4.1 THE FILE

The **file** is a container for storing information. As a first approximation, we can treat it simply as a sequence of characters. If you name a file `foo` and write three characters `a`, `b` and `c` into it, then `foo` will contain only the string `abc` and nothing else. Unlike the old DOS files, a UNIX file doesn't contain the eof (end-of-file) mark. A file's size is not stored in the file, nor even its name. All file attributes are kept in a separate area of the hard disk, not directly accessible to humans, but only to the kernel.

UNIX treats directories and devices as files as well. A directory is simply a folder where you store filenames and other directories. All physical devices like the hard disk, memory, CD-ROM, printer and modem are treated as files. The shell is also a file, and so is the kernel. And if you are wondering how UNIX treats the main memory in your system, it's a file too!

So we have already divided files into three categories:

- *Ordinary file*—Also known as *regular file*. It contains only data as a stream of characters.
- *Directory file*—It's commonly said that a directory contains files and other directories, but strictly speaking, it contains their *names* and a number associated with each name.
- *Device file*—All devices and peripherals are represented by files. To read or write a device, you have to perform these operations on its associated file.

There are other types of files, but we'll stick to these three for the time being. The reason why we make this distinction between file types is that the significance of a file's attributes often depends on its type. Read permission for an ordinary file means something quite different from that for a directory. Moreover, you can't directly put something into a directory file, and a device file isn't really a stream of characters. While many commands work with all types of files, some don't. For a proper understanding of the file system you must understand the significance of these files.

4.1.1 Ordinary (Regular) File

An **ordinary file** or **regular file** is the most common file type. All programs you write belong to this type. An ordinary file itself can be divided into two types:

- Text file
- Binary file

A **text file** contains only printable characters, and you can often view the contents and make sense out of them. All C and Java program sources, shell and **perl** scripts are text files. A text file contains lines of characters where every line is terminated with the *newline* character, also known as *linefeed* (LF). When you press *[Enter]* while inserting text, the LF character is appended to every line. You won't see this character normally, but there is a command (**od**) which can make it visible.

A **binary file**, on the other hand, contains both printable and unprintable characters that cover the entire ASCII range (0 to 255). Most UNIX commands are binary files, and the object code and executables that you produce by compiling C programs are also binary files. Picture, sound and video files are binary files as well. Displaying such files with a simple **cat** command produces unreadable output and may even disturb your terminal's settings.

4.1.2 Directory File

A **directory** contains no data, but keeps some details of the files and subdirectories that it contains. The UNIX file system is organized with a number of directories and subdirectories, and you can also create them as and when you need. You often need to do that to group a set of files pertaining to a specific application. This allows two or more files in separate directories to have the same filename.

A directory file contains an entry for every file and subdirectory that it houses. If you have 20 files in a directory, there will be 20 entries in the directory. Each entry has two components:

- The filename.
- A unique identification number for the file or directory (called the *inode number*).

If a directory `bar` contains an entry for a file `foo`, we commonly (and loosely) say that the directory `bar` contains the file `foo`. Though we'll often be using the phrase "contains the file" rather than "contains the filename", you must not interpret the statement literally. *A directory contains the filename and not the file's contents.*

You can't write a directory file, but you can perform some action that makes the kernel write a directory. For instance, when you create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (inode number and filename) associated with the file.

Note: The name of a file can only be found in its directory; the file itself doesn't contain its own name or any of its attributes, like its size or access rights.

4.1.3 Device File

You'll also be printing files, installing software from CD-ROMs or backing up files to tape. All of these activities are performed by reading or writing the file representing the device. For instance, when you restore files from tape, you read the file associated with the tape drive. It is advantageous to treat devices as files as some of the commands used to access an ordinary file also work with device files.

Device filenames are generally found inside a single directory structure, `/dev`. A device file is indeed special; it's not really a stream of characters. *In fact, it doesn't contain anything at all.* You'll soon learn that every file has some attributes that are not stored in the file but elsewhere on disk. The operation of a device is entirely governed by the attributes of its associated file. The kernel identifies a device from its attributes and then uses them to operate the device.

Now that you understand the three types of files, you shouldn't feel baffled by subsequent use of the word in the book. The term "file" will often be used in this book to refer to any of these types, though it will mostly be used to mean an ordinary file. The real meaning of the term should be evident from its context.

