Chapter 3 File Systems and the File Hierarchy

Concepts Covered

UNIX file systems and file hierarchies
Internal structure of a file system
Mounting
i-nodes and file attributes
The dirent structure
Manipulating directories and i-nodes
Creation of files by the kernel
Implementing ls, pwd, and du,
Traversing file hierarchies,
opendir, readdir, closedir, seekdir,
telldir, rewinddir, stat, lstat, fstat,
chmod, chown, creat, link, unlink,
unlinkat, readlink, umask, fnmatch
chgrp, chown, utime,getpwuid,
getgrgid, getpwnam, getgrnam,
rename, ntfw, fts_ open, fts_ read,
fts_ children, fts_ close.

3.1 Introduction

This chapter looks at UNIX file systems from the programmer's perspective. The primary objective is to be able to write programs that use the part of the UNIX API concerning the file system and its components. Of necessity, we will begin with an overview of what the file system is, and to a limited extent, how it is implemented. Although it is not necessary to understand how it is implemented to write programs that use it, a basic understanding of the typical implementation can help in understanding performance considerations and limitations.

3.2 File System Abstraction

A *file system* is an abstraction that supports the creation, deletion, and modification of files, and organization of files into directories. It also supports control of access to files and directories and manages the disk space accorded to it. We tend to use the phrase "file system" to refer to a hierarchical, tree-like structure whose internal nodes are directories and whose external nodes are non-directory-files (or perhaps empty directories), but a file system is actually the flat structure sitting on a linear storage device such as a disk partition. The layout of Linux, for example, is shown in Figure 3.1. This flat structure is completely hidden from the user, but not entirely from the programmer. The user sees this as a hierarchically organized collection of files and directories, which is more properly called the *directory hierarchy* or *file hierarchy*.

3.3 File System Mounting

Multiple storage devices are usually attached to a modern computer. Some operating systems treat the file systems on these devices as independent entities. In Microsoft's DOS, and systems derived from it, for example, each separate disk partition has a drive letter, and the file hierarchy on each separate drive or partition is separate from all others attached to the computer. In effect, DOS has multiple trees whose roots are drive letters. For example, a typical Windows machine may

have a directory `E:\users` on the "`E:`" drive and a directory "`C:\Temp`" on the "`C:`" drive but these directories are in two separate trees, not a single tree.

In UNIX there is a single file hierarchy. It is a tree if you think of the leaf nodes as filenames, but it is not a tree if you think of the leaf nodes as actual files, since a single file can have more than one name, existing as a directory entry in multiple directories, making the topology a directed acyclic graph[1]. We will take the liberty of referring to it as a tree, knowing that this is inaccurate.

In UNIX, every accessible file is in this single file hierarchy, no matter how many disks are attached. There is no such thing as the "C" drive" or "E" drive" in UNIX. This is because of the concept of *mounting*, which will be described in more detail later. In short, in UNIX, a file system may be mounted onto the single file hierarchy by attaching that file system's root to some directory in the hierarchy. It is like grafting a branch onto a tree. By mounting a file system onto the file hierarchy, the file system becomes a subtree of the hierarchy, making it possible to navigate into the file system from the rest of the file hierarchy. The `mount` command without arguments displays a list showing all of the file systems currently mounted on the file hierarchy. As an example, the output of the `mount` command could be:

```
/dev/mapper/root.vg-root.lv on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw,rootcontext="system_u:object_r:tmpfs_t:s0")
/dev/sda1 on /boot type ext3 (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
```

Each line is of the form,

file_system_name `on` *place_where_it_is_mounted* `type` *file_system_type other_options*

The sixth line states that there is a file system named `/dev/sda1` of type `ext3` that is mounted on the directory `/boot`. You may wonder about the meaning of the line

```
proc on /proc type proc (rw)
```

In a subsequent chapter we will explore the `/proc` file system, which is not a file system that manages disk space, but an interface to the kernel's memory. Further interpretation of this output will be delayed until after a discussion of file systems and mounting in greater depth. At this point, the significance of mounting is that different file systems can be, and usually are, part of a single conceptual file hierarchy, making it possible to partition a disk into separate file systems that all become part of a single file hierarchy.

3.4 Disk Partitions

In the early versions of UNIX, the disk was configured as a single partition with a single file system. As disks grew in size it became advantageous in operating system design to partition them into multiple logical devices that were actually distinct physical portions of the same disk. Partitioning a disk allowed for

[1] If you include symbolic links. it may even be cyclic

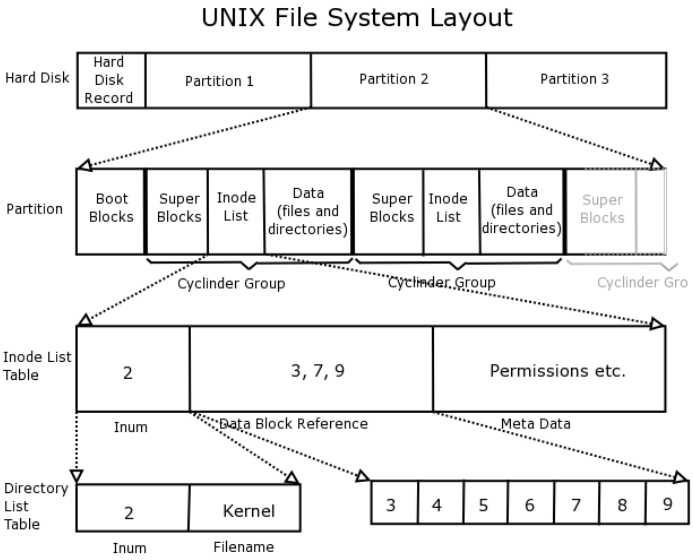

Figure 3.1: Linux file system layout, reproduced from "Linux Internals", by Simone Demblon and Sebastian Spitzner, Courtesy of The Shuttleworth Foundation.

- more control of security – different user groups could be placed into different partitions, and different mounting options could be used on separate partitions, so that some might be read only, and others might have different security options
- more efficient use of the disk – different partitions could use different block sizes and file size limits
- more efficient operation – shorter seek distances would improve disk access times
- improved back-up procedures – backups could be done on partitions, not disks, thereby making it possible to back-up different file systems at different intervals
- improved reliability – damage could be restricted to a single partition rather than the entire disk, and redundancy could be built in

The biggest disadvantage of partitioning a disk is that partitions can not be increased in size, so when they are created, if they are too small and fill up, the entire disk would need to be reorganized. There are other disadvantages of partitioning a disk, but they tend to be outweighed by the advantages.

Modern UNIX systems allow disks to be partitioned into two or more separate physical entities, each containing a distinct file system. The file systems in separate physical partitions are connected to each other by virtue of their being mounted on directories of the one and only directory hierarchy rooted at "/", but they are otherwise unrelated. Each separate file system has its own pool of free blocks, its own set of files, and its own i-node table.

3.5 UNIX File Systems

The different UNIX-like operating systems provide different file systems, each of which may be implemented in its own way. The implementation of the file system is not part of any UNIX

standard: there is no single implementation prescribed or proscribed in any standards document. Therefore, when reading about an implementation of the UNIX file system, you be aware that it is not the only way it is done.

The legacy UNIX file system is not used in many modern systems; modern implementations are more complex because they incorporate many enhancements to the original design. One reason for this is the fact that modern machines must be able to mount file systems of different types. For example, many UNIX systems allow users to mount *FAT*[2] and *NTFS* disk-based file systems, which do not follow the UNIX model. In Chapter 1 it was noted that a directory in UNIX is a file that consists of a list of directory entries, each of which contains the name of a file and its *i-number*, which serves as a pointer to the file's *i-node*. In a FAT system (persisting since early Windows operating systems), directories do not have this structure. UNIX kernels, if they are designed to mount such systems, must create a kernel object in memory to simulate the UNIX directory. Still more importantly, the UNIX kernel cannot hard-code system calls such as `read()` because the implementation of `read()` will depend on the file system. As a result, the actual machine code that is executed when these calls are invoked cannot be bound to the function name when the kernel is compiled.

This situation is analogous to the problem that is solved by pointers in a programming language or virtual functions in C++. When it is not known how much storage a data structure needs at compile time, instead of declaring it statically, the binding of the name to the storage is delayed until run-time by using a pointer instead. In C++, when classes are created with virtual functions, the function code that is executed as a result of a function call is not determined until run-time, another form of delayed binding. In this case, the solution is achieved opaquely through the use of pointers in the underlying implementation[3].

In modern UNIX systems, such as Linux, the implementation of the file system is achieved by dynamically binding the implementations of file system calls to functions that are hard-coded in the particular file system that is mounted, a form of delayed binding as well. How this is accomplished is not important right now; what matters is that modern UNIX file systems are *virtual file systems*, designed to handle many different types of underlying physical file systems. In fact, in Sun's variants of UNIX, from SunOS through Solaris, and in BSD (and FreeBSD), the concepts of i-node and i-number have been replaced by those of v-node and v-number, with the "v" standing for "virtual". Linux continues to use the term "i-node", and in these notes we will do the same.The *Linux Second Extended File System* (*Ext2*) is depicted in Figure 3.2.

The original Linux Virtual File System was developed by Chris Provenzano, and later rewritten by Linus Torvalds. The Linux Ext2 file system was developed in the mid 1990's by Rémy Card, Theodore Ts'o, and Stephen Tweedie. The next Linux file system was Ext3, which was developed by Stephen Tweedie and which differs from Ext2 only in that it contains *journaling*. Journaling is a way to maintain file system consistency in the event of hardware failures. A special journal file is used to record all of the actions that are supposed to be taken on the file system, such as creating and deleting files, changing their contents or attributes, and so on. In a journaling file system, this record can be used to recover the state of the file system without the lengthy task of examining every block and i-node. Ext2 and Ext3 are interchangeable – one can be converted to the other while the file system is mounted because the difference is the journaling. The Fourth Extended File System, Ext4, was released in 2008, mostly to improve performance. While Linux supports many types of file systems, the Ext2, Ext3, and Ext4 file systems are native to it and found on almost all

[2] *FAT* stands for *File Allocation Table*, and is the file system found on many Microsoft operating systems as well as other external storage devices such as memory cards.

[3] A special table called a *virtual dispatch table* is usually how virtual functions are implemented.

Linux systems.

Although there are many different UNIX file systems, most current implementations are built upon ideas from Dennis Ritchie's original implementation. Therefore, the implementations described in these notes should be thought of as generic implementations, meaning they do not actually exist in any one system but approximate many actual implementations. I will, as much as possible, describe how some Linux variant does things, since that is the most prevalent UNIX system that students use, and that is the one running on our host computer.

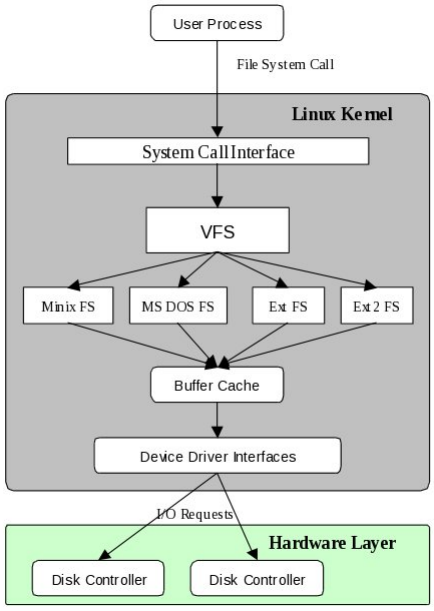

Figure 3.2: The Linux Second Extended File System (Ext2)

3.6 The Principal Components of a UNIX File System

Every file system has (at least) one *superblock* located at the beginning of its allocated storage. The superblock contains all of the information about how the file system is configured, such as block size, block address range, and mount status[4]. Copies of the superblock are almost always stored in several other places within a partition[5].

Each file system in UNIX has at least one table that identifies the files in it. The entries in this table are called *i-nodes* (pronounced "eye-nodes"), and the indices of the i-nodes in this i-node table are called *i-numbers*. The i-nodes contain the file attributes and a map indicating where the blocks of the file are located on the disk. Attributes include properties such as

- the owner,

[4]The superblock actually contains (on most UNIX systems) the block size, a pointer to the i-node table and free i-node list, a pointer to a structure specifying the type of file system, a device identifier for the block device, a structure with the allowed operations, the mount status, and other information as well.

[5]Originally there was a single superblock. In later versions of UNIX, a copy of the superblock was placed in every cylinder group in case of a disk crash.

- the group,
- the permissions allowed on the file and the file type,
- the number of links to the file
- the time of last modification,
- the time of last access,
- the time the attributes were last changed,
- the size in bytes of the file,
- the number of blocks used by the file, and
- the id of the device on which the file resides.

I-nodes and the tables that use them are important components of the UNIX file system. Modern file systems usually have multiple i-node tables, as described below.

Every file system separates the i-node tables from the data blocks. The data blocks are where file contents are stored.

Figure 3.1 depicts the structure and layout of a modern UNIX disk device with several file systems on it. Each successively lower layer of the figure is an enlargement of a structure in the preceding layer. You can see that in a modern file system, not only are there multiple superblocks in a single file system, but multiple i-node tables as well. This is an enhancement added for performance reasons. As disks grew in size, files whose blocks were on the outer edge of the disk became further away from the i-nodes that contained the block addresses and file status. Disk accesses to read or write the file required ever increasing latency caused by the disk seeks. By making several smaller tables, each in its own cylinder group, no file became too far away from the i-node table. The figure shows that the i-node in position 2 of the table usually points to the entry for the root directory file in the file system. We will return to this issue below.

3.6.1 Defining and Creating File Systems

Partitioning a disk divides the disk into logically distinct regions, often named with letters from the beginning of the alphabet, i.e., a, b, c, and so on. In UNIX, partitions are not necessarily disjoint. The "c" partition is almost always the entire disk[6], and typically does not have a file system. It is used by utilities that access the disk block by block. The "b" partition traditionally was reserved as the swapping store, i.e., the partition used for swapping; it did not have a file system written onto it. The innermost partition, "a", is where the kernel is installed and it is typically very small, since little else should be put in it. If a disk has a 100 GB storage capacity, you might make the first 1 GB partition a, the next 10 GB, b, the next 50GB d, the remainder, e, and the whole disk, c.

In order to create files in a partition, a file system must be created in that partition. Creating a file system includes doing the following:

[6]On Solaris hosts, it referred to the entire disk. In FreeBSD, it refers to entire "slices", which can be thought of as collections of partitions.

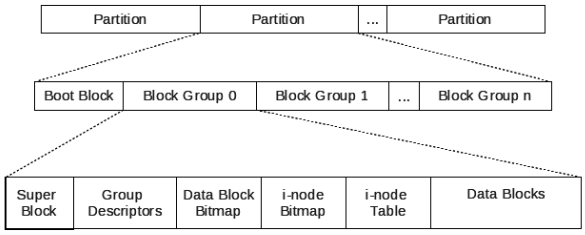

Figure 3.3: Partitions, block groups, and their structures in Ext2

- Dividing the partition into equal size logical blocks, typically anywhere from 1024 to 4096 bytes each, depending upon expected file size. The block size is fixed at file system creation time; it cannot be changed after that without rebuilding the file system. Block size is always a power of two. Larger block sizes result in more wasted disk space in internal fragmentation, whereas smaller sizes result in more disk activity and more disk waits. Larger blocks are appropriate for file systems expecting large files. In the file systems on my personal Linux host, the root file system uses 1024-byte blocks and the second partition, used for user data, uses 4096-byte blocks.

- Deciding how many alternate blocks are needed in each cylinder. (A cylinder is the set of all tracks that are accessible from one position of the disk head assembly. In other words, a cylinder is the set of tracks that are vertically aligned one on top of the other.) When a block becomes bad, it has to be removed from the file system. Alternate blocks are reserved to replace bad blocks.

  – In Linux Ext2, block groups take the place of cylinder groups. Whereas a cylinder group is a physical concept, tied to the geometry of the disk, a block group is a logical concept, independent of the disk geometry, because modern hard disk drives hide the geometry from the operating system. Figure 3.3 illustrates this.

- Deciding how many cylinders are in each *cylinder group*. Grouping cylinders is done for performance and reliability. A cylinder group is a collection of adjacent cylinders that are grouped together to localize information. The file system tries to allocate all data blocks for a given file from the same cylinder group if the file is small enough. Each cylinder group also keeps a copy of the superblock, as described below.

  – In Linux Ext2, each block group contains the following (see Figure 3.3):

    * a redundant copy of the file system's superblock,
    * a redundant copy of a table of block group descriptors; each block group descriptor contains information about the structure of a specific block group and a map of where everything in the group is located; the table is identical in all groups,
    * the block bitmap, which has a bit for each block in the group indicating whether it is free or in use,
    * an i-node bitmap, which has a bit for each i-node in the group indicating whether it is free or in use,
    * an i-node table for the i-nodes in this block group,

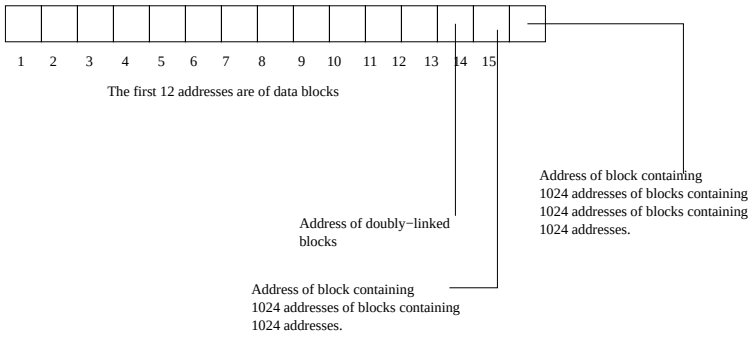

Figure 3.4: Addresses stored in an i-node

  * the actual data blocks.

- Deciding how many bytes to allocate to each i-node. They can be as small as 64 bytes. The i-nodes on the Linux Ext file systems are usually 128 bytes. Larger i-nodes means fewer i-nodes, which means fewer files. Smaller i-nodes allow more files.
- Dividing the cylinder group or partition, depending on the system, into three physical regions:
  - *The superblock.* This stores the map of how the disk is used as well as the file system parameters. In the Linux file system, the superblock contains information such as the block size in bits and bytes, the identifier of the physical device on which the superblock resides, various flags indicating whether it is read-only or locked, or how it is mounted, and queues of mounts waiting to be performed.
  - *The i-node area.* This is where used and free i-nodes are stored. The used and free i-nodes were traditionally arranged into two lists, the i-list and the free-list, with the start of each list stored in the superblock. That method of storage management is obsolete. Later versions of UNIX used a more efficient method[7], in which, when the file system is created, a fixed number of i-nodes was allocated within each cylinder group. This puts i-nodes closer to their data blocks, reducing the overall number of seeks[8].
  - *The data area.* This is where the data blocks are stored.

Many other things need to be done to the partition to make a file system. For example, because the disk rotates while it is reading data, there need to be gaps between blocks. How big should the gaps be? Because the disk head has to advance to a new cylinder to read the next block sometimes, and the disk is rotating while it advances, the next block should not be in the same sector. Which sector should be read next? Also, the superblock is usually replicated on the disk for reliability. How many times? Where should it be placed?

3.6.2 File Storage

The method of storing files in UNIX is flexible and reasonably efficient. Remember that each file has an i-node in an i-node table containing information such as the user-id of the owner, the group-id of

[7] In BSD they called this the Fast FileSystem,

[8] If the i-node is at the beginning of the disk and the data blocks are in the middle, then the head has to go back and forth, reading the address from the i-node, getting the data, going back to the i-node, etc. With the i-nodes near the data blocks, it takes a little more time to reach the i-node, but it is more than made up in the savings in data block access.

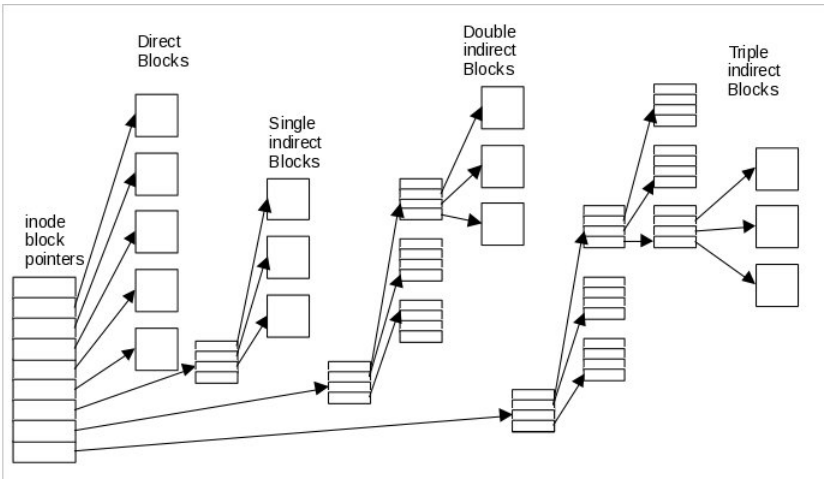

Figure 3.5: Allocation of blocks to a file in a UNIX File System

the file, the file size, permissions and so on. The i-node also contains an array 15 block addresses[9]. A block address is usually 4 bytes long.

