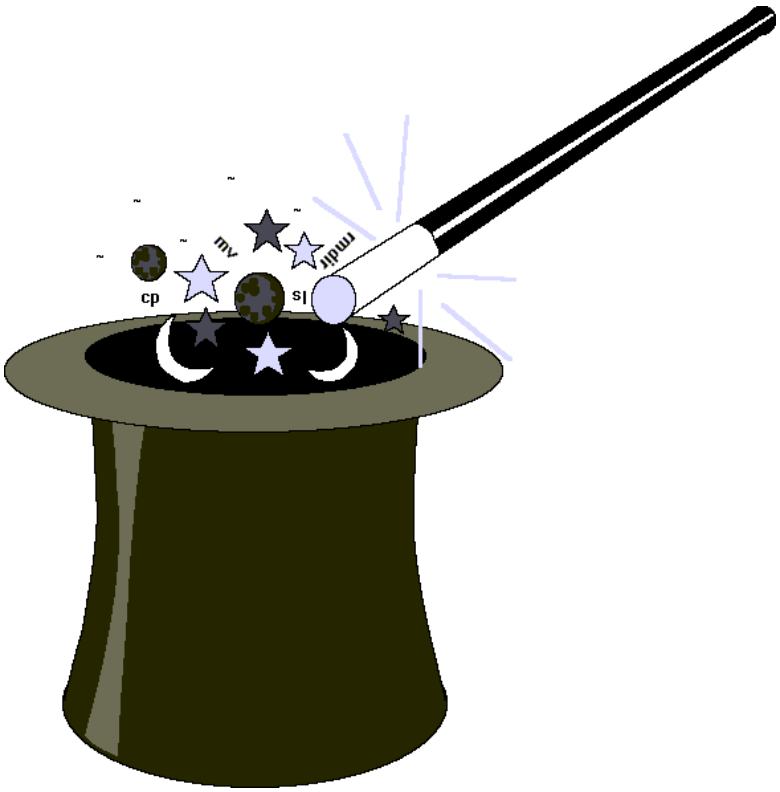
UNIVERSITY OF BIRMINGHAM
Birmingham Environment for Academic Research

Unix – Guide for Beginners (Part 2)



Alan Reed, Aslam Ghumra, Chris Bayliss, and Jenny Harrison

Edition 5.08

Table of Contents

Part 1: Security – File and Directory Access

As stated in the first course, there are three main types of access which can be applied to a file these can be applied selectively to yourself, your groups, or everyone else. These types of access are called file modes. The ***ls -l*** command can be used to display the file mode. The ***chmod*** command is used to change the file modes.

chmod command

The chmod command is used to change the modes (access permissions) associated with a file. The change is specific in the form:

chmod modes filename

The mode is three characters which represents who will have their access changed, the operation (whether the access is to be added, subtracted, or set) and what access permissions are being referred to:

who is affected is specified by one of the following:

u	**u**ser	the owner of the file
g	**g**roup	the group to which the owner belongs
o	**o**ther	everyone else
a	**a**ll	u, g, and o

operation is specified by one of the following:

+	add the specified permissions
-	Subtract the specified permissions
=	Assign the specified permissions, ignoring the current setting

access permission is specified by one or more of the following:

r	**R**ead
w	**W**rite
x	e**X**ecute (or list if a directory)

Note that in order to have access to a file, the user concerned also needs list access (x) to the directory which contains the file, and all directories above it up to and including root. You will need to be aware of this if you set up files which are to be read by others, for example World Wide Web pages.

