15

FILE ATTRIBUTES

In this chapter, we investigate various attributes of files (file metadata). We begin with a description of the *stat()* system call, which returns a structure containing many of these attributes, including file timestamps, file ownership, and file permissions. We then go on to look at various system calls used to change these attributes. (The discussion of file permissions continues in Chapter 17, where we look at access control lists.) We conclude the chapter with a discussion of i-node flags (also known as *ext2* extended file attributes), which control various aspects of the treatment of files by the kernel.

15.1 Retrieving File Information: *stat()*

The *stat()*, *lstat()*, and *fstat()* system calls retrieve information about a file, mostly drawn from the file i-node.

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

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

All return 0 on success, or −1 on error

These three system calls differ only in the way that the file is specified:

- *stat()* returns information about a named file;
- *lstat()* is similar to *stat()*, except that if the named file is a symbolic link, information about the link itself is returned, rather than the file to which the link points; and
- *fstat()* returns information about a file referred to by an open file descriptor.

The *stat()* and *lstat()* system calls don't require permissions on the file itself. However, execute (search) permission is required on all of the parent directories specified in *pathname*. The *fstat()* system call always succeeds, if provided with a valid file descriptor.

All of these system calls return a *stat* structure in the buffer pointed to by *statbuf*. This structure has the following form:

```
struct stat {
    dev_t     st_dev;         /* IDs of device on which file resides */
    ino_t     st_ino;         /* I-node number of file */
    mode_t    st_mode;        /* File type and permissions */
    nlink_t   st_nlink;       /* Number of (hard) links to file */
    uid_t     st_uid;         /* User ID of file owner */
    gid_t     st_gid;         /* Group ID of file owner */
    dev_t     st_rdev;        /* IDs for device special files */
    off_t     st_size;        /* Total file size (bytes) */
    blksize_t st_blksize;     /* Optimal block size for I/O (bytes) */
    blkcnt_t  st_blocks;      /* Number of (512B) blocks allocated */
    time_t    st_atime;       /* Time of last file access */
    time_t    st_mtime;       /* Time of last file modification */
    time_t    st_ctime;       /* Time of last status change */
};
```

The various data types used to type the fields in the *stat* structure are all specified in SUSv3. See Section 3.6.2 for further information about these types.

> According to SUSv3, when *lstat()* is applied to a symbolic link, it needs to return valid information only in the *st_size* field and in the file type component (described shortly) of the *st_mode* field. None of other fields (e.g., the time fields) need contain valid information. This gives an implementation the freedom to not maintain these fields, which may be done for efficiency reasons. In particular, the intent of earlier UNIX standards was to allow a symbolic link to be implemented either as an i-node or as an entry in a directory. Under the latter implementation, it is not possible to implement all of the fields required by the *stat* structure. (On all major contemporary UNIX implementations, symbolic links are implemented as i-nodes.) On Linux, *lstat()* returns information in all of the *stat* fields when applied to a symbolic link.

In the following pages, we look at some of the *stat* structure fields in more detail, and finish with an example program that displays the entire *stat* structure.

Device IDs and i-node number

The *st_dev* field identifies the device on which the file resides. The *st_ino* field contains the i-node number of the file. The combination of *st_dev* and *st_ino* uniquely identifies a file across all file systems. The *dev_t* type records the major and minor IDs of a device (Section 14.1).

If this is the i-node for a device, then the *st_rdev* field contains the major and minor IDs of the device.

The major and minor IDs of a *dev_t* value can be extracted using two macros: `major()` and `minor()`. The header file required to obtain the declarations of these two macros varies across UNIX implementations. On Linux, they are exposed by `<sys/types.h>` if the `_BSD_SOURCE` macro is defined.

The size of the integer values returned by `major()` and `minor()` varies across UNIX implementations. For portability, we always cast the returned values to *long* when printing them (see Section 3.6.2).

File ownership

The *st_uid* and *st_gid* fields identify, respectively, the owner (user ID) and group (group ID) to which the file belongs.

Link count

The *st_nlink* field is the number of (hard) links to the file. We describe links in detail in Chapter 18.

File type and permissions

The *st_mode* field is a bit mask serving the dual purpose of identifying the file type and specifying the file permissions. The bits of this field are laid out as shown in Figure 15-1.

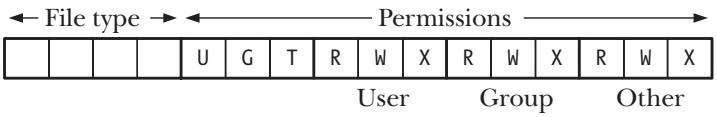

Figure 15-1: Layout of *st_mode* bit mask

The file type can be extracted from this field by ANDing (`&`) with the constant `S_IFMT`. (On Linux, 4 bits are used for the file-type component of the *st_mode* field. However, because SUSv3 makes no specification about how the file type is represented, this detail may vary across implementations.) The resulting value can then be compared with a range of constants to determine the file type, like so:

```
if ((statbuf.st_mode & S_IFMT) == S_IFREG)
    printf("regular file\n");
```

Because this is a common operation, standard macros are provided to simplify the above to the following:

```
if (S_ISREG(statbuf.st_mode))
    printf("regular file\n");
```

The full set of file-type macros (defined in <sys/stat.h>) is shown in Table 15-1. All of the file-type macros in Table 15-1 are specified in SUSv3 and appear on Linux. Some other UNIX implementations define additional file types (e.g., S_IFDOOR, for door files on Solaris). The type S_IFLNK is returned only by calls to *lstat()*, since calls to *stat()* always follow symbolic links.

The original POSIX.1 standard did not specify the constants shown in the first column of Table 15-1, although most of them appeared on most UNIX implementations. SUSv3 requires these constants.

> In order to obtain the definitions of S_IFSOCK and S_ISSOCK() from <sys/stat.h>, we must either define the _BSD_SOURCE feature test macro or define _XOPEN_SOURCE with a value greater than or equal to 500. (The rules have varied somewhat across *glibc* versions: in some cases, _XOPEN_SOURCE must be defined with a value of 600 or greater.)

Table 15-1: Macros for checking file types in the *st_mode* field of the *stat* structure

Constant	Test macro	File type
S_IFREG	S_ISREG()	Regular file
S_IFDIR	S_ISDIR()	Directory
S_IFCHR	S_ISCHR()	Character device
S_IFBLK	S_ISBLK()	Block device
S_IFIFO	S_ISFIFO()	FIFO or pipe
S_IFSOCK	S_ISSOCK()	Socket
S_IFLNK	S_ISLNK()	Symbolic link

The bottom 12 bits of the *st_mode* field define the permissions for the file. We describe the file permission bits in Section 15.4. For now, we simply note that the 9 least significant of the permission bits are the read, write, and execute permissions for each of the categories owner, group, and other.

File size, blocks allocated, and optimal I/O block size

For regular files, the *st_size* field is the total size of the file in bytes. For a symbolic link, this field contains the length (in bytes) of the pathname pointed to by the link. For a shared memory object (Chapter 54), this field contains the size of the object.

The *st_blocks* field indicates the total number of blocks allocated to the file, in 512-byte block units. This total includes space allocated for pointer blocks (see Figure 14-2, on page 258). The choice of the 512-byte unit of measurement is historical—this is the smallest block size on any of the file systems that have been implemented under UNIX. More modern file systems use larger logical block sizes. For example, under *ext2*, the value in *st_blocks* is always a multiple of 2, 4, or 8, depending on whether the *ext2* logical block size is 1024, 2048, or 4096 bytes.

> SUSv3 doesn't define the units in which *st_blocks* is measured, allowing the possibility that an implementation uses a unit other than 512 bytes. Most UNIX implementations do use 512-byte units, but HP-UX 11 uses file system–specific units (e.g., 1024 bytes in some cases).

The *st_blocks* field records the number of disk blocks actually allocated. If the file contains holes (Section 4.7), this will be smaller than might be expected from the corresponding number of bytes (*st_size*) in the file. (The disk usage command, *du −k file*, displays the actual space allocated for a file, in kilobytes; that is, a figure calculated from the *st_blocks* value for the file, rather than the *st_size* value.)

The *st_blksize* field is somewhat misleadingly named. It is not the block size of the underlying file system, but rather the optimal block size (in bytes) for I/O on files on this file system. I/O in blocks smaller than this size is less efficient (refer to Section 13.1). A typical value returned in *st_blksize* is 4096.

