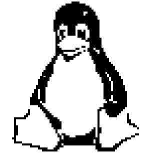
UNIT 7.
The File system



Structure

7.0 Introduction
7.1 Objectives
7.2 LINUX File system
  7.2.1 Home Directory
  7.2.2 Working Directory
  7.2.3 The Directory Tree
  7.2.4 Absolute Pathnames
  7.2.5 Relative Pathnames
  7.2.6 Changing Your Working Directory
    7.2.6.1 pwd
    7.2.6.2 cd
  7.2.7 Files in the Directory Tree
  7.2.8 Listing Files
    7.2.8.1 ls
7.3 Looking Files
  7.3.1 cat
  7.3.2 more
  7.3.3 pg
7.4 Protecting and Sharing Files
  7.4.1 Directory Access Permissions
  7.4.2 File Access Permissions
  7.4.3 More Protection Under Linux
7.5 File Management
  7.5.1 Methods of Creating Files
  7.5.2 File and Directory Names
  7.5.3 File and Directory Wildcards
  7.5.4 Managing Your Files
    7.5.4.1 Creating Directories
      7.5.4.1.1 mkdir
    7.5.4.2 Copying Files
      7.5.4.2.1 cp
      7.5.4.2.2 ftp
    7.5.4.3 Finding Files
    7.5.4.4 Files on Other Operating Systems
7.6 The Second Extended File system (EXT2)
7.7 The Virtual File System (VFS)
  7.7.1 Mounting a File System
    7.7.1.1 /dev
    7.7.1.2 /proc
7.8 Summary
7.9 Check Your Progress

7.0 Introduction

This chapter describes how the Linux kernel maintains the files in the file systems that it supports. It describes the Virtual File System (VFS) and explains how the Linux kernel's real file systems are supported. Once you log in, you can use the many facilities LINUX provides. As an authorized system user, you have an account that gives you:

- A place in the LINUX filesystem where you can store your files.
- A username that identifies you and lets you control access to your files and receive messages from other users.
- A customizable environment that you can tailor to your preferences.

One of the most important features of Linux is its support for many different file systems. This makes it very flexible and well able to coexist with many other operating systems. At the time of writing, Linux supports 15 file systems; `ext`, `ext2`, `xia`, `minix`, `umsdos`, `msdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs` and `ufs`, and no doubt, over time more will be added.

In Linux, as it is for Unix ™, the separate file systems the system may use are not accessed by device identifiers (such as a drive number or a drive name) but instead they are combined into a single hierarchical tree structure that represents the file system as one whole single entity. Linux adds each new file system into this single file system tree as it is mounted. All file systems, of whatever type, are mounted onto a directory and the files of the mounted file system cover up the existing contents of that directory. This directory is known as the mount directory or mount point. When the file system is unmounted, the mount directory's own files are once again revealed.

7.1 Objectives

At the end of this unit, you would be able to

- Undestand the concept of LINUX File system
- Know about different directory structures
- Distinguish among different pathnames
- Describe the usage and manipulations of files
- Examine and illustrate Protecting and Sharing Files

7.2 The LINUX File system

A *file* is the unit of storage in LINUX, as in many other systems. A file can hold anything: text (a report you're writing, a to-do list), a program, digitally encoded pictures or sound, and so on. All of those are just sequences of raw data until they are interpreted by the right program.

In LINUX, files are organized into directories. A *directory* is actually a special kind of file where the system stores information about other files. A directory can be thought of as a place, so that files are said to be contained *in* directories and you are said to work *inside* a directory. (If you've used a Macintosh or Microsoft Windows computer, a LINUX directory is a lot like a folder. MS-DOS and LINUX directories are very similar.)

7.2.1 Home Directory

When you log in to LINUX, you're placed in a directory called *home directory*. This home directory, a unique place in the LINUX filesystem, contains the files you use almost every time you log in. In your home directory, you can make your own files. As you'll see in a minute, you can also store your own directories within your home directory. Like folders in a file cabinet, this is a good way to organize your files.

7.2.2 Working Directory

working directory (sometimes called your current working directory) is the directory you're currently working in. At the start of every session, your home directory is your working directory. You may change to another directory, in which case the directory you move to becomes your working directory.

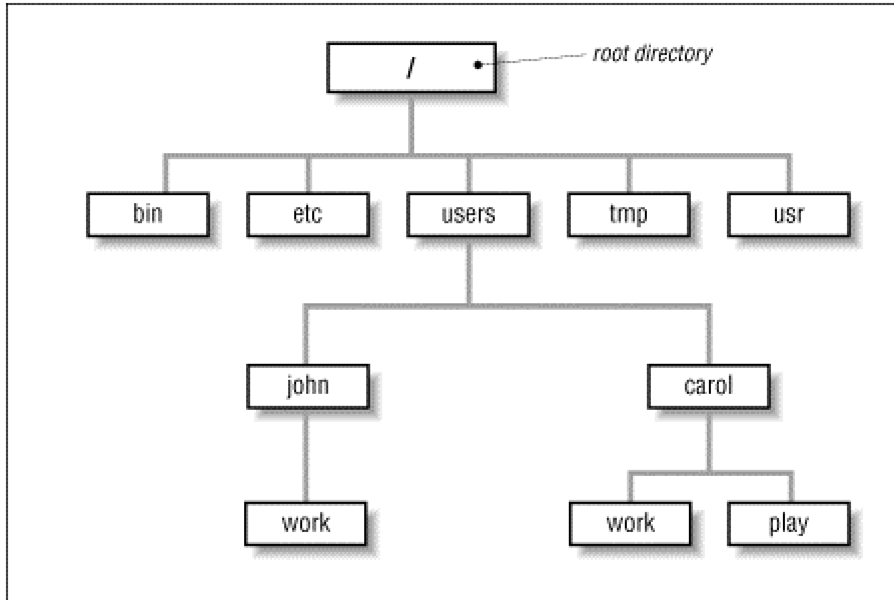
Unless you tell LINUX otherwise, all commands that you enter apply to the files in your working directory. In the same way, when you create files, they're created in your working directory.

