**Filename Expansion**

Under UNIX-based shells such as BASH, wildcards on the command line are expanded

before being passed as a parameter to the application. This is in sharp contrast to the

default mode of operation for DOS-based tools, which often have to perform their own

wildcard expansion. The UNIX method also means that you must be careful where you

use the wildcard characters.

The wildcard characters themselves in BASH are identical to those in command.com:

The asterisk (\*) matches against all filenames, and the question mark (?) matches against

single characters. If you need to use these characters as part of another parameter for

whatever reason, you can escape them by preceding them with a backslash (\) character.

This causes the shell to interpret the asterisk and question mark as regular characters

instead of wildcards.

***NOTE*** Most Linux documentation refers to wildcards as regular expressions. The distinction is

important, since regular expressions are substantially more powerful than just wildcards alone. All

of the shells that come with Linux support regular expressions. You can read more about them in the

shell’s manual page (e.g., man bash, man csh, man tcsh).

**Environment Variables as Parameters**

Under BASH, you can use environment variables as parameters on the command line.

(Although the Windows command prompt can do this as well, it’s not a common practice

and thus is an often-forgotten convention.) For example, issuing the parameter

**$FOO** will cause the value of the FOO environment variable to be passed rather than the

string “$FOO.”

**Multiple Commands**

Under BASH, multiple commands can be executed on the same line by separating the

commands with semicolons (;). For example, to execute this sequence of commands (**cat**

and **ls**) on a single line:

[yyang@fedora-serverA ~]$ **ls -l**

[yyang@fedora-serverA ~]$ **cat /etc/passwd**

you could instead type the following:

[yyang@fedora-serverA ~]$ **ls -l ; cat /etc/passwd**

Since the shell is also a programming language, you can run commands serially only

if the first command succeeds. For example, use the **ls** command to try to list a file that

Chapter 5: The Command Line 109

does *not* exist in your home directory, and then execute the **date** command right after

that on the same line. Type

[yyang@fedora-serverA ~]$ **ls does-not-exist.txt && date**

ls: cannot access does-not-exist.txt: No such file or directory

This command will run the **ls** command, but that command will fail because the file

it is trying to list does not exist, and, therefore, the **date** command will not be executed

either. But if you switch the order of commands around, you will notice that the **date**

command will succeed, while the **ls** command will fail. Try

[yyang@fedora-serverA ~]$ **date && ls does-not-exist.txt**

Sun Jan 30 18:06:37 PDT 2090

ls: cannot access does-not-exist.txt: No such file or directory

**Backticks**

How’s this for wild? You can take the output of one program and make it the parameter

of another program. Sound bizarre? Well, time to get used to it—this is one of the most

useful and innovative features available in all UNIX shells.

Backticks (`) allow you to embed commands as parameters to other commands. You’ll

see this technique used often in this book and in various system scripts. For example,

you can pass the value of a number (a process ID number) stored in a file and then pass

that number as a parameter to the **kill** command. A typical use of this is for killing

(stopping) the Domain Name System (DNS) server **named**. When **named** starts, it writes

its process identification (PID) number into the file **/var/run/named/named.pid**. Thus,

the generic and dirty way of killing the **named** process is to look at the number stored in

**/var/run/named/named.pid** using the **cat** command, and then issue the **kill** command

with that value. For example,

[root@fedora-serverA ~]$ **cat /var/run/named/named.pid**

253

[root@fedora-serverA ~]$ **kill 253**

One problem with killing the **named** process in this way is that it cannot be easily

automated—we are counting on the fact that a human will read the value in **/var/run/**

**named/named.pid** in order to kill the number. Another issue isn’t so much a problem as

it is a nuisance: It takes two steps to stop the DNS server.