File timestamps

The *st_atime*, *st_mtime*, and *st_ctime* fields contain, respectively, the times of last file access, last file modification, and last status change. These fields are of type *time_t*, the standard UNIX time format of seconds since the Epoch. We say more about these fields in Section 15.2.

Example program

The program in Listing 15-1 uses *stat()* to retrieve information about the file named on its command line. If the *−l* command-line option is specified, then the program instead uses *lstat()* so that we can retrieve information about a symbolic link instead of the file to which it refers. The program prints all fields of the returned *stat* structure. (For an explanation of why we cast the *st_size* and *st_blocks* fields to *long long*, see Section 5.10.) The *filePermStr()* function used by this program is shown in Listing 15-4, on page 296.

Here is an example of the use of the program:

```
$ echo 'All operating systems provide services for programs they run' > apue
$ chmod g+s apue          Turn on set-group-ID bit; affects last status change time
$ cat apue                Affects last file access time
All operating systems provide services for programs they run
$ ./t_stat apue
File type:                regular file
Device containing i-node: major=3   minor=11
I-node number:            234363
Mode:                     102644 (rw-r--r--)
    special bits set:     set-GID
Number of (hard) links:   1
Ownership:                UID=1000   GID=100
File size:                61 bytes
Optimal I/O block size:   4096 bytes
512B blocks allocated:    8
Last file access:         Mon Jun  8 09:40:07 2011
Last file modification:   Mon Jun  8 09:39:25 2011
Last status change:       Mon Jun  8 09:39:51 2011
```

Listing 15-1: Retrieving and interpreting file *stat* information

files/t_stat.c

```
#define _BSD_SOURCE     /* Get major() and minor() from <sys/types.h> */
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include "file_perms.h"
#include "tlpi_hdr.h"

static void
displayStatInfo(const struct stat *sb)
{
    printf("File type:                ");

    switch (sb->st_mode & S_IFMT) {
    case S_IFREG:  printf("regular file\n");            break;
    case S_IFDIR:  printf("directory\n");               break;
    case S_IFCHR:  printf("character device\n");        break;
    case S_IFBLK:  printf("block device\n");            break;
    case S_IFLNK:  printf("symbolic (soft) link\n");    break;
    case S_IFIFO:  printf("FIFO or pipe\n");            break;
    case S_IFSOCK: printf("socket\n");                  break;
    default:       printf("unknown file type?\n");      break;
    }

    printf("Device containing i-node: major=%ld   minor=%ld\n",
                (long) major(sb->st_dev), (long) minor(sb->st_dev));

    printf("I-node number:            %ld\n", (long) sb->st_ino);

    printf("Mode:                     %lo (%s)\n",
            (unsigned long) sb->st_mode, filePermStr(sb->st_mode, 0));

    if (sb->st_mode & (S_ISUID | S_ISGID | S_ISVTX))
        printf("    special bits set:     %s%s%s\n",
                (sb->st_mode & S_ISUID) ? "set-UID " : "",
                (sb->st_mode & S_ISGID) ? "set-GID " : "",
                (sb->st_mode & S_ISVTX) ? "sticky " : "");

    printf("Number of (hard) links:   %ld\n", (long) sb->st_nlink);

    printf("Ownership:                UID=%ld   GID=%ld\n",
            (long) sb->st_uid, (long) sb->st_gid);

    if (S_ISCHR(sb->st_mode) || S_ISBLK(sb->st_mode))
        printf("Device number (st_rdev):  major=%ld; minor=%ld\n",
                (long) major(sb->st_rdev), (long) minor(sb->st_rdev));

    printf("File size:                %lld bytes\n", (long long) sb->st_size);
    printf("Optimal I/O block size:   %ld bytes\n", (long) sb->st_blksize);
    printf("512B blocks allocated:    %lld\n", (long long) sb->st_blocks);
```

```
    printf("Last file access:         %s", ctime(&sb->st_atime));
    printf("Last file modification:   %s", ctime(&sb->st_mtime));
    printf("Last status change:       %s", ctime(&sb->st_ctime));
}

int
main(int argc, char *argv[])
{
    struct stat sb;
    Boolean statLink;           /* True if "-l" specified (i.e., use lstat) */
    int fname;                  /* Location of filename argument in argv[] */

    statLink = (argc > 1) && strcmp(argv[1], "-l") == 0;
                                /* Simple parsing for "-l" */
    fname = statLink ? 2 : 1;

    if (fname >= argc || (argc > 1 && strcmp(argv[1], "--help") == 0))
        usageErr("%s [-l] file\n"
                 "        -l = use lstat() instead of stat()\n", argv[0]);

    if (statLink) {
        if (lstat(argv[fname], &sb) == -1)
            errExit("lstat");
    } else {
        if (stat(argv[fname], &sb) == -1)
            errExit("stat");
    }

    displayStatInfo(&sb);

    exit(EXIT_SUCCESS);
}
```
files/t_stat.c

15.2 File Timestamps

The *st_atime*, *st_mtime*, and *st_ctime* fields of the *stat* structure contain file timestamps. These fields record, respectively, the times of last file access, last file modification, and last file status change (i.e., last change to the file's i-node information). Timestamps are recorded in seconds since the Epoch (1 January 1970; see Section 10.1).

Most native Linux and UNIX file systems support all of the timestamp fields, but some non-UNIX file systems may not.

Table 15-2 summarizes which of the timestamp fields (and in some cases, the analogous fields in the parent directory) are changed by various system calls and library functions described in this book. In the headings of this table, *a*, *m*, and *c* represent the *st_atime*, *st_mtime*, and *st_ctime* fields, respectively. In most cases, the relevant timestamp is set to the current time by the system call. The exceptions are *utime()* and similar calls (discussed in Sections 15.2.1 and 15.2.2), which can be used to explicitly set the last file access and modification times to arbitrary values.

Table 15-2: Effect of various functions on file timestamps

Function	File or directory			Parent directory			Notes
	a	m	c	a	m	c	
chmod()			•				Same for *fchmod()*
chown()			•				Same for *lchown()* and *fchown()*
exec()	•						
link()			•		•	•	Affects parent directory of second argument
mkdir()	•	•	•		•	•	
mkfifo()	•	•	•		•	•	
mknod()	•	•	•		•	•	
mmap()	•	•	•				*st_mtime* and *st_ctime* are changed only on updates to `MAP_SHARED` mapping
msync()		•	•				Changed only if file is modified
open(), *creat()*	•	•	•		•	•	When creating new file
open(), *creat()*		•	•				When truncating existing file
pipe()	•	•	•				
read()	•						Same for *readv()*, *pread()*, and *preadv()*
readdir()	•						*readdir()* may buffer directory entries; timestamps updated only if directory is read
removexattr()			•				Same for *fremovexattr()* and *lremovexattr()*
rename()			•		•	•	Affects timestamps in both parent directories; SUSv3 doesn't specify file *st_ctime* change, but notes that some implementations do this
rmdir()					•	•	Same for *remove(directory)*
sendfile()	•						Timestamp changed for input file
setxattr()			•				Same for *fsetxattr()* and *lsetxattr()*
symlink()	•	•	•		•	•	Sets timestamps of link (not target file)
truncate()		•	•				Same for *ftruncate()*; timestamps change only if file size changes
unlink()			•		•	•	Same for *remove(file)*; file *st_ctime* changes if previous link count was > 1
utime()	•	•	•				Same for *utimes()*, *futimes()*, *futimens()*, *lutimes()*, and *utimensat()*
write()		•	•				Same for *writev()*, *pwrite()*, and *pwritev()*

In Sections 14.8.1 and 15.5, we describe *mount(2)* options and per-file flags that prevent updates to the last access time of a file. The *open()* `O_NOATIME` flag described in Section 4.3.1 also serves a similar purpose. In some applications, this can be useful for performance reasons, since it reduces the number of disk operations that are required when a file is accessed.

> Although most UNIX systems don't record the creation time of a file, on recent BSD systems, this time is recorded in a *stat* field named *st_birthtime*.

Nanosecond timestamps

With version 2.6, Linux supports nanosecond resolution for the three timestamp fields of the *stat* structure. Nanosecond resolution improves the accuracy of programs that need to make decisions based on the relative order of file timestamps (e.g., *make(1)*).