7.2.3 The Directory Tree

All directories on a LINUX system are organized into a hierarchical structure that you can imagine as a family tree. The parent directory of the tree is known as the *root directory* and is written as a forward slash (`/`).

The root contains several directories. Figure 7.1 shows the top of an imaginary LINUX filesystem tree - the root directory and some of the directories under the root.

Figure 7.1: Example of a directory tree



bin, *etc*, *users*, *tmp*, and *usr* are some of the *subdirectories* (child directories) of *root*. These are fairly standard directories and usually contain specific kinds of system files. For instance, *bin* contains many LINUX commands. Not all systems have a directory named *users*; it may be called *u*, *home*, and/or be located in some other part of the filesystem.

In our example, the parent directory of *users* (one level above) is *root*. It also has two subdirectories (one level below), *john* and *carol*. On a LINUX system, each directory has one parent directory and may have one or more subdirectories. [1] A subdirectory (like *carol*) can have its own subdirectories (like *work* and *play*), to a limitless depth

To specify a file or directory location, you write its *pathname*. A pathname is like the address of the directory or file in the LINUX filesystem. We'll look at pathnames in a moment.

On a basic LINUX system, all files in the filesystem are stored on disks connected to your computer. It isn't always easy to use the files on someone else's computer or for someone on another computer to use your files. Your system may have an easier way: a *networked filesystem* (with a name like NFS or RFS). Networked filesystems make a remote computer's files appear as if they're part of your computer's directory tree. For instance, your computer in Los Angeles might have a directory named *boston*. When you look in that subdirectory, you'll see some (or all) of the directory tree from your company's computer in Boston. Your system administrator can tell you if your computer has any networked filesystems.

7.2.4 Absolute Pathnames

As you saw above, the LINUX filesystem organizes its files and directories in an inverted tree structure with the root directory at the top. An *absolute pathname* tells you the path of directories you must travel to get from the root to the directory or file you want. In a pathname, put slashes (/) between the directory names.

For example, */users/john* is an absolute pathname. It locates just one directory. Here's how:

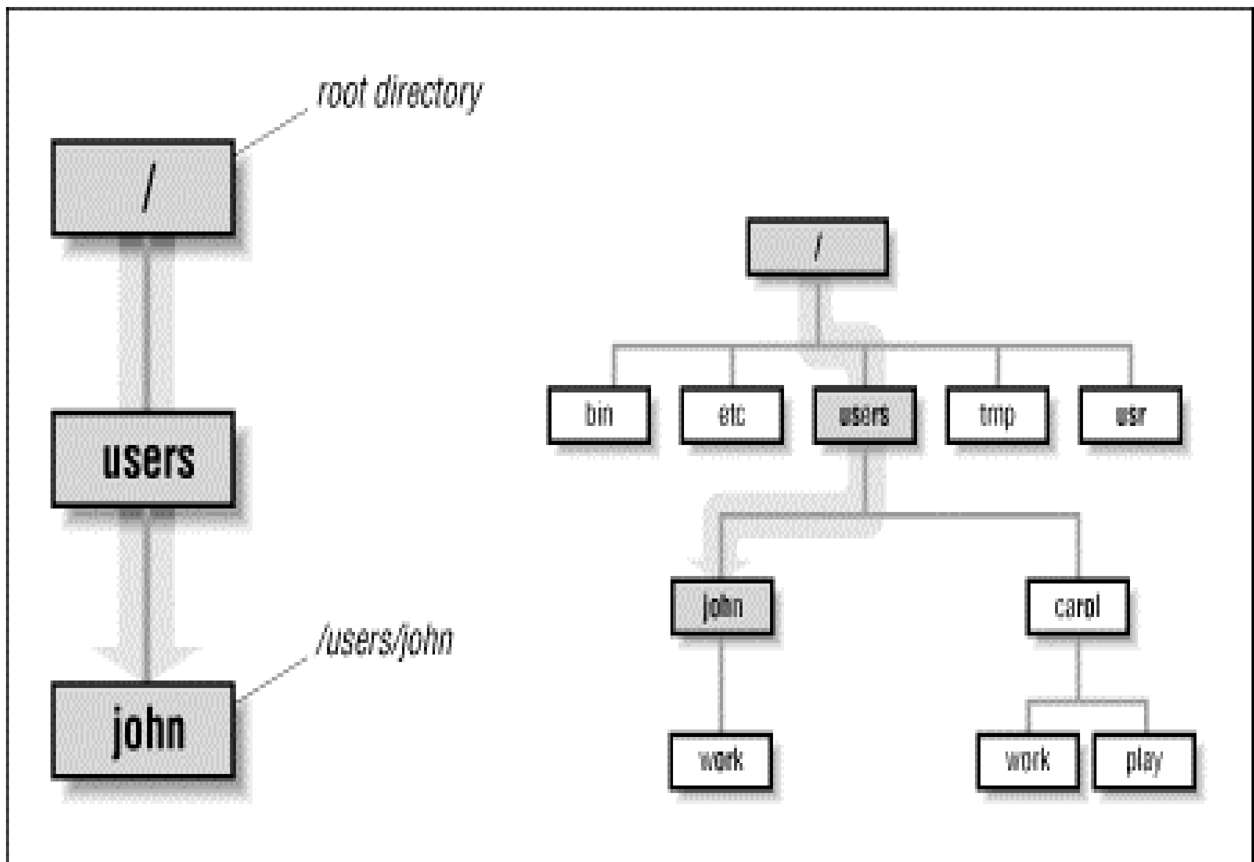
- the root is the first "/"
- the directory *users* (a subdirectory of *root*)
- the directory *john* (a subdirectory of *users*)

Be sure not to type spaces anywhere in the pathname. Figure 7.2 shows this structure.

If you look at Figure 7.2, you'll see that the directory *john* has a subdirectory named *work*. Its absolute pathname is */users/john/work*.