Using backticks, however, we can combine the steps into one and do it in a way that

can be automated. The backticks version would look like this:

[root@fedora-serverA ~]$ **kill `cat /var/run/named/named.pid`**

When BASH sees this command, it will first run **cat /var/run/named/named.**

**pid** and store the result. It will then run **kill** and pass the stored result to it. From our

point of view, this happens in one graceful step.

110 Linux Administration: A Beginner’s Guide

***NOTE*** So far in this chapter, we have looked at features that are internal to BASH (or BASH built-ins

as they are sometimes called). The remainder of the chapter explores several common commands

accessible outside of BASH.

**DOCUMENTATION TOOLS**

Linux comes with two superbly useful tools for making documentation accessible: **man**

and **info**. Currently, a great deal of overlap exists between these two documentation

systems because many applications are moving their documentation to the **info** format.

This format is considered superior to **man** because it allows the documentation to

be hyperlinked together in a web-like way, but without actually having to be written in

Hypertext Markup Language (HTML) format.

The **man** format, on the other hand, has been around for decades. For thousands of

utilities, their man (short for *manual*) pages are their only documentation. Furthermore,

many applications continue to utilize the **man** format because many other UNIX-like

operating systems (such as Sun Solaris) use it.

Both the **man** and **info** documentation systems will be around for a long while to

come. It is highly recommended that you get comfortable with them both.

***TIP*** Many Linux distributions also include a great deal of documentation in the **/usr/doc** or **/usr/**

**share/doc** directory.

**The man Command**

We mentioned quite early in this book that man pages are documents found online (on

the local system) that cover the use of tools and their corresponding configuration files.

The format of the **man** command is as follows:

[yyang@fedora-serverA ~]$ **man *program\_name***

where ***program\_name*** identifies the program you’re interested in. For example, to view

the man page for the **ls** utility that we’ve been using, type

[yyang@fedora-serverA ~]$ **man ls**

While reading about UNIX and UNIX-related information sources (newsgroups and

so forth), you may encounter references to commands followed by numbers in parentheses—

for example, **ls** (1). The number represents the section of the manual pages (see

Table 5-1). Each section covers various subject areas to accommodate the fact that some

tools (such as **printf**) are commands/functions in the C programming language as

well as command-line commands.

Chapter 5: The Command Line 111

To refer to a specific man section, simply specify the section number as the first

parameter and then the command as the second parameter. For example, to get the C

programmers’ information on **printf**, you’d enter this:

[yyang@fedora-serverA ~]$ **man 3 printf**

To get the command-line information, you’d enter this:

[yyang@fedora-serverA ~]$ **man 1 printf**

If you don’t specify a section number with the **man** command, the default behavior

is that the lowest section number gets printed first. Unfortunately, this organization

can sometimes be difficult to use, and as a result, there are several other available

alternatives.

***TIP*** A handy option to the **man** command is **-f** preceding the command parameter. With this

option, **man** will search the summary information of all the man pages and list pages matching your

specified command, along with their section number. For example,

[yyang@fedora-serverA ~]$ **man -f printf**

asprintf (3) - print to allocated string

printf (1) - format and print data

printf (3) - formatted output conversion

**Table 5-1.** Man Page Sections

**Manual Section Subject**

1 User tools

2 System calls

3 C library calls

4 Device driver information

5 Configuration files

6 Games

7 Packages

8 System tools

112 Linux Administration: A Beginner’s Guide

**The texinfo System**

Another common form of documentation is texinfo. Established as the GNU standard,

texinfo is a documentation system similar to the hyperlinked World Wide Web format.

Because documents can be hyperlinked together, texinfo is often easier to read, use, and

search than man pages.

To read the texinfo documents on a specific tool or application, invoke **info** with

the parameter specifying the tool’s name. For example, to read about the **grub** program,

type

[yyang@fedora-serverA ~]$ **info grub**

In general, you will want to verify whether a man page exists before using **info**

(there is still a great deal more information available in **man** format than in texinfo). On

the other hand, some man pages will explicitly state that the texinfo pages are more

authoritative and should be read instead.

