that on Linux, the `ftw` and `nftw` functions record all directories seen and avoid processing a directory more than once, so they don't display this behavior.

A loop of this form is easy to remove. We can `unlink` the file `foo/testdir`, as `unlink` does not follow a symbolic link. But if we create a hard link that forms a loop of this type, its removal is much more difficult. This is why the `link` function will not form a hard link to a directory unless the process has superuser privileges.

> Indeed, Rich Stevens did this on his own system as an experiment while writing the original version of this section. The file system got corrupted and the normal `fsck`(1) utility couldn't fix things. The deprecated tools `clri`(8) and `dcheck`(8) were needed to repair the file system.
>
> The need for hard links to directories has long since passed. With symbolic links and the `mkdir` function, there is no longer any need for users to create hard links to directories.

When we open a file, if the pathname passed to `open` specifies a symbolic link, `open` follows the link to the specified file. If the file pointed to by the symbolic link doesn't exist, `open` returns an error saying that it can't open the file. This response can confuse users who aren't familiar with symbolic links. For example,

```
$ ln -s /no/such/file myfile              create a symbolic link
$ ls myfile
myfile                                    ls says it's there
$ cat myfile                              so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                            try -l option
lrwxrwxrwx  1 sar       13 Jan 22 00:26 myfile -> /no/such/file
```

The file `myfile` does exist, yet `cat` says there is no such file, because `myfile` is a symbolic link and the file pointed to by the symbolic link doesn't exist. The `-l` option to `ls` gives us two hints: the first character is an `l`, which means a symbolic link, and the sequence `->` also indicates a symbolic link. The `ls` command has another option (`-F`) that appends an at-sign (@) to filenames that are symbolic links, which can help us spot symbolic links in a directory listing without the `-l` option. □

4.18 Creating and Reading Symbolic Links

A symbolic link is created with either the `symlink` or `symlinkat` function.

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);

int symlinkat(const char *actualpath, int fd, const char *sympath);
```

Both return: 0 if OK, –1 on error

A new directory entry, *sympath*, is created that points to *actualpath*. It is not required that *actualpath* exist when the symbolic link is created. (We saw this in the example at the end of the previous section.) Also, *actualpath* and *sympath* need not reside in the same file system.

The `symlinkat` function is similar to `symlink`, but the *sympath* argument is evaluated relative to the directory referenced by the open file descriptor for that directory (specified by the *fd* argument). If the *sympath* argument specifies an absolute pathname or if the *fd* argument has the special value `AT_FDCWD`, then `symlinkat` behaves the same way as `symlink`.

Because the `open` function follows a symbolic link, we need a way to open the link itself and read the name in the link. The `readlink` and `readlinkat` functions do this.

```
#include <unistd.h>

ssize_t readlink(const char* restrict pathname, char *restrict buf,
                 size_t bufsize);