The root is always indicated by the slash (/) at the start of the pathname**.**

Figure 7.2: Absolute path of directory john



7.2.5 Relative Pathnames

You can also locate a file or directory with a *relative pathname*. A relative pathname gives the location relative to your working directory.

Unless you use an absolute pathname (starting with a slash), LINUX assumes that you're using a relative pathname. Like absolute pathnames, relative pathnames can go through more than one directory level by naming the directories along the path.

For example, if you're currently in the *users* directory (see Figure 7.2), the relative pathname to the *carol* directory below is simply *carol*. The relative pathname to the *play* directory below that is *carol/play*.

Notice that neither of the pathnames in the previous paragraph starts with a slash. That's what makes them relative pathnames! These pathnames start at the working directory, not the root directory.

Absolute and relative pathnames are totally interchangeable. LINUX commands simply follow whatever path you specify to wherever it leads. If you use an absolute pathname, the path starts from the root. If you use a relative pathname, the path starts from your working directory. Choose whichever is easier at the moment.

7.2.6 Changing Your Working Directory

When you know the absolute or relative pathname of a directory, you can move up and down the LINUX directory tree.

7.2.6.1 pwd

To find which directory you're currently in, use the **pwd** (print working directory) command. The **pwd** command takes no arguments.

% **pwd**

/users/john

pwd prints the absolute pathname of your working directory.

7.2.6.2 cd

You can change your working directory to any directory (including another user's directory - if you have permission) with the **cd** (change directory) command.

The **cd** command has the form:

cd *pathname*

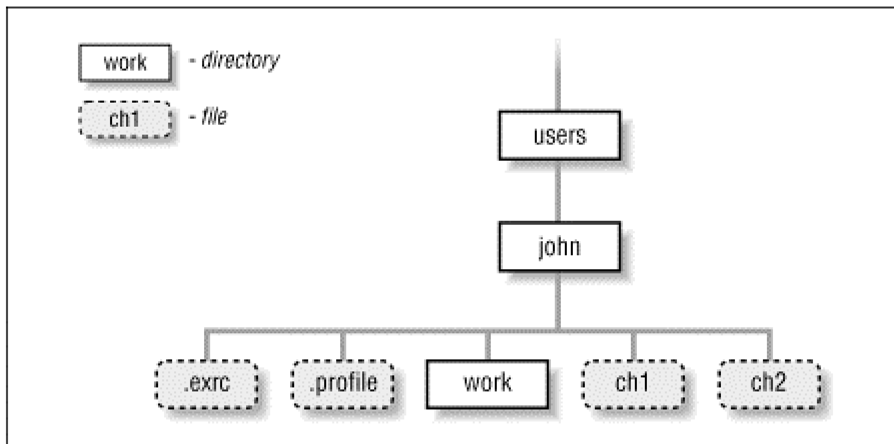
The argument is an absolute or a relative pathname (whichever is easier) for the directory you want to change to.

```
% cd /users/carol
% pwd
/users/carol
```

7.2.7 Files in the Directory Tree

A directory can hold subdirectories. And, of course, a directory can hold files. Figure 7.3 is a close-up of the filesystem around *john*'s home directory. The four files are shown along with the *work* subdirectory.

Figure 7.3: Files in the directory tree



Pathnames to files are made the same way as pathnames to directories. For example, if your working directory is *users*, the relative pathname to the *work* directory below would be *john/work*. The relative pathname to the *ch1* file would be *john/ch1*.

7.2.8 Listing Files

To use the **cd** command, you must decide which entries in a directory are subdirectories and which are files. The **ls** command lists the entries in the directory tree.

7.2.8.1 ls

When you enter the **ls** command, you'll get a listing of the files and subdirectories contained in your working directory. The syntax is:

ls *option(s) directory-and-filename(s)*

```
% ls
ch1    ch10    ch2    ch3   intro
%
```

(Some systems display filenames in a single column. If yours does, you can change the display to columns with the **-x** option.) **ls** has a lot of options that change the information and display format.

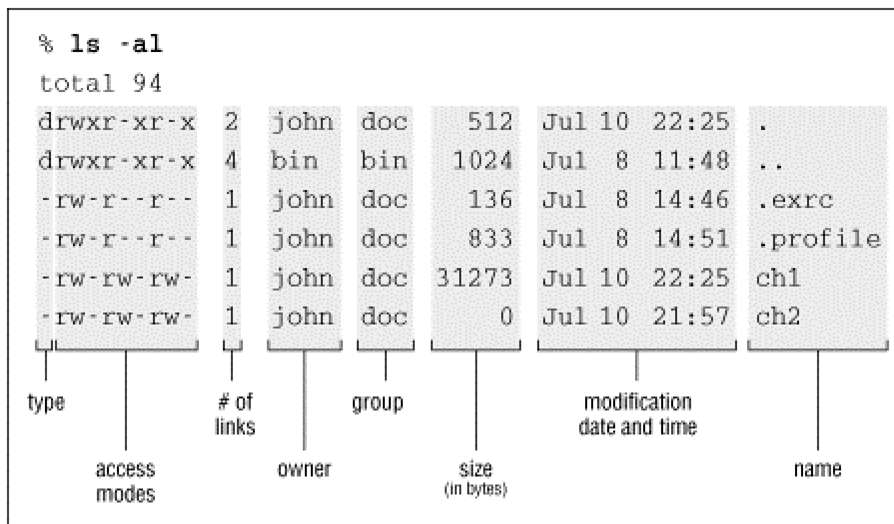
The **-a** option (for *all*) is guaranteed to show you some more files, as in the following example:

```
% ls -a
.      .exrc      ch1     ch2     intro
..     .profile   ch10    ch3
%
```

You'll always see at least two new entries with the names "." (dot) and ".." (dot dot). As mentioned earlier, .. is always the relative pathname to the parent directory, and a single . always stands for any working directory. There may also be other files, like *.profile* or *.exrc*. Any entry whose name begins with a dot is hidden - it will be listed only if you use **ls -a**.