**FILES, FILE TYPES, FILE OWNERSHIP,**

**AND FILE PERMISSIONS**

Managing files under Linux is different from managing files under Windows NT/200*x*/

XP/Vista, and radically different from managing files under Windows 95/98. In this section,

we discuss basic file management tools and concepts under Linux. We’ll start with

specifics on some useful general-purpose commands, and then we’ll step back and look

at some background information.

Under Linux (and UNIX in general), almost everything is abstracted to a file. Originally,

this was done to simplify the programmer’s job. Instead of having to communicate

directly with device drivers, special files (which look like ordinary files to the application)

are used as a bridge. Several types of files accommodate all these file uses.

**Normal Files**

Normal files are just that—normal. They contain data or executables, and the operating

system makes no assumptions about their contents.

**Directories**

Directory files are a special instance of normal files. Directory files list the locations of

other files, some of which may be other directories. (This is similar to folders in Windows.)

In general, the contents of directory files won’t be of importance to your daily operations,

unless you need to open and read the file yourself rather than using existing applications

to navigate directories. (This would be similar to trying to read the DOS file allocation

table directly rather than using command.com to navigate directories or using the findfirst/

findnext system calls.)

Chapter 5: The Command Line 113

**Hard Links**

Each file in the Linux file system gets its own i-node. An i-node keeps track of a file’s

attributes and its location on the disk. If you need to be able to refer to a single file using

two separate filenames, you can create a hard link. The *hard link* will have the same

i-node as the original file and will, therefore, look and behave just like the original. With

every hard link that is created, a reference count is incremented. When a hard link is

removed, the reference count is decremented. Until the reference count reaches zero, the

file will remain on disk.

***NOTE*** A hard link cannot exist between two files on separate partitions. This is because the hard

link refers to the original file by i-node, and a file’s i-node may differ among file systems.

**Symbolic Links**

Unlike hard links, which point to a file by its i-node, a *symbolic link* points to another

file by its name. This allows symbolic links (often abbreviated symlinks) to point to files

located on other partitions, even other network drives.

**Block Devices**

Since all device drivers are accessed through the file system, files of type *block device*

are used to interface with devices such as disks. A block device file has three identifying

traits:

▼ It has a major number.

■ It has a minor number.

▲ When viewed using the **ls -l** command, it shows *b* as the first character of the

permissions field.

For example,

[yyang@fedora-serverA ~]$ **ls -l /dev/sda**

brw-r----- 1 root disk 8, 0 2090-09-30 08:18 /dev/sda

Note the **b** at the beginning of the file’s permissions; the **8** is the major number, and

the **0** is the minor number.

A block device file’s major number identifies the represented device driver. When this

file is accessed, the minor number is passed to the device driver as a parameter, telling

it which device it is accessing. For example, if there are two serial ports, they will share

the same device driver and thus the same major number, but each serial port will have a

unique minor number.

114 Linux Administration: A Beginner’s Guide

**Character Devices**

Similar to block devices, *character devices* are special files that allow you to access devices

through the file system. The obvious difference between block and character devices is

that block devices communicate with the actual devices in large blocks, whereas character

devices work one character at a time. (A hard disk is a block device; a modem is a

character device.) Character device permissions start with a *c,* and the file has a major

number and a minor number. For example,

[yyang@fedora-serverA ~]$ **ls -l /dev/ttyS0**

crw-rw---- 1 root uucp 4, 64 2007-09-30 08:18 /dev/ttyS0

**Named Pipes**

*Named pipes* are a special type of file that allows for interprocess communication.

Using the **mknod** command, you can create a named pipe file that one process can

open for reading and another process can open for writing, thus allowing the two to

communicate with one another. This works especially well when a program refuses

to take input from a command-line pipe, but another program needs to feed the

other one data and you don’t have the disk space for a temporary file.