ssize_t readlinkat(int fd, const char* restrict pathname,
                   char *restrict buf, size_t bufsize);
```

Both return: number of bytes read if OK, –1 on error

These functions combine the actions of `open`, `read`, and `close`. If successful, they return the number of bytes placed into *buf*. The contents of the symbolic link that are returned in *buf* are not null terminated.

The `readlinkat` function behaves the same way as the `readlink` function when the *pathname* argument specifies an absolute pathname or when the *fd* argument has the special value `AT_FDCWD`. However, when the *fd* argument is a valid file descriptor of an open directory and the *pathname* argument is a relative pathname, then `readlinkat` evaluates the pathname relative to the open directory represented by *fd*.

4.19 File Times

In Section 4.2, we discussed how the 2008 version of the Single UNIX Specification increased the resolution of the time fields in the `stat` structure from seconds to seconds plus nanoseconds. The actual resolution stored with each file's attributes depends on the file system implementation. For file systems that store timestamps in second granularity, the nanoseconds fields will be filled with zeros. For file systems that store timestamps in a resolution higher than seconds, the partial seconds value will be converted into nanoseconds and returned in the nanoseconds fields.

Three time fields are maintained for each file. Their purpose is summarized in Figure 4.19.

Field	Description	Example	`ls`(1) option
`st_atim`	last-access time of file data	`read`	`-u`
`st_mtim`	last-modification time of file data	`write`	default
`st_ctim`	last-change time of i-node status	`chmod, chown`	`-c`

Figure 4.19 The three time values associated with each file

Note the difference between the modification time (`st_mtim`) and the changed-status time (`st_ctim`). The modification time indicates when the contents of the file were last modified. The changed-status time indicates when the i-node of the file was last modified. In this chapter, we've described many operations that affect the i-node without changing the actual contents of the file: changing the file access permissions, changing the user 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 the `ls` command 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.

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`	•	•	•		•	•	4.21	
`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`			•		•	•	4.16	for both arguments
`rmdir`					•	•	4.21	
`truncate`, `ftruncate`		•	•				4.13	
`unlink`			•		•	•	4.15	
`utimes`, `utimensat`, `futimens`	•	•	•				4.20	
`write`		•	•				3.8	

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

(The `mkdir` and `rmdir` functions are covered in Section 4.21. The `utimes`, `utimensat`, and `futimens` functions are covered in the next section. The seven `exec` functions are described in Section 8.10. We describe the `mkfifo` and `pipe` functions in Chapter 15.)

4.20 `futimens`, `utimensat`, and `utimes` Functions

Several functions are available to change the access time and the modification time of a file. The `futimens` and `utimensat` functions provide nanosecond granularity for specifying timestamps, using the `timespec` structure (the same structure used by the `stat` family of functions; see Section 4.2).

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

int futimens(int fd, const struct timespec times[2]);

int utimensat(int fd, const char *path, const struct timespec times[2],
              int flag);
```

Both return: 0 if OK, –1 on error

In both functions, the first element of the *times* array argument contains the access time, and the second element contains the modification time. The two time values are calendar times, which count seconds since the Epoch, as described in Section 1.10. Partial seconds are expressed in nanoseconds.

Timestamps can be specified in one of four ways:

1. The *times* argument is a null pointer. In this case, both timestamps are set to the current time.
2. The *times* argument points to an array of two `timespec` structures. If either `tv_nsec` field has the special value `UTIME_NOW`, the corresponding timestamp is set to the current time. The corresponding `tv_sec` field is ignored.
3. The *times* argument points to an array of two `timespec` structures. If either `tv_nsec` field has the special value `UTIME_OMIT`, then the corresponding timestamp is unchanged. The corresponding `tv_sec` field is ignored.
4. The *times* argument points to an array of two `timespec` structures and the `tv_nsec` field contains a value other than `UTIME_NOW` or `UTIME_OMIT`. In this case, the corresponding timestamp is set to the value specified by the corresponding `tv_sec` and `tv_nsec` fields.

The privileges required to execute these functions depend on the value of the *times* argument.

- If *times* is a null pointer or if either `tv_nsec` field is set to `UTIME_NOW`, either the effective user ID of the process must equal the owner ID of the file, the process must have write permission for the file, or the process must be a superuser process.
- If *times* is a non-null pointer and either `tv_nsec` field has a value other than `UTIME_NOW` or `UTIME_OMIT`, 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.

- If *times* is a non-null pointer and both tv_nsec fields are set to UTIME_OMIT, no permissions checks are performed.

With futimens, you need to open the file to change its times. The utimensat function provides a way to change a file's times using the file's name. The *pathname* argument is evaluated relative to the *fd* argument, which is either a file descriptor of an open directory or the special value AT_FDCWD to force evaluation relative to the current directory of the calling process. If *pathname* specifies an absolute pathname, then the *fd* argument is ignored.

The *flag* argument to utimensat can be used to further modify the default behavior. If the AT_SYMLINK_NOFOLLOW flag is set, then the times of the symbolic link itself are changed (if the pathname refers to a symbolic link). The default behavior is to follow a symbolic link and modify the times of the file to which the link refers.

Both futimens and utimensat are included in POSIX.1. A third function, utimes, is included in the Single UNIX Specification as part of the XSI option.