SUSv3 doesn't specify nanosecond timestamps for the *stat* structure, but SUSv4 adds this specification.

Not all file systems support nanosecond timestamps. *JFS*, *XFS*, *ext4*, and *Btrfs* do, but *ext2*, *ext3*, and *Reiserfs* do not.

Under the *glibc* API (since version 2.3), the timestamp fields are each defined as a *timespec* structure (we describe this structure when we discuss *utimensat()* later in this section), which represents a time in seconds and nanoseconds components. Suitable macro definitions make the seconds component of these structures visible using the traditional field names (*st_atime*, *st_mtime*, and *st_ctime*). The nanosecond components can be accessed using field names such *st_atim.tv_nsec*, for the nanosecond component of the last file access time.

15.2.1 Changing File Timestamps with *utime()* and *utimes()*

The last file access and modification timestamps stored in a file i-node can be explicitly changed using *utime()* or one of a related set of system calls. Programs such as *tar(1)* and *unzip(1)* use these system calls to reset file timestamps when unpacking an archive.

```
#include <utime.h>

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

Returns 0 on success, or −1 on error

The *pathname* argument identifies the file whose times we wish to modify. If *pathname* is a symbolic link, it is dereferenced. The *buf* argument can be either `NULL` or a pointer to a *utimbuf* structure:

```
struct utimbuf {
    time_t actime;      /* Access time */
    time_t modtime;     /* Modification time */
};
```

The fields in this structure measure time in seconds since the Epoch (Section 10.1).

Two different cases determine how *utime()* works:

- If *buf* is specified as `NULL`, then both the last access and the last modification times are set to the current time. In this case, either the effective user ID of the process must match the file's user ID (owner), the process must have write permission on the file (logical, since a process with write permission on a file could employ other system calls that would have the side effect of changing these file timestamps), or the process must be privileged (`CAP_FOWNER` or `CAP_DAC_OVERRIDE`).

(To be accurate, on Linux, it is the process's file-system user ID, rather than its effective user ID, that is checked against the file's user ID, as described in Section 9.5.)

- If *buf* is specified as pointer to a *utimbuf* structure, then the last file access and modification times are updated using the corresponding fields of this structure. In this case, the effective user ID of the process must match the file's user ID (having write permission on the file is not sufficient) or the caller must be privileged (`CAP_FOWNER`).

To change just one of the file timestamps, we first use *stat()* to retrieve both times, use one of these times to initialize the *utimbuf* structure, and then set the other as desired. This is demonstrated in the following code, which makes the last modification time of a file the same as the last access time:

```
struct stat sb;
struct utimbuf utb;

if (stat(pathname, &sb) == -1)
    errExit("stat");
utb.actime = sb.st_atime;       /* Leave access time unchanged */
utb.modtime = sb.st_atime;
if (utime(pathname, &utb) == -1)
    errExit("utime");
```

A successful call to *utime()* always sets the last status change time to the current time.

Linux also provides the BSD-derived *utimes()* system call, which performs a similar task to *utime()*.

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

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

Returns 0 on success, or −1 on error

The most notable difference between *utime()* and *utimes()* is that *utimes()* allows time values to be specified with microsecond accuracy (the *timeval* structure is described in Section 10.1). This provides (partial) access to the nanosecond accuracy with which file timestamps are provided in Linux 2.6. The new file access time is specified in *tv[0]*, and the new modification time is specified in *tv[1]*.

> An example of the use of *utimes()* is provided in the file `files/t_utimes.c` in the source code distribution for this book.

The *futimes()* and *lutimes()* library functions perform a similar task to *utimes()*. They differ from *utimes()* in the argument used to specify the file whose timestamps are to be changed.

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

int futimes(int fd, const struct timeval tv[2]);
int lutimes(const char *pathname, const struct timeval tv[2]);
```

Both return 0 on success, or −1 on error

With *futimes()*, the file is specified via an open file descriptor, *fd*.

With *lutimes()*, the file is specified via a pathname, with the difference from *utimes()* that if the pathname refers to a symbolic link, then the link is not dereferenced; instead, the timestamps of the link itself are changed.

The *futimes()* function is supported since *glibc* 2.3. The *lutimes()* function is supported since *glibc* 2.6.

15.2.2 Changing File Timestamps with *utimensat()* and *futimens()*

The *utimensat()* system call (supported since kernel 2.6.22) and the *futimens()* library function (supported since *glibc* 2.6) provide extended functionality for setting a file's last access and last modification timestamps. Among the advantages of these interfaces are the following:

- We can set timestamps with nanosecond precision. This improves on the microsecond precision provided by *utimes()*.
- It is possible to set the timestamps independently (i.e., one at a time). As shown earlier, to change just one of the timestamps using the older interfaces, we must first call *stat()* to retrieve the value of the other timestamp, and then specify the retrieved value along with the timestamp whose value we want to change. (This could lead to a race condition if another process performed an operation that updated the timestamp between these two steps.)
- We can independently set either of the timestamps to the current time. To change just one timestamp to the current time with the older interfaces, we need to employ a call to *stat()* to retrieve the setting of the timestamp whose value we wish to leave unchanged, and a call to *gettimeofday()* to obtain the current time.

These interfaces are not specified in SUSv3, but are included in SUSv4.

The *utimensat()* system call updates the timestamps of the file specified by *pathname* to the values specified in the array *times*.

```
#define _XOPEN_SOURCE 700     /* Or define _POSIX_C_SOURCE >= 200809 */
#include <sys/stat.h>

int utimensat(int dirfd, const char *pathname,
              const struct timespec times[2], int flags);
```

Returns 0 on success, or −1 on error

If *times* is specified as NULL, then both file timestamps are updated to the current time. If *times* is not NULL, then the new last access timestamp is specified in *times[0]* and the new last modification timestamp is specified in *times[1]*. Each of the elements of the array *times* is a structure of the following form:

```
struct timespec {
    time_t tv_sec;     /* Seconds ('time_t' is an integer type) */
    long   tv_nsec;    /* Nanoseconds */
};
```

The fields in this structure specify a time in seconds and nanoseconds since the Epoch (Section 10.1).

To set one of the timestamps to the current time, we specify the special value UTIME_NOW in the corresponding *tv_nsec* field. To leave one of the timestamps unchanged, we specify the special value UTIME_OMIT in the corresponding *tv_nsec* field. In both cases, the value in the corresponding *tv_sec* field is ignored.

The *dirfd* argument can either specify AT_FDCWD, in which case the *pathname* argument is interpreted as for *utimes()*, or it can specify a file descriptor referring to a directory. The purpose of the latter choice is described in Section 18.11.

The *flags* argument can be either 0, or AT_SYMLINK_NOFOLLOW, meaning that *pathname* should not be dereferenced if it is a symbolic link (i.e., the timestamps of the symbolic link itself should be changed). By contrast, *utimes()* always dereferences symbolic links.

The following code segment sets the last access time to the current time and leaves the last modification time unchanged:

```
struct timespec times[2];

times[0].tv_sec = 0;
times[0].tv_nsec = UTIME_NOW;
times[1].tv_sec = 0;
times[1].tv_nsec = UTIME_OMIT;
if (utimensat(AT_FDCWD, "myfile", times, 0) == -1)
    errExit("utimensat");
```

The permission rules for changing timestamps with *utimensat()* (and *futimens()*) are similar to those for the older APIs, and are detailed in the *utimensat(2)* manual page.

The *futimens()* library function updates the timestamps of the file referred to by the open file descriptor *fd*.

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

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

Returns 0 on success, or −1 on error

The *times* argument of *futimens()* is used in the same way as for *utimensat()*.

15.3 File Ownership

Each file has an associated user ID (UID) and group ID (GID). These IDs determine which user and group the file belongs to. We now look at the rules that determine the ownership of new files and describe the system calls used to change a file's ownership.

15.3.1 Ownership of New Files

When a new file is created, its user ID is taken from the effective user ID of the process. The group ID of the new file may be taken from either the effective group ID of the process (equivalent to the System V default behavior) or the group ID of the parent directory (the BSD behavior). The latter possibility is useful for creating project directories in which all files belong to a particular group and are accessible to the members of that group. Which of the two values is used as the new file's group ID is determined by various factors, including the type of file system on which the new file is created. We begin by describing the rules followed by *ext2* and a few other file systems.

