3

Advanced File I/O

3.1 Introduction

This chapter picks up where Chapter 2 left off. First I'll extend the use of the I/O system calls already introduced to work on disk special files. I'll use that to look inside file systems. Then I'll introduce additional system calls that allow us to link to existing files; create, remove, and read directories; and obtain or modify file status information.

You're much less likely to use the advanced features covered in this chapter than those in Chapter 2. You will still benefit from knowing how to use them, however, because that will give you a more complete understanding of how file I/O works.

The program examples in this chapter are more extensive than those that have appeared so far. Careful study of these will be well worth your time, since they illustrate details not covered explicitly in the text.

3.2 Disk Special Files and File Systems

The section explains how to do I/O on disk special files, which is then used to access the internals of a UNIX file system. It also explains how mounting and unmounting work.

3.2.1 I/O on Disk Special Files

Until now we've done I/O exclusively through the kernel file system, using relatively high-level abstractions like file, directory, and i-node. As discussed in Section 2.12, the file system is implemented on top of the block I/O system, which uses the buffer cache.[1] The block I/O system is accessed via a block special file, or block device, that interfaces directly to the disk. The disk is treated as a sequence of blocks whose size is a multiple of the sector size, which is usually 512.

There may be several physical disks, and each physical disk may be divided into pieces, each of which is called a *volume*, *partition*, or *file system*. (The term *file system* is confusing because it also describes part of the kernel. The context in which we use the term will make our intended meaning clear.)

Each volume corresponds to a special file whose name is typically formed from a device name, such as "hd," followed by numbers and letters that indicate which section of which physical disk it occupies. These special files are usually linked into the /dev directory, although they don't have to be. For example, on Linux the file /dev/hdb3 refers to the third partition of the second physical hard disk.

In principle, a disk special file may be operated on with I/O system calls just as if it were a regular file. It may be opened for reading and/or writing, read or written (in arbitrary units), seeked (to any byte boundary), and closed. The buffer cache is used (since it is a block special file), but within the volume, there are no directories, files, i-nodes, permissions, owners, sizes, times, and so on. One just deals with a giant array of numbered blocks.

In practice, most users can't perform any of these operations because permission is denied to them. Reading a disk that contains other users' files compromises their privacy, even though the disk appears haphazard when viewed as a special file (blocks are assigned to UNIX files and directories in no obvious order). Writing to a disk without going through the kernel file system would create even worse havoc.

On the other hand, if a volume is reserved for a user, and not used for other users' files and directories, then there is no conflict. Users implementing database managers or data acquisition systems may indeed want to have a volume set aside so they can access it as a block special file. A limitation, of course, is that there are only so many disk special files to go around. Usually when a disk special file is used by an application, that application is the only one, or certainly the main one, on the computer.

Potentially even faster than block disk special files are *raw* disk special files. These device drivers deal with the same areas of disk. However, these raw spe-

1. Newer systems actually use the virtual-memory system rather than the buffer cache, but the cache is still a useful abstract model for how the kernel handles files.

cial files are *character* devices, not block devices. That means they do not follow the block model, and they do not use the buffer cache. They do something even better.

When a read or write is initiated on a raw special file, the process is locked into memory (prevented from swapping) so no physical data addresses can change. Then, if the hardware and driver support it, the disk is ordered to transfer data using DMA. Data flows directly between the process's data segment and the disk controller, without going through the kernel at all. The size of the transfer may be more than a block at a time.

Usually, I/O on raw devices is less flexible than it is with regular files and block special files. I/O in multiples of a disk sector is required. The DMA hardware may require the process's buffer address to be on a particular boundary. Seeks may be required to be to a block boundary only. These restrictions don't bother designers of database-oriented file systems, since they find it convenient to view a disk as being made of fixed-size pages anyhow.

So far, we have seen that UNIX features vary somewhat from version to version. Here, however, we have a variation that depends also on the hardware, the device drivers, and even the installation. Since computers vary enormously in their I/O hardware, device drivers vary accordingly. Also, the goals of the implementation effort affect the importance of making raw I/O fast. On a general-purpose desktop system, for example, it may be of very little importance.

Like synchronized I/O (Section 2.16), raw I/O has one tremendous advantage and one tremendous disadvantage. The tremendous advantage is that, since the process waits for a write to complete, and since there is no question of which process owns the data, the process can be informed about physical write errors via a –1 return value and an `errno` code. There is a code defined (`EIO`), but whether it is ever passed on is up to the device driver implementor. You'll have to check the system-specific documentation or even look in the code for the device driver to be sure.

The tremendous disadvantage of raw I/O is that, since the process waits for a read or write to complete, and since the design of UNIX allows a process to issue only a single `read` or `write` system call at a time, the process does I/O very fast but not very often. For example, in a multi-user database application with one centralized database manager process (a common arrangement), there will be a colossal I/O traffic jam if raw I/O is used, since only one process does the I/O for everyone. The solutions are multiple database processes, multiple threads (Section 5.17), or asynchronous I/O (Section 3.9).

A minor disadvantage of raw I/O that we can dispose of immediately is that while the process is locked in memory, other processes may not be able to run because there is no room to swap them in. This would be a shame, because DMA does not use the CPU, and these processes could otherwise actually execute. The solution is just to add some more memory. At today's prices there is no excuse for not having enough memory to avoid most, if not all, swapping, especially on a computer that runs an application important enough to be doing raw I/O.

Table 3.1 summarizes the functional differences between I/O on regular files, on block disk devices, and on raw disk devices.

Table 3.1 I/O Feature Comparison

Features	Regular File	Block Disk Device	Raw Disk Device
directories, files, i-nodes, permissions, etc.	yes	no	no
buffer cache	yes	yes	no
I/O error returns, DMA	no	no	yes

To show the speed differences, on Solaris we timed 10,000 reads of 65,536 bytes each. We read a regular file, a block disk device (/dev/dsk/c0d0p0), and a raw disk device (/dev/rdsk/c0d0p0). As I did in Chapter 2, in Table 3.2 I report system time, user time, and real time, in seconds. Since reading a raw disk device provides features somewhat similar to reading a regular file with the `O_RSYNC` flag, as explained in Section 2.16.3, I included times for that as well.

Table 3.2 I/O Timing Comparison

I/O Type	User	System	Real
Regular file	0.40	21.28	441.75
Regular file, `O_RSYNC | O_DSYNC`	0.25	25.50	412.05
Block disk device	0.38	22.50	562.78
Raw disk device	0.10	2.87	409.70

As you can see, the speed advantages of raw I/O on Solaris are enormous, and I got similar results on Linux. I didn't bother running a comparison for writes, since I know that the two file types that use the buffer cache would have won hands down, and because I didn't have a spare volume to scribble on. Of course, it wouldn't have been a fair comparison, since the times with the buffer cache wouldn't have included the physical I/O, and the raw I/O time would have included little else.

3.2.2 Low-Level Access to a File System

It's illuminating to look at the internal structure of a file system by reading it as a disk device (assuming you have read permission). You'll probably never do this in an application program, but that's what low-level file utilities such as `fsck` do.

There are more file-system designs than there are UNIX systems: UFS (UNIX File System) on Solaris, FFS (Fast File System) on FreeBSD (also called UFS), and ReiserFS and Ext2fs on Linux. What follows here is loosely based on the original (1970s) UNIX file system and on FFS.

The raw disk is treated as a sequence of blocks of a fixed size, say 2048 bytes. (The actual size doesn't matter for this discussion.) The first block or so is reserved for a boot program (if the disk is bootable), a disk label, and other such administrative information.

The file system proper starts at a fixed offset from the start of the disk with the *superblock;* it contains assorted structural information such as the number of i-nodes, the total number of blocks in the volume, the head of a linked list of free blocks, and so on. Each file (regular, directory, special, etc.) uses one i-node and must be linked to from at least one directory. Files that have data on disk—regular, directory, and symbolic link—also have data blocks pointed to from their i-nodes. I-nodes start at a location pointed to from the superblock.

I-nodes 0 and 1 aren't used.[2] I-node 2 is reserved for the root directory (/) so that when the kernel is given an absolute path, it can start from a known place. No other i-numbers have any particular significance; they are assigned to files as needed.

2. I-node numbering starts at 1, which allows 0 to be used to mean "no i-node" (e.g., an empty directory). Historically, i-node 1 was used to collect bad disk blocks.

The following program, which is very specific to FreeBSD's implementation of FFS, shows one way to access the superblock and an i-node by reading the disk device /dev/ad0s1g, which is a raw disk device that happens to contain the /usr directory tree. That this is so can be seen by executing the `mount` command (which knows FFS as type "ufs"):

```
$ mount
/dev/ad0s1a on / (ufs, NFS exported, local)
/dev/ad0s1f on /tmp (ufs, local, soft-updates)
/dev/ad0s1g on /usr (ufs, local, soft-updates)
/dev/ad0s1e on /var (ufs, local, soft-updates)
procfs on /proc (procfs, local)
```

Here's the program:

```
#ifndef FREEBSD
#error "Program is for FreeBSD only."
#endif

#include "defs.h"
#include <sys/param.h>
#include <ufs/ffs/fs.h>
#include <ufs/ufs/dinode.h>

#define DEVICE "/dev/ad0s1g"

int main(int argc, char *argv[])
{
    int fd;
    long inumber;
    char sb_buf[((sizeof(struct fs) / DEV_BSIZE) + 1) * DEV_BSIZE];
    struct fs *superblock = (struct fs *)sb_buf;
    struct dinode *d;
    ssize_t nread;
    off_t fsbo, fsba;
    char *inode_buf;
    size_t inode_buf_size;

    if (argc < 2) {
        printf("Usage: inode n\n");
        exit(EXIT_FAILURE);
    }
    inumber = atol(argv[1]);
```

```
    ec_neg1( fd = open(DEVICE, O_RDONLY) )
    ec_neg1( lseek(fd, SBLOCK * DEV_BSIZE, SEEK_SET) )
    switch (nread = read(fd, sb_buf, sizeof(sb_buf))) {
    case 0:
        errno = 0;
        printf("EOF from read (1)\n");
        EC_FAIL
    case -1:
        EC_FAIL
    default:
        if (nread != sizeof(sb_buf)) {
            errno = 0;
            printf("Read only %d bytes instead of %d\n", nread,
sizeof(sb_buf));
            EC_FAIL
        }
    }
    printf("Superblock info for %s:\n", DEVICE);
    printf("\tlast time written = %s", ctime(&superblock->fs_time));
    printf("\tnumber of blocks in fs = %ld\n", (long)superblock->fs_size);
    printf("\tnumber of data blocks in fs = %ld\n",
      (long)superblock->fs_dsize);
    printf("\tsize of basic blocks in fs = %ld\n",
      (long)superblock->fs_bsize);
    printf("\tsize of frag blocks in fs = %ld\n",
      (long)superblock->fs_fsize);
    printf("\tname mounted on = %s\n", superblock->fs_fsmnt);

    inode_buf_size = superblock->fs_bsize;
    ec_null( inode_buf = malloc(inode_buf_size) )

    fsba = ino_to_fsba(superblock, inumber);
    fsbo = ino_to_fsbo(superblock, inumber);

    ec_neg1( lseek(fd, fsbtodb(superblock, fsba) * DEV_BSIZE, SEEK_SET) )
    switch (nread = read(fd, inode_buf, inode_buf_size)) {
    case 0:
        errno = 0;
        printf("EOF from read (2)\n");
        EC_FAIL
    case -1:
        EC_FAIL
    default:
        if (nread != inode_buf_size) {
            errno = 0;
            printf("Read only %d bytes instead of %d\n",
              nread, inode_buf_size);
            EC_FAIL
        }
    }
```

```
    d = (struct dinode *)&inode_buf[fsbo * sizeof(struct dinode)];
    printf("\ninumber %ld info:\n", inumber);
    printf("\tmode = 0%o\n", d->di_mode);
    printf("\tlinks = %d\n", d->di_nlink);
    printf("\towner = %d\n", d->di_uid);
    printf("\tmod. time = %s", ctime((time_t *)&d->di_mtime));
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The complicated declaration for `sb_buf`:

```
char sb_buf[((sizeof(struct fs) / DEV_BSIZE) + 1) * DEV_BSIZE];
```

is to round up its size to an even sector, as that's a requirement for reading raw disk devices on FreeBSD.

The first `lseek` is to `SBLOCK * DEV_BSIZE`, which is the location of the superblock. `SBLOCK` happens to be 16, and `DEV_BSIZE` is the sector size, which is 512, as is true for most disks. The superblock always starts 16 sectors from the start of the file system. The first group of `printf`s print a few fields from the very long `fs` structure. This is that output:

```
Superblock info for /dev/ad0s1g:
        last time written = Mon Oct 14 15:25:25 2002
        number of blocks in fs = 1731396
        number of data blocks in fs = 1704331
        size of basic blocks in fs = 16384
        size of frag blocks in fs = 2048
        name mounted on = /usr
```

Next the program finds the i-node supplied as its argument (`argv[1]`) using some macros that are in one of the header files, as the arithmetic is very complicated on an FFS disk. I got the i-number of a file with the `-i` option on the `ls` command:

```
$ ls -li x
383642 -rwxr-xr-x  1 marc  marc  11687 Sep 19 13:29 x
```

And this was the rest of the output for i-node 383642:

```
inumber 383642 info:
        mode = 0100755
        links = 1
        owner = 1001
        mod. time = Thu Sep 19 13:29:30 2002
```

I'm not going to spend any more time on low-level disk access—all I really wanted was to illustrate how regular files and special files are related. But you might enjoy modifying the program a bit to display other information on a FreeBSD system if you have one, or even modifying it for another UNIX system that you do have.

3.2.3 `statvfs` and `fstatvfs` System Calls

Reading the superblock with `read` after finding it on a block or raw device is fun, but there's a much better way to do it on SUS1-conforming systems (Section 1.5.1) using the `statvfs` or `fstatvfs` system calls:

statvfs—get file system information by path

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

int statvfs(
    const char *path,      /* pathname */
    struct statvfs *buf    /* returned info */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fstatvfs—get file system information by file descriptor

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

int fstatvfs(
    int fd,                /* file descriptor */
    struct statvfs *buf    /* returned info */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Usually, you'll use `statvfs`, which returns information about the file system that contains the path given by its first argument. If you have a file opened, you can use `fstatvfs` instead, which returns the same information.

The standard defines fields for the `statvfs` structure, but implementations are not required to support them all. Also, most implementations support additional fields and additional flags for the `f_flag` field. You'll have to check your system's `man` page for the specifics. If an implementation doesn't support a standard field, the field is still there, but it won't contain meaningful information.

struct statvfs—structure for statvfs and fstatvfs

```
struct statvfs {
    unsigned long f_bsize;     /* block size */
    unsigned long f_frsize;    /* fundamental (fblock) size */
    fsblkcnt_t f_blocks;       /* total number of fblocks */
    fsblkcnt_t f_bfree;        /* number of free fblocks */
    fsblkcnt_t f_bavail;       /* number of avail. fblocks */
    fsfilcnt_t f_files;        /* total number of i-numbers */
    fsfilcnt_t f_ffree;        /* number of free i-numbers */
    fsfilcnt_t f_favail;       /* number of avail. i-numbers */
    unsigned long f_fsid;      /* file-system ID */
    unsigned long f_flag;      /* flags (see below) */
    unsigned long f_namemax;   /* max length of filename */
};
```

Some comments about this structure:

- Most file systems allocate fairly large blocks to files, given by the `f_bsize` field, to speed up access, but then finish off the file with smaller fragment blocks to avoid wasting space. The fragment-block size is called the *fundamental* block size and is given by the `f_frsize` field; we call it an *fblock* for short. The fields of type `fsblkcnt_t` are in units of fblocks, so, to get the total space in bytes, you would multiply `f_blocks` by `f_frsize`.
- The types `fsblkcnt_t` and `fsfilcnt_t` are unsigned, but otherwise implementation defined. They're typically `long` or `long long`. If you need to display one and you have a C99 compiler, cast it to `uintmax_t` and use `printf` format `%ju`; otherwise, cast it to `unsigned long long` and use format `%llu`.
- The SUS standards define only two flags for the `f_flag` field that indicate how the file system was mounted:

  `ST_RDONLY` is set if it's read only, and
  `ST_NOSUID` is set if the set-user-ID-on-execution and set-group-ID-on-execution bits are to be ignored on executable files (a security precaution).
- The term "available," which applies to the `f_bavail` and `f_favail` fields, means "available to nonsuperuser processes." It might be less that the corresponding "free" fields, so as to reserve a minimum amount of free space on systems where performance suffers when free space gets tight.

FreeBSD, being pre-SUS, doesn't support the `statvfs` or `fstatvfs` functions, but it does have a similar function called `statfs` whose structure has a different name and mostly different fields. For the basic information, it's possible to code in a way that works with both `statvfs` and `statfs`, as the following program illustrates.