To get more information about each file, add the **-l** option. (That's a lowercase letter "L" for *long*.) This option can be used alone, or in combination with **-a**, as shown in Figure 7.4.

Figure 7.4: Output from ls -al



The long format provides the following information about each file:

<u>*Total n*</u>*:*
n amount of storage used by the files in this directory.

Type **:**

Tells whether the file is a directory (`d`) or a plain file (-). (There are other less common types that we don't explain here.)

Access modes **:**

Specifies three types of users (yourself, your group, all others) who are allowed to read (`r`), write (`w`), or execute (`x`) your files.

Links **:**

The number of files and directories linked to this one.

Owner :

The person who created or owns the file.

Group:

The group that owns the file. (If your version of LINUX doesn't show this column, add the **-g** option to see it.)

Size (in bytes) :

The size of the file.

Modification date :

The date when the file was last modified.

Name :

The name of the file or directory.

Notice especially the columns that list the owner and group of the files, and the access modes (also called permissions). The person who creates a file is its owner; if you've created any files (or the system administrator did it for you), this column should show your username. You also belong to a group, to which you were assigned by the system administrator. Files you create will either be marked with the name of your group or, in some cases, the group that owns the directory.

The *permissions* control who can read, write (modify), or execute the file (if it's a program). The permissions have ten characters. The first character shows the file type (directory or plain file). The second through the fourth characters show the permissions for the file's owner - yourself if you created the file. The fifth through the seventh characters show permissions for other members of the file's group. The eighth through the tenth characters show permissions for all other users.

For example, the permissions for *.profile* are `-rw-r--r--`, so it's a plain file. You, the owner, have both read and write permissions. But other users of the system can only read the file; they cannot modify the file's contents. No one has execute (`x`) permission, which should only be used for executable files (files that hold programs).

In the case of directories, `x` means the permission to access the directory - for example, to run a command that reads a file there or to use a subdirectory. Notice that the two directories shown in the example are executable (searchable by you, by your group, and by everyone else on the system). A directory with `w` (write) permission allows deleting, renaming, or adding files within the directory. Read (`r`) permission allows listing the directory with **ls**.

You can use the **chmod** command to change the permissions of your files and directories. (ee the section of this chapter called "Protecting and Sharing Files.")

If you need to know only which files are directories and which are executable files, you can use the **-F** option.

If you give the pathname to a directory, **ls** will list the directory but it will *not* change your working directory. The **pwd** command in the following example shows that:

```
% ls -F /users/andy
calendar    goals    ideas/
ch2         guide/   testpgm*
% pwd
/etc
%
```

ls -F puts a / (slash) at the end of each directory name. (The directory name doesn't really have a slash in it; that's just the shortcut **ls -F** uses with a directory.) In our example, *guide* and *ideas* are directories. You can verify this by using **ls -l** and noting the "d" in the first field of the output. Files with an execute status (x), like programs, are marked with an * (asterisk). The file *testpgm* is an executable file. Files that aren't marked are not executable.

On Linux and other systems with the GNU version of **ls**, you may be able to see names in color. For instance, directories could be green and program files could be yellow.

Go to your home directory.	Enter **`cd`**
Find your working directory.	Enter **`pwd`**
Change to new working directory.	Enter **`cd /etc`**
List files in new working directory.	Enter **`ls`**
Change directory to root and list files.	Enter **`cd /; ls`**
Change to a new directory.	Enter **`cd usr`**
Give a wrong pathname.	Enter **`cd xqk`**
Change to a new directory with its absolute pathname.	Enter **`cd /etc`**
List files in another directory.	Enter **`ls /bin`**
Find your working directory (notice that **ls** didn't change it).	Enter **`pwd`**
Return to your home directory.	Enter **`cd`**

7.3 Looking Files

By now, you're probably tired of looking at files from the outside. It's kind of like going to a bookstore and looking at the covers, but never getting to read a word. Let's look at three programs for reading files: **cat**, **more**, and **pg**.

7.3.1 cat

Most first-time users of LINUX think that **cat** is a strange name for a program. As we'll see later, **cat**, which is short for "concatenate," puts files together (concatenates them) to make a bigger file. It can also display files on your screen.

To display files on the standard output), use:

cat *file(s)*

If you enter **cat** without a filename, get out by pressing `[RETURN]` followed by a single `[CTRL-D]`.

7.3.2 more

If you want to "read" a long file on the screen, your system may have the **more** command to display one screen or "page" of text at a time. A standard terminal screen can usually display 24 lines of text; a window can display almost any number of lines. The syntax is:

more *file(s)*

7.3.3 pg

The **more** command isn't available on some LINUX systems. Most of those systems have the **pg** command instead. It works like **more** but has different features. For example, **pg** lets you move to specific lines.

The syntax is:

pg *filename(s)*

7.4 Protecting and Sharing Files

LINUX makes it easy for users to share files and directories. Controlling exactly who has access takes some explaining, though - more explaining than we can do here. So here's a cookbook set of instructions.

7.4.1 Directory Access Permissions

A directory's access permissions help to control access to the files in it. These affect the overall ability to use files and subdirectories in the directory. In the commands below, replace *dirname* with the directory's pathname. An easy way to change permissions on the working directory is by using its relative pathname, **.** (dot), as in "**chmod 755 .**".

- To keep yourself from accidentally removing files (or adding or renaming files) in a directory, use **chmod 555** *dirname*.
- To protect the files in a directory and all its subdirectories from everyone else on your system - but still be able to do anything *you* want to dp there - use **chmod 700** *dirname*.
- To let other people on the system see what's in a directory - and read or edit the files if the file permissions let them - but not rename, remove, or add files - use **chmod 755** *dirname*.
- To let people in your LINUX group add, delete, and rename files in a directory of yours - and read or edit other people's files if the file permissions let them - use **chmod 775** *dirname*.
- To give full access to everyone on the system, use **chmod 777** *dirname*.