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

int utimes(const char *pathname, const struct timeval times[2]);
```

Returns: 0 if OK, –1 on error

The utimes function operates on a pathname. The *times* argument is a pointer to an array of two timestamps—access time and modification time—but they are expressed in seconds and microseconds:

```
struct timeval {
        time_t tv_sec;    /* seconds */
        long   tv_usec;   /* microseconds */
};
```

Note that we are unable to specify a value for the changed-status time, st_ctim—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 one of these functions. Also, the standard archive programs, tar(1) and cpio(1), optionally call these functions to set a file's times 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 futimens function.

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

int
main(int argc, char *argv[])
```

```
{
    int             i, fd;
    struct stat     statbuf;
    struct timespec times[2];

    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;
        }
        times[0] = statbuf.st_atim;
        times[1] = statbuf.st_mtim;
        if (futimens(fd, times) < 0)        /* reset times */
            err_ret("%s: futimens error", argv[i]);
        close(fd);
    }
    exit(0);
}
```

Figure 4.21 Example of `futimens` function

We can demonstrate the program in Figure 4.21 on Linux with the following commands:

```
$ ls -l changemod times          look at sizes and last-modification times
-rwxr-xr-x  1 sar   13792 Jan 22 01:26 changemod
-rwxr-xr-x  1 sar   13824 Jan 22 01:26 times
$ ls -lu changemod times         look at last-access times
-rwxr-xr-x  1 sar   13792 Jan 22 22:22 changemod
-rwxr-xr-x  1 sar   13824 Jan 22 22:22 times
$ date                           print today's date
Fri Jan 27 20:53:46 EST 2012
$ ./a.out changemod times        run the program in Figure 4.21
$ ls -l changemod times          and check the results
-rwxr-xr-x  1 sar       0 Jan 22 01:26 changemod
-rwxr-xr-x  1 sar       0 Jan 22 01:26 times
$ ls -lu changemod times         check the last-access times also
-rwxr-xr-x  1 sar       0 Jan 22 22:22 changemod
-rwxr-xr-x  1 sar       0 Jan 22 22:22 times
$ ls -lc changemod times         and the changed-status times
-rwxr-xr-x  1 sar       0 Jan 27 20:53 changemod
-rwxr-xr-x  1 sar       0 Jan 27 20:53 times
```

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

4.21 mkdir, mkdirat, and rmdir Functions

Directories are created with the mkdir and mkdirat functions, and deleted with the rmdir function.

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

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

int mkdirat(int fd, const char *pathname, mode_t mode);
```

Both return: 0 if OK, –1 on error

These functions create a new, empty directory. The entries for dot and dot-dot are created automatically. 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 10 and Linux 3.2.0 also have the new directory inherit the set-group-ID bit from the parent directory. Files created in the new directory will then inherit the group ID of that directory. With Linux, the file system implementation determines whether this behavior is supported. For example, the ext2, ext3, and ext4 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 8.0 and Mac OS X 10.6.8 are based on 4.4BSD, they do not require inheriting the set-group-ID bit. On these platforms, newly created files and directories always inherit the group ID of the parent directory, regardless of whether the set-group-ID bit is set.
>
> 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 constraint, 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.

The mkdirat function is similar to the mkdir function. When the *fd* argument has the special value AT_FDCWD, or when the *pathname* argument specifies an absolute pathname, mkdirat behaves exactly like mkdir. Otherwise, the *fd* argument is an open directory from which relative pathnames will be evaluated.

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.)

4.22 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 the process of reading a directory, 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);
DIR *fdopendir(int fd);
                                        Both return: 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 fdopendir function first appeared in version 4 of the Single UNIX Specification. It provides a way to convert an open file descriptor into a DIR structure for use by the directory handling functions.

The telldir and seekdir functions are not part of the base POSIX.1 standard. They are included in the XSI option in the Single UNIX Specification, 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 <dirent.h> is implementation dependent. Implementations define the structure to contain at least the following two members:

```
ino_t  d_ino;                  /* i-node number */
char   d_name[];               /* null-terminated filename */
```

> The d_ino entry is not defined by POSIX.1, because it is an implementation feature, but it is defined as part of the XSI option in POSIX.1. POSIX.1 defines only the d_name entry in this structure.

Note that the size of the d_name entry isn't specified, but it is guaranteed to hold at least NAME_MAX characters, not including the terminating null byte (recall Figure 2.15.) Since the filename is null terminated, however, it doesn't matter how 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 seven 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 returned by opendir and fdopendir is then used with the other five functions. The opendir function initializes things so that the first readdir returns the first entry in the directory. When the DIR structure is created by fdopendir, the first entry returned by readdir depends on the file offset associated with the file descriptor passed to fdopendir. Note that 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 a count of the various types of files shown 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 problem, 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 SUSv4, `nftw` is included as part of the XSI option. Implementations are included in FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. (In SUSv4, the `ftw` function has been marked as obsolescent.) BSD-based systems have a different function, `fts`(3), that provides similar functionality. It is available in FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8.