For a named pipe file, the first character of its file permissions is a *p*. For example, if a

named pipe called mypipe exists in your present working directory (PWD), a long listing

of the named pipe file would show this:

[yyang@fedora-serverA ~]$ **ls -l mypipe**

prw-r--r-- 1 root root 0 Mar 16 10:47 mypipe

**Listing Files: ls**

Out of necessity, we have been using the **ls** command in previous sections and chapters

of this book. We will look at the **ls** command and its options in more details here.

The **ls** command is used to list all the files in a directory. Of more than 50 available

options, the ones listed in Table 5-2 are the most commonly used. The options can be

used in any combination.

To list all files in a directory with a long listing, type this command:

[yyang@fedora-serverA ~]$ **ls -la**

To list a directory’s nonhidden files that start with the letter A, type this:

[yyang@fedora-serverA ~]$ **ls A\***

Chapter 5: The Command Line 115

***TIP*** Linux/UNIX is case-sensitive. For example, a file named **thefile.txt** is very different from a file

named **Thefile.txt**.

If no such file exists in your working directory, **ls** prints out a message telling

you so.

**Change Ownership: chown**

The **chown** command allows you to change the ownership of a file to someone else. Only

the root user can do this. (Normal users may not give away file ownership or steal ownership

from another user.) The syntax of the command is as follows:

[root@fedora-serverA ~]# **chown [-R] *username filename***

where ***username*** is the login of the user to whom you want to assign ownership, and

***filename*** is the name of the file in question. The filename may be a directory as well.

The **-R** option applies when the specified filename is a directory name. This option

tells the command to recursively descend through the directory tree and apply the new

ownership, not only to the directory itself, but also to all of the files and directories

within it.

**Table 5-2.** Common ls Options

**Option for ls Description**

**-l** Long listing. In addition to the

filename, shows the file size, date/time,

permissions, ownership, and group

information.

**-a** All files. Shows all files in the directory,

including hidden files. Names of hidden

files begin with a period.

**-t** Lists in order of last modified time.

**-r** Reverses the listing.

**-1** Single-column listing.

**-R** Recursively lists all files and

subdirectories.

116 Linux Administration: A Beginner’s Guide

***NOTE*** The **chown** command supports a special syntax that allows you to also specify a group

name to assign to a file. The format of the command becomes **chown username.groupname**

**filename**.

**Change Group: chgrp**

The **chgrp** command-line utility lets you change the group settings of a file. It works

much like **chown**. Here is the format:

[root@fedora-serverA ~]# **chgrp [-R] *groupname filename***

where ***groupname*** is the name of the group to which you want to assign filename ownership.

The filename may be a directory as well.

The **-R** option applies when the specified filename is a directory name. As with

**chown**, the **-R** option tells the command to recursively descend through the directory

tree and apply the new ownership, not only to the directory itself, but also to all of the

files and directories within it.

**Change Mode: chmod**

Directories and files within the Linux system have permissions associated with them.

By default, permissions are set for the owner of the file, the group associated with the

file, and everyone else who can access the file (also known as owner, group, and other,

respectively). When you list files or directories, you see the permissions in the first column

of the output. Permissions are divided into four parts. The first part is represented

by the first character of the permission. Normal files have no special value and are represented

with a hyphen (**-**) character. If the file has a special attribute, it is represented by

a letter. The two special attributes we are most interested in here are directories (**d**) and

symbolic links (**l**).

The second, third, and fourth parts of a permission are represented in three-character

chunks. The first part indicates the file owner’s permission. The second part indicates

the group permission. The last part indicates the world permission. In the context of

UNIX, “world” means all users in the system, regardless of their group settings.

Following are the letters used to represent permissions and their corresponding values.

When you combine attributes, you add their values. The **chmod** command is used

to set permission values.

**Letter Permission Value**

**R** Read 4

**W** Write 2

**X** Execute 1

Chapter 5: The Command Line 117

Using the numeric command mode is typically known as the *octal* permissions, since

the value can range from 0–7. To change permissions on a file, you simply add these

values together for each permission you want to apply.