> To be accurate, on Linux, all uses of the terms *effective user* or *group ID* in this section should really be *file-system user* or *group ID* (Section 9.5).

When an *ext2* file system is mounted, either the *−o grpid* (or the synonymous *−o bsdgroups*) option or the *−o nogrpid* (or the synonymous *−o sysvgroups*) option may be specified to the *mount* command. (If neither option is specified, the default is *−o nogrpid*.) If *−o grpid* is specified, then a new file always inherits its group ID from the parent directory. If *−o nogrpid* is specified, then, by default, a new file takes its group ID from the process's effective group ID. However, if the set-group-ID bit is enabled for the directory (via *chmod g+s*), then the group ID of the file is inherited from the parent directory. These rules are summarized in Table 15-3.

> In Section 18.6, we'll see that when the set-group-ID bit is set on a directory, then it is also set on new subdirectories created within that directory. In this manner, the set-group-ID behavior described in the main text is propagated down through an entire directory tree.

Table 15-3: Rules determining the group ownership of a newly created file

File system mount option	Set-group-ID bit enabled on parent directory?	Group ownership of new file taken from
−o grpid, *−o bsdgroups*	(ignored)	parent directory group ID
−o nogrpid, *−o sysvgroups* (*default*)	no	process effective group ID
	yes	parent directory group ID

At the time of writing, the only file systems that support the *grpid* and *nogrpid* mount options are *ext2*, *ext3*, *ext4*, and (since Linux 2.6.14) *XFS*. Other file systems follow the *nogrpid* rules.

15.3.2 Changing File Ownership: *chown()*, *fchown()*, and *lchown()*

The *chown()*, *lchown()*, and *fchown()* system calls change the owner (user ID) and group (group ID) of a file.

```
#include <unistd.h>

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

#define _XOPEN_SOURCE 500     /* Or: #define _BSD_SOURCE */
#include <unistd.h>

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

All return 0 on success, or −1 on error

The distinction between these three system calls is similar to the *stat()* family of system calls:

- *chown()* changes the ownership of the file named in the *pathname* argument;
- *lchown()* does the same, except that if *pathname* is a symbolic link, ownership of the link file is changed, rather than the file to which it refers; and
- *fchown()* changes the ownership of a file referred to by the open file descriptor, *fd*.

The *owner* argument specifies the new user ID for the file, and the *group* argument specifies the new group ID for the file. To change just one of the IDs, we can specify −1 for the other argument to leave that ID unchanged.

> Prior to Linux 2.2, *chown()* did not dereference symbolic links. The semantics of *chown()* were changed with Linux 2.2, and the new *lchown()* system call was added to provide the behavior of the old *chown()* system call.

Only a privileged (CAP_CHOWN) process may use *chown()* to change the user ID of a file. An unprivileged process can use *chown()* to change the group ID of a file that it owns (i.e., the process's effective user ID matches the user ID of the file) to any of the groups of which they are a member. A privileged process can change the group ID of a file to any value.

If the owner or group of a file is changed, then the set-user-ID and set-group-ID permission bits are both turned off. This is a security precaution to ensure that a normal user could not enable the set-user-ID (or set-group-ID) bit on an executable file and then somehow make it owned by some privileged user (or group), thereby gaining that privileged identity when executing the file.

> SUSv3 leaves it unspecified whether the set-user-ID and set-group-ID bits should be turned off when the superuser changes the owner or group of an executable file. Linux 2.0 did turn these bits off in this case, while some of the early 2.2 kernels (up to 2.2.12) did not. Later 2.2 kernels returned to the 2.0 behavior, where changes by the superuser are treated the same as everyone else, and this behavior is maintained in subsequent kernel versions. (However, if we use the *chown(1)* command under a *root* login to change the ownership of a file, then, after calling *chown(2)*, the *chown* command uses the *chmod()* system call to reenable the set-user-ID and set-group-ID bits.)

When changing the owner or group of a file, the set-group-ID permission bit is not turned off if the group-execute permission bit is already off or if we are changing the ownership of a directory. In both of these cases, the set-group-ID bit is being used for a purpose other than the creation of a set-group-ID program, and therefore it is undesirable to turn the bit off. These other uses of the set-group-ID bit are as follows:

- If the group-execute permission bit is off, then the set-group-ID permission bit is being used to enable mandatory file locking (discussed in Section 55.4).
- In the case of a directory, the set-group-ID bit is being used to control the ownership of new files created in the directory (Section 15.3.1).

The use of *chown()* is demonstrated in Listing 15-2, a program that allows the user to change the owner and group of an arbitrary number of files, specified as command-line arguments. (This program uses the *userIdFromName()* and *groupIdFromName()* functions from Listing 8-1, on page 159, to convert user and group names into corresponding numeric IDs.)

Listing 15-2: Changing the owner and group of a file

files/t_chown.c

```
#include <pwd.h>
#include <grp.h>
#include "ugid_functions.h"             /* Declarations of userIdFromName()
                                           and groupIdFromName() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    uid_t uid;
    gid_t gid;
    int j;
    Boolean errFnd;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s owner group [file...]\n"
                 "        owner or group can be '-', "
                 "meaning leave unchanged\n", argv[0]);

    if (strcmp(argv[1], "-") == 0) {            /* "-" ==> don't change owner */
        uid = -1;
    } else {                                    /* Turn user name into UID */
        uid = userIdFromName(argv[1]);
        if (uid == -1)
            fatal("No such user (%s)", argv[1]);
    }

    if (strcmp(argv[2], "-") == 0) {            /* "-" ==> don't change group */
        gid = -1;
```

```
    } else {                                    /* Turn group name into GID */
        gid = groupIdFromName(argv[2]);
        if (gid == -1)
            fatal("No group user (%s)", argv[1]);
    }

    /* Change ownership of all files named in remaining arguments */

    errFnd = FALSE;
    for (j = 3; j < argc; j++) {
        if (chown(argv[j], uid, gid) == -1) {
            errMsg("chown: %s", argv[j]);
            errFnd = TRUE;
        }
    }

    exit(errFnd ? EXIT_FAILURE : EXIT_SUCCESS);
}
```

files/t_chown.c

15.4 File Permissions

In this section, we describe the permission scheme applied to files and directories. Although we talk about permissions here mainly as they apply to regular files and directories, the rules that we describe apply to all types of files, including devices, FIFOs, and UNIX domain sockets. Furthermore, the System V and POSIX interprocess communication objects (shared memory, semaphores, and message queues) also have permission masks, and the rules that apply for these objects are similar to those for files.

15.4.1 Permissions on Regular Files

As noted in Section 15.1, the bottom 12 bits of the *st_mode* field of the *stat* structure define the permissions for a file. The first 3 of these bits are special bits known as the set-user-ID, set-group-ID, and sticky bits (labeled U, G, and T, respectively, in Figure 15-1). We say more about these bits in Section 15.4.5. The remaining 9 bits form the mask defining the permissions that are granted to various categories of users accessing the file. The file permissions mask divides the world into three categories:

- *Owner* (also known as *user*): The permissions granted to the owner of the file.

  > The term *user* is used by commands such as *chmod(1)*, which uses the abbreviation *u* to refer to this permission category.

- *Group*: The permissions granted to users who are members of the file's group.
- *Other*: The permissions granted to everyone else.

Three permissions may be granted to each user category:

- *Read*: The contents of the file may be read.
- *Write*: The contents of the file may be changed.

- *Execute*: The file may be executed (i.e., it is a program or a script). In order to execute a script file (e.g., a *bash* script), both read and execute permissions are required.

The permissions and ownership of a file can be viewed using the command *ls –l*, as in the following example:

```
$ ls -l myscript.sh
-rwxr-x---    1 mtk      users        1667 Jan 15 09:22 myscript.sh
```

In the above example, the file permissions are displayed as `rwxr-x---` (the initial hyphen preceding this string indicates the type of this file: a regular file). To interpret this string, we break these 9 characters into sets of 3 characters, which respectively indicate whether read, write, and execute permission are enabled. The first set indicates the permissions for owner, which has read, write, and execute permissions enabled. The next set indicates the permissions for group, which has read and execute enabled, but not write. The final set are the permissions for other, which doesn't have any permissions enabled.

The `<sys/stat.h>` header file defines constants that can be ANDed (`&`) with *st_mode* of the *stat* structure, in order to check whether particular permission bits are set. (These constants are also defined via the inclusion of `<fcntl.h>`, which prototypes the *open()* system call.) These constants are shown in Table 15-4.

Table 15-4: Constants for file permission bits

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

In addition to the constants shown in Table 15-4, three constants are defined to equate to masks for all three permissions for each of the categories owner, group, and other: `S_IRWXU` (0700), `S_IRWXG` (070), and `S_IRWXO` (07).

The header file in Listing 15-3 declares a function, *filePermStr()*, which, given a file permissions mask, returns a statically allocated string representation of that mask in the same style as is used by *ls(1)*.

Listing 15-3: Header file for `file_perms.c`

```
#ifndef FILE_PERMS_H
#define FILE_PERMS_H

#include <sys/types.h>

#define FP_SPECIAL 1            /* Include set-user-ID, set-group-ID, and sticky
                                   bit information in returned string */

char *filePermStr(mode_t perm, int flags);

#endif
```

files/file_perms.h

If the `FP_SPECIAL` flag is set in the *filePermStr() flags* argument, then the returned string includes the settings of the set-user-ID, set-group-ID, and sticky bits, again in the style of *ls(1)*.

The implementation of the *filePermStr()* function is shown in Listing 15-4. We employ this function in the program in Listing 15-1.

Listing 15-4: Convert file permissions mask to string

```
#include <sys/stat.h>
#include <stdio.h>
#include "file_perms.h"                 /* Interface for this implementation */

#define STR_SIZE sizeof("rwxrwxrwx")

char *          /* Return ls(1)-style string for file permissions mask */
filePermStr(mode_t perm, int flags)
{
    static char str[STR_SIZE];

    snprintf(str, STR_SIZE, "%c%c%c%c%c%c%c%c%c",
        (perm & S_IRUSR) ? 'r' : '-', (perm & S_IWUSR) ? 'w' : '-',
        (perm & S_IXUSR) ?
            (((perm & S_ISUID) && (flags & FP_SPECIAL)) ? 's' : 'x') :
            (((perm & S_ISUID) && (flags & FP_SPECIAL)) ? 'S' : '-'),
        (perm & S_IRGRP) ? 'r' : '-', (perm & S_IWGRP) ? 'w' : '-',
        (perm & S_IXGRP) ?
            (((perm & S_ISGID) && (flags & FP_SPECIAL)) ? 's' : 'x') :
            (((perm & S_ISGID) && (flags & FP_SPECIAL)) ? 'S' : '-'),
        (perm & S_IROTH) ? 'r' : '-', (perm & S_IWOTH) ? 'w' : '-',
        (perm & S_IXOTH) ?
            (((perm & S_ISVTX) && (flags & FP_SPECIAL)) ? 't' : 'x') :
            (((perm & S_ISVTX) && (flags & FP_SPECIAL)) ? 'T' : '-'));

    return str;
}
```

files/file_perms.c

15.4.2 Permissions on Directories

Directories have the same permission scheme as files. However, the three permissions are interpreted differently:

- *Read*: The contents (i.e., the list of filenames) of the directory may be listed (e.g., by *ls*).

  > If experimenting to verify the operation of the directory read permission bit, be aware that some Linux distributions alias the *ls* command to include flags (e.g., *–F*) that require access to i-node information for files in the directory, and this requires execute permission on the directory. To ensure that we are using an unadulterated *ls*, we can specify the full pathname of the command (`/bin/ls`).

- *Write*: Files may be created in and removed from the directory. Note that it is not necessary to have any permission on a file itself in order to be able to delete it.
- *Execute*: Files within the directory may be accessed. Execute permission on a directory is sometimes called *search* permission.

When accessing a file, execute permission is required on all of the directories listed in the pathname. For example, reading the file `/home/mtk/x` would require execute permission on `/`, `/home`, and `/home/mtk` (as well as read permission on the file `x` itself). If the current working directory is `/home/mtk/sub1` and we access the relative pathname `../sub2/x`, then we need execute permission on `/home/mtk` and `/home/mtk/sub2` (but not on `/` or `/home`).

Read permission on a directory only lets us view the list of filenames in the directory. We must have execute permission on the directory in order to access the contents or the i-node information of files in the directory.

Conversely, if we have execute permission on a directory, but not read permission, then we can access a file in the directory if we know its name, but we can't list the contents of (i.e., the other filenames in) the directory. This is a simple and frequently used technique to control access to the contents of a public directory.

To add or remove files in a directory, we need both execute and write permissions on the directory.

15.4.3 Permission-Checking Algorithm

The kernel checks file permissions whenever we specify a pathname in a system call that accesses a file or directory. When the pathname given to the system call includes a directory prefix, then, in addition to checking for the required permissions on the file itself, the kernel also checks for execute permission on each of the directories in this prefix. Permission checks are made using the process's effective user ID, effective group ID, and supplementary group IDs. (To be strictly accurate, for file permission checks on Linux, the file-system user and group IDs are used instead of the corresponding effective IDs, as described in Section 9.5.)

> Once a file has been opened with *open()*, no permission checking is performed by subsequent system calls that work with the returned file descriptor (such as *read()*, *write()*, *fstat()*, *fcntl()*, and *mmap()*).

The rules applied by the kernel when checking permissions are as follows:

1. If the process is privileged, all access is granted.
2. If the effective user ID of the process is the same as the user ID (owner) of the file, then access is granted according to the *owner* permissions on the file. For example, read access is granted if the owner-read permission bit is turned on in the file permissions mask; otherwise, read access is denied.
3. If the effective group ID of the process or any of the process supplementary group IDs matches the group ID (group owner) of the file, then access is granted according to the *group* permissions on the file.
4. Otherwise, access is granted according to the *other* permissions on the file.

> In the kernel code, the above tests are actually constructed so that the test to see whether a process is privileged is performed only if the process is not granted the permissions it needs via one of the other tests. This is done to avoid unnecessarily setting the ASU process accounting flag, which indicates that the process made use of superuser privileges (Section 28.1).

The checks against owner, group, and other permissions are done in order, and checking stops as soon as the applicable rule is found. This can have an unexpected consequence: if, for example, the permissions for group exceed those of owner, then the owner will actually have fewer permissions on the file than members of the file's group, as illustrated by the following example:

```
$ echo 'Hello world' > a.txt
$ ls -l a.txt
-rw-r--r--   1 mtk    users    12 Jun 18 12:26 a.txt
$ chmod u-rw a.txt              Remove read and write permission from owner
$ ls -l a.txt
----r--r--   1 mtk    users    12 Jun 18 12:26 a.txt
$ cat a.txt
cat: a.txt: Permission denied   Owner can no longer read file
$ su avr                        Become someone else...
Password:
$ groups                        who is in the group owning the file...
users staff teach cs
$ cat a.txt                     and thus can read the file
Hello world
```

Similar remarks apply if other grants more permissions than owner or group.

Since file permissions and ownership information are maintained within a file i-node, all filenames (links) that refer to the same i-node share this information.

Linux 2.6 provides access control lists (ACLs), which make it possible to define file permissions on a per-user and per-group basis. If a file has an ACL, then a modified version of the above algorithm is used. We describe ACLs in Chapter 17.

Permission checking for privileged processes

Above, we said that if a process is privileged, all access is granted when checking permissions. We need to add one proviso to this statement. For a file that is not a directory, Linux grants execute permission to a privileged process only if that permission is granted to at least one of the permission categories for the file. On some

other UNIX implementations, a privileged process can execute a file even when no permission category grants execute permission. When accessing a directory, a privileged process is always granted execute (search) permission.

> We can rephrase our description of a privileged process in terms of two Linux process capabilities: CAP_DAC_READ_SEARCH and CAP_DAC_OVERRIDE (Section 39.2). A process with the CAP_DAC_READ_SEARCH capability always has read permission for any type of file, and always has read and execute permissions for a directory (i.e., can always access files in a directory and read the list of files in a directory). A process with the CAP_DAC_OVERRIDE capability always has read and write permissions for any type of file, and also has execute permission if the file is a directory or if execute permission is granted to at least one of the permission categories for the file.

15.4.4 Checking File Accessibility: *access()*

As noted in Section 15.4.3, the *effective* user and group IDs, as well as supplementary group IDs, are used to determine the permissions a process has when accessing a file. It is also possible for a program (e.g., a set-user-ID or set-group-ID program) to check file accessibility based on the *real* user and group IDs of the process.

The *access()* system call checks the accessibility of the file specified in *pathname* based on a process's real user and group IDs (and supplementary group IDs).