```
#if _XOPEN_SOURCE >= 4
#include <sys/statvfs.h>
#define FCN_NAME statvfs
#define STATVFS 1

#elif defined(FREEBSD)
#include <sys/param.h>
#include <sys/mount.h>
#define FCN_NAME statfs

#else
#error "Need statvfs or nonstandard substitute"
#endif

void print_statvfs(const char *path)
{
    struct FCN_NAME buf;

    if (path == NULL)
        path = ".";
    ec_neg1( FCN_NAME(path, &buf) )
#ifdef STATVFS
    printf("block size = %lu\n", buf.f_bsize);
    printf("fundamental block (fblock) size = %lu\n", buf.f_frsize);
#else
    printf("block size = %lu\n", buf.f_iosize);
    printf("fundamental block size = %lu\n", buf.f_bsize);
#endif
    printf("total number of fblocks = %llu\n",
      (unsigned long long)buf.f_blocks);
    printf("number of free fblocks = %llu\n",
      (unsigned long long)buf.f_bfree);
    printf("number of avail. fblocks = %llu\n",
      (unsigned long long)buf.f_bavail);
    printf("total number of i-numbers = %llu\n",
      (unsigned long long)buf.f_files);
    printf("number of free i-numbers = %llu\n",
      (unsigned long long)buf.f_ffree);
#ifdef STATVFS
    printf("number of avail. i-numbers = %llu\n",
      (unsigned long long)buf.f_favail);
    printf("file-system ID = %lu\n", buf.f_fsid);
    printf("Read-only = %s\n",
      (buf.f_flag & ST_RDONLY) == ST_RDONLY ? "yes" : "no");
    printf("No setuid/setgid = %s\n",
      (buf.f_flag & ST_NOSUID) == ST_NOSUID ? "yes" : "no");
    printf("max length of filename = %lu\n", buf.f_namemax);
#else
```

```
    printf("Read-only = %s\n",
      (buf.f_flags & MNT_RDONLY) == MNT_RDONLY ? "yes" : "no");
    printf("No setuid/setgid = %s\n",
      (buf.f_flags & MNT_NOSUID) == MNT_NOSUID ? "yes" : "no");
#endif
    printf("\nFree space = %.0f%%\n",
      (double)buf.f_bfree * 100 / buf.f_blocks);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("print_statvfs");
EC_CLEANUP_END
}
```

Here's the output we got on Solaris for the file system that contains the directory /home/marc/aup (Linux produces similar output):

```
block size = 8192
fundamental block (fblock) size = 1024
total number of fblocks = 4473046
number of free fblocks = 3683675
number of avail. fblocks = 3638945
total number of i-numbers = 566912
number of free i-numbers = 565782
number of avail. i-numbers = 565782
file-system ID = 26738695
Read-only = no
No setuid/setgid = no
max length of filename = 255

Free space = 82%
```

On FreeBSD this was the output for /usr/home/marc/aup:

```
block size = 16384
fundamental block size = 2048
total number of fblocks = 1704331
number of free fblocks = 1209974
number of avail. fblocks = 1073628
total number of i-numbers = 428030
number of free i-numbers = 310266
Read-only = no
No setuid/setgid = no

Free space = 71%
```

The `statvfs` (or `statfs`) system call is the heart of the well-known `df` ("disk free") command (see Exercise 3.2). Here's what it reported on FreeBSD:

```
$ df /usr
Filesystem  1K-blocks   Used   Avail Capacity  Mounted on
/dev/ad0s1g   3408662 988696 2147274    32%    /usr
```

The reported number of 1k-blocks, 3408662, equates to the number of 2k-blocks that our `print_statvfs` program showed, 1704331.

There's also a standard way to read the information in an i-node, so you don't have to read the device file as we did in the previous section. We'll get to that in Section 3.5.

3.2.4 Mounting and Unmounting File Systems

A UNIX system can have lots of disk file systems (hard disks, floppies, CD-ROMs, DVDs, etc.), but they're all accessible within a single directory tree that starts at the root. Connecting a file system to the existing hierarchy is called *mounting,* and disconnecting it *unmounting*. Figure 3.1 shows a large file system that contains the root (i-node 2, with directory entries x and y) and a smaller unconnected file system, with its own root, also numbered 2. (Recall that 2 is always the i-number for a root directory.)

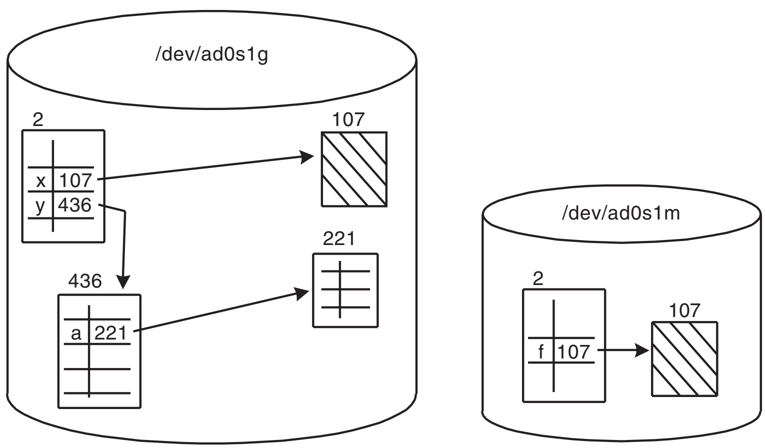

Figure 3.1 Two disconnected file systems.

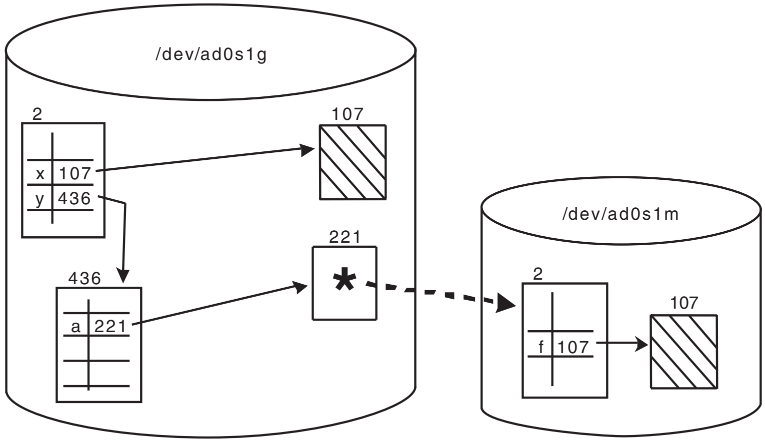

Figure 3.2 Mounted file system.

To mount the second file system, you need two things: Its device name, which is /dev/ad0s1m, and the directory where you want to connect it, for which we'll choose /y/a (i-node 221). Here's the command that creates the tree in Figure 3.2:

```
# mount /dev/ad0s1m /y/a
```

The old contents of directory /y/a are now hidden, and all references to that directory automatically become references to i-node ad0s1m:2, a notation that I'll use to mean "i-node 2 on device ad0s1m." Thus, the file ad0s1m:107 is now accessible as /y/a/f. When ad0s1m is unmounted, its contents are no longer accessible, and the old contents of ad0s1g:221 reappear.[3]

Every UNIX system has a superuser-only system call named `mount` and its undo-function `umount` (sometimes called `unmount`), but their arguments and exact behavior vary from system to system. Like other superuser-only functions, they're not standardized. In practice, these systems calls are used to implement the `mount` and `umount` commands and are almost never called

3. It's rare to actually have anything in a directory whose only purpose is to serve as a mount point, but it's allowed and occasionally used by system administrators to hide a directory.

directly. As an example, here are the Linux synopses, which I won't bother to explain completely:

mount—mount file system (nonstandard)

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

int mount(
    const char *source,        /* device */
    const char *target,        /* directory */
    const char *type,          /* type (e.g., ext2) */
    unsigned long flags,       /* mount flags (e.g., MS_RDONLY) */
    const void *data           /* file-system-dependent data */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

umount—unmount file system (nonstandard)

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

int umount(
    const char *target         /* directory */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Normally, `umount` fails with the error `EBUSY` if any file or directory on it is in use. There's usually an alternative system call named `umount2` that has a flag as a second argument that forces the unmount, which is occasionally essential when the system has to be shut down.

From the application-programming perspective, except for one situation, you're normally not concerned with the fact that a given file or directory you're trying to access is on a file system separately mounted from some other file or directory, as long as you have a path name that works. In fact, *all* accessible file systems, even the root, had to be mounted at one time, perhaps during the boot sequence. The one situation has to do with links, which is what the next section is about.

3.3 Hard and Symbolic Links

An entry in a directory, consisting of a name and an i-number, is called a *hard link*. The other kind of link is a *symbolic link*, which I'll talk about in a moment. Figure 3.3 illustrates both. (The hex numbers in parentheses are device IDs, which I'll explain later.)

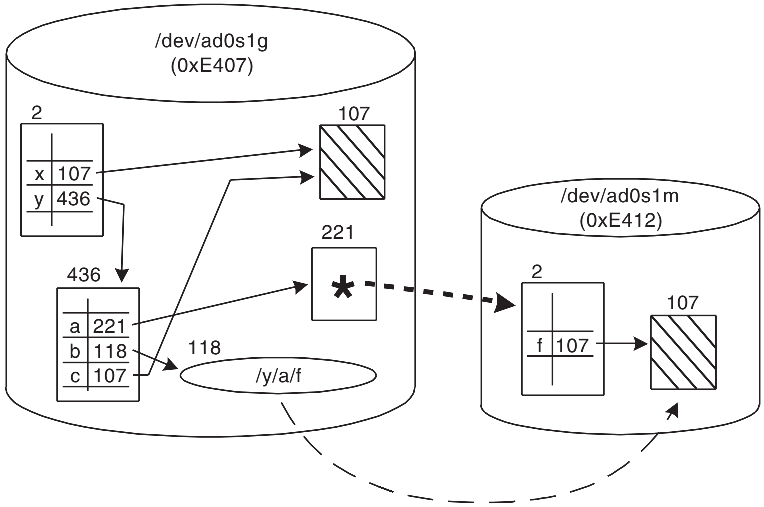

Figure 3.3 Hard and symbolic links.

3.3.1 Creating Hard Links (`link` System Call)

You get a hard link when a file of any type, including directory, is created. You can get additional hard links to nondirectories[4] with the `link` system call:

link—create hard link

```
#include <unistd.h>

int link(
     const char *oldpath,       /* old pathname */
     const char *newpath        /* new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The first argument, `oldpath`, must be an existing link; it provides the i-number to be used. The second argument, `newpath`, indicates the name of the new link. The links are equal in every way, since UNIX has no notion of primary and secondary links. The process must have write permission on the directory that is to contain

4. On some systems the superuser can link to an existing directory, but doing so means the directory structure is no longer a tree, complicating system administration.

the new link. The link specified by the second argument must not already exist; `link` can't be used to change an existing link. In this case the old link must first be removed with `unlink` (Section 2.6), or the `rename` system call can be used (next section).

3.3.2 Renaming a File or Directory (`rename` System Call)

At first thought it seems easy to rename a file, if by that you mean "change the name in the directory entry." All you do is make a second hard link with `link` and remove the old link with `unlink`, but there are many complications:

- The file may be a directory, which you can't make a second hard link to.
- Sometimes you want the new name to be in a different directory, which is no problem if it's on the same file system, but a big problem if it isn't, because `link` only works within a file system. You can, of course, create a symbolic link (next section) in the new directory, but that's not what most people mean by "rename."
- If it's a directory that you want to "rename" ("move," really) between file systems, you have to move the whole subtree beneath it. Moving only empty directories is too restrictive.
- If there are multiple hard links and you're renaming the file within the same file system, they're going to be OK, because they reference the i-number, which won't change. But if you somehow manage to move the file to another file system, the old links will no longer be valid. What might happen, since you have to copy the file and then unlink the original, is that the old copy will stay around with all but the moved hard link still linked to it. Not good.
- If there are symbolic links to a path and you change that path, the symbolic links become dead ends. Also not good.

There are probably even more complications that we could list if we thought about it some more, but you get the idea, and we're already getting a headache. That's why the UNIX `mv` command, which is what you use to "rename" a file or directory, is a very complex piece of code. And it doesn't even deal with the last two problems.

Anyway, the `mv` command does pretty well, but to move a directory within a file system it needs one of the following alternative mechanisms.