For example, if you want to make it so that just the user (owner) can have full access

(RWX) to a file called **foo**, you would type

[yyang@fedora-serverA ~]$ **chmod 700 foo**

What is important to note is that using the octal mode, you always *replace* any permissions

that were set. So if there was a file in **/usr/local** that was SetUID and you ran the

command **chmod -R 700 /usr/local**, that file will no longer be SetUID. If you want

to change certain bits, you should use the symbolic mode of **chmod**. This mode turns out

to be much easier to remember, and you can add, subtract, or overwrite permissions.

The symbolic form of **chmod** allows you to set the bits of the owner, the group, or

others. You can also set the bits for all. For example, if you want to change a file called

**foobar.sh** so that it is executable for the owner, you can run the following command:

[yyang@fedora-serverA ~]$ **chmod u+x foobar.sh**

If you want to change the group’s bit to execute also, use the following:

[yyang@fedora-serverA ~]$ **chmod ug+x foobar.sh**

If you need to specify different permissions for others, just add a comma and its permission

symbols, as here:

[yyang@fedora-serverA ~]$ **chmod ug+x,o-rwx foobar.sh**

If you do not want to add or subtract a permission bit, you can use the equal (=) sign

instead of a plus (+) sign or minus (-) sign. This will write the specific bits to the file and

erase any other bit for that permission. In the previous examples, we used + to add the

execute bit to the User and Group fields. If you want *only* the execute bit, you would

replace the + with =. There is also a fourth character you can use: **a**. This will apply the

permission bits to all of the fields.

The following list shows the most common combinations of the three permissions.

Other combinations, such as **-wx**, do exist, but they are rarely used.

**Letter Permission Value**

**---** No permissions 0

**r--** Read only 4

**Rw-** Read and write 6

**Rwx** Read, write, and execute 7

**r-x** Read and execute 5

**--x** Execute only 1

118 Linux Administration: A Beginner’s Guide

For each file, three of these three-letter chunks are grouped together. The first chunk

represents the permissions for the owner of the file, the second chunk represents the permissions

for the file’s group, and the last chunk represents the permissions for all users

on the system. Table 5-3 shows some permission combinations, their numeric equivalents,

and their descriptions.

**Table 5-3.** File Permissions

**Permission Numeric Equivalent Description**

**-rw-------** 600 Owner has read and write

permissions.

**-rw-r--r--** 644 Owner has read and write

permissions; group and world

have read-only permission.

**-rw-rw-rw-** 666 Everyone has read and write

permissions. Not recommended;

this combination allows the file

to be accessed and changed by

anyone.

**-rwx------** 700 Owner has read, write, and

execute permissions. Best

combination for programs or

executables that the owner

wishes to run.

**-rwxr-xr-x** 755 Owner has read, write, and

execute permissions. Everyone

else has read and execute

permissions.

**-rwxrwxrwx** 777 Everyone has read, write, and

execute permissions. Like the

666 setting, this combination

should be avoided.

**-rwx--x--x** 711 Owner has read, write,

and execute permissions;

everyone else has executeonly

permissions. Useful for

programs that you want to let

others run but not copy.

Chapter 5: The Command Line 119

**FILE MANAGEMENT AND MANIPULATION**

This section covers the basic command-line tools for managing files and directories.

Most of this will be familiar to anyone who has used a command-line interface—same

old functions, but new commands to execute.

**Copy Files: cp**

The **cp** command is used to copy files. It has a substantial number of options. See its man

page for additional details. By default, this command works silently, only displaying

status information if an error condition occurs. Following are the most common options

for **cp**:

**Option for cp Description**

**-f** Forces copy; does not ask for verification

**-I** Interactive copy; before each file is copied, verifies with user

**Table 5-3.** File Permissions