```
#include <unistd.h>

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

Returns 0 if all permissions are granted, otherwise −1

If *pathname* is a symbolic link, *access()* dereferences it.

The *mode* argument is a bit mask consisting of one or more of the constants shown in Table 15-5, ORed (|) together. If all of the permissions specified in *mode* are granted on *pathname*, then *access()* returns 0; if at least one of the requested permissions is not available (or an error occurred), then *access()* returns −1.

Table 15-5: *mode* constants for *access()*

Constant	Description
F_OK	Does the file exist?
R_OK	Can the file be read?
W_OK	Can the file be written?
X_OK	Can the file be executed?

The time gap between a call to *access()* and a subsequent operation on a file means that there is no guarantee that the information returned by *access()* will still be true at the time of the later operation (no matter how brief the interval). This situation could lead to security holes in some application designs.

Suppose, for example, that we have a set-user-ID-*root* program that uses *access()* to check that a file is accessible to the real user ID of the program, and, if so, performs an operation on the file (e.g., *open()* or *exec()*).

The problem is that if the pathname given to *access()* is a symbolic link, and a malicious user manages to change the link so that it refers to a different file before the second step, then the set-user-ID-*root* may end up operating on a file for which the real user ID does not have permission. (This is an example of the type of time-of-check, time-of-use race condition described in Section 38.6.) For this reason, recommended practice is to avoid the use of *access()* altogether (see, for example, [Borisov, 2005]). In the example just given, we can achieve this by temporarily changing the effective (or file system) user ID of the set-user-ID process, attempting the desired operation (e.g., *open()* or *exec()*), and then checking the return value and *errno* to determine whether the operation failed because of a permissions problem.

> The GNU C library provides an analogous, nonstandard function, *euidaccess()* (or synonymously, *eaccess()*), that checks file access permissions using the effective user ID of the process.

15.4.5 Set-User-ID, Set-Group-ID, and Sticky Bits

As well as the 9 bits used for owner, group, and other permissions, the file permissions mask contains 3 additional bits, known as the *set-user-ID* (bit 04000), *set-group-ID* (bit 02000), and *sticky* (bit 01000) bits. We have already discussed the use of the set-user-ID and set-group-ID permission bits for creating privileged programs in Section 9.3. The set-group-ID bit also serves two other purposes that we describe elsewhere: controlling the group ownership of new files created in a directory mounted with the *nogrpid* option (Section 15.3.1), and enabling mandatory locking on a file (Section 55.4). In the remainder of this section, we limit our discussion to the use of the sticky bit.

On older UNIX implementations, the sticky bit was provided as a way of making commonly used programs run faster. If the sticky bit was set on a program file, then the first time the program was executed, a copy of the program text was saved in the swap area—thus it *sticks* in swap, and loads faster on subsequent executions. Modern UNIX implementations have more sophisticated memory-management systems, which have rendered this use of the sticky permission bit obsolete.

> The name of the constant for the sticky permission bit shown in Table 15-4, `S_ISVTX`, derives from an alternative name for the sticky bit: the *saved-text* bit.

In modern UNIX implementations (including Linux), the sticky permission bit serves another, quite different purpose. For directories, the sticky bit acts as the *restricted deletion* flag. Setting this bit on a directory means that an unprivileged process can unlink (*unlink()*, *rmdir()*) and rename (*rename()*) files in the directory only if it has write permission on the directory *and* owns either the file or the directory. (A process with the `CAP_FOWNER` capability can bypass the latter ownership check.) This makes it possible to create a directory that is shared by many users, who can each create and delete their own files in the directory but can't delete files owned by other users. The sticky permission bit is commonly set on the `/tmp` directory for this reason.

A file's sticky permission bit is set via the *chmod* command (*chmod +t file*) or via the *chmod()* system call. If the sticky bit for a file is set, *ls −l* shows a lowercase or uppercase letter *T* in the other-execute permission field, depending on whether the other-execute permission bit is on or off, as in the following:

```
$ touch tfile
$ ls -l tfile
-rw-r--r--   1 mtk    users     0 Jun 23 14:44 tfile
$ chmod +t tfile
$ ls -l tfile
-rw-r--r-T   1 mtk    users     0 Jun 23 14:44 tfile
$ chmod o+x tfile
$ ls -l tfile
-rw-r--r-t   1 mtk    users     0 Jun 23 14:44 tfile
```

15.4.6 The Process File Mode Creation Mask: *umask()*

We now consider in more detail the permissions that are placed on a newly created file or directory. For new files, the kernel uses the permissions specified in the *mode* argument to *open()* or *creat()*. For new directories, permissions are set according to the *mode* argument to *mkdir()*. However, these settings are modified by the file mode creation mask, also known simply as the *umask*. The umask is a process attribute that specifies which permission bits should always be turned *off* when new files or directories are created by the process.

Often, a process just uses the umask it inherits from its parent shell, with the (usually desirable) consequence that the user can control the umask of programs executed from the shell using the shell built-in command *umask*, which changes the umask of the shell process.

The initialization files for most shells set the default umask to the octal value 022 (`----w--w-`). This value specifies that write permission should always be turned off for group and other. Thus, assuming the *mode* argument in a call to *open()* is 0666 (i.e., read and write permitted for all users, which is typical), then new files are created with read and write permissions for owner, and only read permission for everyone else (displayed by *ls −l* as `rw-r--r--`). Correspondingly, assuming that the *mode* argument to *mkdir()* is specified as 0777 (i.e., all permissions granted to all users), new directories are created with all permissions granted for owner, and just read and execute permissions for group and other (i.e., `rwxr-xr-x`).

The *umask()* system call changes a process's umask to the value specified in *mask*.

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

mode_t umask(mode_t mask);
                                Always successfully returns the previous process umask
```

The *mask* argument can be specified either as an octal number or by ORing (|) together the constants listed in Table 15-4.

A call to *umask()* is always successful, and returns the previous umask.

Listing 15-5 illustrates the use of *umask()* in conjunction with *open()* and *mkdir()*. When we run this program, we see the following:

```
$ ./t_umask
Requested file perms: rw-rw----            This is what we asked for
Process umask:        ----wx-wx            This is what we are denied
Actual file perms:    rw-r-----            So this is what we end up with

Requested dir. perms: rwxrwxrwx
Process umask:        ----wx-wx
Actual dir. perms:    rwxr--r--
```

In Listing 15-5, we employ the *mkdir()* and *rmdir()* system calls to create and remove a directory, and the *unlink()* system call to remove a file. We describe these system calls in Chapter 18.

Listing 15-5: Using *umask()*

files/t_umask.c

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

#define MYFILE "myfile"
#define MYDIR  "mydir"
#define FILE_PERMS    (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)
#define DIR_PERMS     (S_IRWXU | S_IRWXG | S_IRWXO)
#define UMASK_SETTING (S_IWGRP | S_IXGRP | S_IWOTH | S_IXOTH)

int
main(int argc, char *argv[])
{
    int fd;
    struct stat sb;
    mode_t u;

    umask(UMASK_SETTING);

    fd = open(MYFILE, O_RDWR | O_CREAT | O_EXCL, FILE_PERMS);
    if (fd == -1)
        errExit("open-%s", MYFILE);
    if (mkdir(MYDIR, DIR_PERMS) == -1)
        errExit("mkdir-%s", MYDIR);

    u = umask(0);               /* Retrieves (and clears) umask value */

    if (stat(MYFILE, &sb) == -1)
        errExit("stat-%s", MYFILE);
    printf("Requested file perms: %s\n", filePermStr(FILE_PERMS, 0));
    printf("Process umask:        %s\n", filePermStr(u, 0));
    printf("Actual file perms:    %s\n\n", filePermStr(sb.st_mode, 0));
```

```
    if (stat(MYDIR, &sb) == -1)
        errExit("stat-%s", MYDIR);
    printf("Requested dir. perms: %s\n", filePermStr(DIR_PERMS, 0));
    printf("Process umask:        %s\n", filePermStr(u, 0));
    printf("Actual dir. perms:    %s\n", filePermStr(sb.st_mode, 0));

    if (unlink(MYFILE) == -1)
        errMsg("unlink-%s", MYFILE);
    if (rmdir(MYDIR) == -1)
        errMsg("rmdir-%s", MYDIR);
    exit(EXIT_SUCCESS);
}
```

files/t_umask.c

15.4.7 Changing File Permissions: *chmod()* and *fchmod()*

The *chmod()* and *fchmod()* system calls change the permissions of a file.