In BSD, i-nodes contains an array of 15 block addresses, which are used as follows:

- For regular files, the first 12 block addresses in this array are the addresses of the first 12 blocks of the file. If the block size is 4096 bytes (4 KB), then a file with at most $12 \cdot 4 = 48$ KB can be accessed by one level of indirection.
- If a file is larger than 48 KB, then the 13^{th} address is the address of a *single indirect block*, which is a 4096-byte block used to store block addresses. Since a block address is 4 bytes, there are $4096/4 = 1024$ block addresses in this block. Since each of these 1024 blocks is 4096 bytes, using the first 12 addresses plus the 13^{th} address allows for addressing files whose size is at most 48 KB + $1024 \cdot 4096$ bytes, or 48 KB + 4 MB.
- For still larger files, the 14^{th} address is the address of a *double indirect block* that contains 1024 addresses of single indirect blocks, each of which contains 1024 device addresses. This accommodates files whose size does not exceed 48 KB + 4 MB + $1024 \cdot 4096$ KB , roughly 4 GB.
- The 15^{th} address is that of a *triple indirect block*, which, needless to say points to 1024 double indirect blocks and so on. It allows for $1024 \cdot 1024 \cdot 4096$ additional KB, or a maximum file size of roughly 4 terabytes.

Figure 3.4 shows how the block pointers are laid out in the i-node and Figure 3.5 illustrates the distribution of blocks that results. With this design, there can be a lot of disk seeking, because the different parts of a file can be very far away from each other. For most files though, the blocks tend to be physically contiguous and so actual I/O overhead is acceptable. For very large files requiring several addressing steps before accessing the disk blocks, overhead can be large. This is offset by the use of the kernel I/O buffering described in Chapter 2.

[9]The number of addresses varies. On early UNIX systems, it was 13. In 4.4BSD it is 15. In some implementations of Linux, it is 13 and in others, it is 15.

3.6.3 How the Kernel Creates Files

Suppose that the current working directory is `/home/sweiss/scratch`, and the command

```
$ gcc -o testprog testprog.c
```

is executed. Assuming that the program is compiled and linked, and that I have write and execute permission for the directory `/home/sweiss/scratch`, `gcc` will create the file named `testprog` in this directory. This section answers the question, "What steps are taken by the kernel to create the file `testprog`" on behalf of `gcc`? These steps can be outlined as follows:

1. The kernel creates an i-node for the file, if possible.
2. The kernel fills in the i-node with the file status.
3. The kernel allocates data blocks for the file and stores the file data in these blocks.
4. The kernel records the addresses of the data blocks in the i-node.
5. The kernel creates a directory entry in the `scratch` directory with the i-node number and file name `testprog`.

Each of these steps is explained below.

Creating the i-node

The very first step is to create an i-node for the file. To do this, the kernel must get a free i-node in the i-node table. If there isn't a free one, it must report the error and stop here. If this happens, the user will get a message that the file system is full. This is one reason to enable disk quotas on the system. A table of active i-nodes is kept in memory with a copy on disk as well.

Updating the i-node

Assume that the kernel obtained an i-node, and that it has index 47 in the i-node table. The kernel fills the i-node in with the owner, permissions, time of last modification, and so on. It then saves the i-number, 47, of this i-node. The i-node update takes place in the memory copy of the table, not the disk resident copy. This is updated periodically by the kernel.

Allocating Data Blocks

The next step is to store the file's data. To do this the kernel must acquire the right number of blocks. As `gcc` runs, it is generating the file, writing it in pieces at a time. Because the kernel does output buffering, the file is being stored in kernel buffers, which are not written to disk until the buffers are flushed. If the file is small, all of it will fit in memory and the kernel will know how many disk blocks are needed for it. If the file is very large, the kernel may start allocating blocks before it knows the file's actual size. Assuming that the file system has storage for it, it will first allocate direct blocks. If the file is larger than the number of bytes that can fill all of the direct blocks, the kernel allocates single-indirect blocks as needed. If it is larger than the amount of storage they can provide, it starts allocating double indirect blocks. It continues this procedure, using a triple indirect block if not even all of the double indirect blocks will suffice.

Recording Locations of Data Blocks

The kernel records the order and location of the data blocks in the i-node. Some file systems also have file maps for faster access to the files. Whatever data structures are used to record the block locations, these must be updated at this point.

Recording the File Name in the Directory

If all of the preceding steps were successful, then the kernel will create a new entry in the current working directory consisting of the pair (47, `testprog`), because 47 is the i-number and `testprog` is the name.

3.6.4 How the Kernel Accesses Files

When a user enters a command that has to open a file for reading, what actually happens? For example, when the command

```
$ cat myfile
```

is executed by the shell, what steps are taken and by which programs? The first step is that the shell parses the command and determines that the `cat` command is the executable and that it has a single file argument named `myfile`. It will arrange for the name `myfile` to be placed into the `argv` array that `cat` will see in its `main()` parameter list:

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

`cat` will ask the kernel to open the file for reading via the `open()` system call. The `open()` system call tries to find the i-number of the file whose name is passed to it. To do so, it has to resolve the pathname, which is a fact that we overlook for now. But just to give you an idea of the complexity, what if the filename were

```
../../shared/temp1/coursework/workspace/myfile
```

This would be a little more challenging than if it were in the working directory. For now suppose that `open()` figures out what directory to open and then searches through that directory for the name "`myfile`". When it finds it, it obtains the i-number for it. Once it has the i-number, it can retrieve the i-node information from its i-node table. If the file is already open, the i-node will be in the active i-node table in kernel memory. If not, it is retrieved from the i-node table on disk, and the i-node is copied into the active i-node table.

Once the kernel has a copy of the i-node in its memory space, it can access the i-node's information. One of the first things it has to do is check that the permission bits allow the `cat` program to access the file. The `cat` program runs with an effective user-id of the user who invoked it. If this user does not have appropriate permission to access this file in the given directory, the `open()` call will return –1 and set the static variable `errno` to `EPERM`. which is the error code that system calls assign to `errno` when they fail because of a permission problem.

If the access is allowed, the `open()` call will have returned a file descriptor for the file, and that descriptor will be pointing to a data structure that, among other things, has a pointer to the i-node for the file. The `cat` command, as it makes successive calls to read the file, is essentially asking the kernel to access the i-node and get the data from the file's data blocks. The kernel uses the i-node to determine which blocks to access. From the file size, the kernel can determine whether the data is entirely in direct blocks (e.g., with a block size of 4096 bytes and 12 direct blocks, at most 48 KB can be stored in them) or whether to read the first single indirect block. If the file is stored in indirect blocks, it will be slower to access, because most of the data blocks will require three disk accesses or even four. If the block size were 8192 bytes, then files of up to 96 KB would be accessed more quickly. The kernel also knows from the active i-node, whether the file's blocks are in a system buffer and can be accessed without disk I/O.

3.7 The `ls` Command

Equipped with a bit more understanding of what a file system does and how it does it, we can try a small exercise by writing a program that uses the file system's programming interface. The `ls` command is a simple enough start.

The `ls` command can be given a list of filenames of all kinds (e.g., regular files, special files, and directories) as arguments:

```
ls [ options ] FILE FILE ...
```

where `FILE`, `FILE`, ... are filenames, whether they are regular files, special files, symbolic links, or directories. Of course, as with almost all commands, the filename arguments can be arbitrary pathnames. If an argument is a directory, it displays the file links contained in that directory, and if the appropriate option is given, the file attributes as well. If it is not a directory, `ls` displays the filename, and if the `-l` option is specified, the file attributes as well. There are many options that change the default behavior of `ls`. You can read the man page for the full list of them.

Summarizing, `ls` does two different things, depending on whether the argument is a directory or a non-directory file.

- When the argument is a directory, `ls` displays its contents.
- When the argument is not a directory, `ls` displays its name.

In either case, the options might require that some subset of its attributes are displayed as well, in a specified order.

Somehow `ls` can determine whether an argument is a directory or not. To implement `ls`, we have to figure out

1. how to determine if a file name is that of a directory;
2. how to list the contents of a directory; and
3. how to list the attributes of a file.

When the argument to `ls` is a non-directory file, it may have to obtain the attributes of the file and display them. When the argument is a directory, say `dir`, `ls` has to do something like

```
open the directory dir
while not all directory entries in dir have been read
    read the entry
    optionally display information about the entry
close the directory
```

The questions are, how can we open, read, and close a directory, and how can we obtain information about file attributes. Fortunately none of these tasks are particularly difficult, owing to the simplicity of the file system interface.

3.8 The Directory Interface

Recall that, as far as the kernel is concerned, regular files are just sequences of bytes. Directories are like regular files except that

1. They are never empty:

   Directories are not empty because every directory has two unique entries, "." and ".." that refer respectively to the directory itself and to the parent directory. These are created when the directory is created.

2. They cannot be written to by unprivileged programs:

   They can only be modified by very specific system calls, unlike regular files. In contrast, anyone with appropriate permission may read the contents of a directory. Some commands that operate on regular files can be applied to directories, but with unexpected results.

3. They have a specific structure:

   A directory is a file that contains a collection of (name, i-number) pairs. In other words, a directory is a table of records. You can think of it as a hash table, since it is accessed with a name in order to retrieve the i-number.

Some commands that read files also read directories, but the results are not useful. For example, on some systems, the `cat` command and the `od` command may display the contents of a directory as a stream of bytes, but because a directory is not a text file, the output of `cat` will look garbled. The output of `od` may look normal. Usually, these commands, like the `more` command, check whether their argument is a directory, and if so, refuse to execute, displaying instead an error message:

```
$ more projects
*** projects: Is a directory ***
```

The `more` command refused to display `projects` because it is a directory and a directory displayed as a sequence of characters would not be meaningful. The output of `od` may reveal the structure of the directory to you if you really examine it carefully. The `-c` option to `od` displays bytes that can

be converted to characters as characters. On Linux, neither `od`, nor `cat`, nor `more` will display the contents of a directory.

If you are wondering what system calls might work with directories, the `open()`, `read()`, and `close()` system calls will act on directories because they do not modify the directory, but because a directory has a specific structure, there is not much point to using them. They are not intended to be used with directories. It is better to use the specific system calls or library functions designed to open, read, modify, and close directories.

If you use a man page search to look for man pages that refer to the word "directory" you will find a long list of them. It is best to filter the output of the search, using something like

```
$ man -k directory | grep read
```

which will display just a few entries. You might see, among other things

```
readdir (2) - read directory entry
readdir (3) - read a directory
readdir (3p)- read a directory
readdir_r [readdir] (3p)- read a directory
seekdir (3) - set the position of the next
readdir()     call in the directory stream
```

Different systems will deliver slightly different sets of man pages, depending on which version of UNIX they are running. All should list the `readdir()` system call in Section 2, and most should list a call named `readdir()` in Section 3 or 3P or both. On almost all systems, if you read the man page for `readdir()` system call in Section 2, it will advise you not to use this system call and instead to use the POSIX library function of the same name. It will refer you to the `readdir()` C library function whose man page is in Section 3 or 3P. You might need to use a lowercase p in your man command, as in

```
$man 3p readdir
```

Because of the wide variation among the different systems, we cannot know exactly what you will see, but all pages should have, at a minimum the following:

```
SYNOPSIS
     #include <sys/types.h>
     #include <dirent.h>

     struct dirent *readdir(DIR *dir);
DESCRIPTION
     The  readdir() function returns a pointer to a dirent
     structure representing the next directory entry in the
     directory stream pointed to by dir. It returns NULL on
     reaching the end-of-file or if an error occurred.
```

This tells us that `readdir()` requires an argument of type `DIR*` and returns a pointer to a `dirent` structure. The rest of the man page may contain a definition of this `dirent` structure, but it may not, depending on the system. If it does not, then you need to use the strategy described earlier for finding it. It will not explain what a `DIR*` is or where it is defined. That will almost certainly require further research. To be clear, in case the page does not contain the `dirent` definition, you should look in the `SEE ALSO` section for man pages that might give us more information; the list might look like the following:

```
SEE ALSO
       read(2), closedir(3), dirfd(3), ftw(3), opendir(3),
       rewinddir(3), scandir(3), seekdir(3), telldir(3),
       feature_test_macros(7)
```

Among the pages listed above, the most likely candidate would be the man page for `opendir()`; this is the function that most likely is the first one to call to do any directory processing. If that page does not have the definition, then we would have to read the man page for `<dirent.h>` or read the header file itself.

One way or another, you will discover that the `dirent` structure is defined as follows:

```
struct dirent {
       ino_t          d_ino;       /* inode number */
       off_t          d_off;       /* offset to the next dirent */
       unsigned short d_reclen;    /* length of this record */
       unsigned char  d_type;      /* type of file; not supported
                                      by all system types */
       char           d_name[256]; /* filename */
};
```

But you will find, whether in the Section 3 or 3P man page for `readdir()` or in the page for the `<dirent.h>` header file, a *caveat* such as the following:

> According to POSIX, the `dirent` structure contains a field `char d_name[]` of unspecified size, with at most `NAME_MAX` characters preceding the terminating null byte. POSIX.1-2001 also documents the field `ino_t d_ino` as an XSI extension. Use of other fields will harm the portability of your programs.

Translation: the only two members that are guaranteed to be present in a `dirent` structure are the `d_ino` and the `d_name` members, and the `d_name` member is not necessarily a fixed length string; POSIX only guarantees that it is a `NULL`-terminated string whose length is at most `NAME_MAX` characters. Therefore, whatever we do, we should not use those other members, and we should make sure that any variables that store a name are large enough to store `NAME_MAX` characters. We will return to the question of how to use `NAME_MAX` in a program.

We need to figure out how to use `readdir()`, what a `DIR` is, and how we get a `DIR*` to use in `readdir()`. The man page for `readdir()` basically says that successive calls to `readdir()` with the same *directory stream* pointer return successive entries in the directory pointed to, and that

when all entries have been accessed, a `NULL` pointer is returned. This is very much like how the `getutent()` function worked with `utmp` structures, and that function required that the library be initialized by a call to `setutent()`. It would be a good guess that the `opendir()` function listed in the `SEE ALSO` section performs this initialization for `readdir()`.

The man page for `opendir()` begins as follows:

```
SYNOPSIS
       #include <sys/types.h>
       #include <dirent.h>

       DIR *opendir(const char *name);

DESCRIPTION
      The opendir() function opens a directory stream corres-
      ponding to the directory name, and returns a pointer to the
      directory stream. The stream is positioned at the first
      entry in the directory.

RETURN VALUE
      The opendir() function returns a pointer to the directory
      stream. On error, NULL is returned, and errno is set
      appropriately.
```

In other words, given the pathname of a directory as a string, `opendir()` returns a pointer to something called a `DIR` that the documentation refers to as a *directory stream*. Names in uppercase letters are usually macros, defined in a header file with a "`#define`" directive, and the `DIR` type may be a macro. It could be a `typedef` as well. We do not need to know what a `DIR` is to open a directory and read from it, so for now we will move on.

In the same way that correct use of `getutent()` requires that we call `endutent()` to clean up after accessing all `utmp` records, correct use of `readdir()` requires that we clean up afterwards. The `SEE ALSO` section contains a reference to a `closedir()` function. If we look at its man page we will see that `closedir()`, given a pointer to a `DIR`, closes the connection to the directory, which is exactly what we must do when `readdir()` has returned a `NULL` pointer.

From the man pages for `readdir()` (in 3 and 3P), `opendir()`, and `dirent.h`, we can piece together how the directory reading interface works. When a directory is opened using `openddir()`, a static variable points to the first entry in the directory. The `readdir()` call reads the entry pointed to by this hidden variable, and increments the pointer so that it points to the next entry. Thus, successive calls iterate through the directory. If at any time, the program needs to know which entry it is about to read, `telldir()` :

```
#include <dirent.h>
long telldir(DIR *dirp);
```

returns its index, which is an integer offset from the beginning of the directory stream. If the program needs to start all over again without closing the directory, `rewinddir()` will do that, and `seekdir()` will move the pointer to a specified index:

```
#include <dirent.h>
void seekdir(DIR *dirp, long offset);
```

The offset returned by `telldir()` can be passed to `seekdir()`.

Although we do not need to know the structure of a `DIR` to use these calls, curiosity beckons us. What you will discover is that `DIR` is not a macro, but a typedef:

```
typedef struct __dirstream DIR;
```

and that there is no definition of the `struct __dirstream` in any exposed header files in the system. This is because `__dirstream` is an incomplete type. An incomplete type is a type that describes an object but lacks the information needed to determine its size. Each implementation of UNIX must define it, and is free to define it as it chooses, but it need not expose that implementation[10] in any header files. The `<dirent.h>` header file declares the `struct __dirstream` and makes `DIR` equivalent to it, but does not define its members. This gives programmers the ability to declare objects of type `DIR*`, but not the ability to access the members of a `DIR` object.

Our curiosity satisfied, we set this aside and tackle the `NAME_MAX` problem that arose earlier. There is no man page search that can tell you where a constant such as `NAME_MAX` might be defined, i.e. which header file needs to be included in a program to access its definition. One can do a search for that string in all of the header files on the system, but then one has to know all of the places containing header files. Such a search will probably fail in `/usr/include` for example, because `NAME_MAX` is a system-dependent value contained someplace where implementation-dependent header files are stored. One can do an exhaustive recursive `grep` for " `NAME_MAX` " (the spaces are important) and one will find it:

```
$ grep -R " NAME_MAX " /usr/* 2>/dev/null
```

In `bash`, the redirect `2>/dev/null` throws away all messages sent to standard error. The output of this command may look something like

```
/usr/include/linux/limits.h:#define NAME_MAX         255
/* # chars in a file name */
/usr/include/glib-1.2/glib.h:#    define NAME_MAX 255
```

followed by lines from other files that may or may not define the macro.

`NAME_MAX` is a system limit in UNIX. It specifies the maximum number of characters in a file name. As such it is one of many such constants defined in the header file `<limits.h>`. This fact is not as easily discovered as some of the others so far. One must include this header file to use the constant. In the next chapter, we will cover how to discover and use system limits in programs. For the moment, we use just a few.

We have enough knowledge now to make an attempt at writing `ls`.

[10] If you write a program that references a `__dirstream` object, the compiler will give an error that its size is unknown. If you instead declare a `__dirstream*` pointer, the program will compile, because the pointer's size is known. You will not be able to dereference this pointer and use what it points to because your program does not have an implementation of it. But it is implemented in the libraries that use it. This is an example of information hiding in C.

3.9 Implementing `ls`

The first attempt at writing `ls` will handle multiple command line arguments and not much more. For each argument, if it is a directory, it will display its contents, in no particular order. One problem is solved for us by the `opendir()` call; the call will fail, returning a -1 if the argument is not a directory. This will be our test for whether or not a filename is that of a directory.

```
1  Listing: ls.c Version 1
2  #include  <stdio.h>
3  #include  <sys/types.h>
4  #include  <dirent.h>
5
6  #define   HERE      "."
7  void ls(char []);
8
9  int main(int argc, char *argv[])
10 {
11     int i = 1;
12
13     if ( 1 == argc )     // no arguments; use .
14         ls( HERE );
15     else
16        // for each command line argument, display contents
17         while ( i < argc ){
18             printf("%s:\n", argv[i] );
19             ls( argv[i] );
20             i++;
21         }
22 }
23
24 // List the contents of the directory named dirname
25 // Uses opendir to check whether argument is a directory
26 // Doesn't check if argument is or "." or ".."
27 void ls( char dirname[] )
28 {
29     DIR             *dir_ptr;     // directory stream
30     struct dirent   *direntp;     // hold one entry
31
32     if ( ( dir_ptr = opendir( dirname ) ) == NULL )
33         // Could not open -- maybe it was not a directory
34         fprintf(stderr,"Cannot open %s\n", dirname);
35     else
36     {
37         while ( ( direntp = readdir( dir_ptr ) ) != NULL )
38             printf("%s\n", direntp->d_name );
39         closedir(dir_ptr);
40     }
```

```
41  }
```

Comments

- The real work is done by the `ls()` function, which will print an error if the file is not a directory. It discovers this when it tries to open the directory using the `opendir()` call. This call will return a `NULL` pointer if it cannot open the directory or if it is not a directory. It sets `errno` to a specific value to indicate the cause of the error, but we ignore `errno` in this version.
- This version lists all files, including dot-files.
- It does not sort the files, which the real `ls` does.
- It does not put the output into columns, which the real `ls` does.
- It does not handle any command-line options.
- It does not display information about the three special bits of the file.