- To always copy a directory and its subtree, followed by a deletion of the old directory, even if the directory is just renamed within its parent.
- A `link` system call that will work on directories, even if restricted to the superuser (The `mv` command can run with the set-user-ID bit on, to temporarily take on the privileges of the superuser.)
- A new system call that can rename a directory within a file system

The first alternative is bad—that's way too much work if all you want to do is change a few characters in the name! The second was ruled out by POSIX in favor of the third—a new system call, `rename`. Actually, it isn't new—it was in Standard C and had been in BSD. It's the only way to rename a directory without a full copy/unlink.

rename—rename file

```
#include <stdio.h>

int rename(
    const char *oldpath,      /* old pathname */
    const char *newpath       /* new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`rename` is roughly equivalent to the sequence:

1. If `newpath` exists, remove it with `unlink` or `rmdir`.
2. `link(oldpath, newpath)`, even if `oldpath` is a directory.
3. Remove `oldpath` with `unlink` or `rmdir`.

What `rename` brings to the table are some additional features and rules:

- As I mentioned, step 2 works on directories, even if the process isn't running as the superuser. (You need write permission in `newpath`'s parent, though.)
- If `newpath` exists, it and `oldpath` have to be both files or both directories.
- If `newpath` exists and is a directory, it has to be empty. (Same rule as for `rmdir`.) In step 3, `oldpath`, if a directory, is removed even if nonempty, since its contents are also in `newpath`.
- If `rename` fails somewhere along the way, everything is left unchanged.

So you can see that while steps 1, 2, and 3 look like you could write them as a library function, there's no way to get it to work like the system call.

The `mv` command can do several things that the `rename` system call can't, such as move files and directories between file systems, and move groups of files and directories to a new parent directory. `rename` just supplies a small, but essential, part of `mv`'s functionality.

A final note: If `oldpath` is a symbolic link, `rename` operates on the symbolic link, not on what it points to; therefore, it might have been called `lrename`, but that would confuse the Standard C folks.

3.3.3 Creating Symbolic Links (`symlink` System Call)

As shown in Figure 3.3, if we wanted to create a second link to the file /x (ad0s1g:107) from directory /y, we could execute this call:

```
ec_neg1( link("/x", "/y/c") )
```

This makes /x and /y/c equivalent in every way; ad0s1g:107 isn't "in" either directory.

But now the transparency of the UNIX mounting mechanism breaks down: It's impossible to create a hard link from /y to /y/a/f (ad0s1m:107) because a directory entry has just an i-number to identify the linked-to object, not a device:i-number combination, which is what it takes to be unique. (I took the trouble in the figures to make 107 the i-number of two completely different files.) If you try to execute:

```
ec_neg1( link("/y/a/f", "/y/b") )
```

it will produce an `EXDEV` error.

The solution, as every experienced UNIX user knows, is to use a symbolic link. Unlike a hard link, where the i-number you want to link to goes right in the directory, a symbolic link is a small file containing the text of a path to the object you want to link to. The link we want is shown by the ellipse in Figure 3.3. It has an i-node of its own (ad0s1g:118), but normally any attempt to access it as /y/b causes the kernel to interpret the access as one to /y/a/f, which is on another file system.

A symbolic link can reference another symbolic link. Normally, such as when a symbolic link is passed to `open`, the kernel keeps dereferencing symbolic links until it finds something that isn't a symbolic link. This chaining can't happen with a hard link, as it directly references an i-node, which cannot be a hard link. (Although it can be a directory that *contains* a hard link.) In other words, if a path leads to a hard link, the path is followed literally. If it leads to a symbolic link, the actual path followed depends on what that symbolic link references, until the end of the chain is reached.

Users create a symbolic link with the –s option of the `ln` command, but the system call that does the work is `symlink`:

symlink—create symbolic link

```
#include <unistd.h>

int symlink(
     const char *oldpath,       /* old pathname */
     const char *newpath        /* possible new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Mostly, `symlink` works like `link`. In both cases, a hard link is created whose path is given by `newpath`; however, with `symlink`, that hard link is to a symbolic-link file that contains the string given by `newpath`.

The comment in the synopsis says "possible" because there is no validation of `newpath` at all—not to ensure that it's a valid path, or a valid anything else. You can even do this:

```
ec_neg1( symlink("lunch with Mike at 11:30", "reminder") )
```

and then see your reminder with the `ls` command (output wrapped to fit on the page):

```
$ ls -l reminder
lrwxrwxrwx   1 marc     sysadmin      24
Oct 16 12:04 reminder -> lunch with Mike at 11:30
```

This rather loose behavior is purposeful and leads to a useful feature of a symbolic link: What it references need not exist. Or, it may exist, but the file system that it's on may not be mounted.

The flip-side of the kernel's lack of interest in whether a symbolic-link target exists is that nothing is done to a symbolic link when its target is unlinked. By "unlinked," I mean, of course, "un-hard-linked," as the hard-link count going to zero is still what the kernel uses to determine if a file is no longer needed (i-node and data reclaimable). It could be impossible to do anything to adjust the symbolic links, because some of them could be on unmounted file systems. That's not possible with hard links, as they're internal to a file system.

So how do you get rid of a symbolic link? With `unlink` (refer to Figure 3.3):

```
ec_neg1( unlink("/y/b") )
```

This unlinks the symbolic link /y/b, and has no effect on the file it refers to, /y/a/f. Think of `unlink` as removing the hard link that its argument directly specifies.

OK, so how *do* you unlink a file that you want to refer to via its symbolic link? You could use another path, as the file has to be hard linked to *some* directory. But if all you have is the symbolic link path, you can read its contents with the `readlink` system call:

readlink—read symbolic link

```
#include <unistd.h>

ssize_t readlink(
    const char *path,          /* pathname */
    char *buf,                 /* returned text */
    size_t bufsize             /* buffer size */
);
/* Returns byte count or -1 on error (sets errno) */
```

`readlink` is unusual for a UNIX system call in that you can't assume that the returned string is NUL-terminated. You need to ensure that `buf` points to enough space to hold the contents of the symbolic link plus a NUL byte, and then pass its size less one as the third argument. Upon return, you'll probably want to force in a NUL byte, like this:

```
ssize_t n;
char buf[1024];

ec_neg1( n = readlink("/home/marc/mylink", buf, sizeof(buf) - 1) )
buf[n] = '\0';
```

If you want, you can now remove whatever /home/marc/mylink linked to and the symbolic link itself like this:

```
ec_neg1( unlink(buf) )
ec_neg1( unlink("/home/marc/mylink") )
```

`readlink` isn't the only case where we want to deal with a symbolic link itself, rather than what it references. Another is the `stat` system call (Section 3.5.1), for getting i-node information, which has a variant that doesn't follow symbolic links called `lstat`.

A problem with the `readlink` example code is the constant 1024, meant to represent a size large enough for the largest path name. Our code isn't going to blow up if the path is longer—we were too careful for that. It will simply truncate the returned path, which is still not good.

But if a file name in a directory is limited to, say, 255 bytes, and there's no limit on how deeply nested directories can get, how much space is enough? Certainly 1024 isn't the answer—that's only enough to handle four levels! Getting the answer is somewhat messy, so it gets a section all to itself. Read on.

3.4 Pathnames

This section explains how to determine the maximum length for a pathname and how to retrieve the pathname of the current directory.

3.4.1 How Long Is a Pathname?

If there were a fixed limit on the length of a pathname—a constant `_POSIX_MAX_PATH`, say—there would also be a limit on how deeply directories could nest. Even if the number were large, it would create problems, as it's pretty common for a UNIX system to effectively mount file systems on other computers via facilities like NFS. Thus, the limit has to be dynamically determined at runtime, and it has to be able to vary by file system.

That's exactly what `pathconf` and `fpathconf` are for (Section 1.5.6). On a system that conforms to POSIX1990—essentially all of them—you call it like this:

```
static long get_max_pathname(const char *path)
{
    long max_path;

    errno = 0;
    max_path = pathconf(path, _PC_PATH_MAX);
    if (max_path == -1) {
        if (errno == 0)
            max_path = 4096; /* guess */
        else
            EC_FAIL
    }
    return max_path + 1;
```

```
EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

I added one to the value returned by `pathconf` because the documentation I've looked at is a little fuzzy as to whether the size includes space for a NUL byte. I'm inclined to assume it does not, figuring that the OS implementors are as unsure as I am.

I called this function for a locally mounted file system and for an NFS-mounted file system on each of my three test systems, and I got 1024 on FreeBSD and Solaris, and 4096 on Linux. But that was with my versions of those systems, and with my configuration. You need to use `pathconf` in your own code, rather than relying on my numbers.

Technically, for `_PC_PATH_MAX`, `pathconf` returns the size of the longest relative pathname from its argument path, not that of the longest absolute path; therefore, for complete accuracy you should call it with path of the root of the file system that you're interested in, and then add to that number the size of the path from the root to the root of that file system. But that's probably going too far—most everyone just calls it with an argument of `"/"` or `"."`.

The preceding was according to the POSIX and SUS standards. In practice, even systems that return some number from `pathconf` and `fpathconf` don't enforce that as a limit when creating directories, but they do enforce it when you try to pass a too-long path name to a system call that takes a path, such as `open` (the error is `ENAMETOOLONG`). On most systems, `getcwd` (next section) can return a string longer than the maximum returned by `pathconf` or `fpathconf`, if you give it a buffer that's big enough.

3.4.2 `getcwd` System Call

There's a straightforward system call for getting the path to the current directory. Like `readlink` (Section 3.3.3), the only tricky thing about it is knowing how big a buffer to pass in.

getcwd—get current directory pathname

```
#include <unistd.h>

char *getcwd(
    char *buf,              /* returned pathname */
    size_t bufsize          /* size of buf */
);
/* Returns pathname or NULL on error (sets errno) */
```

Here's a function that makes it even easier to call `getcwd`, as it automatically allocates a buffer to hold the path string. A call with a `true` argument tells it to free the buffer.

```
static char *get_cwd(bool cleanup)
{
    static char *cwd = NULL;
    static long max_path;

    if (cleanup) {
        free(cwd);
        cwd = NULL;
    }
    else {
        if (cwd == NULL) {
            ec_neg1( max_path = get_max_pathname(".") )
            ec_null( cwd = malloc((size_t)max_path) )
        }
        ec_null( getcwd(cwd, max_path) )
        return cwd;
    }
    return NULL;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

Here's some code that does what the standard `pwd` command does:

```
char *cwd;

ec_null( cwd = get_cwd(false) )
printf("%s\n", cwd);
(void) get_cwd(true);
```

When I ran it I got this on Solaris:

```
/home/marc/aup
```

There's enough functionality in this chapter to program `getcwd` ourselves, and I'll show the code in Section 3.6.4.

There's a similar system call, `getwd`, but it's obsolete and shouldn't be used in new programs.

3.5 Accessing and Displaying File Metadata

This section explains how to retrieve file metadata, such as the owner or modification time, and how to display it.

3.5.1 `stat`, `fstat`, and `lstat` System Calls

An i-node contains a a file's *metadata*—all the information about it other than its name, which really doesn't belong to it anyway, and its data, which the i-node points to. Reading an i-node straight from the disk, as we did in Section 3.2.2, is really primitive. Fortunately, there are three standardized system calls for getting at an i-node, `stat`, `fstat`, and `lstat`, and one very well-known command, `ls`.

stat—get file info by path

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

int stat(
     const char *path,      /* pathname */
     struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

lstat—get file info by path without following symbolic link

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

int lstat(
     const char *path,      /* pathname */
     struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fstat—get file info by file descriptor

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

int fstat(
     int fd,                /* file descriptor */
     struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`stat` takes a path and finds the i-node by following it; `fstat` takes an open file descriptor and finds the i-node from the active i-node table inside the kernel. `lstat` is identical to `stat`, except that if the path leads to a symbolic link, the metadata is for the symbolic link itself, not for what it links to.[5] For all three, the same metadata from the i-node is rearranged and placed into the supplied `stat` structure.

Here's the `stat` structure, but be aware that implementations are free to add or rearrange fields, so make sure you include your local system's sys/stat.h header.

struct stat—structure for stat, fstat, and lstat

```
struct stat {
    dev_t st_dev;              /* device ID of file system */
    ino_t st_ino;              /* i-number */
    mode_t st_mode;            /* mode (see below) */
    nlink_t st_nlink;          /* number of hard links */
    uid_t st_uid;              /* user ID */
    gid_t st_gid;              /* group ID */
    dev_t st_rdev;             /* device ID (if special file) */
    off_t st_size;             /* size in bytes */
    time_t st_atime;           /* last access */
    time_t st_mtime;           /* last data modification */
    time_t st_ctime;           /* last i-node modification */
    blksize_t st_blksize;      /* optimal I/O size */
    blkcnt_t st_blocks;        /* allocated 512-byte blocks */
};
```

Some comments about this structure:

- A device ID (type `dev_t`) is a number that uniquely identifies a mounted file system, even when it's mounted with NFS. So, the combination `st_dev` and `st_ino` uniquely identifies the i-node. This is essentially what we were doing with notations like "ad0s1g:221" back in Section 3.2.4. Look again at Figure 3.3, where I've put the device IDs in hex below each device name.

  The standard doesn't specify how to break down a device ID, but essentially all implementations treat it as combination of major and minor device numbers, where the major number identifies the driver, and the minor number identifies the actual device, as the same driver can interface with all devices of the same type. Typically the minor number is the rightmost byte.

5. There's no `flstat` because there's no way to get a file descriptor open to a symbolic link. Any attempt to use a symbolic link in an `open` will open the linked-to file, not the symbolic link. And, if there were such a way, `fstat` would suffice, as the file descriptor would already specify the right object.

- The field st_dev is the device that contains the i-node; the field st_rdev, used only for special files, is the device that the special file represents. As an example, the special file /dev/ad0s1m is on the root file system, so its st_dev is that of the root file system. But st_rdev is the device that it refers to, which is a different disk entirely.
- The field st_size has different interpretations, depending on the type of i-node and the implementation. For an i-node that represents data on disk—regular files, directories, and symbolic links—it's the size of that data (the path, for symbolic links). For shared memory objects, it's the memory size. For pipes, it's the amount of data in the pipe, but that's nonstandard.
- The access time (st_atime) is updated whenever a file of any type is read, but not when a directory that appears in a path is only searched.
- The data modification time (st_mtime) is updated when a file is written, including when a hard link is added to or removed from a directory.
- The i-node modification time (st_ctime), also called the status-change time, is updated when the file's data is written or when the i-node is explicitly modified (e.g., by changing the owner or link count), but not when the access time is changed only as a side-effect of reading the file.
- The field st_blksize is in the stat structure so that an implementation can vary it by file, if it chooses to do so. In most cases it's probably the same as what's in the superblock (Section 3.2.3).
- If a file has holes, formed by seeking past its end and writing something, the value of st_blocks * 512 could be less than st_size.
- fstat is especially useful when you have a file descriptor that did not come from opening a path, such as one for an un-named pipe or a socket. For these, st_mode, st_ino, st_dev, st_uid, st_gid, st_atime, st_ctime, and st_mtime are required to have valid values, but whether the other fields do is implementation dependent. Usually, though, for named and un-named pipes the st_size field contains the number of unread bytes in the pipe.

The field st_mode consists of bits that indicate the type of file (regular, directory, etc.), the permissions, and a few other characteristics. Rather than assuming specific bits, a portable application is supposed to use macros. First come the macros for the type of file.

st_mode—bit mask and values for type of file

```
S_IFMT                  /* all type-of-file bits */
S_IFBLK                 /* block special file */
S_IFCHR                 /* character special file */
S_IFDIR                 /* directory */
S_IFIFO                 /* named or un-named pipe */[6]
S_IFLNK                 /* symbolic link */
S_IFREG                 /* regular file */
S_IFSOCK                /* socket */
```

The macro `S_IFMT` defines the bits for the type of file; the others are values of those bits, *not* bit masks. Thus, the test for, say, a socket must not be coded as

```
if ((buf.st_mode & S_IFSOCK) == S_IFSOCK) /* wrong */
```

but as

```
if ((buf.st_mode & S_IFMT) == S_IFSOCK)
```

Or, you can use one of these testing macros each of which returns zero for false and nonzero for true, so they work as C Boolean expressions:

st_mode—type-of-file testing macros

```
S_ISBLK(mode)           /* is a block special file */
S_ISCHR(mode)           /* is a character special file */
S_ISDIR(mode)           /* is a directory */
S_ISFIFO(mode)          /* is a named or un-named pipe */
S_ISLNK(mode)           /* is a symbolic link */
S_ISREG(mode)           /* is a regular file */
S_ISSOCK(mode)          /* is a socket */
```

The test for a socket could be written as:

```
if (S_ISSOCK(buf.st_mode))
```

There are nine bits someplace in the mode for the permissions, and I already introduced the macros for them in Section 2.3, as the same macros are used with the `open` system call and a few others. The macros are of the form `S_I`**pwww** where **p** is the permission (`R`, `W`, or `X`) and **www** is for whom (`USR`, `GRP`, or `OTH`).