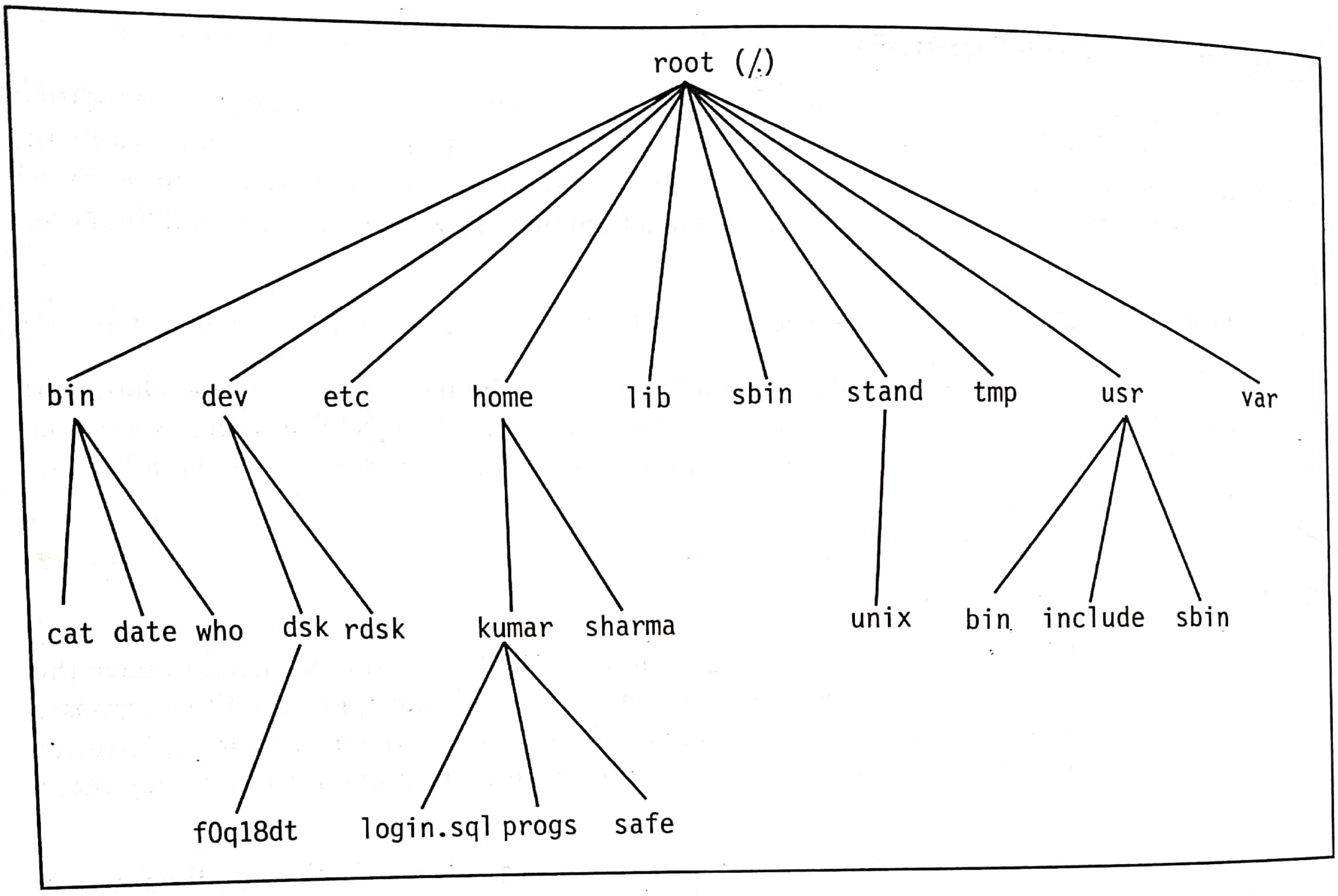

Fig. 4.1 The UNIX File System Tree

4.9 ABSOLUTE PATHNAMES

Many UNIX commands use file and directory names as arguments, which are presumed to exist in the current directory. For instance, the command

```
cat login.sql
```

will work only if the file `login.sql` exists in your current directory. However, if you are placed in `/usr` and want to access `login.sql` in `/home/kumar`, you can't obviously use the above command, but rather the pathname of the file:

```
cat /home/kumar/login.sql
```

As stated before, if the first character of a pathname is /, the file's location must be determined with respect to root (the first /). Such a pathname, as the one above, is called an **absolute pathname**. When you have more than one / in a pathname, for each such /, you have to descend one level in the file system. Thus, `kumar` is one level below `home`, and two levels below root.

When you specify a file by using frontslashes to demarcate the various levels, you have a mechanism of identifying a file uniquely. No two files in a UNIX system can have identical absolute pathnames. You can have two files with the same name, but in different directories; their pathnames will also be different. Thus, the file `/home/kumar/progs/c2f.pl` can coexist with the file `/home/kumar/safe/c2f.pl`.

4.9.1 Using the Absolute Pathname for a Command

More often than not, a UNIX command runs by executing its disk file. When you specify the **date** command, the system has to locate the file date from a list of directories specified in the PATH variable, and then execute it. However, if you know the location of a particular command, you can precede its name with the complete path. Since **date** resides in /bin (or /usr/bin), you can also use the absolute pathname:

```
$ /bin/date
Thu Sep  1 09:30:49 IST 2005
```

Nobody runs the **date** command like that. For any command that resides in the directories specified in the PATH variable, you don't need to use the absolute pathname. This PATH, you'll recall *(2.4.1)*, invariably has the directories /bin and /usr/bin in its list.

If you execute programs residing in some other directory that isn't in PATH, the absolute pathname then needs to be specified. For example, to execute the program **less** residing in /usr/local/bin, you need to enter the absolute pathname:

```
/usr/local/bin/less
```

If you are frequently accessing programs in a certain directory, it's better to include the directory itself in PATH. The technique of doing that is shown in Section 10.3.

4.10 RELATIVE PATHNAMES

You would have noted that in a previous example *(4.8)*, we didn't use an absolute pathname to move to the directory progs. Nor did we use one as an argument to **cat** *(4.9)*:

```
cd progs
cat login.sql
```

Here, both progs and login.sql are presumed to exist in the current directory. Now, if progs also contains a directory scripts under it, you still won't need an absolute pathname to change to that directory:

```
cd progs/scripts                                    progs is in current directory
```

Here we have a pathname that has a /, but it is not an absolute pathname because it doesn't begin with a /. In these three examples, we used a rudimentary form of relative pathnames though they are generally not labeled as such. Relative pathnames, in the sense they are known, are discussed next.