This was just a warm-up exercise. Now we know some of the problems that we have to address. We write a second version, trying to solve some, but not all, of the problems. We also add the ability to display long listings.

3.10 A Second Version of ls

3.10.1 Adding the -l Option to ls

The `-l` option to `ls` displays file attributes. The set of displayed attributes for files that are not device files is defined in the table below.

Field Name	Description
mode	A ten character field in which the first character is a single letter that denotes the file type. A "-" means it is a regular file; a "d" means it is a directory. There are other types as well. This was described in Chapter 1. The remaining nine characters define the file mode, also described in Chapter 1.
number of links	The number of names this file has in all directories combined. A single file may have entries in multiple directories, possibly with the same name or with different names.
owner name	The user-name of the user who owns this file.
group name	The name of the group to which this file belongs. If the group does not have a name its group-id might appear instead.
size in bytes	The actual number of data bytes in the file, not the number of bytes in the blocks allocated to it, except that if the file is a directory, it is the number of bytes in the blocks allocated to the directory and is therefore a multiple of the block size.

Field Name	Description
date of last modification	The date and time that the file was modified last. If the file is relatively new, it shows month, day, time, but if it is older it shows month, day , year. The definition of new varies, depending on which standard the implementation adheres to. In POSIX compliant ls, a file is recent if the date is within the past 6 months. Also, the exact date format can be changed with various options.
file name	The name of the file in the directory.

If a file is a symbolic link, then the default behavior of `ls` is to display this information about the link itself, not the file that it references. To display the referenced file's attributes, `ls` must be given the `-L` option (on most systems) in addition to the `-l` option.

The question is, how does the `ls` command access this information, which we know is in the i-nodes associated to the file? We need a function that accesses the information in an i-node. Once again, the way to find out is by searching the man pages. This time it is likely to be difficult to find the information. Searching for man pages referring to i-nodes will come up empty. We can try searching for a piece of information from the i-node, such as mode, size, or date of last modification, but these terms are too common and might produce long lists. The key is that the information in an i-node is called the file *status*. Searching for status will succeed:

```
$ man -k status | grep file
fileno [ferror]      (3)  - check and reset stream status
fstat                (3p) - get file status
fstat [stat]         (2)  - get file status
ifcfg-ppp0 [pppoe]   (5)  - Configuration file used by adsl-start(8), adsl-stop(8),
                            adsl-status(8) and adsl-connect(8)
lam_rfstate          (2)  - Report status of remote LAM file descriptors
lstat [stat]         (2)  - get file status
stat                 (1)  - display file or filesystem status
stat                 (2)  - get file status
stat                 (3p) - get file status
```

The returned list may be different than this, but it will contain the `stat` family of calls: `stat()`, `lstat()`, and `fstat()`, probably with two pages for `stat()`, one in Section (2) and another in (3) or (3P). The `stat` (3P) man page is the POSIX version. The man page for `stat` (2) begins with:

```
NAME
     stat, lstat, fstat - get file status

SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>

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

```
     int fstat(int fildes, struct stat *buf);

DESCRIPTION

     These functions return information about the specified
     file. You do not need any access rights to the file to get
     this information but you need search rights to all
     directories named in the path leading to the file.

     stat stats the file pointed to by file_name and fills in
     buf.

     lstat is identical to stat, except in the case of a
     symbolic link, where the link itself is stat-ed, not the
     file that it refers to.

     fstat is identical to stat, only the open file pointed
     to by filedes (as returned by open(2)) is stat-ed in place
     of file_name.

     They all return a stat structure which has the following
     fields:

    mode_t   st_mode;   /* File mode (see mknod(2)) */
    ino_t    st_ino;    /* Inode number */
    dev_t    st_dev;    /* ID of device containing a */
                        directory entry for this file */
    dev_t    st_rdev;   /* ID of device */
                        /* This entry is defined only for */
                        /* char special or block special files */
    nlink_t  st_nlink;  /* Number of links */
    uid_t    st_uid;    /* User ID of the file's owner */
    gid_t    st_gid;    /* Group ID of the file's group */
    off_t    st_size;   /* File size in bytes */
    time_t   st_atime;  /* Time of last access */
    time_t   st_mtime;  /* Time of last data modification */
    time_t   st_ctime;  /* Time of last file status change */
                        /* Times measured in seconds since */
                        /* 00:00:00 UTC, Jan. 1, 1970 */
    long     st_blksize;/* Preferred I/O block size */
    blkcnt_t st_blocks; /* Number of 512 byte blocks allocated*/
```

In short, we pass the `stat()` or `lstat()` function the pathname to the file and it fills in the `stat` structure pointed to by the second argument. If the pathname is not absolute, `stat()` and `lstat()` treat it as relative to the working directory. This is important, because if we pass it just a filename, the working directory of the process must be the one containing that filename. If a file is a symbolic link, `lstat()` displays information about the link itself, rather than the file to which it points, so if we want our program to behave like the real `ls`, we should use `lstat()` instead of `stat()`.

There is more information returned by a call to one of the `stat()` functions than is displayed by `ls -l`, so we have to select which members of the `stat` structure we need to display. If you were wondering, the POSIX description of the structure in Section (3P) is similar[11]. It will point out

[11] In fact, the POSIX page now contains example source code that solves the ls problem.

that some of the members of the `stat` structure are required and others are not. All of the members that we need are in the POSIX definition of this structure.

Much of the work is writing a function that, when given a filename, displays a line of output in the form of `ls -l`; therefore we concentrate on that function as a start. Initially we will create a function that displays the individual pieces of information one per line. Once we have that working, we can format it to fit on a line. The "one-per-line" version will be named `print_file_status()`. We can create a driver program that simply calls this function, using its command line argument as the file that we want to "stat". Since we already learned how to display time in Chapter 2, we can reuse that function here. A first pass at this function would be something like the following:

```
void print_file_status(char *fname, struct stat *buf)
{
    printf(" mode: %o\n", buf->st_mode);      // type + mode
    printf("links: %d\n", buf->st_nlink);     // # links
    printf(" user: %d\n", buf->st_uid);       // user id
    printf("group: %d\n", buf->st_gid);       // group id
    printf(" size: %d\n", buf->st_size);      // file size
    printf("mtime: %s\n",
            time2str(buf->st_mtime));         // last modified
    printf(" name: %s\n", fname );            // filename
}
```

Here, `time2str()` is a simple function that converts the `time_t` into a string without the "day of week" name at the start. It will be replaced by something else later. In this first attempt, the mode is numeric, the user and group are displayed as user and group-ids instead of with actual names. We will take each line one at a time and refine it so that it displays the information in the proper form. First we will fix up the mode.

3.10.2 Converting File Mode to String Format

The file mode is stored in the `st_mode` member of the `stat` structure as a 16-bit quantity. The 16 bits are used for different purposes.

The first four bits are reserved for the file type, such as regular file or directory. The next three bits are the special bits: the set-user-id bit, the set-group-id bit, and the sticky bit. We will get to these afterwards. The next nine bits are three sets of three bits each. Each set has a read, write, and execute bit. The three sets of bits are the user, group, and others sets of bits. A 1-bit means the permission or property is on, and a 0-bit, that it is off. With this in mind, it is easy to write code to convert this 16-bit quantity into a ten-character mode string using bit masks.

Other than the file type, which is stored in bits 12 to 15, all other flags are single bits in `st_mode`. In fact, UNIX provides all of the masks we need, in the file `<sys/stat.h>`. These masks are standardized in POSIX and are described in one of the `stat` man pages. The header file `<sys/stat.h>` has the definitions, which are replicated here. The single-bit masks are

```
S_ISUID     0004000    set UID bit
S_ISGID     0002000    set-group-ID bit (see below)
S_ISVTX     0001000    sticky bit (see below)
S_IRWXU     00700      mask for file owner permissions
S_IRUSR     00400      owner has read permission
S_IWUSR     00200      owner has write permission
S_IXUSR     00100      owner has execute permission
S_IRWXG     00070      mask for group permissions
S_IRGRP     00040      group has read permission
S_IWGRP     00020      group has write permission
S_IXGRP     00010      group has execute permission
S_IRWXO     00007      mask for permissions for others
S_IROTH     00004      others have read permission
S_IWOTH     00002      others have write permission
S_IXOTH     00001      others have execute permission
```

and the masks for extracting file type are

```
#define    S_IFMT      0170000   /* type of file */
#define    S_IFREG     0100000   /* regular */
#define    S_IFDIR     0040000   /* directory */
#define    S_IFBLK     0060000   /* block special */
#define    S_IFCHR     0020000   /* character special */
#define    S_IFIFO     0010000   /* FIFO */
```

POSIX defines macros for testing the file type, which are much easier than using the masks:

```
S_ISREG(m)     is it a regular file?
S_ISDIR(m)     directory?
S_ISCHR(m)     character device?
S_ISBLK(m)     block device?
S_ISFIFO(m)    FIFO (named pipe)?
S_ISLNK(m)     symbolic link?
S_ISSOCK(m)    socket?
```

We can use these masks to write a function to convert the numeric mode to the string that is displayed by ls. For now it does not handle the setuid, setgid, and sticky bits. Before incorporating them into the string, we should understand what they do and how they can be displayed in the mode string. After all, the string has ten characters and we need all ten – 1 for type, and 3 sets of 3 for permissions – so where could we display the values of the setuid, setgid, and sticky bits anyway?

The following function, `mode2str()`, given the mode as a 16-bit integer, fills the character string `str` with a permission string in the standard format.

```
void mode2str( int mode, char str[] )
{
    strcpy( str, "----------" );

    if ( S_ISDIR(mode) )        str[0] = 'd';    // directory?
    else if ( S_ISCHR(mode) )   str[0] = 'c';    // char devices
    else if ( S_ISBLK(mode) )   str[0] = 'b';    // block device
    else if ( S_ISLNK(mode) )   str[0] = 'l';    // symbolic link
    else if ( S_ISFIFO(mode))   str[0] = 'p';    // Named pipe (FIFO)
    else if ( S_ISSOCK(mode))   str[0] = 's';    // socket

    if ( mode & S_IRUSR ) str[1] = 'r';     // 3 bits for user
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';     // 3 bits for group
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';     // 3 bits for other
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';
}
```

3.10.3 Converting User/Group ID to Strings

The next step is to convert the user-id and the group-id to names. We need a function that is given a user-id and returns the name of the person with that user-id, and a function that is given the group-id and returns the name of the group. How do we find such a function? We can try man page searches using keywords such as username, name, user, and so on. After a few tries like this, you could hone it down to something like

```
$man -k name | grep user
```

You will come up with a not-too-long list that includes the following prospect:

```
getpwnam (3p) - search user database for a name
```

If we look at this man page, we will discover that, while `getpwnam()` is not what we want, `getpwuid()` is. Both of these functions access the password database, regardless of where the actual file is located, i.e., even if it is the network database, and return a `passwd` structure. The function `getpwuid()` accesses the password database by user-id and `getpwnam()` accesses it by username. We have a user-id, but we need the name, so we want the `getpwuid()` function. It is given a user-id as an argument and it returns a pointer to a `passwd` structure, which is defined in `/usr/include/pwd.h` as

```
struct passwd {
        char  *pw_name;       /* Username */
        char  *pw_passwd;     /* Password */
        _uid_t pw_uid;        /*  User ID */
        _gid_t pw_gid;        /*  Group ID */
        char  *pw_gecos;      /* Real name */
        char  *pw_dir;        /* Home directory */
        char  *pw_shell;      /* Shell program */
}
```

The pointer returned by `getpwuid()` can be dereferenced to access an individual member of the `passwd` structure, provided that the pointer is not `NULL`. The man page for `getpwuid()` notes that the returned pointer will be `NULL` if there was no matching entry in the password database. The application must check for this.

You may wonder why there may not be a matching name. It sometimes happens that a user account is deleted from a UNIX system, but some files created by that user are left behind. For example, the `/tmp` directory is usually world-writable, and anyone can put files there. If the deleted user account had created files there, the system administrator probably did not delete them, since searching through the entire file system for traces of a user's files is resource-consuming. If `ls` were trying to list the contents of `/tmp`, it would not be able to display that username.

It does not end there. Suppose that someone else had hard links to the deleted user's files. Even if the administrator deleted the "original" files created by that user, the actual files still exist because their link counts are not zero. The user-id of the owner of the file is stored in the i-node of the file, not the directory entry. So, for example, if George linked to a file owned by Sherry and Sherry's account was deleted, then George's links are still owned by Sherry. The administrator cannot rightly delete these links if George needs them, and a listing of George's files would fail to display a username.

It is also possible that the user-id does correspond to a legitimate username, but the username and user-id are defined in an NIS map from a different domain than the one to which the host belongs. For example, if a file system is remotely mounted from a different NIS domain, then the owners of the files will have user-ids and names from that domain, and the `getpwuid()` function will not be able to find the username. As a concrete example, suppose a host named `earth` is in the `geography` domain, but I have an account on `earth` in that domain (say `sweiss@geography`) and I, for a while, was remotely mounting `earth`'s file system on my desktop machine, which lived in the `csci` domain. All of the files that I saw belonged to me, `sweiss`, but it was not the me I knew; it was `sweiss@geography`, so the user-id was different than the one in the `csci` NIS maps. As a result, `getpwuid()` would fail to find that user-id in the `csci` maps. So you see that `getpwuid()` can fail for that reason as well.

Since there may not be an entry in the passwd file for a given user, there may not be a user-name associated with the file. There is still a user-id, but no user-name. In this case, the best that ls can do is to print the user-id.

This is not the end of it. It is possible that `ls` finds a name but it is the wrong name. Suppose that the administrator reused the deleted user-id for a new user, not knowing that there were still files from the old user. When this happens, `getpwuid()` will find the user-id in the password database, but it will be wrong. It will now be associated to a different person, an impostor, if you like, for the old owner. `ls` will display the new user-name as the owner of the files, as the old account rolls

over in its cybergrave, and as far as UNIX is concerned, the new user is now the inadvertent owner. There is nothing we can do to prevent this.

What about group names? It turns out that a similar database, similar functions, and similar problems exist for groups. The `/etc/group` file lists all of the groups in the system, together with the users who are in each group. The file has an entry for each group consisting of the group name, the group password, the group-id and a comma-separated list of user-names, such as

```
root::9:root
other::1:
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
staff::4:tbw,snw
```

From the list above you can see that `root` belongs to more than one group. In general, users can belong to more than one group. The primary group of a user is the group specified in the password file entry for that user. When a user creates a file, the user's primary group is made the group of the file. The owner of a file can change the group of a file with the `chgrp` command.

The `getgrgid()` system call, given a group-id, searches the list of groups and returns a pointer to the group structure of the group whose group-id matches, or `NULL` if no such group is found. Similarly, the `getgrnam()` function searches the group database by name and returns a pointer to the group structure that matches that name, or `NULL`. The group structure is defined in `/usr/include/grp.h` as

```
struct group {
    char   *gr_name;        /* group name */
    char   *gr_passwd;      /* group password */
    gid_t   gr_gid;         /* group ID */
    char  **gr_mem;         /* group members */
};
```

It is easy to write a function to convert group-ids to group-names, provided that such groups have names. The same arguments about passwords apply to groups; it is possible that the group-id will not resolve to a correct group name.

3.10.4 Formatting Time

In Chapter 2, we used `ctime()` to format time values. Here we will use `strftime()` and `localtime()` together, so that our program can behave more closely to the real `ls` program. The real `ls` program formats times differently depending upon the user's locale. In addition, for files whose time of last modification is not "recent", the format is different. The definition of "recent" is six months or less. Of course six months is different at different times of the year, so this is an approximate definition. We will use the financial sector's definition of six months – 182 days.

For files that are not recent, ls displays the month name, the day of the month, and the year. For recent files, it displays the month, the day of the month, and the hour and minute. The exact format

depends upon the locale settings. We will put this logic into a function named `get_date_no_day()`, since it never displays the day of the week. this function is given a time_t value and produces a pointer to a statically allocated string as its return value. If that value is not copied before the function is called again, it will be overwritten. The function's logic is

1. Check if the file is recent by comparing current_time - given_time to the number of seconds in 182 days.
2. Create a struct `tm` from the given_time value.
3. If the file is not recent, format it in the format `MMM DD YYYY`.
4. If the file is recent, first try to format it using the user's locale settings.
5. If that fails, format it using the format `MMM DD HH:MM`

The function is shown in the listing below.

```
char* get_date_no_day( time_t timeval )
{
    const int  sixmonths = 15724800; /* number of secs in 182 days */
    static char outstr[200];
    struct tm *tmp;
    time_t current_time = time(NULL);
    int     recent = 1;

    if ( ( current_time - timeval ) > sixmonths )
        recent = 0;

    tmp = localtime(&timeval);
    if (tmp == NULL) {
        perror("get_date_no_day: localtime");
    }

    if ( ! recent ) {
        strftime(outstr, sizeof(outstr), "%b %e  %Y", tmp);
        return outstr;
    }
    else if (strftime(outstr, sizeof(outstr), "%c", tmp) > 0)
        return outstr+4;
    else {
        printf("error with strftime\n");
        strftime(outstr, sizeof(outstr), "%b %e %H:%M", tmp);
        return outstr;
    }
}
```

3.10.5 Getting the Name of the Reference of a Link

If the file is a symbolic link, we need to print out the pathname of the file to which it points. If we do a manpage search for functions that might work with links, i.e.

```
man -k link | grep '([23])'
```

we will see, among the set of choices,

```
readlink (2)         - read value of a symbolic link
```

This is apparently what we need. Reading its manpage we see that the `readlink()` function reads the value of a link:

```
SYNOPSIS
       #include <unistd.h>
       ssize_t readlink(const char *path, char *buf, size_t bufsiz);
   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
       readlink(): _BSD_SOURCE || _XOPEN_SOURCE >= 500 ||
       _POSIX_C_SOURCE >= 200112L
DESCRIPTION
       readlink() places the contents of the symbolic link path in the  buffer
       buf,  which has size bufsiz.  readlink() does not append a null byte to
       buf.  It will truncate the contents (to a length of bufsiz characters),
       in case the buffer is too small to hold all of the contents.
```

`readlink()` fills its second argument with the contents of the symbolic link path. This storage is allocated by the calling program, not by `readlink()`, so you need to declare a large enough string to hold the pathname. You could declare this to be of size `PATH_MAX`[12], but this will be extremely large and it is probably better to truncate the name if it is very large. Because `readlink()` does not append the null byte to the string, the proper usage of this function is roughly as follows, if `filename` is the link being read:

```
ssize_t count;
if ( -1 == (count = readlink(filename, buf, NAME_MAX-1)) )
    perror("function calling this ");
else {
    buf[count] = '\0';
    // use the buf contents here
}
```

`NAME_MAX` is the maximum length of a file name, usually around 255. This is also probably too large.

A second version of `ls` follows. This version still does not parse the command line to detect whether the `-l` option is present, but the code is revised to make it easy to incorporate that change later. The other deficiencies are noted after the listing.

[12]Recall from Chapter 1 that `PATH_MAX` is a system dependent limit specifying the maximum number of bytes in a pathname.