This time you do use them as bit masks, like this, which tests for group read *and* write permission:

```
if ((buf.st_mode & (S_IRGRP | S_IWGRP)) == (S_IRGRP | S_IWGRP))
```

In an attempt to clarify things (or perhaps make them worse—sorry!), here's a test for group read *or* write permission:

6. Actually, `S_IFIFO`, while correct, is misspelled; it should have been called `S_IFFIFO`.

```
if ((buf.st_mode & S_IRGRP) == S_IRGRP ||
  (buf.st_mode & S_IWGRP) == S_IWGRP)
```

We could have coded the read *and* write case like this if we wanted to:

```
if ((buf.st_mode & S_IRGRP) == S_IRGRP &&
  (buf.st_mode & S_IWGRP) == S_IWGRP)
```

There are a few other bits in the `st_mode` field, and I'll repeat the permission bits in the box to make them easier to find if you flip back to this page later:

st_mode—permission and miscellaneous bit masks

```
S_Ixwww     /* x = R|W|X, www = USR|GRP|OTH */
            /* examples: S_IRUSR, S_IWOTH */
S_ISUID     /* set-user-ID on execution */
S_ISGID     /* set-group-ID on execution */
S_ISVTX     /* directory restricted-deletion */
```

I explained set-user-ID and set-group-ID in Section 1.1.5. The `S_ISVTX` flag, if set, means that a file may be unlinked from a directory only by the superuser, the owner of that directory, or the owner of the file. If the flag is not set (the normal case), having write permission in the directory is good enough.[7]

A good way to show how to use the mode macros is to display the modes as the `ls` command does, with a sequence of 10 letters (e.g., `drwxr-xr-x`). Briefly, here are the rules that `ls` uses:

- The first letter indicates the type of file.
- The next nine letters are in three groups of three, for owner, group, and others, and are normally `r`, `w`, or `x` if the permission is set, and a dash if not.
- The owner and group execute letters become an `s` (lower case) if the set-user-ID or set-group-ID bit is on, in addition to the execute bit, and an `S` (upper case) if the set bit is on but the execute position is not. The second combination is possible, but meaningless, except for the combination of set-group-ID on and group execute off, which on some systems causes mandatory file locking, as explained in Section 7.11.5.
- The execute letter for others becomes a `t` if the restricted-deletion bit (`I_SVTX`) is set along with the execute (search) bit, and a `T` if the restricted-deletion bit is set but the execute bit is not.

7. That's the SUS definition. Historically, this bit was called the sticky bit, and it was used on executable-program files to keep the instruction segment of an often-used program (e.g., the shell) on the swap device. It's not usually used with modern, paging UNIX systems. (The letters "SVTX" come from "SaVe TeXt," as the instructions are also called the "text.")

The following function should help make the algorithm clear:

```
#define TYPE(b) ((statp->st_mode & (S_IFMT)) == (b))
#define MODE(b) ((statp->st_mode & (b)) == (b))

static void print_mode(const struct stat *statp)
{
    if (TYPE(S_IFBLK))
        putchar('b');
    else if (TYPE(S_IFCHR))
        putchar('c');
    else if (TYPE(S_IFDIR))
        putchar('d');
    else if (TYPE(S_IFIFO)) /* sic */
        putchar('p');
    else if (TYPE(S_IFREG))
        putchar('-');
    else if (TYPE(S_IFLNK))
        putchar('l');
    else if (TYPE(S_IFSOCK))
        putchar('s');
    else
        putchar('?');
    putchar(MODE(S_IRUSR) ? 'r' : '-');
    putchar(MODE(S_IWUSR) ? 'w' : '-');
    if (MODE(S_ISUID)) {
        if (MODE(S_IXUSR))
            putchar('s');
        else
            putchar('S');
    }
    else if (MODE(S_IXUSR))
        putchar('x');
    else
        putchar('-');
    putchar(MODE(S_IRGRP) ? 'r' : '-');
    putchar(MODE(S_IWGRP) ? 'w' : '-');
    if (MODE(S_ISGID)) {
        if (MODE(S_IXGRP))
            putchar('s');
        else
            putchar('S');
    }
    else if (MODE(S_IXGRP))
        putchar('x');
    else
        putchar('-');
    putchar(MODE(S_IROTH) ? 'r' : '-');
    putchar(MODE(S_IWOTH) ? 'w' : '-');
    if (MODE(S_IFDIR) && MODE(S_ISVTX)) {
```

```
        if (MODE(S_IXOTH))
            putchar('t');
        else
            putchar('T');
    }
    else if (MODE(S_IXOTH))
        putchar('x');
    else
        putchar('-');
}
```

This function doesn't terminate its output with a newline, for a reason that will be very clear soon, but I don't want to spoil the surprise. Here's some test code:

```
struct stat statbuf;

ec_neg1( lstat("somefile", &statbuf) )
print_mode(&statbuf);
putchar('\n');
ec_neg1( system("ls -l somefile") )
```

and the output, in case you didn't believe anything I was saying:

```
prw--w--w-
prw--w--w-   1 marc     sysadmin       0 Oct 17 13:57 somefile
```

Note that we called `lstat`, not `stat`, because we want information about the symbolic link itself if the argument is a symbolic link. We don't want to follow the link.

Next, a function to print the number of links, which is way easier than printing the mode. (Do you see where we're headed?)

```
static void print_numlinks(const struct stat *statp)
{
    printf("%5ld", (long)statp->st_nlink);
}
```

Why the cast to `long`? Well, we really don't know what the type `nlink_t` is, other than that it's an integer type. We need to cast it something concrete to ensure that it matches the format in the `printf`. The same goes for some other types in the `stat` structure, as we'll see later in this chapter.

Next we want to print the owner and group names, but for that we need two more library functions.

3.5.2 `getpwuid`, `getgrgid`, and `getlogin` System Calls

The owner and group numbers are easy to get at, as they're in the `stat` structure's `st_uid` and `st_gid` fields, but we want their names. For that there are two functions, `getpwuid` and `getgrgid`, that aren't really system calls, since the information they need is in the password and group files, which any process can read for itself if it wants to take the trouble. A problem with that, though, is that the file layout isn't standardized.

Here's what the password-file entry for "marc" looks like on Solaris, which is fairly typical:

```
$ grep marc /etc/passwd
marc:x:100:14::/home/marc:/bin/sh
```

(The one thing that is *not* in the password file is the password! It's stored encrypted in another file that only the superuser can read.)

On this system "marc" is a member of a few groups, but group 14 is his login group:

```
$ grep 14 /etc/group
sysadmin::14:
```

getpwuid—get password-file entry

```
#include <pwd.h>

struct passwd *getpwuid(
    uid_t uid          /* user ID */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

struct passwd—structure for `getpwuid`

```
struct passwd {
    char *pw_name;     /* login name */
    uid_t pw_uid;      /* user ID */
    gid_t pw_gid;      /* group ID */
    char *pw_dir;      /* login directory */
    char *pw_shell;    /* login shell */
};
```

getgrgid—get group-file entry

```
#include <grp.h>

struct group *getgrgid(
    gid_t gid          /* group ID */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

struct group—structure for `getgrgid`

```
struct group {
    char *gr_name;      /* group name */
    gid_t gr_gid;       /* group ID */
    char **gr_mem;      /* member-name array (NULL terminated */
};
```

As I've said before, the two structures here show just the standardized fields. Your implementation may have more, so use the structures as defined in the headers.

Now we can use the two look-up functions to print the login and group names, or just the numbers if the names aren't known. Not finding the name is a common occurrence when a file system is mounted across a network, as a user on one system may not have a login on another.

```
static void print_owner(const struct stat *statp)
{
    struct passwd *pwd = getpwuid(statp->st_uid);

    if (pwd == NULL)
        printf(" %-8ld", (long)statp->st_uid);
    else
        printf(" %-8s", pwd->pw_name);
}

static void print_group(const struct stat *statp)
{
    struct group *grp = getgrgid(statp->st_gid);

    if (grp == NULL)
        printf(" %-8ld", (long)statp->st_gid);
    else
        printf(" %-8s", grp->gr_name);
}
```

While we're at it, here's a function to get the name under which the user logged in:

getlogin—get login name

```
#include <unistd.h>

char *getlogin(void);
/* Returns name or NULL on error (sets errno) */
```

3.5.3 More on Displaying File Metadata

Continuing with the `stat` structure, here's a function to print the file size. In the case of special files, we print the major and minor device numbers instead, as the size isn't meaningful. As I said in Section 3.5, how they're encoded in the device ID isn't standardized, but taking the rightmost 8 bits as the minor number usually works:

```
static void print_size(const struct stat *statp)
{
    switch (statp->st_mode & S_IFMT) {
    case S_IFCHR:
    case S_IFBLK:
        printf("%4u,%4u", (unsigned)(statp->st_rdev >> 8),
          (unsigned)(statp->st_rdev & 0xFF));
        break;
    default:
        printf("%9lu", (unsigned long)statp->st_size);
    }
}
```

Next comes the file data-modification date and time, using the Standard C function `strftime` to do the hard work. The functions `time` and `difftime` are in Standard C, too. (See Section 1.7.1 for all three.) We don't normally print the year, unless the time is 6 months away, in which case we skip the time to make room:[8]

```
static void print_date(const struct stat *statp)
{
    time_t now;
    double diff;
    char buf[100], *fmt;

    if (time(&now) == -1) {
        printf(" ????????????");
        return;
    }
    diff = difftime(now, statp->st_mtime);
    if (diff < 0 || diff > 60 * 60 * 24 * 182.5) /* roughly 6 months */
        fmt = "%b %e  %Y";
    else
        fmt = "%b %e %H:%M";
    strftime(buf, sizeof(buf), fmt, localtime(&statp->st_mtime));
    printf(" %s", buf);
}
```

The last thing we want to print is the file name, which isn't in the `stat` structure, of course, because it isn't in the i-node. It's a little tricky, however, because if the name is a symbolic link, we want to print both it and its contents:

---

8. Generally, our policy is to check for all errors except when formatting (for printing) and printing. `difftime` isn't an exception—it's one of those functions that has no error return.

```
static void print_name(const struct stat *statp, const char *name)
{
    if (S_ISLNK(statp->st_mode)) {
        char *contents = malloc(statp->st_size + 1);
        ssize_t n;

        if (contents != NULL && (n = readlink(name, contents,
          statp->st_size)) != -1) {
            contents[n] = '\0'; /* can't assume NUL-terminated */
            printf(" %s -> %s", name, contents);
        }
        else
            printf(" %s -> [can't read link]", name);
        free(contents);
    }
    else
        printf(" %s", name);
}
```

Recall that back in Section 3.3.3 when I introduced `readlink` I pointed out that we needed to know the length of the longest pathname in order to size the buffer. This function shows a more straightforward way, since the exact size of the path (not including the NUL byte) is in the `st_size` field for symbolic links.[9] Note that I allocate one byte more than `st_size`, but pass `st_size` to `readlink` so as to guarantee room for a NUL byte, in case `readlink` doesn't supply one, which it isn't required to do.

Now, for what you've been waiting for, a program that puts all the printing functions together:

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

    for (i = 1; i < argc; i++) {
        ec_neg1( lstat(argv[i], &statbuf) )
        ls_long(&statbuf, argv[i]);
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

9. Most modern file systems store short symbolic links right in the i-node, rather than taking up data blocks, but still the `st_size` field gives the size.

```
static void ls_long(const struct stat *statp, const char *name)
{
    print_mode(statp);
    print_numlinks(statp);
    print_owner(statp);
    print_group(statp);
    print_size(statp);
    print_date(statp);
    print_name(statp, name);
    putchar('\n');
}
```

Here's our simplified `ls` command in action:

```
$ aupls /dev/tty
crw-rw-rw-    1 root     root       5,   0 Mar 23  2002 /dev/tty
$ aupls a.tmp a.out util
lrwxrwxrwx    1 marc     sysadmin        5 Jul 29 13:30 a.tmp -> b.tmp
-rwxr-xr-x    1 marc     sysadmin     8392 Aug  1  2001 a.out
drwxr-xr-x    3 marc     sysadmin      512 Aug 28 12:26 util
```

Notice from the last line that it doesn't know how to list a directory—it only lists the names given as arguments. To add that feature we need to find out how to read a directory to get at its links. That's coming right up.

3.6 Directories

This section covers reading and removing directories, changing the current directory, and walking up and down the directory tree.

3.6.1 Reading Directories

Underneath, UNIX systems nearly always implement directories as regular files, except that they have a special bit set in their i-node and the kernel does not permit writing on them. On some systems you're allowed to read a directory with `read`, but the POSIX and SUS standards don't require this, and they also don't specify the internal format of the directory information. But it's interesting to snoop, so I wrote a little program to read the first 96 bytes of the current directory and dump out the bytes as characters (if they're printable) and in hex:

```
static void dir_read_test(void)
{
```

```
    int fd;
    unsigned char buf[96];
    ssize_t nread;

    ec_neg1( fd = open(".", O_RDONLY) )
    ec_neg1( nread = read(fd, buf, sizeof(buf)) )
    dump(buf, nread);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("dir_read_test");
EC_CLEANUP_END
}

static void dump(const unsigned char *buf, ssize_t n)
{
    int i, j;

    for (i = 0; i < n; i += 16) {
        printf("%4d  ", i);
        for (j = i; j < n && j < i + 16; j++)
            printf("  %c", isprint((int)buf[j]) ? buf[j] : ' ');
        printf("\n      ");
        for (j = i; j < n && j < i + 16; j++)
            printf(" %.2x", buf[j]);
        printf("\n\n");
    }
    printf("\n");
}
```

This was the output (minus the "ec" tracing stuff) on Linux:

```
*** EISDIR (21: "Is a directory") ***
```

So, no luck there. But, on FreeBSD (and Solaris), pay dirt:

```
   0    `                          .           V
       60 d8 05 00 0c 00 04 01 2e 00 00 00 56 d8 05 00

  16                   .  .        b
       0c 00 04 02 2e 2e 00 00 62 d8 05 00 0c 00 08 02

  32    m  2        y                          k     C
       6d 32 00 c0 79 d8 05 00 0c 00 08 01 6b 00 43 c6

  48                               p  c  s  y  n  c  _  s
       b3 da 05 00 18 00 08 0c 70 63 73 79 6e 63 5f 73

  64    i  g  .  o
       69 67 2e 6f 00 00 00 00 e3 e3 05 00 10 00 08 06

  80    t  i  m  e  .  o        r
       74 69 6d 65 2e 6f 00 c6 72 d8 05 00 0c 00 08 02
```

This appears at first to be a mishmash, but looking closer it starts to makes some sense. Six names are visible: `.`, `..`, `m2`, `k`, `pcsync_sig.o`, and `time.o`. We know that their i-numbers have to be in there, too, and to find them in hex we use the `-i` option of the `ls` command and the `dc` ("desk calculator") to translate from decimal to hex (comments in italics were added later):[10]

```
$ ls -ldif . .. m2 k
383072 drwxr-xr-x  2 marc  marc  2560 Oct 18 11:02 .
383062 drwxr-xr-x  9 marc  marc   512 Oct 14 18:05 ..
383074 -rwxrwxrwx  1 marc  marc    55 Jul 25 11:14 m2
383097 -rwxr--r--  1 marc  marc   138 Sep 19 13:28 k
$$$ dc
16o          make 16 the output radix
383072p      push the i-number onto the stack and print it
5D860        dc printed i-number in hex (radix 16)
383062p      … ditto …
5D856
q            quit
```

And, sure enough, the numbers 5D860 and 5D856 show up on the second line (first hex line) of the dump. There are some other numbers for each entry there as well, for such things as string sizes, but we really don't want to dig any deeper. There's a better, standardized way to read a directory, using system calls designed just for that purpose:

opendir—open directory

```
#include <dirent.h>

DIR *opendir(
    const char *path       /* directory pathname */
);
/* Returns DIR pointer or NULL on error (sets errno) */
```

closedir—close directory

```
#include <dirent.h>

int closedir(
    DIR *dirp              /* DIR pointer from opendir */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

10. The `-ldif` options to `ls` mean long listing, don't traverse directories, show i-numbers, and don't sort.

readdir—read directory