7.4.2 File Access Permissions

The access permissions on a file control what can be done to the file's *contents*. The access permissions on the *directory* where the file is kept control whether the file can be renamed or removed.

- To make a private file that only you can edit, use **chmod 600** *filename*. To protect it from accidental editing, use **chmod 400** *filename*.
- To edit a file yourself, and let everyone else on the system read it without editing, use **chmod 644** *filename*.
- To let you and all members of your LINUX group edit a file, but keep any other user from reading or editing it, use **chmod 660** *filename*.
- To let nongroup users read but not edit the file, use **chmod 664** *filename*.
- To let anyone read or edit the file, use **chmod 666** *filename*.

7.4.3 More Protection Under Linux

Most Linux systems have a command that gives you more choices on file and directory protection: **chattr**. **chattr** is being developed, and your version may not have all of the features that it will have in later versions of Linux. For instance, **chattr** can make a Linux file *append-only* (so it can't be overwritten, only added to); *compressed* (to save disk space automatically); *immutable* (so it can't be changed at all); *undeletable*, and more.

7.5 File Management

7.5.1 Methods of Creating Files

Two common LINUX editors are **vi** (pronounced "vee-eye") and **emacs** ("ee-macs").**vi** and **emacs** are very sophisticated, extremely flexible editors for all kinds of text files: programs, email messages, and so on. Many LINUX systems also support easy-to-use word processors. **Pico** is a simple editor (not word processor) that has been added to many LINUX systems.

Since there are several editor programs, you can choose one you're comfortable with. **vi** is probably the best choice because almost all LINUX systems have it, but **emacs** is also widely available. Although **pico** is much less powerful than **emacs** or **vi**, it's also a lot easier to learn.

7.5.2 File and Directory Names

Both files and directories are identified by their names. A directory is really just a special kind of file, so the rules for naming directories are the same as the rules for naming files.

Filenames may contain any character except `/`, which is reserved as the separator between files and directories in a pathname. Unlike some operating systems, LINUX doesn't require a dot (`.`) in a filename.A filename must be unique inside its directory, but other directories may have files with the same names.

7.5.3 File and Directory Wildcards

When you have a number of files named in series (for example, *chap1* to *chap12*) or filenames with common characters (like *aegis*, *aeon*, and *aerie*), you can use *wildcards* (also called *metacharacters*) to specify many files at once. These special characters are * (asterisk), `?` (question mark), and `[ ]` (square brackets). When used in a filename given as an argument to a command:

*****

An asterisk is replaced by any number of characters in a filename. For example, *ae** would match *aegis*, *aerie*, *aeon*, etc.

?

A question mark is replaced by any single character (so *h?p* matches *hop* and *hip*, but not *help*).

[ ]

Square brackets can surround a choice of characters you'd like to match. Any one of the characters between the brackets will be matched. For example, [*Cc*]*hapter* would match either *Chapter* or *chapter*, *chap*[*13*] would match *chap1*, *chap2*, or *chap3*.

```
% ls chap?
```

```
chap2    chap5    chap7
chap4    chap6
% ls chap[5-8]
chap5    chap6    chap7
% ls chap??
chap10   chap1b
% ls *old
chap1a.old   chap3.old    cold
% ls *a*a*
chap1a.old   haha
```

Wildcards are useful for more than listing files. Most LINUX commands accept more than one filename, and you can use wildcards to put multiple files on the command line. For example, the command **more** is used to display a file on the screen. Let's say you want to display files *chap3.old* and *chap1a.old*. Instead of specifying these files individually, you could enter the command as:

```
% more *.old
```

This is equivalent to "**more chap1a.old chap3.old**".

7.5.4 Managing Your Files

The tree structure of the LINUX filesystem makes it easy to organize your files. After you make and edit some files, you may want to copy or move files from one directory to another, rename files to distinguish different versions of a file, or give several names to the same file. You may want to create new directories each time you start working on a different project.

7.5.4.1 Creating Directories

7.5.4.1.1 mkdir

To create a new directory, use the **mkdir** command. The format is:

mkdir *dirname(s)*

dirname is the name of the new directory. To make several directories, put a space between each directory name. To continue our example, you would enter:

```
% mkdir spy boston.dine
```

7.5.4.2 Copying Files

If you're about to edit a file, you may want to save a copy of it first. Doing that makes it easy to get back the original version.

7.5.4.2.1 cp

The **cp** command can put a copy of a file into the same directory or into another directory. **cp** doesn't affect the original file, so it's a good way to keep an identical backup of a file.

To copy a file, use the command:

cp *old new*

where *old* is a pathname to the original file and *new* is the pathname you want for the copy.

For example, let's say your working directory was */users/carol*. To copy three files called *ch1*, *ch2*, and *ch3* from */users/john* to a subdirectory called *work* (that's */users/carol/work*), by entering:

```
% cp ../john/ch1 ../john/ch2 ../john/ch3 work
```

Or, you could use wildcards and let the shell find all the appropriate files. This time, let's add the **-i** option for safety:

```
% cp -i ../john/ch[1-3] work
cp: overwrite work/ch2? n
```

There was already a file named *ch2* in the *work* directory. When **cp** asked, I answered **n** to prevent copying *ch2*. Answering **y** would overwrite the old *ch2*.

The shorthand forms `.` and `..` will put the copy in the working directory or its parent. For example:

```
% cp ../john/ch[1-3] .
```

puts the copies into the working directory.

7.5.4.2.2 ftp

The command **ftp** (file transfer protocol) is a flexible way to copy files between two computers. (Some systems have a friendlier version of **ftp** named **ncftp**.) Both computers don't need to be running LINUX, though they do need to be connected by a network (like the Internet) that **ftp** can use. To start **ftp**, give the hostname of the remote computer:

ftp *hostname*

ftp will prompt for your username and password on the remote computer. This is something like a remote login, but **ftp** doesn't start your usual shell. Instead, **ftp** prints its

own prompt and uses a special set of commands for transferring files. Table 7.5.4.2.2 lists the most important **ftp** commands.