```
Listing ls.c Version 2
void    print_file_status( char *, struct stat *);
void    mode2str( int , char [] );
char * get_date_no_day( time_t * time );
char * uid2name( uid_t );
char * gid2name( gid_t );
void    ls(char [], int);

int main(int argc, char *argv[])
{
    int i = 1;

    if ( argc == 1 )     // no arguments; use .
        ls( HERE );
    else
        while ( i < argc ){
            printf("%s:\n", argv[i] );
            ls( argv[i], 1 );
            i++;
        }
}

// Does not check whether entries are files or directories
// or . files
void ls( char dirname[], int do_longlisting )
{
    DIR             *dir_ptr;     // directory stream
    struct dirent   *direntp;     // holds one entry
    struct stat     info;         // stores stat results

    if ( ( dir_ptr = opendir( dirname ) ) == NULL )
        // could not open -- maybe it was not a directory
        fprintf(stderr,"ls1: cannot open %s\n", dirname);
    else {
        while ( ( direntp = readdir( dir_ptr ) ) != NULL ) {
            if ( do_longlisting ) {
              if (lstat(direntp->d_name, &statbuf) == -1) {
                  perror(direntp->d_name);
                  continue; // stat call failed but we go on
              }
              print_file_status(direntp->d_name, &statbuf);
            }
            else // not long -- just print name
               printf("%s\n", direntp->d_name);
        }
        closedir(dir_ptr);
    }
}

void print_file_status( char *filename, struct stat *info_p )
{
    char    modestr[11];
    ssize_t count;
    char    buf[NAME_MAX];
```

```
    mode2str( info_p->st_mode, modestr );
    printf( "%s"    , modestr );
    printf( "%4d "  , (int) info_p->st_nlink);
    printf( "%-8s " , uid2name(info_p->st_uid) );
    printf( "%-8s " , gid2name(info_p->st_gid) );
    printf( "%8ld " , (long)info_p->st_size);
    printf( "%.12s ", get_date_no_day(info_p->st_mtime));
    printf( "%s"    , filename );
    if ( S_ISLNK(info_p->st_mode) ) {
        if ( -1 == (count = readlink(filename, buf, NAME_MAX-1)) )
            perror("print_file_status: ");
        else {
            buf[count] = '\0';
            printf("->%s", buf );
        }
    }
    printf("\n");
}

// Given user-id, return user-name if possible
char *uid2name    ( uid_t uid )
{
    struct  passwd *pw_ptr;
    static  char numstr[10];  // must be static!

    if ( ( pw_ptr = getpwuid( uid ) ) == NULL ) {
        // convert uid to a string; using sprintf is easiest
        sprintf(numstr,"%d", uid);
        return numstr;
    }
    else
        return pw_ptr->pw_name;
}

char *gid2name     ( gid_t gid )
{
        struct group *grp_ptr;
        static  char numstr[10];

        if ( ( grp_ptr = getgrgid(gid) ) == NULL ) {
        // convert gid to string
                sprintf(numstr,"%d", gid);
                return numstr;
        }
        else
                return grp_ptr->gr_name;
}
```

What Is Still Wrong?

- It still treats all arguments like directories and displays an error if given a regular file.

- It still will not correctly display the files in directories specified on the command-line unless they are within the current working directory.
- The program does not print the total lines printed.
- It does not sort by filename.
- It displays all entries, including . and ..
- It does not display information about the three special bits of the file.

The most serious of these problems is that this version will fail if the directory argument is not in the current working directory. For example, if this is called as follows:

```
$ ls2 /
```

passing it the root directory, and there are subdirectories `/etc`, `/usr`, and `/bin`, the `stat()` function will be given the filenames `"etc"`, `"usr"`, and `"bin"`. It will treat these as if they are in the current working directory since it has no way of knowing their absolute pathnames. It will fail to find them in the current working directory and will thus fail. The solution is to pass `lstat()` the absolute pathnames of these directories, or their correct relative names. This is accomplished by passing `lstat()`, for example, `"/etc"` instead. This implies that we need to concatenate the path of the directory argument to its file names. We could also change the working directory each time we need to list its contents, saving the old one to return to afterwards.

3.11 The Three Special Bits

A file's i-node and the `st_mode` member of the `stat` structure returned by the various `stat()` calls, each use 16 bits to define the file's permissions and type. We have accounted for a 4-bit type and 9 bits of permissions. That leaves three unexplained bits. The three bits are the *set-user-id* bit (setuid bit), the *set-group-id* bit (setgid bit) and the *save-text-image* bit (the sticky bit).

3.11.1 The Set-User-ID Bit

The setuid bit plays a critical role in UNIX. Consider the problem of allowing users to change their own passwords. A user should be able to change her password, but no one else should be able to do so, except the super-user. Since all passwords are stored in a single file, the user has to have permission to modify the password file. This implies that the user needs write permission on the password file. But if the user has write permission on the password file, then she can modify anyone else's password too, which is not acceptable. So users cannot have write permission on the password file; the file should be owned by root, and no user except the superuser should have write permission to the file. So we are back where we started, right?

Not exactly. To change a password, a user runs the `passwd` command. The `passwd` command accesses the password file and changes it. Ordinarily, when a user runs a command, the process executing the command has the same effective user-id as the user who invoked the command from the shell, and the permissions associated with that effective user-id. If this were true for the `passwd`

command, then the `passwd` program would not be able to modify the password file, since only `root` has write permission on it. What if somehow we could give the `passwd` program the permissions associated with `root` so that it could modify the password file. Enter the setuid bit.

Recall that, at any given time, a process has two associated user-ids called its real user-id and its effective user-id. The real user-id can never be changed. The effective user-id can vary. Ordinarily, when a program is run, its effective user-id is set to be the effective user-id of the process that created it, such as the shell. This is usually the real user-id of the user who indirectly ran the process. But if the setuid bit is set on an executable file, the effective user-id of the process that executes the program in that file is the user-id of the owner of the file. In this case, `root` owns the `/usr/bin/passwd` program file, so when `passwd` runs, its effective user-id is that of `root`.

Recapping, the `passwd` program is owned by `root`, but it has its setuid bit turned on. When a user executes this program, it runs with `root` as the effective user-id, not the user. Because the program checks what the real user-id of the caller is by calling `getuid()`, it knows that it is being run by a specific user and it can access the appropriate line of the password file. This prevents `passwd` and consequently the user from modifying anything other than the password of its own entry. Other uses of the setuid bit include protecting global game data, protecting the print spooler, protecting global databases in general.

3.11.2 The Set-Group-ID Bit

The set-group-id bit is similar to the set-user-id bit except it sets the effective-group-id of the running program. If a program belongs to a group and has the setgid bit on, then when the program runs, it runs with the privileges accorded to the group that owns it rather than the privileges of the group that runs it.

The set-group-id bit is used by the `write` command. The `write` command (`/usr/bin/write`), not the `write()` system call, is a command that lets users write to a terminal. The syntax is

```
write username [ ttyname]
```

After entering this command, all input will be displayed on the given user's terminal, until an end-of-input signal (Ctrl-D) is received. To try it, type `who` to see who is logged on, and which terminals they are using. Suppose I am logged in on terminal `/dev/pts/2`. You could type

```
$ write sweiss /dev/pts/2
Can I bother you?
Ctrl-D
```

and wait for my response. Your typing will appear on my terminal window. How is it possible that one person can write on another person's terminal?

The `write` command needs write permission on the terminal on which it wants to write. First take a look at the list of pseudo-terminal devices in `/dev/pts`. The list will look something like

```
crw-------  1 tlewis   tty 136, 1 Mar  5 17:50 1
crw--w----  1 tbw      tty 136, 3 Mar  3 16:22 3
crw-------  1 nguyen04 tty 136, 4 Mar  5 10:36 4
crw--w----  1 tbw      tty 136, 5 Mar  3 15:40 5
crw--w----  1 sweiss   tty 136, 7 Mar  5 18:00 7
```

Notice that some of these have the group write bit set and others do not. Notice that all terminals belong to the `tty` group. This means that any process that runs with an effective group-id of `tty` can write to those terminals whose write bit is set. Now take a look at the write command's status:

```
$ ls -l /usr/bin/write
-rwxr-sr-x 1 root tty 10124 Jul 27 2005 /usr/bin/write*
```

and observe that the `write` executable is in the `tty` group and its setgid bit is set. When a user runs `write`, the process that executes it runs with the effective group-id of the `write` program, which is the `tty` group. This implies that the `write` command will be able to write to any terminal whose group write bit is set. Since it can be annoying to receive messages on your terminal while you are working, UNIX provides a simple command to query, enable, or disable this bit:

```
$ mesg [y/n]
```

If you type `mesg` alone, it will display `y` or `n`, depending on whether the bit is set. Typing `mesg y` turns it on, and `mesg n` turns it off.

3.11.3 The Sticky-Bit

The *sticky bit*, also called the *save-text-image bit*, serves two different purposes when it is applied to files and directories. Originally, UNIX was a pure swapping operating system. Processes were swapped in and out of memory to maintain the multiprogramming level. The swapping store was a separate disk or a separate partition of a disk that was used exclusively for storing process images when they were swapped out. The executable code and other data were kept in contiguous bytes on the swapping store, making reads and writes faster.

A program that was used by many people might go through many memory loads and unloads each day. Putting it in the swapping store made loads and unloads easier, because the file was in one piece. Setting the sticky bit on a program file prevented it from being removed from the swapping store.

If a directory has the sticky bit on, then any file you place in the directory will be protected from being deleted by anyone except you. You put a file in the directory and no one but you can remove that file. The directory is readable and writable by everyone, but the sticky bit prevents one person from deleting another person's files in that directory. This is how UNIX can implement directories such as `/tmp`, which is used to store temporary files.

3.11.4 The Special Bits and `ls`

How does `ls -l` display these three bits? If the set-uid bit is turned on, the permission string has an `s` instead of an `x` in the owner set:

```
-rws------
```

instead of

```
-rwx------
```

If the setgid bit is set, the second `x` becomes an `s`:

```
-rwxrws---
```

If the sticky bit is set, the rightmost character is a `t` instead of an `x` or a dash:

```
-rwxrwxrwt
```

We simply have to test these bits and modify the string accordingly. The revised `mode2str()` function is included in the listings that follow.

3.12 A Final Version of `ls`

The final version of the `ls` program can use the bit masks described above to display the appropriate letter in case the set-uid, set-gid, or sticky bit is set on a file. The problem of not displaying the files when the directory argument is not within the current working directory has an easy solution. The argument to the `stat()` call must be constructed by concatenating the directory name, a slash, and the directory entry, as in

```
...
while ((dp = readdir(dir)) != NULL) {
    sprintf(fname,"%s/%s",dirname, dp->d_name);  // changed here
    if (stat(fname, &statbuf) == -1) {
...
```

The `sprintf()` function is used to concatenate the directory name and the directory entry with the slash in between. This way, the argument to `stat()` will be either an absolute path name, if the directory argument was, or it will be relative to the working directory, since if it were not, the `opendir()` call would fail, and that would be the user's error in passing a non-existent directory name to the program.

The error that occurs when a command-line argument of `ls` is not a directory is slightly more work to remove. The problem is that the `ls` function called by `main()` starts out with

```
if ( ( dir_ptr = opendir( dirname ) ) == NULL )
    // could not open -- maybe it was not a directory
    fprintf(stderr,"Cannot open %s\n", dirname);
else {
    ...
```

The `opendir()` call fails because the argument is not a directory, and the program displays an error, e.g.

```
Cannot open foo
```

Instead we need to first test whether the file is a regular file or a directory. In order to do this, we need to call `stat()` with the file name, and check whether the file is a directory by using the `S_ISDIR()` macro on the `st_mode` member. The following is how the function should start:

```
if ( lstat(dirname, &statbuf) == -1 ) {
    perror(fname);
    return;       // stat call failed so we quit this call
}
else if ( ! S_ISDIR(statbuf.st_mode) ) {
    if ( do_longlisting )
        print_file_status(dirname, statbuf);
    else
        printf("%s\n", dirname);
    return;
}

if ( ( dir = opendir( dirname ) ) == NULL )
    fprintf(stderr,"Cannot open %s\n", dirname);
else {
    ...
```

The main program needs to be modified to parse the command line, using the `getopt()` function that POSIX requires of a compliant UNIX system. We need to include the `<unistd.h>` header file to use it. For this version we have a single option letter, "`l`", but the program can easily be extended to include other options. The main program is in the following listing, and the supporting functions are in the listing that follows it.

```
Listing ls.c Final Version of main()

void ls( char dirname[], int do_longlisting );
void print_file_status  ( char *fname,
                          struct stat statbuf );
char* mode2str          ( int mode  );
char *uid2name          ( uid_t uid );
char *gid2name          ( gid_t gid );

int main(int argc, char *argv[])
{
    int longlisting = 0;
    int ch;
    char options[] = ":l";

    opterr = 0;  // turn off error messages by getopt()

    while  (1) {
        ch = getopt(argc, argv, options);
```

```
        // it returns -1 when it finds no more options
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'l':
            longlisting = 1;
            break;
        case '?' :
            printf("Illegal option ignored.\n");
            break;
        default:
           printf ("getopt returned character code 0%o ??\n",
                   ch);
           break;
        }
    }

    if ( optind == argc )     // no arguments; use .
        ls( ".", longlisting );
    else
        while ( optind < argc ){
            ls( argv[optind], longlisting );
            optind++;
        }

    return 0;
}
```

The supporting functions follow. Some functions are the same as in the previous version. Those are not listed to save space. A comment indicates this when appropriate.

```
void ls( char dirname[], int do_longlisting )
{
    DIR           *dir;         // pointer to directory struct
    struct dirent *dp;          // pointer to directory entry
    char          fname[PATH_MAX];// string to hold path name
    struct  stat  statbuf;      // to store stat results

    /* test if a regular file, and if so, just display it */
    if ( lstat(dirname, &statbuf) == -1 ) {
        perror(fname);
        return;       // stat call failed so we quit this call
    }
    else if ( ! S_ISDIR(statbuf.st_mode) ) {
        if ( do_longlisting )
            print_file_status(dirname, statbuf);
        else
            printf("%s\n", dirname);
        return;
    }

    if ( ( dir = opendir( dirname ) ) == NULL )
```

```
        fprintf(stderr,"Cannot open %s\n", dirname);
    else {
        printf("\n%s:\n", dirname);
        // Loop through directory entries
        while ((dp = readdir(dir)) != NULL) {
            if (strcmp(dp->d_name, ".") == 0 ||
                strcmp(dp->d_name,"..") == 0 )
                // skip dot and dot-dot entries
                continue;

            if ( do_longlisting ) {
                // construct a pathname for the file using the
                // directory name passed to the program and the
                // directory entry
                sprintf(fname,"%s/%s",dirname, dp->d_name);

                // fill the stat buffer
                if (lstat(fname, &statbuf) == -1) {
                    perror(fname);
                    continue;  // stat call failed but we go on
                }
                print_file_status(dp->d_name, statbuf);
            }
            else
                printf("%s\n", dp->d_name);
        }
    }
}


void print_file_status     ( char        *dname,
                            struct stat statbuf  )
{
    ssize_t count;
    char    buf[NAME_MAX];

    /* Print out type, permissions, and number of links */
    printf("%10.10s", mode2str (statbuf.st_mode));
    printf("%3d", (int) statbuf.st_nlink);

    /* Print out owner's name if it is found using getpwuid() */
    printf(" %-8.8s", uid2name(statbuf.st_uid));

    /* Print out group name if it is found using getgrgid() */
    printf(" %-8.8s", gid2name(statbuf.st_gid));

    /* Print size of file */
    printf(" %8jd", (intmax_t)statbuf.st_size);

    /* Print time of last modification */
    printf( " %.12s ", get_date_no_day(statbuf.st_mtime));

     /* print  file name  and if a link, the linked file */
    printf(" %s",  dname);
```

```
    if ( S_ISLNK(statbuf.st_mode) ) {
        if ( -1 == (count = readlink(dname, buf, NAME_MAX-1)) )
            perror("print_file_status: ");
        else {
            buf[count] = '\0';
            printf("->%s", buf );
        }
    }
    printf("\n");

}

char* mode2str              ( int mode  )
{
    static char str[11];
    strcpy( str, "----------" );           // default=no perms

    if ( S_ISDIR(mode) )        str[0] = 'd';   // directory?
    else if ( S_ISCHR(mode) )   str[0] = 'c';   // char devices
    else if ( S_ISBLK(mode) )   str[0] = 'b';   // block device
    else if ( S_ISLNK(mode) )   str[0] = 'l';   // symbolic link
    else if ( S_ISFIFO(mode))   str[0] = 'p';   // Named pipe (FIFO)
    else if ( S_ISSOCK(mode))   str[0] = 's';   // socket

    if ( mode & S_IRUSR ) str[1] = 'r';     // 3 bits for user
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';     // 3 bits for group
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';     // 3 bits for other
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';

    if ( mode & S_ISUID ) str[3] = 's';     // set-uid
    if ( mode & S_ISGID ) str[6] = 's';     // set-gid
    if ( mode & S_ISVTX ) str[9] = 't';     // sticky bit
    return str;
}

char *uid2name      ( uid_t uid   )
// same as previous version and omitted here.

char *gid2name      ( gid_t gid   )
// same as previous version and omitted here.
```

This last version of `ls` solves all of the important problems. The goal was really to understand how to interface to various system functions, and writing `ls` correctly was a way to get experience with a few important structures: the `dirent` object and the i-node.

3.13 Modifying File Attributes

The preceding exercise accessed attributes stored in the i-nodes, but did not modify any of them. This is a good time to consider which of how attributes can be modified by user level programs, and how. When appropriate, various shell commands that are related will be noted.

3.13.1 Type of a File

The type of a file is initialized when the file is first created and it cannot be changed after that. As mentioned before, a file can be a regular file, a directory, a device special file (of which there are several types), a socket, a symbolic link, or a named pipe.

The `creat()` system call creates regular files; the `mkdir()` call makes a directory. The `mkdir` command creates a new directory. In fact it is the only way to make a directory. Other calls make the other types of files. For example, `mknod()` is the system call that creates special files such as device special files. The `mknod` command makes these at the user level. There is also a special function to make FIFO special files, the `mkfifo()` call.

3.13.2 Permission Bits and Special Bits

The permission bits are initialized when the file is created by the kernel. The second argument of the `creat()` call, for example, is a number used to initialize the file mode. However, the mode assigned to the file is not exactly this argument; it is this number modified by applying the process's *umask*. The umask of the process is the umask of the effective user-id of the process.

Every user has a umask. The umask is an inverted mask; a 1 in the umask represents a bit to turn off in the masked value, i.e., the bitwise C operation `(mode & ~umask)` is applied. For example, if the umask is octal 022, then it is binary 000010010. This shows that bits 2 and 5 of the mask are set, which means bits 2 and 5 are turned off in the masked value. Since 2 is write by others and 5 is write by group, this umask sets the default file mode so that no one other than the owner can write to the file. The call,

```
fd = creat("newfile", 0766 );
```

would create the file named `newfile` with permission string `-rwxrw-rw-` if there were no umask. If the umask has value 022, then the umask is subtracted and the file permissions would be `-rwxr-r--`. Remember, the umask is an un-mask mask – it unsets the bits that are set in it. And it only does this when the file is created, not after.

A process inherits its umask value when it is started up[13], but it can change it by calling `umask()`:

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

[13] All processes are created by other processes. When you run a program from the command line, the shell creates the process that executes the program and the shell is its parent process. The umask of the shell becomes the umask of the new process. The umask of the shell is initialized when the shell starts up, usually by reading a shell configuration script such as the .bash_profile file.

The `umask()` system call changes the umask to the bitwise AND of mask and octal 0777 and returns the value of the previous mask. In other words, it ignores the parts of the mode_t value passed to it that define the file type and special bit values. There is also a shell command named `umask` that can be used to inspect or change the user's umask.

A file's permission bits can be changed by a process by calling `chmod()` :

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

As with the mode in the `creat()` system call, the mode may be supplied as an integer or as the bitwise-or of one or more of the bitmasks described earlier – e.g. `S_IRUSR`.

Of course at the user level, `chmod` is the command to change the permission bits of a file. See the man page for all of the details on how to use it. There are many variations on it.

3.13.3 Number of Links to a File

The name of a file is just a name stored in a directory. The total number of names that a file has is called its *link count*. The i-node contains this link count. When the link count is decremented because a name is deleted, it is compared to zero. When it reaches zero, the file is actually removed.

The `link()` system call creates a new name for an existing file and the `unlink()` and `unlinkat()` calls remove a name for a file:

```
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
int unlink(const char *path);

#include <fcntl.h>
int unlinkat(int dirfd, const char *pathname, int flags);
Feature test macro on glibc for  unlinkat():
    Since  glibc 2.10: _XOPEN_SOURCE >= 700 || _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10: _ATFILE_SOURCE
```

If the new pathname exists already, `link()` will not overwrite it, instead returning -1 and setting `errno` to `EEXIST`. Most implementations require that both pathnames be on the same file system, although POSIX.1 allows an implementation to support linking across file systems. There are various other conditions that can cause it to fail as well, and you should read its man page for details.

Unlinking a file, whether with `unlink()` or `unlinkat()`, requires that the process has write permission and execute permission in the directory containing the name to be removed, since it is the directory entry that is removed. If the sticky bit is set in this directory the process must have write permission for the directory and either

- own the file,
- own the directory, or

- have superuser privileges.

If none of these are satisfied, `errno` is set to `EPERM`. The `unlink()` call may do more than just delete the name of the file. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse. However, if one or more processes still have the file open, it will remain in existence until the last file descriptor referring to it is closed, even if the link count went to zero. Some versions of Unix will set `errno` to `EBUSY` in this case (but not Linux.)

The `unlinkat()` system call does the same thing as `unlink()`, but it does so relative to a given directory that has been opened[14]. Specifically, one can use the `open()` system call to open a directory and get a file descriptor for it, and use that file descriptor as the reference point for the `pathname` second argument. Thus, a path relative to a given directory can be supplied. The last argument is a flag that can be used to make `unlinkat()` behave more like the `rmdir()` system call. If it is passed `AT_REMOVEDIR`, then it is as if `rmdir()` was called on `pathname`. The following example demonstrates its use.