```
#include <dirent.h>

struct dirent *readdir(
     DIR *dirp              /* DIR pointer from opendir */
);
/* Returns structure or NULL on EOF or error (sets errno) */
```

struct dirent—structure for readdir

```
struct dirent {
     ino_t d_ino;           /* i-number */
     char d_name[];         /* name */
};
```

rewinddir—rewind directory

```
#include <dirent.h>

void rewinddir(
     DIR *dirp              /* DIR pointer from opendir */
);
```

seekdir—seek directory

```
#include <dirent.h>

void seekdir(
     DIR *dirp,             /* DIR pointer from opendir */
     long loc               /* location */
);
```

telldir—get directory location

```
#include <dirent.h>

long telldir(
     DIR *dirp              /* DIR pointer from opendir */
);
/* Returns location (no error return) */
```

These functions work as you would expect: You start with `opendir`, which gives you a pointer to a `DIR` (analogous to calling the Standard C function `fopen` to get a pointer to a `FILE`). That pointer is then used as an argument to the other five functions. You call `readdir` in a loop until it returns `NULL`, which means that you got to the end if `errno` wasn't changed by `readdir`, and an error if it was. (So things don't get confused, you should set `errno` to zero before you call `readdir`.) `readdir` returns a pointer to a `dirent` structure which contains the i-number and name for one entry. When you're done with the `DIR`, you close it with `closedir`.

The i-number returned in the `d_ino` field by `readdir` isn't particularly useful because, if that entry is a mount point, it will be the premount i-number, not the one that reflects the current tree. For example, in Figure 3.3 (back in Section 3.3), `readdir` would give us 221 for directory entry /y/a, but, as it is a mount point, the effective i-node is ad0s1m:2 (plain 2 isn't specific enough, as device ad0s1g also has a 2). I-node 221 isn't even accessible. Therefore, when reading directories to navigate the directory tree, as I will do when I show later how to get the path of the current directory in Section 3.6.4, you have to get the correct i-number for a directory entry with a call to one of the `stat` functions, not from the `d_ino` field.

`rewinddir` is occasionally useful to go back to the beginning, so you can read a directory again without having to close and reopen it. `seekdir` and `telldir` are more rarely used. The `loc` argument to `seekdir` must be something you got from `telldir`; you can't assume it's an entry number, and you can't assume that directory entries take up fixed-width positions, as we've already seen.

`readdir` is one of those functions that uses a single statically allocated structure that the returned pointer points to. It's very convenient but not so good for multi-threaded programs, so there's a stateless variant of `readdir` that uses memory you pass in instead:

readdir_r—read directory

```
#include <dirent.h>

int readdir_r(
    DIR *restrict dirp,        /* DIR pointer from opendir */
    struct dirent *entry,      /* structure to hold entry */
    struct dirent **result     /* result (pointer or NULL) */
);
/* Returns 0 on success or error number on error (errno not set) */
```

You need to pass a pointer to a `dirent` structure into `readdir_r` that's large enough to hold a name of at least `NAME_MAX + 1` elements. You get the value for `NAME_MAX` with a call to `pathconf` or `fpathconf`, with the directory as an argument, similar to what we had to do in Section 3.4 to get the value for the maximum path length. `readdir_r` returns its result through the `result` argument; its interpretation is identical to the return value from `readir`, but this time you don't check `errno`—the error number (or zero) is the return value from the function. You can use the `ec_rv` macro (Section 1.4.2) to check it, as I do in this example, which lists the names and i-numbers in the current directory.

```
static void readdir_r_test(void)
{
    bool ok = false;
    long name_max;
    DIR *dir = NULL;
    struct dirent *entry = NULL, *result;

    errno = 0;
    /* failure with errno == 0 means value not found */
    ec_neg1( name_max = pathconf(".", _PC_NAME_MAX) )
    ec_null( entry = malloc(offsetof(struct dirent, d_name) +
      name_max + 1) )
    ec_null( dir = opendir(".") )
    while (true) {
        ec_rv( readdir_r(dir, entry, &result) )
        if (result == NULL)
            break;
        printf("name: %s; i-number: %ld\n", result->d_name,
          (long)result->d_ino);
    }
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    free(entry);
    if (!ok)
        EC_FLUSH("readdir_r_test");
EC_CLEANUP_END
}
```

Some comments on this code:

- A –1 return from `pathconf` with `errno` zero means that there is no limit for the names in a directory, which can't be. But we want our code to handle this case, so we take the shortcut of setting `errno` to zero and then treating all –1 returns as errors. The comment is for anyone who actually gets an error message with an `errno` value of zero. The idea is that we want to ensure that our code handles the impossible cases without dealing with them in an overly complicated way, which we wouldn't be able to test anyway, as we can't create test cases for impossible occurrences.
- The allocation with `malloc` is tricky. All we know for sure about the `dirent` structure is that the field `d_ino` is in it someplace (there may be other, nonstandard fields), and that `d_name` is last. So, the offset of `d_name` plus the size we need is the only safe way to calculate the total size, taking

into account both holes in the structure (allowed by Standard C) and hidden fields.

- `EC_CLEANUP` jumps to the cleanup code, as explained in Section 1.4.2. The Boolean `ok` tells us whether there was an error.

What a pain! It's much easier to use `readdir`, which is fine if you're not multithreading or you can ensure that only one thread at a time is reading a directory:

```
static void readdir_test(void)
{
    DIR *dir = NULL;
    struct dirent *entry;

    ec_null( dir = opendir(".") )
    while (errno = 0, (entry = readdir(dir)) != NULL)
        printf("name: %s; i-number: %ld\n", entry->d_name,
          (long)entry->d_ino);
    ec_nzero( errno )
    EC_CLEANUP

EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    EC_FLUSH("readdir_test");
EC_CLEANUP_END
}
```

The one tricky thing here is stuffing the call to `readdir` into the `while` test: The first part of the comma expression zeroes `errno`, and the second part determines the value of the expression as a whole, which is what the `while` tests. You may want a less dense version, which is fine, but don't do this:

```
errno = 0;
while ((entry = readdir(dir)) != NULL) { /* wrong */
    /* process entry */
}
```

because `errno` might be reset during the processing of the entry. You should set `errno` to zero each time you call `readdir`.[11]

Anyway, here are the first few lines of output (from either example):

11. I know what you're thinking: That half the difficulty of programming UNIX is dealing with `errno`. You're right! But don't shoot me, I'm only the messenger! See Appendix B for an easier way.

```
name: .; i-number: 383072
name: ..; i-number: 383062
name: m2; i-number: 383074
name: k; i-number: 383097
```

Now that we can read a directory, let's put a `readdir` loop together with the `ls_long` function that appeared at the end of Section 3.5.3 to get an `ls` command that can list a directory:

```
int main(int argc, char *argv[]) /* has a bug */
{
    bool ok = false;
    int i;
    DIR *dir = NULL;
    struct dirent *entry;
    struct stat statbuf;

    for (i = 1; i < argc; i++) {
        ec_neg1( lstat(argv[i], &statbuf) )
        if (!S_ISDIR(statbuf.st_mode)) {
            ls_long(&statbuf, argv[i]);
            ok = true;
            EC_CLEANUP
        }
        ec_null( dir = opendir(argv[i]) )
        while (errno = 0, ((entry = readdir(dir)) != NULL)) {
            ec_neg1( lstat(entry->d_name, &statbuf) )
            ls_long(&statbuf, entry->d_name);
        }
        ec_nzero( errno )
    }
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

This program ran just fine when I listed the current directory, as this output shows (only the first few lines are shown):

```
$ aupls .
drwxr-xr-x    2 marc     marc         2560 Oct 18 12:20 .
drwxr-xr-x    9 marc     marc          512 Oct 14 18:05 ..
-rwxrwxrwx    1 marc     marc           55 Jul 25 11:14 m2
-rwxr--r--    1 marc     marc          138 Sep 19 13:28 k
```

But when I tried it on the /tmp directory, I got this:

```
$ aupls /tmp
drwxr-xr-x    2 marc     marc         2560 Oct 18 12:20 .
drwxr-xr-x    9 marc     marc          512 Oct 14 18:05 ..
ERROR:  0: main [/aup/c3/aupls.c:422] lstat(entry->d_name, &statbuf)
                *** ENOENT (2: "No such file or directory") ***
```

The symptom is that the call to `lstat` in the `readdir` loop fails to find the name, even though we just read it from the directory. The cause is that we're trying to call `lstat` with the path "auplog.tmp," which would be fine if /tmp were the current directory, but it isn't.[12] Do a `cd` first and it works (only first few lines shown):

```
$ cd /tmp
$ /usr/home/marc/aup/aupls .
drwxrwxrwt    3 root     wheel        1536 Oct 18 12:20 .
drwxr-xr-x   18 root     wheel         512 Jul 25 08:01 ..
-rw-r--r--    1 marc     wheel      189741 Aug 30 11:22 auplog.tmp
-rw-------    1 marc     wheel           0 Aug  5 13:23 ed.7GHAhk
```

The fix is to put a call to the `chdir` system before the `readdir` loop, which is a great excuse to go on to the next section.

3.6.2 `chdir` and `fchdir` System Calls

Everybody knows what a shell's `cd` command does, and here's the system call that's behind it:

chdir—change current directory by path

```
#include <unistd.h>

int chdir(
    const char *path          /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchdir—change current directory by file descriptor

```
#include <unistd.h>

int fchdir(
    int fd                    /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

12. The first two entries worked only because `.` and `..` are in every directory. The ones that printed weren't the right ones, however.

As you would expect, the argument to `chdir` can be a relative or absolute path, and whatever i-node it leads to, if a directory, becomes the new current directory.

`fchdir` takes a file descriptor open to a directory as its argument. But wait! Didn't I say in Section 3.6.1 that opening a directory was nonstandard? No, I did not; I said that *reading* one was nonstandard. There is exactly one standard, useful reason to open a directory, and that's to get a file descriptor for use with `fchdir`. You have to open it with `O_RDONLY`, though; you're not allowed to open it for writing.

Since you can't read a directory portably, why have `fchdir`—why not always use `chdir`, with a pathname? Because an `open` paired with an `fchdir` is an excellent way to bookmark your place and return to it. Compare these two techniques:

1. get pathname of current directory	1. open current directory
2. something that changes current directory	2. something that changes current directory
3. chdir using pathname from step 1	3. fchdir using file descriptor from step 1

The technique on the left is inferior because:

- It's a lot of trouble to get a path to the current directory, as we saw in Section 3.4.2.
- It's very time consuming, as we'll see when we do it ourselves, in Section 3.6.4.

In some cases, if you go down just one level, you can get back to where you were with:

```
ec_neg1( chdir("..") )
```

which doesn't require you to have a pathname, but it's still better to use the `open`/`fchdir` approach because it doesn't depend on knowledge of how your program traverses directories. You can't use it, however, if you don't have read permission on the directory.

So now let's fix the version of `aupls` from the previous section so it changes to a directory before reading it. We do need to return to where we were because there may be multiple arguments to process, and each assumes that the current directory is unchanged. Here's just the repaired middle part:

```
ec_null( dir = opendir(argv[i]) )
ec_neg1( fd = open(".", O_RDONLY) )
ec_neg1( chdir(argv[i]) )
while (errno = 0, ((entry = readdir(dir)) != NULL)) {
    ec_neg1( lstat(entry->d_name, &statbuf) )
    ls_long(&statbuf, entry->d_name);
}
ec_nzero( errno )
ec_neg1( fchdir(fd) )
```

There's still a problem. If you want to try to find it before I tell you what it is, stop reading *here*.

The problem is that if there's an error between the `chdir` and `fchdir` calls that jumps to the cleanup code, the call to `fchdir` won't be executed, and the current directory won't be restored. It's OK in this program because all such errors terminate the process, and the current directory is unique to each process. (The shell's current directory won't be affected.) But, still, if the error processing should be changed at some point, or if this code is copied and pasted into another program, things could go awry.

A good fix (not shown) is to put a second call to `fchdir` in the cleanup code, so it gets executed no matter what. You can initialize `fd` to –1 and test for that so you don't use it unless it has a valid value.

3.6.3 `mkdir` and `rmdir` System Calls

There are two system calls for making and removing a directory:

mkdir—make directory

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

int mkdir(
    const char *path,          /* pathname */
    mode_t perms               /* permissions */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

rmdir—remove directory

```
#include <unistd.h>

int rmdir(
    const char *path           /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

For `mkdir`, the permissions, how they interact with the file-creation mask, and the ownership of the new directory are the same as for `open` (Section 2.4). It automatically creates the special `.` and `..` links.

`rmdir` acts pretty much like `unlink`, which isn't allowed on directories.[13] One big restriction is that the directory to be removed has to be empty (except for `.` and `..`). If it's not empty, you have to remove its links first, which could involve multiple calls to `unlink` and multiple calls to `rmdir`, starting at the bottom of the subtree and working up. If that's what you really want to do, it's much easier just to write:

```
ec_neg1( system("rm -rf somedir") )
```

since the `rm` command knows how to walk a directory tree.

Here's an illustration of what we just said:

```
void rmdir_test(void)
{
    ec_neg1( mkdir("somedir", PERM_DIRECTORY) )
    ec_neg1( rmdir("somedir") )
    ec_neg1( mkdir("somedir", PERM_DIRECTORY) )
    ec_neg1( close(open("somedir/x", O_WRONLY | O_CREAT, PERM_FILE)) )
    ec_neg1( system("ls -ld somedir; ls -l somedir") )
    if (rmdir("somedir") == -1)
        perror("Expected error");
    ec_neg1( system("rm -rf somedir") )
    ec_neg1( system("ls -ld somedir") )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("rmdir_test");
EC_CLEANUP_END
}
```

`PERM_DIRECTORY` and `PERM_FILE` were explained in Section 1.1.5. The line that creates the file is a little weird, but handy when you just want to create a file. The only bad thing about it is that if `open` fails, the `errno` value reported will be the one for `close` (which will get an argument of –1), and we'll have to guess why the file couldn't be created.

The output from executing the function:

13. The superuser can use it on directories on some systems, but that's nonstandard.