Table 7.5.4.2.2 Some ftp Commands

Command	Description
put *filename*	Copies the file *filename* from your local computer to the remote computer. If you give a second argument, the remote copy will have that name.
mput *filenames*	Copies the named files (you can use wildcards) from local to remote.
get *filename*	Copies the file *filename* from the remote computer to your local computer. If you give a second argument, the local copy will have that name.
mget *filenames*	Copies the named files (you can use wildcards) from remote to local.
cd *pathname*	Changes the working directory on the remote machine to *pathname* (**ftp** usually starts at your home directory on the remote machine).
lcd *pathname*	Changes **ftp**'s working directory on the local machine to *pathname* (**ftp** starts at your working directory on the local computer). Note that the **ftp lcd** command changes only **ftp**'s working directory. After you quit **ftp**, your shell's working directory will not have changed.
dir	Lists the remote directory (like **ls -l**).
binary	Tells **ftp** to copy the following file(s) without translation. This preserves pictures, sound, or other data.
ascii	Transfers plain text files, translating data if needed.
quit	Ends the **ftp** session and takes you back to a shell prompt.

Here's an example. Carol uses **ftp** to copy the file *todo* from her *work* subdirectory on her account on the remote computer *rhino*:

```
% ls
afile   ch2    somefile
% ftp rhino
Connected to rhino.zoo.com.
Name (rhino:carol): csmith
Password:
ftp> cd work
ftp> dir
total 3
-rw-r--r--  1 csmith   mgmt     47 Feb  5  1997 for.ed
-rw-r--r--  1 csmith   mgmt    264 Oct 11 12:18 message
-rw-r--r--  1 csmith   mgmt    724 Nov 20 14:53 todo
ftp> get todo
ftp> quit
% ls
afile   ch2    somefile    todo
```

We've covered the most basic **ftp** commands here. Entering **help** at an `ftp>` prompt gives a list of all commands; entering **help** followed by an **ftp** command name gives a one-line summary of that command.

7.5.4.3 Finding Files

If your account has lots of files, organizing those files into subdirectories can help you find the files later. Sometimes you may not remember which subdirectory has a file. The **find** command can search for files in many ways; we'll look at two of them.

Change to your home directory so **find** will start its search there. Then carefully enter one of the two **find** commands below. (The syntax is strange and ugly - but **find** does the job!)

```
% cd
% find . -type f -name 'chap*' -print
./chap2
./old/chap10b
% find . -type f -mtime -2 -print
./work/to_do
```

The first command looked in your working (home) directory and all its subdirectories for files (**type f**) whose names start with *chap*. (**find** understands wildcards in filenames.) The second command looked for all files that have been created or modified in the last two days (**-mtime -2**). The relative pathnames that **find** finds start with a dot (**./**), the name of the working directory, which you can ignore.

Linux systems, and some others, have the GNU **locate** command. If it's been set up and maintained on your system, you can use **locate** to search part or all of a filesystem for a file with a certain name. For instance, if you're looking for a file named *alpha-test*, *alphatest*, or something like that, try this:

```
% locate alpha
/users/alan/alpha3
/usr/local/projects/mega/alphatest
```

You'll get the absolute pathnames of files and directories that have *alpha* in their names. **locate** may or may not list protected, private files; its listings usually also aren't completely up to date.

7.5.4.4 Files on Other Operating Systems

You read above about **ftp**, a program for transferring files across a network - possibly to non-LINUX operating systems. Your system may also be able to run operating systems other than LINUX. For instance, many Linux systems can also run

MS-DOS and Windows 95. If yours does, you can probably use those files from your Linux account.

If the DOS or Windows filesystem is *mounted* with your other filesystems, you'll be able to use its files by typing a UNIX-like pathname. For instance, from our PC under Linux, we can access the DOS file *C:\WORD\REPORT.DOC* through the pathname */dosc/word/report.doc*.

Your Linux (or other) system may also have the MTOOLS utilities. These give you DOS-like commands that interoperate with the UNIX-like system. For example, we can put a Windows 95 floppy disk in the A: drive and then copy a file named *summary.txt* into our current directory (.) by entering:

```
% mcopy a:summary.txt .
Copying summary.txt
%
```

Your system administrator should be able to tell you whether other filesystems are mounted, whether you have utilities like MTOOLS, and how to use them.

7.6 The Second Extended File system (EXT2)

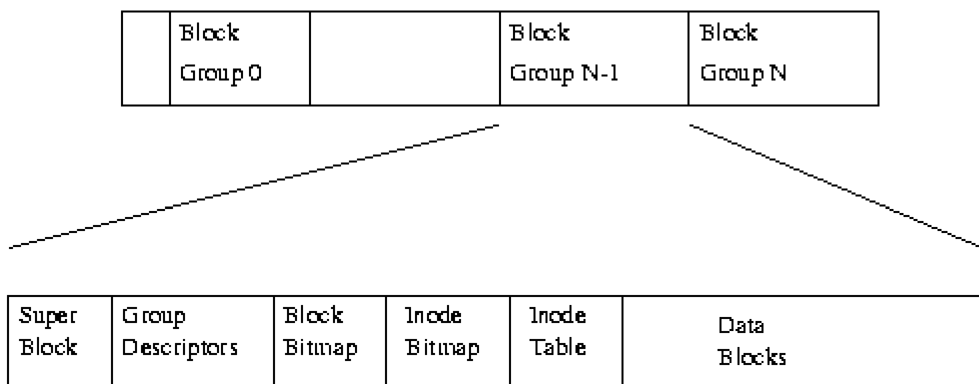


Figure 7.5: Physical Layout of the EXT2 File system

The Second Extended File system was devised (by Rémy Card) as an extensible and powerful file system for Linux. It is also the most successful file system so far in the Linux community and is the basis for all of the currently shipping Linux distributions.