```
Listing unlinkatdemo.c
/* usage:  unlinkatdemo  directory file
           where path is the path to a file relative to
           the given directory.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

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

    /* check args       */
    if ( argc < 3 ){
        fprintf( stderr, "usage: %s directory file\n", *argv);
        exit(1);
    }

    /* open directory   */
    if ( (dirfd = open(argv[1], O_RDONLY)) == -1 ) {
        fprintf( stderr, "could not open %s\n", argv[1]);
        exit(1);
    }
    /* We could check here if argv[2] refers to a directory or a file
       to make this more robust, but we don't. */
```

[14]The reason that `unlinkat()` exists is to prevent possible race conditions which `unlink()` is susceptible to. A component of a pathname given to `unlink()` might change in parallel to the call if the pathname is not within the current working directory of the process. By opening a directory and getting its file descriptor, the race condition is eliminated.

```
    /* unlink the file relative to dirfd */
    if ( -1 == unlinkat(dirfd, argv[2], 0) ) {
        fprintf( stderr, "could not unlink %s/%s\n", argv[1], argv[2]);
        exit(1);
    }
    return 0;
}
```

A descriptor for the directory supplied as the first argument to the program is obtained and stored in `dirfd`. If this is successful then `unlinkat()` is called with this descriptor and the name of a file relative to that directory, which is supplied in the second argument to the program.

At the user level, the commands to create new links and remove old links are `link` and `unlink`. There is also `ln`:

```
/usr/sbin/link existing-file new-file
/usr/sbin/unlink file
/usr/bin/ln [ -fns ] source_file [ target ]
```

Consult their man pages for details.

3.13.4 Owner and Group of a File

The owner and group of a file are established when the file is created. Remember, you as a user do not create any files. The kernel does. Whether you use the shell to create a file, or `vi`, or any other program, the program makes a request to the kernel, sooner or later, by calling `creat()`. The kernel uses the effective-user-id and the effective group-id of the process that issued the `creat()` call as the owner and group of the file. Sometimes though, it uses the group-id of the parent directory.

The owner and group of a file are changed only by the `chown()` and `chgrp()` system calls or their command equivalents. It is very unusual for anyone other than the super-user to change ownership of a file. The `chown` command or system call changes who owns a file. You cannot change the ownership of a file that you do not already own. You should consult the man pages for the details.

3.13.5 Size of a File

A file increases in size as data is written to it using the `write()` system call. Its size is set to zero by the `creat()` call. There is no call to reduce the size of a file other than to zero it. In other words, files can grow and can be reset to zero size, but nothing else. The `lseek()` system call may be used to reposition a file pointer beyond the physical end of the file. This by itself does not change the size of the file. Only actual writes can do that. If data is written to this position, the file's stored size will be increased, with a hole in its middle. The number of disk blocks that it uses will be the number actually required to store data excluding the hole. The following listing illustrates.

```
Listing file_hole.c
#include <stdlib.h>
#include <fcntl.h>
```

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

    /* create a new file named file_with_hole in the pwd */
    if ((fd = creat("file_with_hole", 0644)) < 0)
        exit(1);

    /* put a small string at the beginning */
    if (write(fd, buffer, 10) != 10)
        exit(1);

    /* seek 131072 = 2^17 bytes past the beginning of the file */
    if (lseek(fd, 131072, SEEK_SET) == -1)
        exit(1);

    /* write a small string there as well. */
    if (write(fd, enddata, 10) != 10)
        exit(1);

    /* we now have a large file with a big hole. */
    exit(0);
}
```

If you do an `ls -sl` on the file `file_with_holes` you will see that its size is 131082 bytes and uses 12 blocks. A block is usually 1KB.

```
$ ls -sl file_with_hole
12 -rw-r--r-- 1 stewart stewart 131082 Feb 21 20:09 file_with_hole
```

3.13.6 Modification and Access Time

Every i-node maintains three timestamps:

- `st_mtime`, the time the file was last modified
- `st_ctime`, the time the file attributes were last modified
- `st_atime`, the time the file was last read

These three timestamps are set by the kernel as the file is accessed and modified. You have no control over `st_ctime`, but you can change `st_atime` and `st_mtime` manually using the `utime()` system call:

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
```

where a `utimbuf` is defined as

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

It is possible to change the modification time of a file to a future time! The system does not check that any time values make sense. The listing below can be used to change the timestamps on a file to any time at all.

There are many system calls that affect the time stamps on a file. For example, when you remove an entry from a directory, the directory itself is modified and its `st_mtime` value changes. If you list the contents of a directory, its `st_atime` value changes.

```
Listing changefiletimes.c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <utime.h>

#define CH_ACCESS 1
#define CH_MOD    2

// setclockback
// Given a time_t now, returns a new time_t by subtracting the days
// hours, minutes, and seconds specified
time_t setclockback( time_t now, int days, int hours,
                     int mins, int secs)
{
    struct tm time_tm;
    time_t    newtime;

    localtime_r(&now, &time_tm);
    time_tm.tm_sec -= secs;
    time_tm.tm_min -= mins;
    time_tm.tm_hour -= hours;
    time_tm.tm_mday -= days;
    newtime = mktime(&time_tm);

    return newtime;
}

void backdate(const char* fn, int mode, int times[], int size)
```

```
{
    struct stat buf;
    time_t temp_t;
    time_t access_t, mod_t;

    struct utimbuf utbuf;

    if ( 4 != size ) {
        fprintf(stderr, "wrong number of parameters to backdate.\n");
        exit(1);
    }

    if ( -1 == stat(fn, &buf ) ){
        perror("Could not stat file.\n");
        exit(1);
    }

    time(&access_t);
    if ( mode & CH_ACCESS ) {
        temp_t = setclockback( access_t, times[0], times[1],
                               times[2], times[3]);
        access_t = temp_t;
    }
    time(&mod_t);
    if ( mode & CH_MOD ) {
        temp_t = setclockback( mod_t, times[0], times[1],
                               times[2], times[3]);
        mod_t = temp_t;
    }
    utbuf.actime = access_t;
    utbuf.modtime = mod_t;

    if ( -1 == utime(fn, &utbuf) ) {
       perror("Error changing times");
       exit(1);
    }

}
int main(int argc, char* argv[])
{
    int i = 0;
    time_t now;
    char*  timestr;
    char   filename[255];
    int    times[4] = {0,0,0,0};

    if ( argc < 2) {
        printf("usage: %s filename days hours minutes seconds\n",
```

```
                argv[0]);
        exit(1);
    }

    strcpy(filename, argv[1]);
    while ( (--argc >= 2) && (i < 4) ) {
        times[i] = atoi(argv[i+2]);
        i++;
    }

    time(&now);
    timestr = ctime(&now);
    backdate(filename, CH_ACCESS|CH_MOD, times, 4);
    return 0;
}
```

3.13.7 Name of a File

The system call to rename a file is `rename()`:

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

It returns –1 on failure, 0 on success. `oldname` is the pathname of the file or directory to be renamed and `newname` is the new pathname. The behavior is complex, depending on whether `oldname` refers to a file, a directory, or a symbolic link, and whether `newname` already exists and whether it is on the same file system. To illustrate just some of the complexity:

- If `oldname` specifies a file that is not a directory, then if `newname` exists, it must not refer to a directory. If `newname` exists and is not a directory, it is removed, and `oldname` is renamed to `newname`. The process must have write permission for the directory containing `oldname` and for the directory containing `newname`, since both are being changed.
- If `oldname` specifies a directory, and if `newname` exists, it must refer to an empty directory. If so, it is removed, and `oldname` is renamed to `newname`. Additionally, `newname` cannot contain a path prefix that includes `oldname`. For example, you cannot rename `/class_stuff/notes` to `/class_stuff/notes/systemcalls`, because the old name `/class_stuff/notes` is a path prefix of the new name and cannot be removed.
- As a special case, if `oldname` and `newname` refer to the same file, the function returns successfully without changing anything.

The process calling `rename()` must have appropriate permissions in all cases. A simplistic description is that `rename()` renames a file, and if this involves deleting a name from one directory and creating one in another, that is what it does. For complete details see the man page.

3.14 Traversing the Tree, Up and Down

Two common things that we do with trees are ascending them and descending them. By "ascending" we mean traveling from a given node to the parent and to the grandparent and so on until we reach the root. By "descending" we mean something more extensive, as there are many nodes in the subtree rooted at a given node – we mean visiting all nodes in that subtree in some specified order. This is a descent into the tree.

The first type of traversal is what the `pwd` command does; it travels from the current working directory up the tree so that it can display the path from the root to the current working directory. With a little research you will discover that there is a system call[15], `getcwd()`, that does exactly this. The second type of traversal is what recursive versions of commands such as `ls`, `grep`, `chown`, `chmod`, and many more, do, and what the `find` command must do as well. They start in the given directory and visit all files in the subtree rooted at that directory.

Both of these problems present interesting programming challenges and require us to learn a bit more about the file system and the file hierarchy. We will first solve the problem of finding the path to the current working directory. After that we will look at two different functions provided in the API for "walking" the file tree.

3.15 The pwd Command

Recall that the `pwd` command prints the absolute pathname to the current working directory. As a warm-up, we will exercise our knowledge of directories and i-nodes to make sure we understand the task that lies ahead. Specifically we will try to reconstruct a portion of a file hierarchy from limited information about i-numbers in a set of directories.

Suppose that we are given a directory named `scratch` that contains subdirectories that also have subdirectories, and so on. Each of these subdirectories may have regular files as well. The command

```
$ ls -iaR scratch
```

will recursively display the i-numbers and filenames of all files within these directories, including entries for "`.`" and "`..`". The listing below shows the output of this command on a hypothetical directory named `scratch`, except that directory names that would ordinarily appear in the output and just displayed the files contained in them. From this listing, it is possible to reconstruct the file hierarchy rooted at `scratch`.

```
725 .                                   732 stuff
449 ..
753 temp                                727 .
727 test1                               725 ..
728 test2                               729 file1
                                        730 file2
731 .                                   733 garbage
728 ..
733 junk                                728 .
```

[15] In Linux it is a system call. On other systems it is a library function.

```
725 ..                                  729 temp
731 data
748 srcs
```

Before reconstructing the hierarchy, you should be able to answer the following questions:

- What is the i-number of `scratch`?
- Which filenames are links to the same file?

The key to reconstruction is to use the i-numbers of the parent entries to obtain their names, and use process of elimination for the rest. See Figure 3.6.

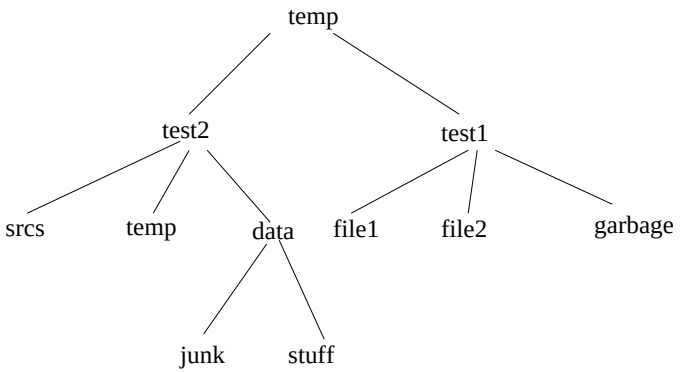

Figure 3.6: Hierarchy below `scratch` directory

This exercise shows that the parent directory entries in a directory play a vital role in the hierarchy, because they are, in essence, back links. They are a way to ascend the tree.

3.15.1 Implementing the pwd Command

Suppose my working directory is `chapter03`, which is located in the directory `unix_demos`, which is in `class_stuff`, which is in `cs82010`, which is in `home`, which is in the root directory. Then typing `pwd` prints the path

```
/home/cs82010/class_stuff/unix_demos/chapter03
```

Of course when it starts, `pwd` does not "know" where it is. It is somewhere in a node of a very large tree, but it does not know the path to it. It could use an exhaustive search to find the path, starting at the root and recursively searching through every directory until it finds one whose i-number matches the i-number of the current working directory, but that would not be a very useful command. From the preceding exercise it should be clear that `pwd` has work backwards to construct the path, using the parent entries in the directories as it goes along.

3.15.2 About pwd

Before proceeding, let us note that the actual `pwd` command is very often a shell built-in, not a separate program. The easiest way to determine if it is a built-in or a program, at least on a Linux host, is to enter

```
$ type pwd
```

If it is a shell built-in, the reply will be

```
pwd is a shell builtin
```

If not, it will be the pathname to the executable, such as `/bin/pwd`. You can then use the `file` command to identify the type of file:

```
$ file /bin/pwd
/bin/pwd: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.9, stripped
```

which shows the type of executable it is. The shell built-in most likely just retrieves the value of the environment variable `PWD`, which is updated as the working directory is changed. The shell will use the kernel's `getcwd()` system call to accomplish this. We exclude the possibility of solving this problem by calling `getcwd()`; that defeats the objective, which is in effect to write our own version of `getcwd()`.

3.15.3 How pwd Works

How does `pwd` construct the path? It is not stored anywhere. In fact, no directory has information about where it is located except for that one little item – the dot-dot entry. The `".."` in a directory always contains the i-number of the parent directory, with one exception: *the root of the file system has no parent.* If you list the i-numbers in the root directory you will see that . and .. have the same i-number:

```
$ cd /
$ ls -iaF | grep '\./'
2 ./
2 ../
```

This provides a stopping criterion for an iterative solution to printing the pathname. The idea is to do the following:

1. Record the i-number, n, of the current directory, i.e., the i-number for ".".
2. Change directory (`chdir()`) to the parent directory.

3. Compare the i-number of the parent directory, which is now the current directory, to n. If the i-number of the parent directory is n, stop. Otherwise, find the name of the link with i-number n and append that name to the left of the current pathname, and append a "/" to the left of that, and go back to step 1.

4. Print the current pathname.

This algorithm seems like it should work. However, as we have to construct the string from the right end to the left end, it is easier to use a recursive solution. Recursion is never a fast strategy and you should not think that the real `pwd` is recursive, however, it is an easy way to create a working solution.

3.15.4 A First Version of `pwd`

The first version of `pwd.c` is below, named `pwd1.c`. As we will see shortly, this program does not work correctly. The bug is only exposed in limited conditions. Some comments are omitted from the listing to save space.

Listing 1. pwd1.c

```
#include    <stdio.h>
#include    <string.h>
#include    <stdlib.h>
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <dirent.h>

// BUFSIZ is defined in stdio.h: it is the maximum string size
#define     MAXPATH      BUFSIZ

// print_pwd prints the pwd to the string abs_pathname and
// NULL-terminates it
void  print_pwd    (  char * abs_pathname );

// print_path prints to str the path from / to the file
// specified by  cur_inum.
void  print_path ( char* str, ino_t cur_inum );

// inum_to_name puts the filename of the file with i-node inum
// into string buf, at most len chars long and 0 terminates it.
void  inum_to_name(ino_t inum, char * buf, int len );

//  get_ino gets the i-node number of file fname, if that fname
//  is the name of a file in the current working directory
//  Returns 0 if successful, -1 if not.
int   get_ino(char * fname, ino_t * inum);

/*********************************************************/
int main(nt argc, char* argv[])
{
    char  path[MAXPATH] = "\0";  //  string to store pwd
```

```
    print_pwd( path );              //  print pwd to string path
    printf("%s\n", path);           //  print path to stdout
    return 0;
}

/*********************************************************/
void print_pwd    (  char * pathname )
{
    ino_t inum;

    get_ino(".", &inum );
    print_path(pathname, inum );
}

/*********************************************************/
void print_path( char * abs_pathname, ino_t this_inode )

{
    ino_t       parent_inode ;
    char        its_name[BUFSIZ];

    // get inumber of parent
    get_ino("..", &parent_inode );

    // At root iff parent inum == cur inum
    if ( parent_inode != this_inode ) {
        chdir( ".." );    // cd up to parent
        //  get filename of current file
        inum_to_name(this_inode, its_name, BUFSIZ);
        // recursively get path to parent directory
        print_path(abs_pathname,  parent_inode );
        strcat(abs_pathname, "/");
        strcat(abs_pathname,  its_name );
    }
    else
        strcat( abs_pathname, "/");
}

/*********************************************************/
void inum_to_name(ino_t inode_to_find , char *namebuf,
                  int buflen)
{
    DIR            *dir_ptr;
    struct dirent  *direntp;

    dir_ptr = opendir( "." );
    if ( dir_ptr == NULL ) {
        perror( "." );
        exit(1);
    }

    // search directory for a file with specified inum
    while ( ( direntp = readdir( dir_ptr ) ) != NULL )
        if ( direntp->d_ino == inode_to_find ) {
```

```
            strncpy( namebuf, direntp->d_name, buflen);
            namebuf[buflen-1] = '\0';
            closedir( dir_ptr );
            return;
        }
    fprintf(stderr, "\nError looking for i-node number %d\n",
            inode_to_find);
    exit(1);
}

/*********************************************************/
int  get_ino( char *fname, ino_t *inum )
{
    struct stat info;
    if ( stat( fname , &info ) == -1 ){
        fprintf(stderr, "Cannot stat ");
        perror(fname);
        return -1;
    }
    *inum   = info.st_ino;
    return 0;
}
```

The recursive function in the above listing is the `print_path()` function. Given an i-number, `this_inode`, of the current working directory file, it checks whether the i-number of the parent directory is equal to `this_inode`. If they are equal, it stops the recursion, because this means it is at the root. In this case, it appends a slash, "/", to the pathname under construction. If it is not, it has to get the filename of the current working directory. Following the logic we described above, it does this by going up one level to the parent directory and looking for the i-number in that directory. When it finds it, it retrieves the name matching the i-number.

Therefore, it begins by getting the i-number of the parent directory into `parent_inode`. It then compares `parent_inode` to `this_inode`, and if unequal it calls `chdir("..")` to step into the parent directory. Once in the parent directory, it calls `inum_to_name()` with `this_inode`. The `inum_to_name()` call will store into `its_name` the actual name of the directory with i-number `this_inode`. `print_path()` then makes a recursive call to itself, passing the i-number of the parent, `parent_inode`, and the string in which it is constructing the pathname. In the recursive call, it is one step closer to the root of the file system, ensuring that the problem size is diminished. After the recursive call, we assume that the absolute pathname from the root down to the parent directory is in the argument `abs_pathname`, so we append a slash followed by the file name we found for the current directory, `its_name`, to `abs_pathname.` and we return. If `parent_inode == this_inode`, then `print_path()` appends a slash to `abs_pathname`. When it does this, `abs_pathname` is the null string, so when this program is called from within the root directory, it will display "/".

The function that obtains the i-number of a filename is simple; it calls `stat()` with the given filename and returns the `st_ino` member of the `stat` structure filled by the `stat()` call. The `inum_to_name()` function is a slightly more complex. It opens the current working directory and repeatedly reads the entries in the directory until it finds an entry with the given i-number. This is a linear search of the directory. It uses the `readdir()` call to retrieve a pointer to a `dirent` structure and pulls the name member out of the `dirent` structure. If it does not find an entry with the given i-number it prints an error message on the standard error stream.

Running the program from various directories, you should discover that

- most pathnames start with a leading double-slash instead of a single slash, and
- the root directory is correctly displayed as `/`.

Now try running the program from within a directory in a file system that has been mounted on the root file system. You will discover that either the program does not print the full absolute path, that it prints an incorrect path, or that it generates the error