```
#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 size_t pathlen;

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(&pathlen);    /* malloc PATH_MAX+1 bytes */
                                        /* (Figure 2.16) */
    if (pathlen <= strlen(pathname)) {
        pathlen = strlen(pathname) * 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    strcpy(fullpath, pathname);
    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, n;

    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);

    n = strlen(fullpath);
    if (n + NAME_MAX + 2 > pathlen) {   /* expand path buffer */
        pathlen *= 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    fullpath[n++] = '/';
```

```
    fullpath[n] = 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(&fullpath[n], dirp->d_name); /* append name after "/" */
        if ((ret = dopath(func)) != 0)      /* recursive */
            break;  /* time to leave */
    }
    fullpath[n-1] = 0;  /* erase everything from slash onward */

    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:   /* directories should have type = FTW_D */
            err_dump("for S_IFDIR for %s", pathname);
        }
        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

To illustrate the ftw and nftw functions, we have provided more generality in this program than needed. 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 using this technique in many standard UNIX System commands—find, ls, tar, and so on—refer to Fowler, Korn, and Vo [1989].

4.23 chdir, fchdir, and getcwd Functions

Every process has a current working directory. This directory is where the search for all relative pathnames starts (i.e., with 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 function.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int fd);
```

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.

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.) As a result, 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 this program, call the executable `mycd`, and run it, 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.

> The Linux kernel can determine the full pathname. Its components are distributed throughout the mount table and the dcache table, and are reassembled, for example, when you read the `/proc/self/cwd` symbolic link.

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 level, 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 already exists that does this work for us.

```
#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 else an error will be 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;
    size_t      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`.

4.24 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 for their use 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`. Consequently, 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 8.0 and Mac OS X 10.6.8 use a 32-bit integer, with 8 bits for the major number and 24 bits for the minor number. On 32-bit systems, Solaris 10 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 10 represents `dev_t` as a 64-bit integer, with 32 bits for each number. On Linux 3.2.0, although `dev_t` is a 64-bit integer, only 12 bits are used for the major number and 20 bits are used for the minor number.

  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 their function prototypes in `<sys/mkdev.h>`, because the macro definitions in `<sys/sysmacros.h>` are considered obsolete in Solaris. 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 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 on Linux gives us the following output:

```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 8/3
/home/sar: dev = 8/4
/dev/tty0: dev = 0/5 (character) rdev = 4/0
/dev/tty1: dev = 0/5 (character) rdev = 4/1
$ mount                          which directories are mounted on which devices?
/dev/sda3 on / type ext3 (rw,errors=remount-ro,commit=0)
/dev/sda4 on /home type ext3 (rw,commit=0)
$ ls -l /dev/tty[01] /dev/sda[34]
brw-rw----  1 root       8, 3 2011-07-01 11:08 /dev/sda3
brw-rw----  1 root       8, 4 2011-07-01 11:08 /dev/sda4
crw--w----  1 root       4, 0 2011-07-01 11:08 /dev/tty0
crw-------  1 root       4, 1 2011-07-01 11:08 /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, which 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 UNIX systems 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 0/5—the `devtmpfs` pseudo file system, which implements the `/dev`—but that their actual device numbers are 4/0 and 4/1. □

4.25 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 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, as follows:

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

4.26 Summary

This chapter has centered on 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 and directories. We've looked at how files and directories might be laid out in a file

system, and we've seen how to navigate the file system namespace. A thorough understanding of all the properties of files and directories and all the functions that operate on them 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 the 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 4.22, how does the system's limit on the number of open files affect the `myftw` function?

4.11 In Section 4.22, our version of `ftw` never changes its directory. Modify this routine so that each time it encounters a directory, it uses the `chdir` function to change 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 `utimes` 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 used 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`?