The EXT2 file system, like a lot of the file systems, is built on the premise that the data held in files is kept in data blocks. These data blocks are all of the same length and, although that length can vary between different EXT2 file systems the block size of a particular EXT2 file system is set when it is created (using mke2fs). Every file's size is

rounded up to an integral number of blocks. If the block size is 1024 bytes, then a file of 1025 bytes will occupy two 1024 byte blocks. Unfortunately this means that on average you waste half a block per file. Usually in computing you trade off CPU usage for memory and disk space utilisation. In this case Linux, along with most operating systems, trades off a relatively inefficient disk usage in order to reduce the workload on the CPU. Not all of the blocks in the file system hold data, some must be used to contain the information that describes the structure of the file system. EXT2 defines the file system topology by describing each file in the system with an inode data structure. An inode describes which blocks the data within a file occupies as well as the access rights of the file, the file's modification times and the type of the file. Every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in inode tables. EXT2 directories are simply special files (themselves described by inodes) which contain pointers to the inodes of their directory entries.

Figure 7.5 shows the layout of the EXT2 file system as occupying a series of blocks in a block structured device. So far as each file system is concerned, block devices are just a series of blocks that can be read and written. A file system does not need to concern itself with where on the physical media a block should be put, that is the job of the device's driver. Whenever a file system needs to read information or data from the block device containing it, it requests that its supporting device driver reads an integral number of blocks. The EXT2 file system divides the logical partition that it occupies into Block Groups.

7.7 The Virtual File System (VFS)

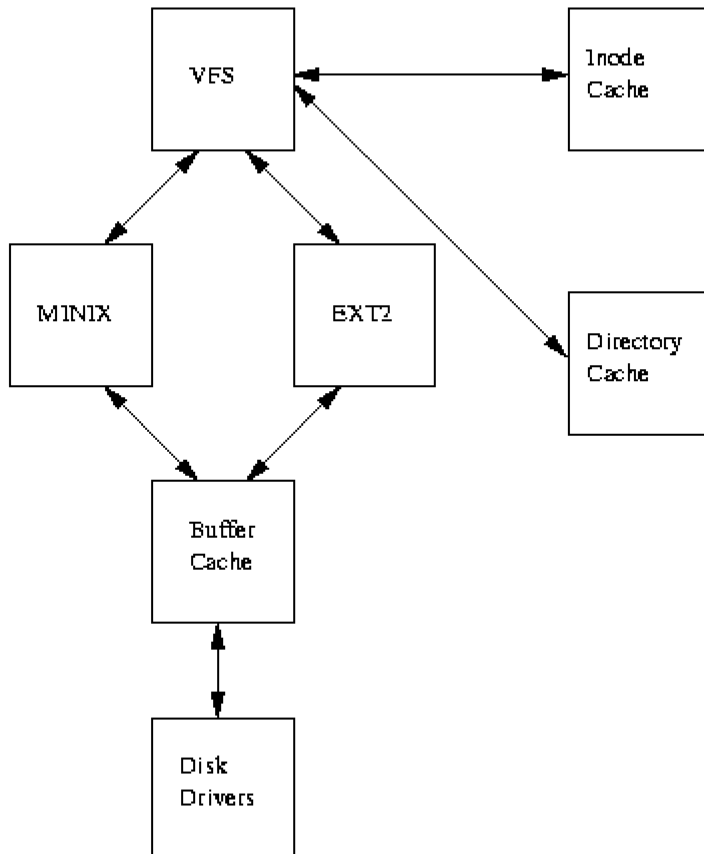


Figure 7.6 A Logical Diagram of the Virtual File System

Figure 7.6 shows the relationship between the Linux kernel's Virtual File System and it's real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems.

Rather confusingly, the VFS describes the system's files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the VFS inodes describe files and directories within the system; the contents and topology of the Virtual File System. From now on, to avoid confusion, I will write about VFS inodes and VFS superblocks to distinquish them from EXT2 inodes and superblocks.

As each file system is initialised, it registers itself with the VFS. This happens as the operating system initialises itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the VFAT file system is implemented as a kernel module, then it is only loaded when a VFAT file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure. The VFS keeps a list of the mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions. So, for example, the superblock representing a mounted EXT2 file system contains a pointer to the EXT2 specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the "/" directory. This mapping of information is very efficient for the EXT2 file system but moderately less so for other file systems.

For example, typing ls for a directory or cat for a file cause the the Virtual File System to search through the VFS inodes that represent the file system. As every file and directory on the system is represented by a VFS inode, then a number of inodes will be being repeatedly accessed. These inodes are kept in the inode cache which makes access to them quicker. If an inode is not in the inode cache, then a file system specific routine must be called in order to read the appropriate inode. The action of reading the inode causes it to be put into the inode cache and further accesses to the inode keep it in the cache. The less used VFS inodes get removed from the cache.

All of the Linux file systems use a common buffer cache to cache data buffers from the underlying devices to help speed up access by all of the file systems to the physical devices holding the file systems.

The VFS also keeps a cache of directory lookups so that the inodes for frequently used directories can be quickly found.

7.7.1 Mounting a File System

When the superuser attempts to mount a file system, the Linux kernel must first validate the arguments passed in the system call. Although mount does some basic checking, it does not know which file systems this kernel has been built to support or that the proposed mount point actually exists. Consider the following mount command:

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

This mount command will pass the kernel three pieces of information; the name of the file system, the physical block device that contains the file system and, thirdly, where in the existing file system topology the new file system is to be mounted.

A Linux or UNIX machine typically has two special-purpose directories, `/dev` and `/proc`.