```
Error looking for i-node number n
```

for some n. This is a result of the program's not taking into account the way in which file systems are mounted. In order to understand this, we will revisit how mounting works.

3.15.5 Multiple File Systems: Mounting

Section 3.3 introduced the concept of file system mounting. In UNIX, unlike DOS or Windows, all files on all volumes are part of a single directory hierarchy; this is achieved by mounting one file system onto another. What exactly do we mean by mounting one file system onto another? To make it clear, suppose that we have a root file system that looks in part like the one in Figure 3.7.

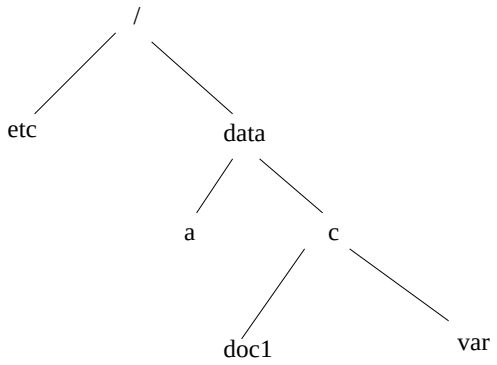

Figure 3.7: Root file system before mount

It has a subdirectory named `data` with two subdirectories named `a` and `c`. Notice that `c` is not empty. Suppose this file system is located on one internal disk of the computer, and suppose that there is a second internal disk that contains its own file system, as shown in Figure 3.8. The second disk is represented by a device special file named `/dev/hdb`. There is a file system on this second disk, whose root has two subdirectories named `staff` and `students`, and `students` has subdirectories `grad` and `undergrad`. To make the files in this second file system available to the users of the system, it has to be mounted on a *mount point*. A *mount point* is a directory in the root file system that will be replaced by the root of the mounted file system.

If the directory `/data/c` is a mount point for the `/dev/hdb` file system, then we say that `/dev/hdb` is mounted on `/data/c`. The following `mount` command will mount the `/dev/hdb` file system on `/data/c`. It does not matter that `c` contains file links already; the `mount` merely hides them while it is there. They would disappear from view until the file system was unmounted, when they would reappear.

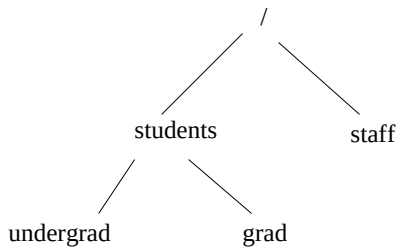

Figure 3.8: File system `/dev/hdb`

```
$ mount /dev/hdb /data/c
```

After this command, the root file system will be as in Figure 3.9.

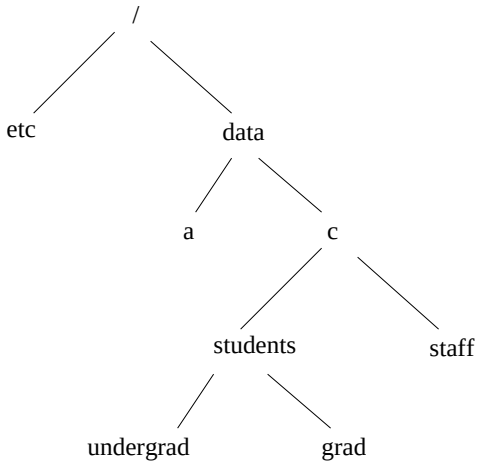

Figure 3.9: Root system after mount of `/dev/hdb`

The absolute pathname of the `grad` directory would then be `/data/c/students/grad`.

In reality, we would have to specify the type of file system written on `/dev/hdb`, unless it is the default file system, and only the superuser is allowed to run the `mount` command, with a few exceptions, such as mounting removable storage devices such as CD-ROMs, DVDs, and USB devices.

When a directory becomes a mount point, the kernel restructures the directory hierarchy. The directory contents are not lost; they are masked by the root directory of the mounted file system. The kernel records in a list of mount points that this file system is mounted on this directory. Different versions of UNIX implement mounting in different ways. This relevant part of this discussion is that operations that traverse the tree can identify mount points because they are directories that are the roots of different file systems than their parents.

3.15.6 Duplicate I-node Numbers and Cross-Device Links

The advantage of mounting is that it simplifies the user's conceptualization and navigation of the file hierarchy. It is a single hierarchy. One problem that the kernel must deal with is that there may be many files with the same i-number, since i-numbers are unique only within a single file system. In fact, the root of every file system has i-number 2, and its `".."` entry is also 2, so there will be

many i-nodes with i-number 2. The kernel is able to distinguish i-nodes because i-nodes are also known by the name of the device on which they are located. On all modern UNIX systems, the i-node contains a member that stores the name of the device on which it is located. It is like having a dog tag on a dog; if the dog gets lost, someone can look at it and know where it came from. (I'll admit i-nodes can't wander around, but a process looking at a memory copy of an i-node may need to know which device it got it from.) So i-nodes are more accurately represented by (i-number, device number) pairs.

To pursue this line of thought, suppose there is a file in the root file system with i-number 52. Suppose it has a hard link named `/data/a/doc1`. Suppose also that, referring to the figure above, `/dev/hdb` has a file `/students/undergrad/hwk1` with i-number 52. If UNIX allowed the creation of a hard link across the two file systems with the call

```
link( "/data/a/doc1", "/data/c/students/grad/doc1")
```

then, after the call, there would be two links in the `/dev/hdb` file system, each having the same i-number, but these i-numbers would refer to different i-nodes. This would break the file system, unless directories were able to store device numbers as well as i-numbers with file names, which would require rewriting almost the entire kernel. Therefore, UNIX systems generally do not allow anyone to create hard links that span across file systems. All hard links to a file must be in a single file system. For the same reason, you cannot use the `rename()` call to move a file across file systems. The `rename()` call makes a new name in the specified directory pointing to the original file. The call,

```
rename( "/data/a/doc1", "/data/c/students/grad/doc1")
```

would cause the file system to have two links to different files but with the same i-number.

3.15.7 A Second Version of `pwd`

It is now possible a correct implementation of the `pwd` command, which we call `pwd2.c`. The problem with the original program is what it does when it reaches a mount point as it works its way towards the root. Suppose that the current working directory is

```
/data/mnt/backups/backup1/current_backup/home/class_stuff
```

and that `/data/mnt/backups` is actually a mount point on which the file system `/dev/sdc1` has been mounted. The device `/dev/sdc1` is an external drive that is mounted to backup and restore files as needed, and it has a top-level directory named `/backup1`, under which one can find the directory `current_backup/home/class_stuff`.

As our program ascends the tree, it eventually reaches the directory `backup1`. Suppose this has i-number 6643. When the program has made `backup1` its current working directory, it obtains the i-number of its parent, which is the root of the mounted file system `/dev/sdc1` and therefore has i-number 2.

It then changes directory to this parent directory, the `/data/mnt/backups` directory, and calls `inum_to_name()` with i-number 6643. `inum_to_name()` searches through the directory entries in the directory `/data/mnt/backups`, finds 6643, and sets its_name to `"backup1"`.

Then the program makes the next recursive call, in which `this_inode` = 2. When it changes directory to its parent though, it has just crossed the mount point. There will not be a directory entry in this directory with the i-number 2. Although `backups` has i-number 2 as the root of the `/dev/sdc1` file system, the directory for `/data/mnt` does not have a `dirent` entry for it with the i-number 2. As a result `inum_to_name()` will fail (unless by coincidence, it has an entry with i-number 2 for some other name); it will search through the entire directory and then report that it could not find i-number 2.

Although the dirent entry for backups in /data/mnt does not have i-number 2, its i-node does have that i-number. This is why, if we were to issue the command

```
ls -i /data/mnt
```

we would see the i-number 2 displayed for `backups`, because `ls` calls `stat()` to retrieve the i-node contents. We must do the same thing. We make several changes to make the program correct.

1. We replace `get_ino()` by a function, `get_deviceid_inode()` that gets the i-number and device-id for the given file name. We call this function when the program starts so that we have the device-id of the current working directory, and pass it to the recursive function, `print_path()`.
2. We change the test within `print_path()` for when we have reached the root: we have reached the root if and only if the parent i-number matches the child i-number and they have the same device ids, otherwise we are not there yet.
3. We change the way we find the name of the current directory in `inum_to_name()`. We pass `inum_to_name()` the device-id and the i-number of the current working directory. `inum_to_name()` calls `stat()` for every entry in the current working directory, trying to match the i-number and the device-id stored in that entry's i-node against the i-number and device-d it was passed. If the i-number and device match the pair passed to `inum_to_name()`, then the `d_name` member of the `dirent` structure is the name of the current working directory, and this is what is returned to `print_path()`.
4. To fix the problem with the terminating slash, which should not be at the end of the path, we can keep a static variable in the recursive function that keeps track of the level of recursion. The trailing slash will only be printed if we are in a recursive call, not at the top-level call.

All of these changes are incorporated into `pwd2.c`, whose code is below in Listing 2.

```
Listing 2. pwd2.c
// same includes as before

#define     MAXPATH      BUFSIZ

/*********************************************************/
void  print_pwd    (  char * abs_pathname );

// print_path prints to str the path from / to the file
// specified by  cur_inum on the device with device id
// dev_num.
```

```
void  print_path( char* str, ino_t cur_inum, dev_t  dev_num);

// inum_to_name puts the filename of the file with inode inum
// on device with device number dev_num into string buf, at
// most len chars long and 0 terminates it.
void  inum_to_name ( ino_t inum, dev_t dev_num,
                     char * buf, int len );

// get_dev_ino gets the device id and the inode number of
// file fname, if that fname is the name of a file in the
// current working directory. Returns 0 if success, -1 if not.
int   get_deviceid_inode  ( const char *fname, dev_t *dev_id,
                      ino_t *inum );

/*********************************************************/

int main(int argc, char* argv[])
{
    char  path[MAXPATH] = "\0";

    print_pwd( path );
    printf("%s\n", path);
    return 0;
}

/*********************************************************/
void print_pwd    (  char * abs_pathname )
{
    ino_t inum;
    dev_t devnum;

    get_deviceid_inode(".", &devnum, &inum );
    print_path(abs_pathname, inum, devnum);
}

/*********************************************************/
void print_path  (  char *  abs_pathname,
                    ino_t   this_inode,
                    dev_t   this_dev )
// Recursively prints path leading down to file with this
// inode on this_dev  Uses static int height to determine
// which recursive level it is in
{
    ino_t       parent_inode ;
    char        its_name[BUFSIZ];
    dev_t       dev_of_node,
                dev_of_parent;
    static int  height = 0;

    // get device id and inumber of parent
    get_deviceid_inode("..", &dev_of_parent, &parent_inode );

    // At root iff parent inum == cur inum & device ids
    // are same
```

```
    if ( ( parent_inode != this_inode )
         || ( dev_of_parent != this_dev) ) {
        chdir( ".." );

        inum_to_name(this_inode, this_dev, its_name, BUFSIZ);
        height++;  // about to make recursive call
        print_path(abs_pathname, parent_inode, dev_of_parent);
        strcat(abs_pathname,  its_name );

        if ( 1 < height )
            /*  Since height is decremented whenever we
             *  leave call it can only be > 1 if we have not
             *  yet popped all calls  from the stack
             */
            strcat(abs_pathname,  "/" );
        height--;
    }
    else  // must be at root
         strcat(abs_pathname, "/");
}
/*********************************************************/
void inum_to_name(ino_t inode_to_find, dev_t devnum,
                  char *name, int buflen)
{
    DIR             *dir_ptr;
    struct dirent    *direntp;
    struct stat      statbuf;

    if ( NULL == (dir_ptr = opendir( "." ) ) ) {
        perror( "." );
        exit(1);
    }

    while ( ( direntp = readdir( dir_ptr ) ) != NULL ) {
        if ( -1 == (stat(direntp->d_name, &statbuf)) ) {
              fprintf(stderr, "could not stat");
              perror(direntp->d_name);
              exit(1);
        }
        if ( ( statbuf.st_ino == inode_to_find ) &&
             ( statbuf.st_dev == devnum) ) {
            strncpy( name, direntp->d_name, buflen);
            name[buflen-1] = '\0';   //  just in case
            closedir( dir_ptr );
            return;
        }
    }
    fprintf(stderr, "Error looking for i-node %d\n",
            inode_to_find);
    exit(1);
}

/*********************************************************/
int  get_deviceid_inode( const char *fname, dev_t *dev_id,
```

```
                    ino_t *inum )
{
    struct stat info;
    if ( stat( fname , &info ) == -1 ){
        fprintf(stderr , "Cannot stat ");
        perror(fname);
        return -1;
    }
    *inum   = info.st_ino;
    *dev_id = info.st_dev;
    return 0;
}
```

This concludes the implementation of the `pwd` command.

3.15.8 Symbolic Links

Users should be able to create links across file systems, and they should also be able to make links to directories. For example, I might have a directory,

```
/data/c/sweiss/development/sourcecode/c-sources
```

that I access frequently while working on lecture notes. If my lecture notes are in `/data/c/sweiss/classes/notes`, then it would be convenient to have a directory

```
/data/c/sweiss/classes/notes/c-sources
```

that is just a link to the `c-sources` directory under `sourcecode`. Then if my working directory were `/data/c/sweiss/classes/notes`, I could type

```
$ ls c-sources
```

and it would display the contents of `/data/c/sweiss/development/sourcecode/c-sources`.

Similarly, users should not be limited in their ability to make links to files. The fact that the file system depends on the uniqueness of i-numbers within a file system should not limit a user's ability to make links that are convenient, even if they span file systems.

For these reasons, most UNIX systems provide an alternative kind of link called a *symbolic link* (or *soft link*). A symbolic link is a file that contains a reference to the name of the file to which it links. The command `ln -s` creates a symbolic link instead of a hard link. To illustrate, suppose we have a directory named `temp`, and inside it we run the commands

```
$ who > users
$ ln users whoson          # hard link
$ ln -s users ulist        # soft link
$ ls -1i
628 lrwxrwxrwx   1 sweiss  staff    5 Aug 30 00:22 ulist -> users
614 -rw-------   2 sweiss  staff  676 Aug 30 00:22 users
614 -rw-------   2 sweiss  staff  676 Aug 30 00:22 whoson
```

You can see a new notation. First, the "`l`" in the file type specifies that the file is a symbolic link. Second, the notation "`ulist -> users`" indicates that `ulist` is a symbolic link to the file `users`. The `symlink()` system call does the same thing.

Symbolic links are actual files. They have i-nodes, as demonstrated above. But the i-node contains different information than the i-node of a hard link, and it refers to the name of the file to which it is linked. In other words, the actual filename is stored in the link.

Symbolic links can be broken easily. If I now type

```
$ rm users
rm: remove users (yes/no)? y
$ ls -li
total 4
628 lrwxrwxrwx 1 sweiss student 5 Aug 30 00:22 ulist -> users
614 -rw------- 1 sweiss student 676 Aug 30 00:22 whoson
$ more ulist
ulist: No such file or directory
```

you see that `ulist` still points to `users`, even though it does not exist. This is not an error. The error only occurs when the process tries to access the file.

3.15.9 System Calls Related to Symbolic Links

The following system calls are related to the use of symbolic links.

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
int lstat(const char *file_name, struct stat *buf);
```

The system call that creates symbolic links is `symlink()`. The `readlink()` call obtains the name of the file to which a symbolic link is pointing. The `lstat()` call obtains a `stat` structure for a file that is a link, not the file to which it is linked.

3.16 Tree Walks

There are four different ways that we can visit all of the nodes in a given subtree of the hierarchy:

- Write our own recursive function to traverse the tree;
- Write a non-recursive function to traverse the tree;
- Use the `nftw()` POSIX library function;
- Use the `fts()` function available in systems including the 4.4BSD API.

Commands such as `grep`, `chmod`, `chown`, `rm`, `cp`, and `chgrp` use the `fts()` functions to perform their recursive tree traversals. The `GNU` version of the `ls` command, written by Richard Stallman and David MacKenzie, uses internal stacks and queues to recurse the tree. The `GNU find` command, originally written by Eric Decker, uses mutually recursive functions whereas the versions on BSD systems use the `fts()` functions. In this section we describe the functions provided by the libraries, beginning with the POSIX `nftw()` function and following with `fts()`.

3.16.1 The nftw() Tree Walk Function

`nftw()` replaces the older `ftw()` function, which still exists, but is deprecated. Its prototype is

```
#include <ftw.h>
int nftw(const char *path,
         int (*fn)(const char *, const struct stat *, int, struct FTW *),
         int fd_limit, int flags);
```

The `nftw()` function recursively descends the directory hierarchy rooted in `path`. For each object in the hierarchy, it calls the function pointed to by `fn`, passing it

- a pointer to a null-terminated character string containing the pathname of the object (e.g. a file's path from the root of the walk),
- a pointer to a `stat` structure containing information about the object, filled in as if `stat()` or `lstat()` had been called to retrieve the information,
- an integer that gives more information about the object, whose value is one of the following predefined constants:

`FTW_D`	The object is a directory.
`FTW_DNR`	The object is a directory that cannot be read. The `fn` function shall not be called for any of its descendants.
`FTW_DP`	The object is a directory and subdirectories have been visited. (This condition occurs if the `FTW_DEPTH` flag is included in `flags`.)
`FTW_F`	The object is a file.
`FTW_NS`	The `stat()` function failed on the object because of lack of appropriate permission. The `stat` buffer passed to `(*fn)` is undefined. Failure of `stat()` for any other reason is considered an error and `nftw()` returns -1.
`FTW_SL`	The object is a symbolic link. (This condition occurs if the `FTW_PHYS` flag is included in `flags`.)
`FTW_SLN`	The object is a symbolic link that does not name an existing file. (This condition shall only occur if the `FTW_PHYS` flag is not included in `flags`.)

- a pointer to an `FTW` structure, which is defined as

```
struct FTW {
     int base;
     int level;
};
```

The `FTW` structure gives syntactic information about the filename and depth information about its place in the search. Specifically, the value of `base` is the offset of the object's filename in the pathname passed as the first argument to `(*fn)`. This makes it possible to extract the filename from the pathname easily. The value of `level` indicates depth relative to the root of the walk, where the root level is 0. For example, if the pathname of the file is `documents/pictures/2012/january/nyc`, and `nyc` is the file object being processed by the call to `(*fn)`, then `base` would be the length of the string "`documents/pictures/2012/january/`" and `level` would be 4, since `nyc` is at level 4 in the tree rooted at `documents`.

The `(*fn)` function is a programmer-defined function that is called for every object that `ntfw()` visits in the tree. It can have any semantics provided that has the prototype described above. The function has access to the information returned by a call to `stat()` as well as information contained in the integer flag. It can be designed to extract and utilize this information, but the problem, which is a big one, is that the function has no hooks to pass in user-supplied data or pointers, so that it is not possible to use this function to modify any variables unless they are globally scoped. For example, to do something relatively simple such as computing the total number of bytes used by all objects in the subtree rooted at a given directory, and printing that value when it is totaled, we would either have to make it a static variable within `(*fn)` and do a post-order traversal, printing only when we return to the root directory, or declare the variable outside of the function with file scope and do any printing in the function that calls `nftw()` when the call terminates. It is expected that the programmer will use global variables when using this function. This will be clear when we look at the example code below. We will have more to say about this below.

The third parameter to the `nftw()` function itself is an integer `fd_limit` that sets the maximum number of file descriptors that should be used by `nftw()` while traversing the file tree. Only one descriptor is needed for each level of the tree. If `fd_limit` is smaller than the depth of the tree, then performance will be degraded because the function will have to keep opening and closing directories.

The last parameter is a flag consisting of a bitwise-OR of zero or more of the following constants, which control how `ntfw()` handles mount points and soft links and what it uses as its current working directory, and whether it follows a pre-order or post-order traversal of the tree. Specifically,

`FTW_CHDIR` If set, `nftw()` changes the current working directory to each directory as it reports files in that directory. If clear, `nftw()` does not change the current working directory.

`FTW_DEPTH` If set, `nftw()` reports all files in a directory before reporting the directory itself. If clear, `nftw()` reports any directory before reporting the files in that directory.

`FTW_MOUNT` If set, `nftw()` reports files only in the same file system as `path`.

`FTW_PHYS` If set, `nftw()` performs a physical walk and does not follow symbolic links. If it is clear, it follows symbolic links but does not visit any file twice. If `FTW_PHYS` is clear and `FTW_DEPTH` is set, `nftw()` follows soft links but does not report on any directory that would be a descendant of itself. If both `FTW_PHYS` and `FTW_DEPTH` are clear, `nftw()` follows soft links but does not report on the contents of any directory that would be a descendant of itself.

There is one other flag that not in the above list, `FTW_ACTIONRETVAL`, only available with `_GNU_SOURCE` set, which we ignore for now.

The `nftw()` function runs until the first of the following conditions occurs:

- an invocation of `(*fn)` returns a non-zero value, in which case `nftw()` returns that value;
- it detects an error other than `EACCES`, in which case it returns -1 and sets `errno` to indicate the error; or
- the tree is exhausted, in which case it returns 0.

As usual, we will look at an example to see how it is used. The following program displays the size and name of every file in the tree rooted at its argument. It indents the name in proportion to its depth in the tree, and prints the total size in bytes of all files visited. It lets the user control whether to cross mount points with the `-m` option, to do a post-order in stead of a pre-order with the `-d` option, and does not follow symbolic links unless the `-p` option is provided.