```
drwx------    2 marc     users          72 Oct 18 15:32 somedir
total 0
-rw-r--r--    1 marc     users           0 Oct 18 15:32 x
Expected error: Directory not empty
ls: somedir: No such file or directory
```

3.6.4 Implementing `getcwd` (Walking Up the Tree)

We really had no fun at all calling `getcwd` back in Section 3.4.2, what with all the trouble it took to size the buffer. Let's have some fun now by *implementing* it!

The basic idea is to start with the current directory, get its i-number, and then go to its parent directory to find the entry with that i-node, thereby getting the name of the child. Then we repeat, until we can go no higher.

What does "no higher" mean? It means that either

```
chdir("..")
```

returns –1 with an `ENOENT` error, or that it succeeds but leaves us in the same directory—either behavior is possible, and both mean that we're at the root.

As we walk up the tree, we'll accumulate the components of the path as we discover them in a linked list with the newest (highest-level) entry at the top. So if the name of the current directory is `grandchild`, and then we move to its parent, `child`, and then to its parent, `parent`, and then find that we're at the root, the linked list we'll end up with is shown in Figure 3.4.

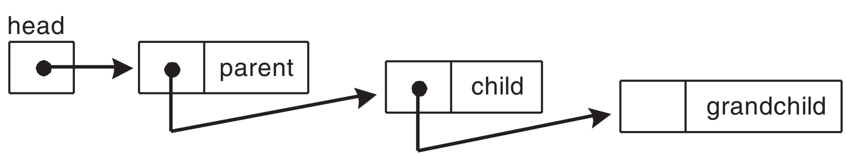

Figure 3.4 Liked list of pathname components.

Here's the structure for each linked-list node and the function that creates a new node and places it at the head of the list:

```
struct pathlist_node {
    struct pathlist_node *c_next;
    char c_name[1]; /* flexible array */
};

static bool push_pathlist(struct pathlist_node **head, const char *name)
{
    struct pathlist_node *p;

    ec_null( p = malloc(sizeof(struct pathlist_node) + strlen(name)) )
    strcpy(p->c_name, name);
    p->c_next = *head;
    *head = p;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The size of each `pathlist_node` is variable; the structure has enough space for the NUL byte that terminates the string, and, when allocating a node, we have to add to this space for the name, which you can see in the argument to `malloc`. Note that we put each node at the head of the list to keep the list in reverse order of when we encountered each node. That's exactly the order in which we want to assemble the components of the path, as shown by the function `get_pathlist`:

```
static char *get_pathlist(struct pathlist_node *head)
{
    struct pathlist_node *p;
    char *path;
    size_t total = 0;

    for (p = head; p != NULL; p = p->c_next)
        total += strlen(p->c_name) + 1;
    ec_null( path = malloc(total + 1) )
    path[0] = '\0';
    for (p = head; p != NULL; p = p->c_next) {
        strcat(path, "/");
        strcat(path, p->c_name);
    }
    return path;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

This function makes two passes over the list. The first just totals the space we need for the path (the +1 in the loop is for the slash), and the second builds the path string.

The final path-manipulation function frees the linked list, presumably to be called after `get_pathlist`:

```
static void free_pathlist(struct pathlist_node **head)
{
    struct pathlist_node *p, *p_next;

    for (p = *head; p != NULL; p = p_next) {
        p_next = p->c_next;
        free(p);
    }
    *head = NULL;
}
```

We used the `p_next` variable to hold the pointer to the next node because `p` itself becomes invalid as soon as we call `free`.

Here's some test code that builds the list shown in Figure 3.4:

```
struct pathlist_node *head = NULL;
char *path;

ec_false( push_pathlist(&head, "grandchild") )
ec_false( push_pathlist(&head, "child") )
ec_false( push_pathlist(&head, "parent") )
ec_null( path = get_pathlist(head) );
free_pathlist(&head);
printf("%s\n", path);
free(path);
```

And this is what got printed:

```
/parent/child/grandchild
```

A really convenient feature of handling the path this way is that we didn't have to preallocate space for the path, as we did when we called the standard `getcwd` function, in Section 3.4.2.

Now we're ready for `getcwdx`, our version of `getcwd`, which walks up the tree, calling `push_pathlist` at each level once it identifies an entry name as that of a child, and stops at the root. First comes a macro that tests whether two `stat` structures represent the same i-node, by testing both the device IDs and i-numbers:

```
#define SAME_INODE(s1, s2)  ((s1).st_dev == (s2).st_dev &&\
                            (s1).st_ino == (s2).st_ino)
```

We'll use this macro twice in the `getcwdx` function:

```
char *getcwdx(void)
{
    struct stat stat_child, stat_parent, stat_entry;
    DIR *sp = NULL;
    struct dirent *dp;
    struct pathlist_node *head = NULL;
    int dirfd = -1, rtn;
    char *path = NULL;

    ec_neg1( dirfd = open(".", O_RDONLY) )
    ec_neg1( lstat(".", &stat_child) )
    while (true) {
        ec_neg1( lstat("..", &stat_parent) )

        /* change to parent and detect root */
        if (((rtn = chdir("..")) == -1 && errno == ENOENT) ||
          SAME_INODE(stat_child, stat_parent)) {
            if (head == NULL)
                ec_false( push_pathlist(&head, "") )
            ec_null( path = get_pathlist(head) )
            EC_CLEANUP
        }
        ec_neg1( rtn )

        /* read directory looking for child */
        ec_null( sp = opendir(".") )
        while (errno = 0, (dp = readdir(sp)) != NULL) {
            ec_neg1( lstat(dp->d_name, &stat_entry) )
            if (SAME_INODE(stat_child, stat_entry)) {
                ec_false( push_pathlist(&head, dp->d_name) )
                break;
            }
        }
        if (dp == NULL) {
            if (errno == 0)
                errno = ENOENT;
            EC_FAIL
        }
        stat_child = stat_parent;
    }
```

```
EC_CLEANUP_BGN
    if (sp != NULL)
        (void)closedir(sp);
    if (dirfd != -1) {
        (void)fchdir(dirfd);
        (void)close(dirfd);
    }
    free_pathlist(&head);
    return path;
EC_CLEANUP_END
}
```

Some comments on this function:

- We use the `open`/`fchdir` technique to get and reset the current directory, as the function changes it as it walks the tree.
- In the loop, `stat_child` is for the child whose name we're trying to get the entry for, `stat_entry` is for each entry we get from a call to `readdir`, and `stat_parent` is for the parent, which becomes the child when we move up.
- The test for arriving at the root was explained at the beginning of this section: Either `chdir` fails or it doesn't take us anywhere.
- If we get to the root with the list still empty, we push an empty name onto the stack so the pathname will come out as `/`.
- In the `readdir` loop, we could have just skipped nondirectory entries, but it really wasn't necessary, as the `SAME_INODE` test is just as fast and accurate.

Finally, here's a little program that makes this into a command that behaves like the standard `pwd` command:

```
int main(void)
{
    char *path;
    ec_null( path = getcwdx() )
    printf("%s\n", path);
    free(path);
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

3.6.5 Implementing `ftw` (Walking Down the Tree)

There's a standard library function, `ftw` ("file tree walk"), which I won't describe specifically here, that provides a generalized way to recursively process the entries in a directory tree. You give it pointer to a function which it calls for each object it encounters, but it's more interesting for us to implement our own tree-walking function, using what we've learned in this chapter.

The first thing to clear up is whether the directory structure is really a tree, because our recursive algorithm depends on that. If there are any loops our program might get stuck, and if the same directory is linked to twice (by entries other than `.` and `..`), we might visit the same directory (and its subtree) more than once. There are two kinds of problems to think about:

1. It's pretty common for symbolic links to link to a directory, and that creates at least two links, since every directory is also hard linked to. There's no protection against symbolic links creating a loop, either.
2. On some systems the superuser can create a second hard link to a directory, with the `link` system call. This is almost never done, but it's a possibility our program might encounter.

Problem 1 can easily be dealt with by simply not following any symbolic links. We'll ignore Problem 2 for now, although Exercise 3.5 at the end of this chapter deals with it. So, for our purposes, the tree formed by hard links is indeed a tree.

We want our program, `aupls` (an extension of the one presented in Section 3.5.3) to act somewhat like `ls`, in that a `-R` argument makes it recurse, and a `-d` argument tells it to just list the information for a directory, rather than for that directory's contents. Without either of these arguments it behaves like the earlier `aupls`.

With the `-R` argument, `aupls` (like `ls`) first lists the path to a directory and all of the entries at that level, including directories, and then for each directory does the same thing, recursively, as in this example:

```
$ aupls -R /aup/common

/aup/common:
-rwxr-xr-x    1 root    root      1145 Oct  2 10:21 makefile
-rwxr-xr-x    1 root    root      171 Aug 23 10:41 logf.h
-rwxr-xr-x    1 root    root      1076 Aug 26 15:24 logf.c
drwxr-xr-x    1 root    root      4096 Oct  2 12:20 cf
-rwxr-xr-x    1 root    root      245 Aug 26 15:29 notes.txt

/aup/common/cf:
-rwxr-xr-x    1 root    root      1348 Oct  3 13:52 cf-ec.c-ec_push.htm
-rwxr-xr-x    1 root    root      576 Oct  3 13:52 cf-ec.c-ec_print.htm
```

```
-rwxr-xr-x    1 root    root      450 Oct  3 13:52 cf-ec.c-ec_reinit.htm
-rwxr-xr-x    1 root    root      120 Oct  3 13:52 cf-ec.c-ec_warn.htm
```

As you'll see when we get to the code, for each level (a `readdir` loop), there are two passes. In Pass 1 we print the "stat" information, and then in Pass 2, for each directory, we recurse, with a Pass 1 in that directory, and so on.

It's convenient to represent the traversal information with a structure that's passed to each function in the program, rather than using global variables or a long argument list:

```
typedef enum {SHOW_PATH, SHOW_INFO} SHOW_OP;

struct traverse_info {
    bool ti_recursive;                              /* -R option? */
    char *ti_name;                                  /* current entry */
    struct stat ti_stat;                            /* stat for ti_name */
    bool (*ti_fcn)(struct traverse_info *, SHOW_OP); /* callback fcn */
};
```

Here's the callback function we're going to use:

```
static bool show_stat(struct traverse_info *p, SHOW_OP op)
{
    switch (op) {
    case SHOW_PATH:
        ec_false( print_cwd(false) )
        break;
    case SHOW_INFO:
        ls_long(&p->ti_stat, p->ti_name);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

`show_stat` is called with `op` set to `SHOW_PATH` to print a pathname heading, and with `SHOW_INFO` for the detail lines. `ls_long` is from Section 3.5.3, and `print_cwd` is almost identical to the sample code we showed in Section 3.4.2:

```
static bool print_cwd(bool cleanup)
{
    char *cwd;
```

```
    if (cleanup)
        (void)get_cwd(true);
    else {
        ec_null( cwd = get_cwd(false) )
        printf("\n%s:\n", cwd);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Later we'll see a `print_cwd(true)` call in the overall cleanup code.

Now let's jump up to the top level for the `main` function to see how everything is initialized and how the listing gets started:

```
#define USAGE "Usage: aupls [-Rd] [dir]\n"

static long total_entries = 0, total_dirs = 0;

int main(int argc, char *argv[])
{
    struct traverse_info ti = {0};
    int c, status = EXIT_FAILURE;
    bool stat_only = false;

    ti.ti_fcn = show_stat;
    while ((c = getopt(argc, argv, "dR")) != -1)
        switch(c) {
        case 'd':
            stat_only = true;
            break;
        case 'R':
            ti.ti_recursive = true;
            break;
        default:
            fprintf(stderr, USAGE);
            EC_CLEANUP
        }
    switch (argc - optind) {
    case 0:
        ti.ti_name = ".";
        break;
    case 1:
        ti.ti_name = argv[optind];
        break;
```

```
    default:
        fprintf(stderr, USAGE);
        EC_CLEANUP
    }
    ec_false( do_entry(&ti, stat_only) )
    printf("\nTotal entries: %ld; directories = %ld\n", total_entries,
      total_dirs);
    status = EXIT_SUCCESS;
    EC_CLEANUP

EC_CLEANUP_BGN
    print_cwd(true);
    exit(status);
EC_CLEANUP_END
}
```

The two globals `total_entries` and `total_dirs` are used to keep overall counts, which we found interesting to display at the end; we'll see where they're incremented shortly.

`getopt` is a standard library function (part of POSIX/SUS, not Standard C). Its third argument is a list of allowable option letters. When it's finished with options, it sets the global `optind` to the index of the next argument to be processed, which in our example is an optional pathname. If none is given, we assume the current directory.

The real work is done by `do_entry`, which looks like this:

```
static bool do_entry(struct traverse_info *p, bool stat_only)
{
    bool is_dir;

    ec_neg1( lstat(p->ti_name, &p->ti_stat) )
    is_dir = S_ISDIR(p->ti_stat.st_mode);
    if (stat_only || !is_dir) {
        total_entries++;
        if (is_dir)
            total_dirs++;
        ec_false( (p->ti_fcn)(p, SHOW_INFO) )
    }
    else if (is_dir)
        ec_false( do_dir(p) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

do_entry is used in one of two ways: If the stat_only argument is true or if the current entry isn't a directory, it increments the global counters and then calls the callback function in SHOW_INFO mode. This is done when the -d argument is specified, when a nondirectory is specified on the aupls command line, or during Pass 1 of a directory listing. Otherwise, for a directory, it calls the function do_dir that contains the readdir loops to process a directory's entries:

```
static bool do_dir(struct traverse_info *p)
{
    DIR *sp = NULL;
    struct dirent *dp;
    int dirfd = -1;
    bool result = false;

    ec_neg1(  dirfd = open(".", O_RDONLY) )
    if ((sp = opendir(p->ti_name)) == NULL || chdir(p->ti_name) == -1) {
        if (errno == EACCES) {
            fprintf(stderr, "%s: Permission denied.\n", p->ti_name);
            result = true;
            EC_CLEANUP
        }
        EC_FAIL
    }
    if (p->ti_recursive)
        ec_false( (p->ti_fcn)(p, SHOW_PATH) )
    while (errno = 0, ((dp = readdir(sp)) != NULL)) {
        if (strcmp(dp->d_name, ".") == 0 ||
          strcmp(dp->d_name, "..") == 0)
            continue;
        p->ti_name = dp->d_name;
        ec_false( do_entry(p, true) )
    }
    if (errno != 0)
        syserr_print("Reading directory (Pass 1)");
    if (p->ti_recursive) {
        rewinddir(sp);
        while (errno = 0, ((dp = readdir(sp)) != NULL)) {
            if (strcmp(dp->d_name, ".") == 0 ||
              strcmp(dp->d_name, "..") == 0)
                continue;
            p->ti_name = dp->d_name;
            ec_false( do_entry(p, false) )
        }
        if (errno != 0)
            syserr_print("Reading directory (Pass 2)");
    }
    result = true;
    EC_CLEANUP
```

```
EC_CLEANUP_BGN
    if (dirfd != -1) {
        (void)fchdir(dirfd);
        (void)close(dirfd);
    }
    if (sp != NULL)
        (void)closedir(sp);
    return result;
EC_CLEANUP_END
}
```

`do_dir` is where the recursion takes place. It opens `dirfd` to the current directory so it can be restored on cleanup. Then it opens the directory with `opendir` and changes to it so the entries can be processed relative to their parent directory. It's very common to get an `EACCES` error because a directory isn't readable (`opendir` fails) or not searchable (`chdir` fails), so we just want to print a message in those cases—not terminate processing.

Next, if the `-R` argument was specified, we tell the callback function to print the current directory path. This is so all normal printing can be localized to one callback function.

Then we have the Pass 1 `readdir` loop (in which `do_entry` is called with `stat_only` set to true), a `rewinddir` system call, and, if `-R` was specified, the Pass 2 `readdir` loop (calling `do_entry` with `stat_only` set to `false`). The recursion occurs because a Pass 2 call to `do_entry` might call `do_dir` recursively. Note that both `readdir` loops skip the `.` and `..` entries, which `ls` also omits by default.

We decided to treat errors from `readdir` as nonfatal, to keep things going, rather than bailing out with the `ec_nzero` macro, which explains the two calls to `syserr_print`, which is:

```
void syserr_print(const char *msg)
{
    char buf[200];

    fprintf(stderr, "ERROR: %s\n", syserrmsg(buf, sizeof(buf), msg,
      errno, EC_ERRNO));
}
```

`syserrmsg` was in Section 1.4.1.

That's it—our own fully recursive version of `ls`!

3.7 Changing an I-Node

The `stat` family of system calls (Section 3.5.1) retrieves information from an i-node, and I explained there how various data fields in the i-node get changed as a file is manipulated. Some of the fields can also be changed directly by system calls that are discussed in this section. Table 3.3 indicates what system calls do what. The notation "fixed" means the field isn't changeable at all—you have to make a new i-node if you want it to be something else—and the notation "side-effect" means it's changeable only as a side-effect of doing something else, such as making a new link with `link`, but isn't changeable directly.

Table 3.3 Changing I-Node Fields

I-Node Field	Description	Changed by…
`st_dev`	device ID of file system	fixed
`st_ino`	i-number	fixed
`st_mode`	mode	`chmod, fchmod`
`st_nlink`	number of hard links	side-effect
`st_uid`	user ID	`chown, fchown, lchown`
`st_gid`	group ID	`chown, fchown, lchown`
`st_rdev`	device ID (if special file)	fixed
`st_size`	size in bytes	side-effect
`st_atime`	last access	`utime`
`st_mtime`	last data modification	`utime`
`st_ctime`	last i-node modification	side-effect
`st_blksize`	optimal I/O size	fixed
`st_blocks`	allocated 512-byte blocks	side-effect

3.7.1 `chmod` and `fchmod` System Calls

chmod—change mode of file by path

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

int chmod(
    const char *path,      /* pathname */
    mode_t mode            /* new mode */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchmod—change mode of file by file descriptor

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

int fchmod(
    int fd,                /* file descriptor */
    mode_t mode            /* new mode */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `chmod` system call changes the mode of an existing file of any type. It can't be used to change the type itself, however, so only the `S_ISUID`, `S_ISGID`, and `S_ISVTX` flags or the permission bits can be changed. Use the macros as with the `stat` structure (Section 3.5.1). `fchmod` is similar but takes an open file descriptor instead of a path.

The caller's effective user-ID must match the user-ID of the file, or the caller must be the superuser. It's not enough to have write permission or for the caller's effective group-ID to match that of the file. In addition, if `S_ISGID` is being set, the effective group-ID must match (except for the superuser).

Unless you're going to set the entire mode, you'll first have to get the existing mode with a call to one of the `stat` functions, set or clear the bits you want to change, and then execute `chmod` with the revised mode.

It's uncommon for `chmod` to be called from within an application, as the mode is usually set when a file is created. Users, however, frequently use the `chmod` command.

3.7.2 `chown`, `fchown`, and `lchown` System Calls

`chown` changes the user-ID and group-ID of a file. Only the owner (process's effective user-ID equal to the file's user-ID) or the superuser may execute it. If either `uid` or `gid` is –1, the corresponding ID is left alone.

chown—change owner and group of file by path

```
#include <unistd.h>

int chown(
    const char *path,      /* pathname */
    uid_t uid,             /* new user ID */
    gid_t gid              /* new group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchown—change owner and group of file by file descriptor

```
#include <unistd.h>

int fchown(
    int fd,                /* file descriptor */
    uid_t uid,             /* new user ID */
    gid_t gid              /* new group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

lchown—change owner and group of symbolic link by path

```
#include <unistd.h>

int lchown(
    const char *path,      /* pathname */
    uid_t uid,             /* user ID */
    gid_t gid              /* group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`fchown` is similar but takes an open file descriptor instead of a path. `lchown` is also similar but acts on the path argument directly, rather than what it leads to if it's a symbolic link.

Unless the caller is the superuser, these system calls clear the set-user-ID and set-group-ID bits. This is to prevent a rather obvious form of break-in:

```
$ cp /bin/sh mysh          [get a personal copy of the shell]
$ chmod 4700 mysh          [turn on the set-user-ID bit]
$ chown root mysh          [make the superuser the owner]
$ my sh                    [become superuser]
```

If you want your own superuser shell, you must reverse the order of `chmod` and `chown`; however, unless you are already the superuser, you won't be allowed to execute the `chmod`. So the loophole is plugged.

Some UNIX systems are configured to operate with a slightly different rule if the macro `_POSIX_CHOWN_RESTRICTED` is set (tested with `pathconf` or `fpathconf`).

Only the superuser can change the owner (user-ID), but the owner can change the group-ID to the process's effective group-ID or to one of its supplemental group-IDs. In other words, the owner can't give the file away—at most the group-ID can be changed to one of the group-IDs associated with the process.

As ownership is usually changed by the `chown` command (which calls the `chown` system call), these rules don't usually affect applications directly. They do affect system administration and how users are able to use the system.

3.7.3 `utime` System Call

utime—set file access and modification times

```
#include <utime.h>

int utime(
     const char *path,              /* pathname */
     const struct utimbuf *timbuf   /* new times */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct utimbuf—structure for utime

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

`utime` changes the access and modification times of a file of any type.[14] The type `time_t` in the structure is the number of seconds since the epoch, as defined in Section 1.7.1. Only the owner or superuser can change the times with a `timbuf` argument.

If the `timbuf` argument is `NULL`, however, the access and modification times are set to the current time. This is done to force a file to appear up-to-date without rewriting it and is primarily for the benefit of the `touch` command. Anyone with write permission can do this, since writing, which they're allowed, would also change the times.

Aside from `touch`, `utime` is most often used when restoring files from a dump tape or when receiving files across a network. The times are reset to the values they had originally, from data that comes along with the file somehow. The status-

14. On some systems there's a similar `utimes` system call, but it's obsolete.

change time can't be reset, but that's appropriate since the i-node didn't move anywhere—a new one got created.

3.8 More File-Manipulation Calls

This section covers some additional file-manipulation system calls that didn't fit in the previous sections.

3.8.1 `access` System Call

Unlike any other system call that deals with permissions, `access` checks the *real* user-ID or group-ID, not the *effective* ones.

access—determine accessibility of file

```
#include <unistd.h>

int access(
     const char *path,      /* pathname */
     int what               /* permission to be tested */
);
/* Returns 0 if allowed or -1 if not or on error (sets errno) */
```

The argument `what` uses the following flags, the first three of which can be ORed together:

```
R_OK                        /* read permission */
W_OK                        /* write permission */
X_OK                        /* execute (search) permission */
F_OK                        /* test for existence */
```

If the process's real user-ID matches that of the path, the owner permissions are checked; if the real group-ID matches, the group permissions are checked; and if neither, the other permissions are checked.

There are two principal uses of `access`:

- To check whether the real user or group has permission to do something with a file, in case the set-user-ID or set-group-ID bits are set. For example, there might be a command that sets the user-ID to superuser on execution, but it wants to check whether the real user has permission to unlink a file from a directory before doing so. It can't just test to see if `unlink` will fail with an `EACCES` error because, as the effective user-ID is the superuser, it cannot fail for that reason.

- To check whether a file exists. You could use `stat` instead, but `access` is simpler. (Actually, the whole `access` system call could be written as a library function calling `stat`, and, in some implementations, it even might be implemented that way.)

If `path` is a symbolic link, that link is followed until a nonsymbolic link is found.[15]

When you call `access`, you probably don't want to put it in an `ec_neg1` macro because, if it returns –1, you'll want to distinguish an `EACCES` error (for `R_OK`, `W_OK`, and/or `X_OK`) or an `ENOENT` error (for `F_OK`) from other errors. Like this:

```
if (access("tmp", F_OK) == 0)
    printf("Exists\n");
else if (errno == ENOENT)
    printf("Does not exist\n");
else
    EC_FAIL
```

3.8.2 `mknod` System Call

mknod—make file

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

int mknod(
    const char *path,          /* pathname */
    mode_t perms,              /* mode */
    dev_t dev                  /* device-ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`mknod` makes a regular file, a directory, a special file, or a named pipe (FIFO). The only portable use (and the only one for which you don't need to be superuser) is to make a named pipe, but for that you have `mkfifo` (Section 7.2.1). You also don't need it for regular files (use `open`) or for directories (use `mkdir`) either. You can't use it for symbolic links (use `symlink`) or for sockets (use `bind`).

15. This is true of almost all system calls that take a pathname, unless they start with the letter l. Two exceptions are `unlink` and `rename`.

Therefore, the main use of `mknod` is to make the special files, usually in the /dev directory, that are used to access devices. As this is generally done only when a new device driver is installed, the `mknod` command is used, which executes the system call.

The `perms` argument uses the same bits and macros as were defined for the `stat` structure in Section 3.5.1, and `dev` is the device-ID as defined there. If you've installed a new device driver, you'll know what the device-ID is.

3.8.3 `fcntl` System Call

In Section 2.2.3, which you may want to reread before continuing, I explained that several open file *descriptors* can share the same open file *description,* which holds the file offset, the status flags (e.g., `O_APPEND`), and access modes (e.g., `O_RDONLY`). Normally, you set the status flags and access modes when a file is opened, but you can also get and set the status flags at any time with the `fcntl` system call. You can also use it to get, but not set, the access modes.

fcntl—control open file

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(
    int fd,             /* file descriptor */
    int op,             /* operation */
    ...                 /* optional argument depending on op */
);
/* Returns result depending on op or -1 on error (sets errno) */
```

There are ten operations in all, but I'm only going to discuss four of them right here. The others will be discussed elsewhere, as indicated in Table 3.4.

Table 3.4 `fcntl` Operations

Operation	Purpose	Where
`F_DUPFD`	Duplicate file descriptor	Section 6.3
`F_GETFD`	Get file-descriptor flags	Here
`F_SETFD`	Set file-descriptor flags (uses third `int` argument)	Here
`F_GETFL`	Get file-description status flags and access modes	Here

Table 3.4 `fcntl` Operations (cont.)

Operation	Purpose	Where
`F_SETFL`	Set file-description status flags (uses third `int` argument)	Here
`F_GETOWN`	Used with sockets*	Section 8.7
`F_SETOWN`	Used with sockets*	Section 8.7
`F_GETLK`	Get a lock	Section 7.11.4
`F_SETLK`	Set or clear a lock	Section 7. 11.4
`F_SETLKW`	Set or clear a lock	Section 7. 11.4

* Too complicated to even summarize here.

For all of the "get" and "set" operations, you should get the current value, set or clear the flags you want to modify, and then do the set operation, even if only one flag is defined. That way your code will still work if more flags are added later or if there are implementation-dependent, nonstandard flags that you don't know about. For example, the wrong way to set the `O_APPEND` flag is:

```
ec_neg1( fcntl(fd, F_SETFL, O_APPEND) ) /* wrong */
```

and the correct way is:

```
ec_neg1( flags = fcntl(fd, F_GETFL) )
ec_neg1( fcntl(fd, F_SETFL, flags | O_APPEND) )
```

If we wanted to clear the `O_APPEND` flag, the second line would have been:

```
    ec_neg1( fcntl(fd, F_SETFL, flags & ~O_APPEND) )
```

(The rule is "OR to set, AND complement to clear.")

The only standard file-descriptor flag that's defined is the close-on-exec flag, `FD_CLOEXEC`, which indicates whether the file descriptor will be closed when an `exec` system call is issued. There's more about close-on-exec in Section 5.3. You use the `F_GETFD` and `F_SETFD` operations for it. This is an important use for `fcntl`, as this flag can't be set when a file is opened.

You use the `F_GETFL` and `F_SETFL` operations for the file-description status flags and access modes. The access modes, which you can get but not set, are one of

O_RDONLY, O_WRONLY, or O_RDWR. As these are values, not bit-masks, the mask O_ACCMODE must be used to pick them out of the returned value, like this:

```
ec_neg1( flags = fcntl(fd, F_GETFL) )
if ((flags & O_ACCMODE) == O_RDONLY)
    /* file is opened read-only */
```

The following two if statements are both wrong:

```
if (flags & O_RDONLY)                /* wrong */
if ((flags & O_RDONLY) == O_RDONLY)  /* still wrong */
```

The status flags, all of which are listed in Table 2.1 in Section 2.4.4, are more interesting. You can get and set O_APPEND, O_DSYNC, O_NOCTTY, O_NONBLOCK, O_RSYNC, and O_SYNC. You can get O_CREAT, O_TRUNC, and O_EXCL, but it makes no sense to set them because they play a role only with the open system call (after that it's too late). I showed an example using O_APPEND previously.

There's more on the O_NONBLOCK flag in Sections 4.2.2 and 7.2.

3.9 Asynchronous I/O

This section explains how to initiate an I/O operation so that your program doesn't have to wait around for it to be completed. You can go off and do something else, and then check back later for the operation's status.

3.9.1 Synchronized vs. Synchronous, Once Again

Before reading this section, make sure you've read Section 2.16.1, where synchronized (vs. nonsynchronized) I/O is explained. Here we're concerned with asynchronous I/O, which means that the I/O operation is initiated by a system call (e.g., aio_read) that returns before it's completed, and its completion is dealt with separately. That is:

- *Synchronized* means that the I/O is completed when physical I/O is completed. *Nonsynchronized* means that I/O between the process and the buffer cache is good enough for completion.
- *Synchronous* means that the system call returns only when the I/O has completed, as defined by the previous paragraph. *Asynchronous* means that the system call returns as soon as the I/O has been initiated, and completion is tested for by other system calls.

To say the same thing in one sentence: Synchronous/asynchronous refers to whether the call waits for completion, and synchronized/nonsynchronized specifies what "completion" means.

Some of this section also assumes you know how to work with signals, so you may want to read Chapter 9 before reading this section, or at least refer to parts of Chapter 9 as you read, especially Section 9.5.6.

As I've noted (Section 2.9), most `write`s in UNIX are already somewhat asynchronous because the system call returns as soon as the data has been transferred to the buffer cache, and the actual output takes place later. If the `O_SYNC` or `O_DSYNC` flags (Section 2.16.3) are set, however, a `write` doesn't return until the actual output is completed. By contrast, normal `read`s usually involve waiting for the actual input because, unless there's been read-ahead or the data just happens to be in a buffer, a `read` waits until the data can be physically read.

With asynchronous I/O (AIO), a process doesn't have to wait for a `read`, or for a synchronized (`O_SYNC` or `O_DSYNC`) `write`. The I/O operation is initiated, the AIO call returns right away, and later the process can call `aio_error` to find out whether the operation has completed or even be notified when it completes via a signal or the creation of a new thread (Section 5.17).

The term "completed" as used here means just what it does with `read` and `write`. It doesn't mean that the I/O is synchronized unless the file descriptor has been set for synchronization, as explained in Section 2.16.3. Thus, there are four cases, as shown in Table 3.5.

Table 3.5 Synchronized vs. Synchronous read and write

	Synchronous	Asynchronous
Nonsynchronized	`read/write`; `O_SYNC` and `O_DSYNC` clear	`aio_read/aio_write`; `O_SYNC` and `O_DSYNC` clear
Synchronized	`read/write`; `O_SYNC` or `O_DSYNC` set	`aio_read/aio_write`; `O_SYNC` or `O_DSYNC` set

AIO can increase performance on `read`s, synchronized or not, if you can organize your application so that it can initiate `read`s before it needs the data and then go do something else useful. If it doesn't have anything else it can do in the meantime, however, it may as well just block in the `read` and let another process or thread run. AIO also helps with synchronized `write`s but doesn't do much for nonsynchronized `write`s, as the buffer cache already does much the same thing.

Don't confuse asynchronous I/O with nonblocking I/O that you specify by setting the `O_NONBLOCK` flag (Sections 2.4.4 and 4.2.2). Nonblocking means that the call returns if it would block, but it doesn't do the work.[16] Asynchronous means that it initiates the work and then returns.

The AIO functions are part of the Asynchronous Input and Output option, as represented by the `_POSIX_ASYNCHRONOUS_IO` macro. You can test it at run-time with `pathconf`, and I showed example code that does that in Section 1.5.4.