7.7.1.1 /dev

The `/dev` directory contains entries for the physical *devices* that may or may not be present in the hardware. The hard drive partitions containing the mounted filesystem(s) have entries in `/dev`, as a simple df shows.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
 Mounted on
 /dev/hda6               495876    222748    247527  48% /
 /dev/hda1                50755      3887     44248   9% /boot
 /dev/hda8               367013     13262    334803   4% /home
 /dev/hda5              1714416   1123624    503704  70% /usr
```

Among other things, the `/dev` directory also contains *loopback* devices, such as `/dev/loop0`. A loopback device is a gimmick allows an ordinary file to be accessed as if it were a block device. This enables mounting an entire filesystem within a single large file.

7.7.1.2 /proc

The `/proc` directory is actually a pseudo-filesystem. The files in the `/proc` directory mirror currently running system and kernel *processes* and contain information and statistics about them.

The `/proc` directory contains subdirectories with unusual numerical names. Every one of these names maps to the process ID of a currently running process. Within each of these subdirectories, there are a number of files that hold useful information about the corresponding process. The `stat` and `status` files keep running statistics on the process, the `cmdline` file holds the command-line arguments the process was invoked with, and the `exe` file is a symbolic link to the complete path name of the invoking process. There are a few more such files, but these seem to be the most interesting from a scripting standpoint.

7.8 Summary

The most basic concept of a file, and one you may already be familiar with from other computer systems, defines a file as a distinct chunk of information that is found on your hard drive. Distinct means that there can be many different files, each with its own particular contents. To keep files from getting confused with each other, every file must have a unique identity. In Linux, you identify each file by its name and location. In each location or directory, there can be only one file by a particular name. So, for instance, if

you create a file called novel, and you get a second great idea, you will either have to call it something different, such as novel2, or put it in a different place, to keep from overwriting the contents already in your original novel.

Linux allows filenames to be up to 256 characters long. These characters can be lower- and uppercase letters, numbers, and other characters, usually the dash (-), the underscore (_), and the dot (.).

They can't include reserved metacharacters such as the asterisk, question mark, backslash, and space, because these all have meaning to the shell.

7.9 Check Your Progress

I. Choose apropriate answer

1. File contains _____.
   a. digitally encoded pictures b. program c. sound d. all of these.
2. In case of ___________ pathname ,the path starts from root.
   a. absolute b. relative c. both of these d. none of these
3. _______ command used to change the permissions of your files & directories.
   a. ls b. chmod c. more d. ls -F
4. **chmod 555** *dirname* is used ____________________________.
   a. To protect the files in a directory and all its subdirectories from everyone else on your system.
   b. To keep yourself from accidentally removing files in a directory.
   c. To let other people on the system see what's in a directory - and read or edit the files if the file permissions let them - but not rename, remove, or add files .
   d. none of the above.
5. **chmod 644** *filename* is used to________________________.
   a. To let nongroup users read but not edit the file
   b. To let anyone read or edit the file
   c. To make a private file that only you can edit
   d. none of the above.
6. For using FTP ____________________________.
   a. Both computers don't need to be running LINUX
   b. Both computers need to be running LINUX
   c. one computer must be running LINUX & other any operating system
   d. none of the above.
7. The acronym for EXT2 is ________________________.
   a.Second Extended File system
   b Extended File system
   c Expanded File system
   d all of the above
8. mount command contains the ____________.
   a.physical block device that contains the file system

b name of the file system
c both of above
d none of these

II. Say True or False

1. A *directory* is actually a special kind of file. True/False
2. In case of absolute pathname, the path starts from your working directory. True/False
3. ls -l puts a / (slash) at the end of each directory name. True/False
4. chmod 755 *dirname* gives full access to everyone on the system. True/False
5. chmod 664 *filename* allows nongroup users to read but not edit the file True/False
6. Filenames can have any character including `/`. True/False

III. Essay type Questions

1. Briefly explain the concept of LINUX file system..
2. Explain the following:
   i Directory Tree
   ii Absolute Pathname
   iii Files in the Directory Tree.
   iv Directory Access Permission
   v File Access Permission
3. Differentiate among Absolute Pathname & Relative Pathname.
4. Explain all the attributes related to ls –al .
5. Describe the concept of File Management.
6. Write a note on the following commands.
   i locate
   ii find
7. What is VFS? How it is usefull in Linux.
8. Elaborate second Extended File system with physical layout.
9. How to mount a File system.

IV. Further readings and other activities

1. Get more information using man or info or any help command for
   ftp, ssh, ping, mount, ls, pwd, mkdir
2. Exercise: Manipulating files

In this exercise, you'll create, rename and delete files. Find out if your site has one or more printers as well as the appropriate command to use for printing.

Go to home directory. Enter **`cd`**

Copy distant file to working directory. Enter **`cp /etc/passwd myfile`**

Create new directory.	Enter **`mkdir temp`**
List working directory.	Enter **`ls -F`**
Move file to new directory.	Enter **`mv myfile temp`**
Change working directory.	Enter **`cd temp`**
Copy file to working directory.	Enter **`cp myfile myfile.two`**
Print the file.	Enter your printer command and the filename
List filenames with wildcard.	Enter **`ls -l myfile*`**
Remove files.	Enter **`rm myfile*`**
Go up to parent directory.	Enter **`cd ..`**
Remove directory.	Enter **`rmdir temp`**
Verify that directory was removed.	Enter **`ls -F`**

3. More information on files can be obtained from text book
   title: Introduction to the Shell programming
   Author: Yashwanth Kanetkar

IV. Further readings and other activities

1. Get more information using man or info or any help command for
   fs, ls, diff, du, df, apropos, bash, more, pg, chmod, chown, passwd
   EX: 1. $ man du 2. $ apropos bash

Check the amount of space used, amount of space free for different drives and different directories.

2. For further readings you can refer the following books
   1. Title: The Complete Reference Linux Foutth Edition
      Author: Richard Petersen
      Publication: Tata McGraw-Hill
   2. Title: Red Hat Linux Starter kit in 24 hours
      Author: Judith Samson etal
      Publication: Sams Techmedia

Reference e-mails: raomvp@yahoo.com roopasindhe@lycos.com
URL / Web Site: http://www.raomvp.bravepages.com

Good-Luck