Listing nftwdemo.c

```
#define _XOPEN_SOURCE 500
#include <ftw.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <limits.h>

#define  TRUE  1
#define  FALSE 0
#define  MAXDEPTH    20

/* For getopt() we need to use this feature test macro */
#if  ( _POSIX_C_SOURCE >= 2 || _XOPEN_SOURCE )

static int display_info(const char *fpath, const struct stat *sb,
             int tflag, struct FTW *ftwbuf)
{
    char  blanks[PATH_MAX];
    const char *filename = fpath + ftwbuf->base;
    int    width          = 4*ftwbuf->level;

    /* fill blanks with a string of 4*level blanks */
    memset(blanks, ' ', width);
    blanks[width] = '\0';

    /* print out blanks then filename (not full path) */
    printf("%s%s", blanks, filename );

    /* Check flags and print a message if need be */
    if ( tflag == FTW_DNR )
        printf(" (unreadable directory)");
    else if ( tflag == FTW_SL )
        printf(" (symbolic link)" );
    else if ( tflag == FTW_SLN )
        printf(" (broken symbolic link)" );
    else if ( tflag == FTW_NS )
        printf("stat failed " );

    printf("\n");
```

```
    return 0;           /* To tell nftw() to continue */
}

int main(int argc, char *argv[])
{
    int flags = 0;
    int status;
    int ch;
    char options[] = ":cdpm";
    opterr = 0;  /* turn off error messages by getopt() */

    while  (TRUE) {
        /* call getopt, passing argc and argv and the options string */
        ch = getopt(argc, argv, options);

        /* it returns -1 when it finds no more options  */
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'c':
            flags |= FTW_CHDIR;
            break;
        case 'd':
            flags |= FTW_DEPTH;
            break;
        case 'p':
            flags |= FTW_PHYS;
            break;
        case 'm':
            flags |= FTW_MOUNT;
            break;
        default:
           fprintf (stderr, "Bad option found.\n");
           return 1;
        }
    }

    if (optind < argc) {
        while (optind < argc) {
            status = nftw(argv[optind], display_info,
                          MAXDEPTH, flags);
            if ( -1 == status ) {
               perror("nftw");
               exit(EXIT_FAILURE);
            }
            else
                optind++;
        }
    }
    else {
        status = nftw(".", display_info, MAXDEPTH, flags);
        if ( -1 == status ) {
           perror("nftw");
           exit(EXIT_FAILURE);
```

```
            }
        }
        exit(EXIT_SUCCESS);
}
#endif
```

Notes.

- The main program allows multiple filename arguments .
- The `display()` function uses `memset()` to set the string blanks to the correct number of blanks.
- To print the file name without the leading path, it prints the string starting at `fpath+ftwbuf->base`.
- It also displays information about whether the object is a symbolic link, broken or otherwise, or an unreadable directory, or if the `stat()` call failed on the object.

This was a warm-up exercise. In general, the `nftw()` is challenging to use if the task requires changing state information that must be preserved across calls to the `(*fn)` function. We must use variables that are either static and locally scoped or global. To see what the problem is, we will try to implement a simple version of the `du` command.

3.16.2 The `du` Command

The `du` command in its simplest form is invoked with

```
$ du file file ...
```

The `du` command estimates disk usage for each file it is given, and if any are directories, it does this recursively on each. In other words, when it is given a directory as an argument, it traverses the tree rooted at that directory, and for each directory that it visits, it prints its disk usage. The disk usage of a directory is the sum of the usage required for the directory file itself together with all files in its tree. It seems like an ideal candidate for a command to be implemented using `nftw()`, and a good exercise in using `nftw()` for a realistic application.

The `du` command has several options, but we will write a simple version of it that accepts no options. The actual `du` command by default displays the total number of blocks that each file uses. Rather than displaying block usage, ours will display the actual number of bytes. This is equivalent to the command `du -b`. Also, `du` by default does not display the usage of all files; to do that it needs the `-a` option. Therefore, we will write the equivalent of the command

```
$ du -ab file file ...
```

The du command should not follow symbolic links, otherwise it may double count files or count files that are not within the directory argument. For this first version, we will also disable crossing mount points, so that it measures usage only within a single file system. It is easy enough to provide an option to let it cross mount points.

Obviously it has to do a post-order traversal of the tree, because otherwise when it visits a directory it will not have the total usage of that directory's children. Thus, in Figure 3.10, assuming a left-to-right traversal of the children of a single node (since we can draw them in any order we wish, we can assume it is always left-to-right), the files visited will be `srcs`, `cpy`, `bin`, `pics`, `stuff`, `data`, `work`, `ideas`, `projects`, `file2`, `garbage`, `misc`, `A`. Henceforth it is convenient to assume that the children of a node are always visited in a left-to-right order.

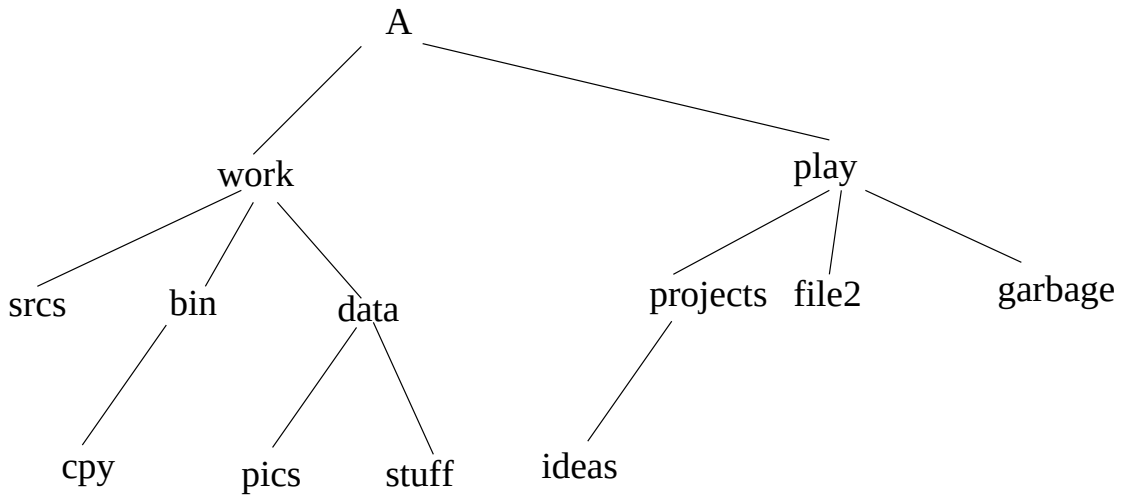

Figure 3.10: Sample tree hierarchy.

The interesting problem is how to recursively accumulate the sizes of the files that it visits. It has to be able to print out the size of each file that it visits, and when it reaches a directory, to print out that directory's total usage. Thus, in Figure 3.10, when it reaches data, it has to print out the sum of the sizes of `pics`, `stuff`, and `data`, but it also has to add the sizes of `srcs`, and the current accumulation in `bin` to a running total to pass up to `work` when it reaches it. This suggests that if we keep a set of running totals indexed by level in the tree, it should suffice to record total size, provided that we do this carefully.

Because the only way to share data between the main program and the function argument to `nftw()` is by making it global, we will declare the array

```
#define MAXDEPTH  50
static  uintmax_t totalsize[MAXDEPTH];
```

in file scope. `uintmax_t` is a type defined in `<stdint.h>`. It is the largest unsigned integer type available on the machine. It is often equivalent to `unsigned long long int`. We will assume that the depth of the tree is never greater than 50 in out implementation. This array must be made global because the main program zeroes it initially and the function that we pass to `nftw()`, which we will call `file_usage()`, must be able to modify it and access it. The prototype for this function is

```
int file_usage(const char *fpath, const struct stat *sb,
                  int tflag, struct FTW *ftwbuf)
```

At any instant of time, `file_usage()` will be visiting a specific file in the tree. Let us call this the current file, and its level, the current level, and let us use the variable `cur_level` to represent this. We call the level of the file processed immediately before the current file the previous level and we will use the variable `prev_level` to store that level. Both of these take on values up to `MAXDEPTH` and no larger. The current file has a size which we will store in the variable `cur_size`. This is the size of the actual file, not the sum of the sizes of any children it may have. Even directories have size – they are usually allocated a single block (4096 bytes these days on most machines). We can obtain this value from the `stat` structure passed into the function; it is `sb->st_size`.

The action that `file_usage()` must take depends on the values of `cur_level` and `prev_level` and nothing else. This is what we will now justify. The following invariant will be maintained by `file_usage()` after it has processed a file:

> `totalsize[cur_level]` is the sum of the sizes of all subtrees in the tree whose roots are at level `cur_level` and are siblings to the left of the currently processed node plus the size of the subtree rooted at the current node.

Suppose first that `prev_level < cur_level`. This implies that we just descended from a node closer to the root of the tree. There is only one way in which this can happen during a post-order traversal: when we reach a leaf node that is leftmost in its tree. For any other node, either the previous node will be at the same level or will be below it. Thus, we have just reached a bottom level and we must set `totalsize[cur_level]` to 0 and add the current file's size to it. Equivalently,

```
cur_size = sb->st_size;
totalsize[cur_level] = cur_size;
```

Notice that `totalsize[cur_level]` satisfies the invariant in this case.

Suppose instead that `prev_level == cur_level`. In this case we are visiting a file that is a sibling of the one previously visited and it cannot be a directory, otherwise the previous node would have been at a lower level. This implies that we have visited all siblings to the left of the current file and that it has no children. Therefore, we need to update `totalsize[cur_level]` by adding the file's size (i.e., `cur_size`) to it:

```
cur_size = sb->st_size;
totalsize[cur_level] += cur_size;
```

Assuming that the invariant was true prior to entering `file_usage()`, it remain true as a result of adding `cur_size` to it in this case since `cur_size` is the size of the subtree rooted at this file.

The last case to consider is when `prev_level > cur_level`. This can only mean one thing – we have just returned in the post-order traversal to a directory all of whose children have just been visited. It can only be the case that `prev_level == cur_level+1` but we do not need this fact to do what needs to be done. This is where the transfer of sizes between levels takes place. First, the size that we have to report for this file is not the directory size itself, but the directory size plus the sum of the sizes passed up the tree to each of its children.

We take advantage of the invariant regarding `totalsize[prev_level]`. What we know is that, since the last node processed was the rightmost node in the subtree rooted at the current directory, and its level is prev_level, `totalsize[prev_level]` must be the sum of the sizes of all subtrees of this directory. Therefore the size to display for it is its own size plus `totalsize[prev_level]`:

```
    cur_size    = totalsize[prev_level] + sb->st_size;
```

To maintain the invariant, we add to `totalsize[cur_level]` the new value of `cur_size`. You can verify that it will hold for the current level in so doing. The last step is the less obvious one, and it will need explanation. We must set `totalsize[prev_level]` to 0. All together, the actions in this case are

```
    cur_size    = totalsize[prev_level] + sb->st_size;
    totalsize[cur_level] += cur_size;
    totalsize[prev_level] = 0;
```

To see why we must zero `totalsize[prev_level]`, consider what will happen when `file_usage()` exits and is called for the next file. Using the file tree in Figure 3.10, suppose the current file is the directory `work`, and `file_usage()` just processed the directory named `data`. `cur_level` = 1 and `prev_level` = 2. The next file that `file_usage()` will process is `ideas`, and then `projects`. After it visits `ideas` and returns to `projects`, `totalsize[1]` must start with the value 0, otherwise the size of `projects` will include the sizes of `srcs`, `bin`, and `data`. In other words, every time that we finish a row of siblings in a subtree, having reached the rightmost sibling, and return to the parent, we must zero out the entry in the `totalsize[]` array for that level. The only chance to do this is when we have recorded its value in the parent and are finished with that node. Doing this preserves the invariant, as no nodes are currently being visited in that level anymore.

We put all of this together in the following listing, which is our initial version of a simple `du` command. Our command will print a message next to a file name if that file had a problem such as being a broken link or an unreadable directory. We could do instead what the real `du` does and print the message to the standard error stream. Comments are omitted to reduce the size of the listing.

Listing simpledu.c

```
#define _XOPEN_SOURCE 500
#include <ftw.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <limits.h>

#define  TRUE  1
#define  FALSE 0
#define  MAXDEPTH  200

/* For getopt() we need to use this feature test macro */
#if  ( _POSIX_C_SOURCE >= 2 || _XOPEN_SOURCE )

static  uintmax_t totalsize[MAXDEPTH];

int file_usage(const char *fpath, const struct stat *sb,
                int tflag, struct FTW *ftwbuf)
{
    static int prev_level = -1;
    int        cur_level;
```

```
    uintmax_t  cur_size;

    cur_level = ftwbuf->level;
    if ( cur_level >= MAXDEPTH ) {
        fprintf(stderr, "Exceeded maximum depth.\n");
        return -1;
    }

    if ( prev_level == cur_level ) {
        cur_size = sb->st_size;
        totalsize[cur_level] += cur_size;
    }
    else if ( prev_level > cur_level ) {
        cur_size    = totalsize[prev_level] + sb->st_size;
        totalsize[cur_level] += cur_size;
        totalsize[prev_level] = 0;
    }
    else {
        cur_size = sb->st_size;
        totalsize[cur_level] = cur_size;
    }

    printf("%ju\t%s", cur_size, fpath);
    prev_level = cur_level;

    if ( tflag == FTW_DNR )
        printf(" (unreadable directory)");
    else if ( tflag == FTW_SL )
        printf(" (symbolic link)" );
    else if ( tflag == FTW_SLN )
        printf(" (broken symbolic link)" );
    else if ( tflag == FTW_NS )
        printf("stat failed " );
    printf("\n");

   return 0;
}

int main(int argc, char *argv[])
{
    int flags = FTW_DEPTH | FTW_PHYS | FTW_MOUNT;
    int status;
    int i = 1;

    if ( argc < 2 )  {
        memset( totalsize, 0, MAXDEPTH*sizeof(uintmax_t));
        status = nftw(".", file_usage, 20, flags);
        if ( -1 == status ) {
           fprintf(stderr, "nftw exited abnormally.\n");
           exit(EXIT_FAILURE);
        }
    }
    else
        while (i < argc) {
```

```
            memset( totalsize, 0, MAXDEPTH*sizeof(uintmax_t));
            status = nftw(argv[i], file_usage, MAXDEPTH, flags);
            if ( -1 == status ) {
               fprintf(stderr, "nftw exited abnormally.\n");
               exit(EXIT_FAILURE);
            }
            else {
                i++;
            }
        }
    exit(EXIT_SUCCESS);
}
#endif
```

If you run this and compare the output to `du -ab`, you might discover that it is not always the same. But try comparing it to the output of `du-alb` and you will see it is the same. Consult the manpage and you will see that the `-l` option tells `du` to count sizes multiple times for files that have multiple links. What is the problem?

This version of `du`, as it stands, counts files with multiple links as many times as they have links. If a file has two names in two different subdirectories of our root directory, each will be counted. What is the way to prevent this? The `stat` structure has both the i-number of the file and the number of links. We can inspect the number of links for non-directory files. If it is greater than one, then it has another name somewhere. We do not know where, but we can store the i-number in a lookup table and set its count to the number of links-1. Each time that we find a file whose link count is greater than one, we can look up its i-number in the table. If it is there with a positive link count, we decrement the link count in the table and do not add the file's size to the running total, and do not even print its name. If the link count reaches zero, we remove it from the table. If the file is not there, we add it to the table with its link count.

Exercise 1. Write an implementation of `du` that does not count files with multiple links more than once.

Exercise 2. Write an implement ion of `du` that has the option to count either 512-byte blocks or bytes.

3.16.3 The `fts` File Hierarchy Traversal Functions

Unlike `nftw()`, `fts` is a family of functions, in much the same way that `opendir()`, `readdir()`, `rewinddir()`, and `closedir()` are inter-related functions, which conform to 4.4BSD but are not POSIX functions. The `fts` set of functions includes `fts_open()`, `fts_read()`, `fts_set()`, `fts_children()`, and `fts_close()`. Just as `opendir()` creates a directory stream object and returns a pointer to it, `fts_open()` creates a *handle* that is used by the other functions. A *handle* is a pointer to an `FTS` structure. Unlike `nftw()`, which does not allow the application to control the order in which files are searched other than whether it is pre-order or post-order, `fts` allows the calling program to specify this order. We begin by looking at the manpage for `fts`. The synopsis is as follows:

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

```
FTS *fts_open(char * const *path_argv, int options,
        int (*compar)(const FTSENT **, const FTSENT **));
FTSENT *fts_read(FTS *ftsp);
FTSENT *fts_children(FTS *ftsp, int options);
int     fts_set(FTS *ftsp, FTSENT *f, int options);
int     fts_close(FTS *ftsp);
```

The paradigm for using `fts` is that we begin by calling `fts_open()`, passing an array of strings representing the roots of trees that we wish to traverse, an integer encoding options, and a function to be used for determining the order in which files are visited. It returns a handle, i.e., a pointer to an `FTS` structure that is then passed to `fts_read()`. Each time that `fts_read()` is called, it visits a file. Each file in the tree is visited just once, except for directories, which are visited before and after their children. `fts_read()` returns a pointer to an `FTSENT` structure for each file that it visits. Files and `FTSENT` structures are in one-to-one correspondence. `FTSENT` structures have a member that allows them to be linked together; the function `fts_children()` returns a pointer to a linked list of these, representing all of the children in a directory, exactly which will be explained below. The fts_set() function allows a file to be reprocessed after it has been returned by a call to `fts_read()`. When one is completely finished processing the directory tree passed to `fts_open()`, it should be closed with `fts_close()`.

There is no need to know the internal structure of a `DIR` structure because the application should treat it opaquely. However, we do need to know the content of the `FTSENT` structure, because that is what characterizes each visited file. The two structures are defined in the header file `<fts.h>`, found in the standard directory, `/usr/include`. Their definitions (omitting some macros here) are

```
typedef struct {
struct _ftsent *fts_cur;    /* current node */
struct _ftsent *fts_child;  /* linked list of children */
struct _ftsent **fts_array; /* sort array */
dev_t fts_dev;              /* starting device # */
char *fts_path;             /* path for this descent */
int fts_rfd;                /* fd for root */
int fts_pathlen;            /* sizeof(path) */
int fts_nitems;             /* elements in the sort array */
int (*fts_compar) (const void *, const void *); /* compare fn */
int fts_options;            /* fts_open options, global flags */
} FTS;
typedef struct _ftsent {
    unsigned short fts_info;     /* flags for FTSENT structure */
    char          *fts_accpath;  /* access path */
    char          *fts_path;     /* root path */
    short          fts_pathlen;  /* strlen(fts_path) */
    char          *fts_name;     /* filename */
    short          fts_namelen;  /* strlen(fts_name) */
    short          fts_level;    /* depth (-1 to N) */
    int            fts_errno;    /* file errno */
    long           fts_number;   /* local numeric value */
```

```
        void          *fts_pointer;  /* local address value */
        struct ftsent *fts_parent;   /* parent directory */
        struct ftsent *fts_link;     /* next file structure */
        struct ftsent *fts_cycle;    /* cycle structure */
        struct stat   *fts_statp;    /* stat(2) information */
    } FTSENT;
```

The `FTSENT` structure has many members, and they all need to be understood to use the `fts` functions to their maximum extent, but for relatively simple applications it is not necessary to understand them all. The most important members, with brief descriptions are

`fts_info` An integer that encodes information about the type of object represented by this structure.

`fts_accpath` A path for accessing the file from the current directory.

`fts_path` The path for the file relative to the root of the traversal. This path contains the path specified to `fts_open()` as a prefix.

`fts_name` The file name.

`fts_errno` Upon return of a `FTSENT` structure from the `fts_children()` or `fts_read()` functions, with its `fts_info` field set to `FTS_DNR`, `FTS_ERR` or `FTS_NS`, the `fts_errno` field contains the value of the external variable errno specifying the cause of the error. Otherwise, the contents of the `fts_errno` field are undefined.

`fts_number` This field is provided for the use of the application program and is not modified by the `fts` functions. It is initialized to 0.

`fts_pointer` This field is provided for the use of the application program and is not modified by the `fts` functions. It is initialized to `NULL`.

`fts_parent` A pointer to the `FTSENT` structure referencing the file in the hierarchy immediately above the current file, i.e. the directory of which this file is a member. A parent structure for the initial entry point is provided as well, however, only the `fts_level`, `fts_number` and `fts_pointer` fields are guaranteed to be initialized.