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

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

#define _XOPEN_SOURCE 500     /* Or: #define _BSD_SOURCE */
#include <sys/stat.h>

int fchmod(int fd, mode_t mode);
```

Both return 0 on success, or −1 on error

The *chmod()* system call changes the permissions of the file named in *pathname*. If this argument is a symbolic link, *chmod()* changes the permissions of the file to which it refers, rather than the permissions of the link itself. (A symbolic link is always created with read, write, and execute permissions enabled for all users, and these permission can't be changed. These permissions are ignored when dereferencing the link.)

The *fchmod()* system call changes the permissions on the file referred to by the open file descriptor *fd*.

The *mode* argument specifies the new permissions of the file, either numerically (octal) or as a mask formed by ORing (|) the permission bits listed in Table 15-4. In order to change the permissions on a file, either the process must be privileged (CAP_FOWNER) or its effective user ID must match the owner (user ID) of the file. (To be strictly accurate, on Linux, for an unprivileged process, it is the process's file-system user ID, rather than its effective user ID, that must match the user ID of the file, as described in Section 9.5.)

To set the permissions on a file so that only read permission is granted to all users, we could use the following call:

```
    if (chmod("myfile", S_IRUSR | S_IRGRP | S_IROTH) == -1)
        errExit("chmod");
    /* Or equivalently: chmod("myfile", 0444); */
```

In order to modify selected bits of the file permissions, we first retrieve the existing permissions using *stat()*, tweak the bits we want to change, and then use *chmod()* to update the permissions:

```
struct stat sb;
mode_t mode;

if (stat("myfile", &sb) == -1)
    errExit("stat");
mode = (sb.st_mode | S_IWUSR) & ~S_IROTH;
       /* owner-write on, other-read off, remaining bits unchanged */
if (chmod("myfile", mode) == -1)
    errExit("chmod");
```

The above is equivalent to the following shell command:

```
$ chmod u+w,o-r myfile
```

In Section 15.3.1, we noted that if a directory resides on an *ext2* system mounted with the *−o bsdgroups* option, or on one mounted with the *−o sysvgroups* option *and* the set-group-ID permission bit is turned on for the directory, then a newly created file in the directory takes its ownership from the parent directory, not the effective group ID of the creating process. It may be the case that the group ID of such a file doesn't match any of the group IDs of the creating process. For this reason, when an unprivileged process (one that doesn't have the `CAP_FSETID` capability) calls *chmod()* (or *fchmod()*) on a file whose group ID is not equal to the effective group ID or any of the supplementary group IDs of the process, the kernel always clears the set-group-ID permission bit. This is a security measure designed to prevent a user from creating a set-group-ID program for a group of which they are not a member. The following shell commands show the attempted exploit that this measure prevents:

```
$ mount | grep test                 Hmmm, /test is mounted with −o bsdgroups
/dev/sda9 on /test type ext3 (rw,bsdgroups)
$ ls -ld /test                      Directory has GID root, writable by anyone
drwxrwxrwx   3 root   root    4096 Jun 30 20:11 /test
$ id                                I'm an ordinary user, not part of root group
uid=1000(mtk) gid=100(users) groups=100(users),101(staff),104(teach)
$ cd /test
$ cp ~/myprog .                     Copy some mischievous program here
$ ls -l myprog                      Hey! It's in the root group!
-rwxr-xr-x   1 mtk    root   19684 Jun 30 20:43 myprog
$ chmod g+s myprog                  Can I make it set-group-ID to root?
$ ls -l myprog                      Hmm, no...
-rwxr-xr-x   1 mtk    root   19684 Jun 30 20:43 myprog
```

15.5 I-node Flags (*ext2* Extended File Attributes)

Some Linux file systems allow various *i-node flags* to be set on files and directories. This feature is a nonstandard Linux extension.

> The modern BSDs provide a similar feature to i-node flags in the form of file flags set using *chflags(1)* and *chflags(2)*.

The first Linux file system to support i-node flags was *ext2*, and these flags are sometimes referred to as *ext2 extended file attributes*. Subsequently, support for i-node flags has been added on other file systems, including *Btrfs*, *ext3*, *ext4*, *Reiserfs* (since Linux 2.4.19), *XFS* (since Linux 2.4.25 and 2.6), and *JFS* (since Linux 2.6.17).

> The range of i-node flags supported varies somewhat across file systems. In order to use i-node flags on a *Reiserfs* file system, we must use the *mount −o attrs* option when mounting the file system.

From the shell, i-node flags can be set and viewed using the *chattr* and *lsattr* commands, as shown in the following example:

```
$ lsattr myfile
-------- myfile
$ chattr +ai myfile                    Turn on Append Only and Immutable flags
$ lsattr myfile
----ia-- myfile
```

Within a program, i-node flags can be retrieved and modified using the *ioctl()* system call, as detailed shortly.

I-node flags can be set on both regular files and directories. Most i-node flags are intended for use with regular files, although some of them also (or only) have meaning for directories. Table 15-6 summarizes the range of available i-node flags, showing the corresponding flag name (defined in `<linux/fs.h>`) that is used from programs in *ioctl()* calls, and the option letter that is used with the *chattr* command.

> Before Linux 2.6.19, the `FS_*` constants shown in Table 15-6 were not defined in `<linux/fs.h>`. Instead, there was a set of file system–specific header files that defined file system–specific constant names, all with the same value. Thus, *ext2* had `EXT2_APPEND_FL`, defined in `<linux/ext2_fs.h>`; *Reiserfs* had `REISERFS_APPEND_FL`, defined with the same value in `<linux/reiser_fs.h>`; and so on. Since each of the header files defines the corresponding constants with the same value, on older systems that don't provide the definitions in `<linux/fs.h>`, it is possible to include any of the header files and use the file system–specific names.

Table 15-6: I-node flags

Constant	*chattr* option	Purpose
`FS_APPEND_FL`	`a`	Append only (privilege required)
`FS_COMPR_FL`	`c`	Enable file compression (not implemented)
`FS_DIRSYNC_FL`	`D`	Synchronous directory updates (since Linux 2.6)
`FS_IMMUTABLE_FL`	`i`	Immutable (privilege required)
`FS_JOURNAL_DATA_FL`	`j`	Enable data journaling (privilege required)
`FS_NOATIME_FL`	`A`	Don't update file last access time
`FS_NODUMP_FL`	`d`	No dump
`FS_NOTAIL_FL`	`t`	No tail packing
`FS_SECRM_FL`	`s`	Secure deletion (not implemented)
`FS_SYNC_FL`	`S`	Synchronous file (and directory) updates
`FS_TOPDIR_FL`	`T`	Treat as top-level directory for Orlov (since Linux 2.6)
`FS_UNRM_FL`	`u`	File can be undeleted (not implemented)

The various FL_* flags and their meanings are as follows:

FS_APPEND_FL
: The file can be opened for writing only if the O_APPEND flag is specified (thus forcing all file updates to append to the end of the file). This flag could be used for a log file, for example. Only privileged (CAP_LINUX_IMMUTABLE) processes can set this flag.

FS_COMPR_FL
: Store the file on disk in a compressed format. This feature is not implemented as a standard part of any of the major native Linux file systems. (There are packages that implement this feature for *ext2* and *ext3*.) Given the low cost of disk storage, the CPU overhead involved in compression and decompression, and the fact that compressing a file means that it is no longer a simple matter to randomly access the file's contents (via *lseek()*), file compression is undesirable for many applications.

FS_DIRSYNC_FL (since Linux 2.6)
: Make directory updates (e.g., *open(pathname, O_CREAT)*, *link()*, *unlink()*, and *mkdir()*) synchronous. This is analogous to the synchronous file update mechanism described in Section 13.3. As with synchronous file updates, there is a performance impact associated with synchronous directory updates. This setting can be applied only to directories. (The MS_DIRSYNC mount flag described in Section 14.8.1 provides similar functionality, but on a per-mount basis.)

FS_IMMUTABLE_FL
: Make the file immutable. File data can't be updated (*write()* and *truncate()*) and metadata changes are prevented (e.g., *chmod()*, *chown()*, *unlink()*, *link()*, *rename()*, *rmdir()*, *utime()*, *setxattr()*, and *removexattr()*). Only privileged (CAP_LINUX_IMMUTABLE) processes can set this flag for a file. When this flag is set, even a privileged process can't change the file contents or metadata.

FS_JOURNAL_DATA_FL
: Enable journaling of data. This flag is supported only on the *ext3* and *ext4* file systems. These file systems provide three levels of journaling: *journal*, *ordered*, and *writeback*. All modes journal updates to file metadata, but the *journal* mode additionally journals updates to file data. On a file system that is journaling in *ordered* or *writeback* mode, a privileged (CAP_SYS_RESOURCE) process can enable journaling of data updates on a per-file basis by setting this flag. (The *mount(8)* manual page describes the difference between the *ordered* and *writeback* modes.)

FS_NOATIME_FL
: Don't update the file last access time when the file is accessed. This eliminates the need to update the file's i-node each time the file is accessed, thus improving I/O performance (see the description of the MS_NOATIME flag in Section 14.8.1).

`FS_NODUMP_FL`
: Don't include this file in backups made using *dump(8)*. The effect of this flag is dependent on the *−h* option described in the *dump(8)* manual page.

`FS_NOTAIL_FL`
: Disable tail packing. This flag is supported only on the *Reiserfs* file system. It disables the *Reiserfs* tail-packing feature, which tries to pack small files (and the final fragment of larger files) into the same disk block as the file metadata. Tail packing can also be disabled for an entire *Reiserfs* file system by mounting it with the *mount −notail* option.

`FS_SECRM_FL`
: Delete the file securely. The intended purpose of this unimplemented feature is that, when removed, a file is securely deleted, meaning that it is first overwritten to prevent a disk-scanning program from reading or re-creating it. (The issue of truly secure deletion is rather complex: it can actually require multiple writes on magnetic media to securely erase previously recorded data; see [Gutmann, 1996].)

`FS_SYNC_FL`
: Make file updates synchronous. When applied to files, this flag causes writes to the file to be synchronous (as though the `O_SYNC` flag was specified on all opens of this file). When applied to a directory, this flag has the same effect as the synchronous directory updates flag described above.

`FS_TOPDIR_FL` (since Linux 2.6)
: This marks a directory for special treatment under the *Orlov* block-allocation strategy. The Orlov strategy is a BSD-inspired modification of the *ext2* block-allocation strategy that tries to improve the chances that related files (e.g., the files within a single directory) are placed close to each other on disk, which can improve disk seek times. For details, see [Corbet, 2002] and [Kumar, et al. 2008]. `FS_TOPDIR_FL` has an effect only for *ext2* and its descendants, *ext3* and *ext4*.

`FS_UNRM_FL`
: Allow this file to be recovered (undeleted) if it is deleted. This feature is not implemented, since it is possible to implement file-recovery mechanisms outside the kernel.

Generally, when i-node flags are set on a directory, they are automatically inherited by new files and subdirectories created in that directory. There are exceptions to this rule:

- The `FS_DIRSYNC_FL` (*chattr +D*) flag, which can be applied only to a directory, is inherited only by subdirectories created in that directory.
- When the `FS_IMMUTABLE_FL` (*chattr +i*) flag is applied to a directory, it is not inherited by files and subdirectories created within that directory, since this flag prevents new entries being added to the directory.

Within a program, i-node flags can be retrieved and modified using the *ioctl()* FS_IOC_GETFLAGS and FS_IOC_SETFLAGS operations. (These constants are defined in <linux/fs.h>.) The following code shows how to enable the FS_NOATIME_FL flag on the file referred to by the open file descriptor *fd*:

```
int attr;

if (ioctl(fd, FS_IOC_GETFLAGS, &attr) == -1)    /* Fetch current flags */
    errExit("ioctl");
attr |= FS_NOATIME_FL;
if (ioctl(fd, FS_IOC_SETFLAGS, &attr) == -1)    /* Update flags */
    errExit("ioctl");
```

In order to change the i-node flags of a file, either the effective user ID of the process must match the user ID (owner) of the file, or the process must be privileged (CAP_FOWNER). (To be strictly accurate, on Linux, for an unprivileged process it is the process's file-system user ID, rather than its effective user ID, that must match the user ID of the file, as described in Section 9.5.)

15.6 Summary

The *stat()* system call retrieves information about a file (metadata), most of which is drawn from the file i-node. This information includes file ownership, file permissions, and file timestamps.

A program can update a file's last access time and last modification time using *utime()*, *utimes()*, and various similar interfaces.

Each file has an associated user ID (owner) and group ID, as well as a set of permission bits. For permissions purposes, file users are divided into three categories: *owner* (also known as *user*), *group*, and *other*. Three permissions may be granted to each category of user: *read*, *write*, and *execute*. The same scheme is used with directories, although the permission bits have slightly different meanings. The *chown()* and *chmod()* system calls change the ownership and permissions of a file. The *umask()* system call sets a mask of permission bits that are always turned off when the calling process creates a file.

Three additional permission bits are used for files and directories. The set-user-ID and set-group-ID permission bits can be applied to program files to create programs that cause the executing process to gain privilege by assuming a different effective user or group identity (that of the program file). For directories residing on file systems mounted using the *nogrpid* (*sysvgroups*) option, the set-group-ID permission bit can be used to control whether new files created in the directory inherit their group ID from the process's effective group ID or from the parent directory's group ID. When applied to directories, the sticky permission bit acts as the restricted deletion flag.

I-node flags control the various behaviors of files and directories. Although originally defined for *ext2*, these flags are now supported on several other file systems.

15.7 Exercises

15-1. Section 15.4 contained several statements about the permissions required for various file-system operations. Use shell commands or write programs to verify or answer the following:

a) Removing all owner permissions from a file denies the file owner access, even though group and other do have access.

b) On a directory with read permission but not execute permission, the names of files in the directory can be listed, but the files themselves can't be accessed, regardless of the permissions on them.

c) What permissions are required on the parent directory and the file itself in order to create a new file, open a file for reading, open a file for writing, and delete a file? What permissions are required on the source and target directory to rename a file? If the target file of a rename operation already exists, what permissions are required on that file? How does setting the sticky permission bit (*chmod +t*) of a directory affect renaming and deletion operations?

15-2. Do you expect any of a file's three timestamps to be changed by the *stat()* system call? If not, explain why.

15-3. On a system running Linux 2.6, modify the program in Listing 15-1 (`t_stat.c`) so that the file timestamps are displayed with nanosecond accuracy.

15-4. The *access()* system call checks permissions using the process's real user and group IDs. Write a corresponding function that performs the checks according to the process's effective user and group IDs.

15-5. As noted in Section 15.4.6, *umask()* always sets the process umask and, at the same time, returns a copy of the old umask. How can we obtain a copy of the current process umask while leaving it unchanged?

15-6. The *chmod a+rX file* command enables read permission for all categories of user, and likewise enables execute permission for all categories of user if *file* is a directory or execute permission is enabled for any of the user categories for *file*, as demonstrated in the following example:

```
$ ls -ld dir file prog
dr--------  2 mtk users    48 May  4 12:28 dir
-r--------  1 mtk users 19794 May  4 12:22 file
-r-x------  1 mtk users 19336 May  4 12:21 prog
$ chmod a+rX dir file prog
$ ls -ld dir file prog
dr-xr-xr-x  2 mtk users    48 May  4 12:28 dir
-r--r--r--  1 mtk users 19794 May  4 12:22 file
-r-xr-xr-x  1 mtk users 19336 May  4 12:21 prog
```

Write a program that uses *stat()* and *chmod()* to perform the equivalent of *chmod a+rX*.

15-7. Write a simple version of the *chattr(1)* command, which modifies file i-node flags. See the *chattr(1)* man page for details of the *chattr* command-line interface. (You don't need to implement the *−R*, *−V*, and *−v* options.)