One system call in this section, `lio_listio`, can be used for either synchronous or asynchronous I/O.

3.9.2 AIO Control Block

All of the AIO system calls use a control block to keep track of the state of an operation, and different operations have to use distinct control blocks. When an operation is complete, its control block can be reused, of course.

struct aiocb—AIO control block

```
struct aiocb {
    int aio_fildes;                  /* file descriptor */
    off_t aio_offset;                /* file offset */
    volatile void *aio_buf;          /* buffer */
    size_t aio_nbytes;               /* size of transfer */
    int aio_reqprio;                 /* request priority offset */
    struct sigevent aio_sigevent;    /* signal information */
    int aio_lio_opcode;              /* operation to be performed */
};
```

The first four members are like the ones for `pread` and `pwrite` (Section 2.14). That is, the three arguments that `read` or `write` would take plus a file offset for an implicit `lseek`.

The `sigevent` structure is explained fully in Section 9.5.6. With it, you can arrange for a signal to be generated or a thread to be started when the operation is complete. You can pass an arbitrary integer or pointer value to the signal handler or the thread, and usually this will be a pointer to the control block. If you don't

16. `connect` is an exception; see Section 8.1.2.

want a signal or a thread, you set member aio_sigevent.sigev_notify to SIGEV_NONE.

The aio_reqprio member is used to affect the priority of the operation and is only available when two other POSIX options, _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING, are supported. For more on this feature, see [SUS2002].

The last member, aio_lio_opcode, is used with the lio_listio system call, as explained in Section 3.9.9.

3.9.3 aio_read and aio_write

The basic AIO functions are aio_read and aio_write. Just as with pread and pwrite, the aio_offset member determines where in the file the I/O is to take place. It's ineffective if the file descriptor is open to a device that doesn't allow seeking (e.g., a socket) or, for aio_write, if the O_APPEND flag is set (Section 2.8). The buffer and size are in the control block instead of being passed as arguments.

aio_read—asynchronous read from file

```
#include <aio.h>

int aio_read(
    struct aiocb *aiocbp      /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

aio_write—asynchronous write to file

```
#include <aio.h>

int aio_write(
    struct aiocb *aiocbp      /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

A successful return from these functions just means that the operation was initiated; it certainly doesn't mean that the I/O was successful, or even that the values set in the control block were OK. You always have to check the result with aio_error (next section). An implementation might report an error such as invalid offset right away, with a –1 return from aio_read or aio_write, or it might return 0 and report the bad news later.

3.9.4 `aio_error` and `aio_return`

aio_error—retrieve error status for asynchronous I/O operation

```
#include <aio.h>

int aio_error(
    const struct aiocb *aiocbp    /* control block */
);
/* Returns 0, errno value, or EINPROGRESS (does not set errno) */
```

If the operation is complete, `aio_error` returns 0 if it was successful or the `errno` value that an equivalent `read`, `write`, `fsync`, or `fdatasync` would have returned (Sections 2.9, 2.10, and 2.16.2). If the operation isn't complete, it returns `EINPROGRESS`. If you know that the operation completed (e.g., you got a signal saying so), you can treat `EINPROGRESS` just like any other error and use the `ec_rv` macro, as we did in Section 3.6.1:

```
ec_rv( aio_error(aiocbp) )
```

I should mention, though, that the asynchronous I/O calls are usually used in fairly advanced applications for which our simple "ec" error checking may not be suitable, as I noted in Section 1.4.2. Still, `ec_rv` is fine during development because of its ability to provide a function-call trace.

If `aio_error` says the operation was successful, you still may want the return value from the equivalent `read` or `write` to get the actual number of bytes transmitted. You do that with `aio_return`:

aio_return—retrieve return status of asynchronous I/O operation

```
#include <aio.h>

ssize_t aio_return(
    struct aiocb *aiocbp          /* control block */
);
/* Returns operation return value or -1 on error (sets errno) */
```

You're supposed to call `aio_return` only if `aio_error` reported success. A –1 return from `aio_return` means that you called it wrong, not that the operation returned an error. Also, you can call `aio_return` only once per operation, as the value may be discarded once it's been retrieved.

3.9.5 **aio_cancel**

You can cancel an outstanding asynchronous operation with `aio_cancel`:

aio_cancel—cancel asynchronous I/O request

```
#include <aio.h>

int aio_cancel(
     int fd,                    /* file descriptor */
     struct aiocb *aiocbp       /* control block */
);
/* Returns result code or -1 on error (sets errno) */
```

If the `aiocbp` argument is `NULL`, this call attempts to cancel all asynchronous operations on file descriptor `fd`. Operations that weren't cancelled report their completion in the normal way, but those that got cancelled report (via `aio_error`), an error code of `ECANCELED`.

If the `aiocbp` argument isn't `NULL`, `aio_cancel` tries to cancel just the operation that was started with that control block. In this case the `fd` argument must be the same as the `aio_fildes` member of the control block.

If `aio_cancel` succeeds it returns one of these result codes:

`AIO_CANCELED`	All the requested operations were cancelled.
`AIO_NOTCANCELED`	One or more requested operations (maybe all) were not cancelled because they were already in progress. You have to call `aio_error` on each operation to find out which were cancelled.
`AIO_ALLDONE`	None of the requested operations were cancelled because all were completed.

3.9.6 **aio_fsync**

Aside from `read` and `write` equivalents, there's a third kind of I/O that can be done asynchronously: flushing of the buffer-cache, as is done with `fsync` or `fdatasync` (see Section 2.16.2 for the distinction). Both kinds of flushing are handled with the same call:

aio_fsync—initiate buffer-cache flushing for one file

```
#include <aio.h>

int aio_fsync(
    int op,                           /* O_SYNC or O_DSYNC */
    struct aiocb *aiocbp              /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The control block is used a little differently than it is with the other calls: All buffers associated with the file descriptor given by the `aio_fildes` member are flushed, not just those associated with the operation that was started with the control block, if one even was. The request to synchronize (i.e., flush the buffers) is asynchronous: It doesn't happen right away, but is only initiated, and you can check for completion the usual way (e.g., with `aio_error` or with a signal). So, as with `aio_read` and `aio_write`, a return of zero just means that the initiation went OK.

3.9.7 `aio_suspend`

Instead of getting notified asynchronously via a signal or a thread when I/O is completed, you can decide to become synchronous by simply waiting for it to complete:

aio_suspend—wait for asynchronous I/O request

```
#include <aio.h>

int aio_suspend(
    const struct aiocb *const list[], /* array of control blocks */
    int cbcnt,                        /* number of elements in array */
    const struct timespec *timeout    /* max time to wait */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If `timeout` is `NULL`, `aio_suspend` takes an array of `cbcnt` control blocks, and blocks until *one* of them completes. Each must have been used to initiate an asynchronous I/O operation. If at least one is already complete at the time of the call, `aio_suspend` returns immediately.

To help in reusing the same array, it's OK for an element of `list` to be `NULL`, but such an element is included in `cbcnt`.

If timeout isn't NULL, aio_suspend blocks at most for that amount of time, and then returns –1 with errno set to EAGAIN. The timespec structure is in Section 1.7.2.

Like most other system calls that block, aio_suspend can be interrupted by a signal, as explained further in Section 9.1.4. This leads to one very tricky situation: The completion that aio_suspend is waiting for could, depending on how the control block is set up, generate a signal, and that will cause aio_suspend to be interrupted, which causes it to return –1 with errno set to EINTR. In some cases why it returned doesn't matter, but the problem with the EINTR return is that any signal could have caused it, not just one from the completed I/O request. You can avoid having an asynchronous-completion signal interrupt aio_suspend in one of two ways:

- Use a signal to indicate completion or call aio_suspend, but not both.
- Set the SA_RESTART flag (Section 9.1.6) for the signal.

Or, let a signal interrupt aio_suspend and just call aio_error for each element of the list you pass to aio_suspend when it returns to see what, if anything, happened, and reissue the aio_suspend (perhaps in a loop) if nothing did.

3.9.8 Example Comparing Synchronous and Asynchronous I/O

This section shows an example of how to use the AIO calls and also demonstrates some of their advantages over synchronous I/O.

First, here's a program called sio that uses conventional, synchronous I/O to read a file. Every 8000 reads, it also reads the standard input.

```
#define PATH "/aup/c3/datafile.txt"
#define FREQ 8000

static void synchronous(void)
{
    int fd, count = 0;
    ssize_t nread;
    char buf1[512], buf2[512];

    ec_neg1( fd = open(PATH, O_RDONLY) )
    timestart();
    while (true) {
        ec_neg1( nread = read(fd, buf1, sizeof(buf1)) )
        if (nread == 0)
            break;
```

```
        if (count % FREQ == 0)
            ec_neg1( read(STDIN_FILENO, buf2, sizeof(buf2)) )
        count++;
    }
    timestop("synchronous");
    printf("read %d blocks\n", count);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("synchronous")
EC_CLEANUP_END
}
```

(`timestart` and `timestop` were in Section 1.7.2.)

The standard input is connected to a pipe that's being filled from a program called `feed`:

```
$ feed | sio
```

`feed` is a very reluctant writer—it writes only once every 20 seconds:

```
int main(void)
{
    char buf[512];

    memset(buf, 'x', sizeof(buf));
    while (true) {
        sleep(20);
        write(STDOUT_FILENO, buf, sizeof(buf));
    }
}
```

With the pipeline as shown, `sio` took 0.12 sec. of user CPU time, 0.73 sec. of system CPU time, and 200.05 sec. of elapsed time to run to completion. Obviously, a lot of the elapsed time was spent waiting for input on the pipe.

This is a contrived example, but, nonetheless, it provides an opportunity to improve the elapsed time with asynchronous I/O. If reading of the pipe can be done asynchronously, the file can be read in the meantime, as in this recoding:

```
static void asynchronous(void)
{
    int fd, count = 0;
    ssize_t nread;
    char buf1[512], buf2[512];
    struct aiocb cb;
    const struct aiocb *list[1] = { &cb };
```

```
    memset(&cb, 0, sizeof(cb));
    cb.aio_fildes = STDIN_FILENO;
    cb.aio_buf = buf2;
    cb.aio_nbytes = sizeof(buf2);
    cb.aio_sigevent.sigev_notify = SIGEV_NONE;
    ec_neg1( fd = open(PATH, O_RDONLY) )
    timestart();
    while (true) {
        ec_neg1( nread = read(fd, buf1, sizeof(buf1)) )
        if (nread == 0)
            break;
        if (count % FREQ == 0) {
            if (count > 1) {
                ec_neg1( aio_suspend(list, 1, NULL) )
                ec_rv( aio_error(&cb) )
            }
            ec_neg1( aio_read(&cb) )
        }
        count++;
    }
    timestop("asynchronous");
    printf("read %d blocks\n", count);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("asynchronous")
EC_CLEANUP_END
}
```

Notice how the control block is set up; it's zeroed first to make sure that any additional implementation-dependent members are zeroed. Every 8000 reads (`FREQ` was defined earlier), the program issues an `aio_read` call, and thereafter it waits in `aio_suspend` for the control block to be ready before it issues the next call. While the `aio_read` is working, it continues reading the file, not suspending again for 8000 more reads. This time the user CPU time was worse—0.18 sec.—because there was more work to do. The system time was about the same (0.70 sec.), but the elapsed time was shorter: 180.58 sec.

You won't always see a benefit this great in using AIO, and sometimes you'll see an improvement even better than this example showed. The keys are:

- The program has to have some useful work to do while the I/O is processing asynchronously.
- The benefits have to outweigh the increased bookkeeping costs and the increased number of system calls. You might also consider the increased programming complexity and the fact that AIO is not supported on all systems.

3.9.9 `lio_listio`

There's one other use for a list of control blocks: Batching I/O requests together and initiating them with a single call:

lio_listio—list-directed I/O

```
#include <aio.h>

int lio_listio(
     int mode,                        /* LIO_WAIT or LIO_NOWAIT */
     struct aiocb *const list[],      /* array of control blocks */
     int cbcnt,                       /* number of elements in array */
     struct sigevent *sig             /* NULL or signal to generate */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Like `aio_suspend`, `lio_listio` takes a list of `cbcnt` control blocks, ignoring `NULL` elements in `list`. It initiates the requests in no particular order. Each operation is specified by the `aio_lio_opcode` member of the control block:

`LIO_READ` Same as if `aio_read` or `pread` had been called.

`LIO_WRITE` Same as if `aio_write` or `pwrite` had been called.

`LIO_NOP` A no-op; ignore the control block.

The `mode` argument determines whether the I/O initiated by `lio_listio` is synchronous or asynchronous, as shown by Table 3.6, which resembles Table 3.5 (see Section 3.9.1).

Table 3.6 Synchronized vs. Synchronous `lio_listio`

	Synchronous	Asynchronous
Nonsynchronized	`LIO_WAIT`; `O_SYNC` and `O_DSYNC` clear	`LIO_NOWAIT`; `O_SYNC` and `O_DSYNC` clear
Synchronized	`LIO_WAIT`; `O_SYNC` or `O_DSYNC` set	`LIO_NOWAIT`; `O_SYNC` or `O_DSYNC` set

So, `lio_listio` isn't only for asynchronous I/O like the "aio" system calls—it can be used any time you want to batch I/O calls.

For asynchronous `lio_listio`s, with a mode of `LIO_NOWAIT`, you can request notification via a signal or thread-startup with the `sig` argument, just as with the aio_sigevent member of a control block. But here, the notification means that *all* the requests in the list have completed. You can still be notified of completions via the aio_sigevent members of the individual control blocks.

For LIO_WAIT, the sig argument isn't used.

As for some of the other AIO calls, a successful return from lio_listio only means that that call went well. It says nothing about the I/O operations, which you test for with aio_error.

Exercises

3.1. Modify the program in Section 3.2.2 as suggested by the last paragraph of that section.

3.2. Write the standard df command.

3.3. Modify the aupls command as suggested by the last paragraph of Section 3.6.2.

3.4. Modify the getcwdx function in Section 3.6.4 so it never changes the current directory.

3.5. Fix Problem 2 that's described at the start of Section 3.6.5.

3.6. Modify the aupls command from Section 3. 3.6.5 so it sorts the list by name.

3.7. Modify the aupls command from Section 3. 3.6.5 so it takes a -t option to sort by modified time and name.

3.8. Modify the aupls command from Section 3. 3.6.5 so it takes additional standard options (your choice).

3.9. Write a program to copy an entire tree of directories and files. It should have two arguments: the root of the tree (e.g., /usr/marc/book), and the root of the copy (e.g., /usr/marc/backup/book). Don't bother dealing with symbolic or hard links (i.e., it's OK to make multiple copies), and don't bother with preserving ownership, permissions, or times.

3.10. Same as Exercise 3.9, but, to the extent possible, preserve ownership, permissions, and times. Make "the extent possible" dependent on whether the command is running as superuser.

3.11. Same as Exercise 3.10, but preserve the symbolic and hard link structure.

3.12. Why is there no `lchmod` system call?

3.13. Write `access` as a function. You may use any system call except `access`.

3.14. Implement `readv` and `writev` (Section 2.15 using `lio_listio`. Are there any advantages to implementing them this way? Disadvantages?

3.15. Can you do 3.14 for `read`, `write`, `pread`, `pwrite`, `fsync`, `fdatasync`, `aio_read`, `aio_write`, and `aio_fsync`? If so, do it. Any advantages? Disadvantages?

3.16. List the pros and cons of checking for AIO completion by `aio_suspend`, a signal, a thread-start, or polling with `aio_error`. (Requires information that's in Chapters 5 and 9.)

This page intentionally left blank