`fts_link` Upon return from the `fts_children()` function, the `fts_link` field points to the next structure in the `NULL`-terminated linked list of directory members. Otherwise, the contents of the `fts_link` field are undefined.

`fts_statp` A pointer to a `stat` structure for the file.

The `fts_info` field can have any of the following values:

`FTS_D` A directory being visited in pre-order.

`FTS_DC` A directory that causes a cycle in the tree. (The `fts_cycle` field of the `FTSENT` structure will be filled in as well.)

`FTS_DEFAULT` Any `FTSENT` structure that represents a file type not explicitly described by one of the other `fts_info` values.

`FTS_DNR` A directory which cannot be read. This is an error return, and the fts_errno field will be set to indicate what caused the error.

`FTS_DOT` A file named "." or ".." which was not specified as a file name to `fts_open()` (see `FTS_SEEDOT`).

`FTS_DP` A directory being visited in post-order. The contents of the `FTSENT` structure will be unchanged from when it was returned in pre-order, i.e. with the `fts_info` field set to `FTS_D`.

`FTS_ERR` This is an error return, and the `fts_errno` field will be set to indicate what caused the error.

`FTS_F` A regular file.

`FTS_NS` A file for which no `stat` information was available. The contents of the `fts_statp` field are undefined. This is an error return, and the `fts_errno` field will be set to indicate what caused the error.

`FTS_NSOK` A file for which no `stat` information was requested. The contents of the `fts_statp` field are undefined.

`FTS_SL` A symbolic link.

`FTS_SLNONE` A symbolic link with a non-existent target. The contents of the `fts_statp` field reference the file characteristic information for the symbolic link itself.

Some immediate observations from the above information are that:

- The `fts_info` member has information similar to that found in the integer passed to the `(*fn)` function by `nftw()`.
- Unlike the `nftw()` function, `fts` provides hooks for the application to use. In particular, `fts_number` and `fts_pointer` are two members of the returned structure that can be used for application specific data, making it possible to change state and data among different invocations of the `fts_read()` function.
- The `fts_parent` field provides a means to access the parent node, unlike `ntfw()`.
- It has `stat` information for the returned file in the `fts_statp` member, unless `fts_info` is either `FTS_NS` or `FTS_NSOK`.
- The name of the file is in the `fts_name` member. The `fts_path` member has the pathname of the file relative to the root of the search. For all practical purposes, this is the same path as `fts_accpath`.

It is also to important to know that a single buffer is used for the various path members of all `FSTENT` structures of all files in file hierarchy. Because of this, there is no guarantee that the `fts_path` and `fts_accpath` members of a previously returned `FTSENT` structure are still properly null-terminated, because they might have been written over already. Only the `fts_path` and `fts_accpath` members of the file most recently returned by `fts_read()`.are guaranteed to be null-terminated. To use these fields to reference any files represented by other `FTSENT` structures will require that the path buffer

be modified using the information contained in that `FTSENT` structure's `fts_pathlen` field. Any such modifications should be undone before further calls to `fts_read()` are attempted. In contrast, the `fts_name` field is always null-terminated.

Our first example illustrates the basic concepts. It is derived from a program from *RosettaCode.org*. The program, named `ftsdemo`, is given the name of a directory and a shell-style regular expression, enclosed in single quotes to prevent shell expansion of it, and it displays the names of all files in the given directory's hierarchy that match the expression. For example, `'*.c'` will cause all files ending in `'.c'` to be matched. The listing and an explanation follow.

Listing ftsdemo.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <err.h>
#include <errno.h>
#include <fnmatch.h>
#include <fts.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/*****************************************************************************
                            Utility Function Prototypes
*****************************************************************************/
/** usage()
 *  Print usage message on standard output
 */
void usage(char* progname);

/** entcmp()
 * Compare files by name.
 */
int entcmp(const FTSENT **s1, const FTSENT **s2);

/** pmatch()
 * Print all files in the directory tree that match the glob pattern.
 * Example: pmatch("/usr/src", "*.c");
 */
void pmatch(char *dir, const char *pattern);

/*****************************************************************************
                                Main Program
*****************************************************************************/
int main(int argc, char* argv[])
{
    if ( argc < 3 )  {
        usage(argv[0]);
        exit(1);
    }
    pmatch(argv[1], argv[2]);
    return 0;
}
```

```
/*****************************************************************************/
int entcmp(const FTSENT **s1, const FTSENT **s2)
{
    return (strcoll((*s1)->fts_name, (*s2)->fts_name));
}


/*****************************************************************************/
void usage(char* progname)
{
    printf("usage: %s directory pattern\n", progname);
}

/*****************************************************************************/

void pmatch(char *dir, const char *pattern)
{
    FTS    *tree;   /* pointer to file tree stream returned by fts_open */
    FTSENT *f;      /* pointer to structure returned by fts_read */
    char *argv[] = { dir,  NULL };

    /* Call fts_open(0 with FTS_LOGICAL to follow symbolic links
     * including links to other directories. Since it detects cycles,
     * we do not have to worry about infinite loops.
     */
    tree = fts_open(argv, FTS_LOGICAL , entcmp);
    if (tree == NULL)
        perror("fts_open");

    /* Repeatedly get next file, skipping "." and ".." because
     * FTS_SEEDOT was not set.
     */
    while ((f = fts_read(tree))) {
        switch (f->fts_info) {
        case FTS_DNR:  /* Cannot read directory */
            fprintf(stderr, "Could not read %s\n", f->fts_path);
            continue;
        case FTS_ERR:  /* Miscellaneous error */
            fprintf(stderr, "Error on %s\n", f->fts_path);
            continue;
        case FTS_NS:  /* stat() error */
            /* Show error, then continue to next files. */
            fprintf(stderr, "Could not stat %s\n", f->fts_path);
            continue;
        case FTS_DP:
          /* Returned to directory for second time as part of
              post-order visit to directory, so skip it. */
            continue;
        }

        /*
         * Check if the name matches pattern, and if so, print out its
         * path. This check uses FNM_PERIOD, so "*.c" will not
         * match ".invisible.c".
```

```
             */
            if (fnmatch(pattern, f->fts_name, FNM_PERIOD) == 0)
                printf("%s\n", f->fts_path);

            /*
             * A cycle happens when a symbolic link (or perhaps a
             * hard link) puts a directory inside itself. Tell user
             * when this happens.
             */
            if (f->fts_info == FTS_DC)
                fprintf(stderr, "%s: cycle in directory tree",
                        f->fts_path);
        }

        /* fts_read() sets errno = 0 unless it has an error. */
        if (errno != 0)
          perror("fts_read");

        if (fts_close(tree) < 0)
          perror("fts_close");
}
```

The main program does very little; it checks usage and if the usage is correct, it calls `pmatch()`, passing the name of the directory and the expression to be matched.

`pmatch()` begins by constructing a proper first argument for the call to `fts_open()`. The first argument must be a `NULL`-terminated list of directory names. If we want to traverse just a single directory, we therefore have to create a list of the form `{dirname, NULL}`, where `dirname` is a `NULL`-terminated string.

It then calls `fts_open()`, setting the `FTS_LOGICAL` flag so that it can follow symbolic links. A nice feature of `fts` is that it detects when it is about to complete a cycle caused by symbolic links, and it sets the `fts_info` member of the returned structure to `FTS_DC` in this case, so that the application can handle it. The third argument to `fts_open()` is a pointer to the comparison function that will be used for sorting the files. If a `NULL` pointer is supplied, the directory traversal order is in the order listed in the `argv` array passed as argument one for the root paths, and in the order listed in each directory for everything else.. The function prototype must be

```
int compare(const FTSENT **, const FTSENT **));
```

and it must return a negative value, zero, or a positive value to indicate if the file referenced by its first argument comes before or after, the file referenced by its second argument. The `fts_accpath`, `fts_path` and `fts_pathlen` members of the `FTSENT` structures may never be used in this comparison. If the `fts_info` member is set to `FTS_NS` or `FTS_NSOK`, the `fts_statp` field may not be used either. If `fts_open()` fails, it returns a `NULL` pointer.

In this program, files will be sorted by the default collating order, which is accomplished by calling `strcoll()` with the two file names:

```
int entcmp(const FTSENT **s1, const FTSENT **s2)
{
    return (strcoll((*s1)->fts_name, (*s2)->fts_name));
}
```

After getting the `FTS` pointer, tree, it uses this to repeatedly call `fts_read()`. For each file that it visits, `fts_read()` returns a pointer to an `FTSENT` structure as long as there are files remaining and no error occurred. If an error occurred it returns `NULL` and sets `errno` to some error value, and if there are no more files left, it returns `NULL` and sets `errno` to zero. The program checks the `fts_info` member of the structure to determine whether an error occurred, and if it matters, whether it is a directory or a file. In this case it does not matter what type of file it is, but only whether some type of error occurred. If no error occurred, it calls the `fnmatch()` library function, whose prototype is

```
#include <fnmatch.h>
int fnmatch(const char *pattern, const char *string, int flags);
```

The `fnmatch()` function checks whether its string argument matches its pattern argument, which must be a valid shell pattern, and returns zero if it matches, `FNM_NOMATCH` if it does not match, and some other value if there is an error. The third argument can be used to pass various flags that control how it behaves. In our program, we want the shell pattern to explicitly use a period character if the file is supposed to have a leading period. For example, the pattern `'*.h'` should not match the file `.foo.h`. The flag `FNM_PERIOD` tells `fnmatch()` to match only if that period is the first character in the string. Every file that matches the pattern is printed on standard output.

When the loop terminates, it checks `errno` to see if there was an error, and it calls `fts_close()` to cleanup and release resources.

This program was relatively simple. We now show a slightly more interesting application of `fts` that uses the `fts_children()` function and two different methods of ordering the tree traversal.

As noted above, the `fts_children()` function returns a pointer to a linked list of `FTSENT` structures that represent the files of a single directory in the hierarchy. The burning question is, which directory? The answer is not simple. If `fts_read()` was never called at all after `fts_open()` was called, then `fts_children()` returns a pointer to the list of top-level directories passed to `fts_open()` in its `argv` array. If `fts_read()` was called prior to calling `fts_children()` and the last object returned by a call to `fts_read()` was a directory, then it is this directory whose children are delivered in the linked list returned by `fts_children()`. But if the last call to `fts_read()` returned a non-directory file, then `fts_children()` returns a `NULL` pointer and sets `errno` to zero. This return value is indistinguishable from what it will do if the last object returned by a call to `fts_read()` was an empty directory. In this case also, `fts_children()` will return a `NULL` pointer and set `errno` to zero. So this final state implies that either the last object was an empty directory or a non-directory file. In either case it has no children. Lastly, if an error condition arises, `fts_children()` will return a `NULL` pointer but set `errno` to some non-zero value.

To demonstrate the use of `fts_children()`, we will use it to implement our `ls` command once more, but this time we will allow the user to specify a few different orderings in which to list the files. Specifically, the user can list by collating order, by time of last modification, or by file size, in all cases ascending. To make this possible, the program will implement three different `compare()` functions; which one is used will depend on the command line option supplied by the user. The program's usage is

```
ls_fts [-l] [-m | -s ] [file file ...]
```

The `-m` option sorts by modification time, `-s`, by size, and `-l` turns on long listings. The program listing below omits the helper functions already displayed in previous versions of `ls`, as well as almost all comments.

```
Listing ls_fts.c
#include <sys/types.h>
#include <err.h>
#include <errno.h>
#include <fnmatch.h>
#include <fts.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include "utils.h"

#define  BYTIME  1
#define  BYNAME  2
#define  BYSIZE  3
#define  HERE    "."
#define  TRUE    1
#define  FALSE   0

int entcmp(const FTSENT **a, const FTSENT **b)
{
    return strcoll((*a)->fts_name, (*b)->fts_name);
}

int mtimecmp(const FTSENT **s1, const FTSENT **s2)
{
    return (int) (((*s1)->fts_statp)->st_mtime - ((*s2)->fts_statp)->st_mtime);
}

int szcmp(const FTSENT **s1, const FTSENT **s2)
{
    return (int) (((*s1)->fts_statp)->st_size - ((*s2)->fts_statp)->st_size);
}

void ls  ( char dir[], int do_longlisting, int sortflag )
{
    FTS *tree;
    FTSENT *f;
    char *argv[] = { dir, NULL };

    switch ( sortflag ) {
    case BYTIME:
        tree = fts_open(argv, FTS_LOGICAL , mtimecmp);
        break;
    case BYNAME:
        tree = fts_open(argv, FTS_LOGICAL , entcmp);
        break;
```

```
    case BYSIZE:
        tree = fts_open(argv, FTS_LOGICAL , szcmp);
        break;
    }

    if (tree == NULL)
        perror("fts_open");

    f = fts_read(tree);
    if ( NULL == f ) {
        perror("fts_read");
        return;
    }

    f = fts_children(tree, 0);
    if ( NULL == f )
        if ( errno != 0 )
            perror("fts_children");
        else
            fprintf(stderr, "empty directory\n");

    while ( f != NULL ) {
        switch (f->fts_info) {
        case FTS_DNR:  /* Cannot read directory */
            fprintf(stderr, "Could not read %s\n", f->fts_path);
            continue;
        case FTS_ERR:  /* Miscellaneous error */
            fprintf(stderr, "Error on %s\n", f->fts_path);
            continue;
        case FTS_NS:  /* stat() error */
            fprintf(stderr, "Could not stat %s\n", f->fts_path);
            continue;
        case FTS_DP:
          /* Returned to directory for second time as part of
             post-order visit to directory, so skip it. */
            continue;
        }
        if ( do_longlisting )
            print_file_status(f->fts_name, f->fts_statp);
        else
            printf("%s\n", f->fts_name);

        f = f->fts_link;
    }
    if (errno != 0)
      perror("fts_read");

    if (fts_close(tree) < 0)
      perror("fts_close");
}

/**********************  Main Program ***********************************/

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

```
{
    int longlisting = 0;
    int howtosort   = BYNAME;
    int ch;
    char options[] = ":lms";
    opterr = 0;

    while  (TRUE) {
        ch = getopt(argc, argv, options);
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'l':
            longlisting = 1;
            break;
        case 'm':
            if ( howtosort != BYSIZE )
                howtosort = BYTIME;
            else {
                printf("usage: %s [-l] [-m|-s] [files]\n", argv[0]);
                return 1;
            }
            break;
        case 's':
            if ( howtosort != BYTIME )
                howtosort = BYSIZE;
            else {
                printf("usage: %s [-l] [-m|-s] [files]\n", argv[0]);
                return 1;
            }
            break;

        case '?' :
            printf("Illegal option ignored.\n");
            break;
        default:
           printf ("?? getopt returned character code 0%o ??\n", ch);
           break;
        }
    }

    if ( optind == argc )     /* no arguments; use . */
        ls( HERE, longlisting, howtosort );
    else
        /* for each command line argument, display file */
        while ( optind < argc ) {
            ls( argv[optind], longlisting, howtosort );
            optind++;
        }
    return 0;
}
```

Notes.

- The main program does the option parsing and prevents the user from choosing to sort by both size and modification time, because sorting by both implies having to resort the linked list returned by `fts_children()`.
- The main program calls `ls()` for each command line argument rather than assembling them into an array of strings, because `ls()` is designed to display a single directory's contents only.
- The `mtimecmp()` and `szcmp()` functions use arithmetic rather than conditional evaluation. This makes them faster, but on a non-POSIX system, the time subtraction may not work. Both work by accessing the `stat` structure pointed to by the `FTSENT` structure's `fts_statp` member. No check is made that it is `NULL`.
- `ls()` begins by calling `fts_open()`. It then calls `fts_read()` so that the directory object will be the last object read by a call to `fts_read()`, ensuring that `fts_children()` will return a pointer to the list of children of that directory, if it is non-empty.
- Unlike the previous program, this does not repeatedly call `fts_read()`. It traverses the linked list of `FTSENT` structures until it reaches a `NULL` link and then terminates.

This is just a short overview of the `fts` functions. It does not show how to use the hooks provided to the application inside the `FTSENT` structure.

Exercise. A good exercise would be to implement the `du` command using fts, taking advantage of the members of the `FTSENT` structure available to the application, and the parent pointers.

Appendix A

A.1 Useful Command-Line Options for ls

Some of the most useful command-line options for ls are listed below, with brief descriptions.

Option	Description
`-a`	show dot-files (i.e. hidden files)
`-l`	display a long listing, with file type, permissions, number of links, owner, group, size in bytes, time of last modification
`-lu`	same as `ls -l`, except uses the last time the file was read instead of modified.
`-s`	show the file size in blocks
`-t`	sort the files by timestamp specified (`u` for access, modification by default)
`-F`	show file types
`-i`	show the i-number of the file
`-d`	if `ls` has a directory argument, show the attributes of the directory, instead of its contents
`-R`	recursively descend any directories or their subdirectories, applying any other options listed

As you can see, several of these options can be extremely useful when searching for things in the file system. A "dot-file" is not a file filled with dots. It is a file whose name begins with a dot, such as "`.bashrc`" or "`.login`" or "`.evolution`". You read the first as "dot-bash-R-C", and the second as "dot-login". Dot-files are thought of as hidden files because the plain `ls` command does not display files whose names begin with a dot, and `ls` is the only command to view the content of a directory. Even in desktop environments, the default settings of a browser like Nautilus will hide these files. In your home directory, if you type "`ls -a`" you will see all of the files and directories whose names begin with a dot. In this list you will see "`.`" and "`..`". Recall that these refer to the directory itself and its parent. Note that unlike Windows, the UNIX kernel has no concept of a hidden file.

A.1.1 Bit Masks

Suppose we want to extract the value of bit 8 of some 16-bit number named `flags`. If we perform a bitwise AND of `flags` and the binary number that has 0's everywhere except in bit 8, called `mask8` here, as illustrated:

0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	`mask8`
b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	`flags`
0	0	0	0	0	0	0	b_8	0	0	0	0	0	0	0	0	`mask8 & flags`

then the result, `mask8 & flags`, will have 0's everywhere and the value of bit 8 of `flags` in bit 8. Therefore `mask8 & flags == 0` if and only if bit 8 in `flags` is 0. I can represent the value of this 16-bit mask variable as the octal number `000400`. Octal numbers in C and C++ must begin with a leading 0, so this would be written `0000400`, and the bitwise AND would be written as

```
flags & 0000400
```

in a program. We can then test the bit and take action accordingly with code like

```
if (flags & 0000400 ) ...
```

A.2 The `find` Command

In addition to the commands you have seen in previous chapters that can recursively traverse the file hierarchy, there are a few UNIX commands that are designed to traverse the entire tree at a given directory, without having to specify any special recursive option. The most useful of these is the `find` command. The `du` command also does this:

`find` searches for files in a tree structure rooted at the given directory.

`du` displays the disk usage of all files in a given directory tree

The find command is a very powerful command. its basic usage is

```
find [options] path ... expression
```

where `path ...` means one or more directories, and `expression` is composed of options, tests, and actions joined by various operators. Unless the expression explicitly prunes the tree with the use of the `-prune` option, the entire tree rooted at each directory argument will be traversed. Options are expressions that always return true. Actions are expressions that can return true or false. The operators that combine expressions are boolean operators, `and`, `or`, `not`, and a list operator. If no operator is present between expressions, operator `and` is used instead. The best way to see how `find` can be used is by some examples.

```
$ find . -name '*.c' -print
```

searches for all files in the tree rooted at '.' matching the pattern `*.c`, printing out the matching pathnames. The `-print` action can be omitted because `-print` is the default action. The `-name` test will return true if a file matches the given pattern. When the test is true, the next action will be applied, which is how the file name is printed.

```
$ find . -amin -30
```

searches for all files in the tree rooted at '.' that have been accessed within the past 30 minutes and prints them.

```
$ find . -mmin -120
```

searches for all files in the tree rooted at '.' that have been modified within the past 120 minutes and prints them.

```
find / -links +1 ! -type d
```

searches through the entire file system looking for all files whose link count is greater than 1 and which are not directories (`! -type d`).

```
find . -samefile myfile
```

searches for all files that are hard links to `myfile`

```
find / -name core -exec /bin/rm -f '{}' \;
```

searches for all files named `core` in the file hierarchy and runs `/bin/rm -f` on each one, which deletes them. The notation `{}` means the currently matched file name. To protect the braces from being interpreted by the shell, they are enclosed in single quotes. The command that find runs must be terminated by a semi-colon, but again it must be escaped so that the shell does not interpret it as the end of the find command itself. We could write `\;` or `';'`.

```
find . -perm /o=w
```

finds all files in the current directory and below that others can write.

This is just a handful of search options that are possible. There are options to test file size, check ownership. permissions, group ownership, compare two files, and so on. It is well worth learning how to use this command.