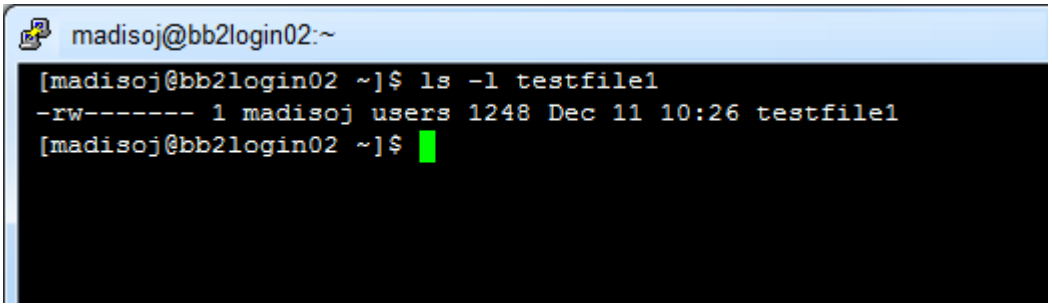
In the example below the ***chmod*** command is given together with the old and new modes for the files concerned.

Example 1: Given a file named **testfile1** with the default modes (read and write for the user), add read mode for other:

To list the default access for testfile1, type:

ls -l testfile1

Before:	User	Group	Others
	rw-	---	---



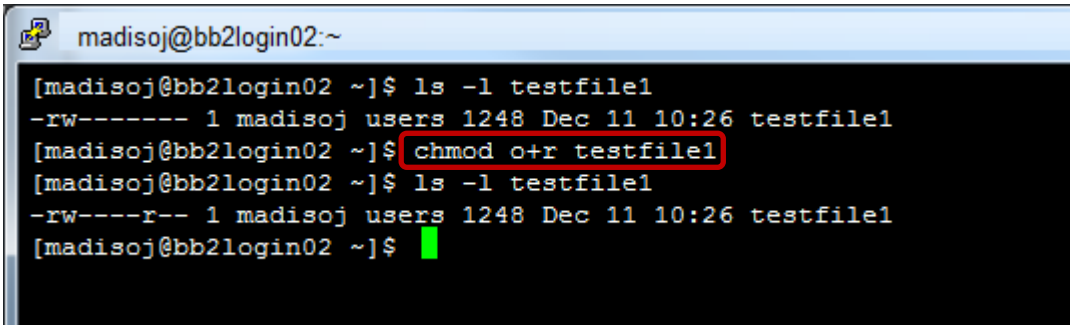
To change the other's access modes, type:

chmod o+r testfile1

To list the new access for testfile1, type:

ls -l testfile1

After:	User	Group	Others
	rw-	---	r--



Example2: Remove read and write access for everyone else from the file testfile1:

To list the access modes for testfile 1, type:

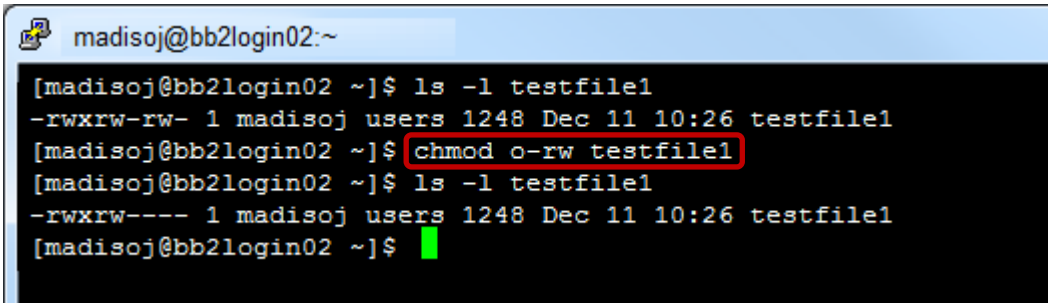
ls -l testfile1 (the access modes should be as follows)

Before:	User	Group	Others
	rwx	rw-	rw-

To change other's access modes, type:

chmod o-rw testfile1

After:	User	Group	Others
	rw-	rw-	---



Example 3: Give everyone (owner, group, and other) read, write, and execute permissions on the file named testfile1.

To check the access mode for test file1, type:

ls –l testfile1

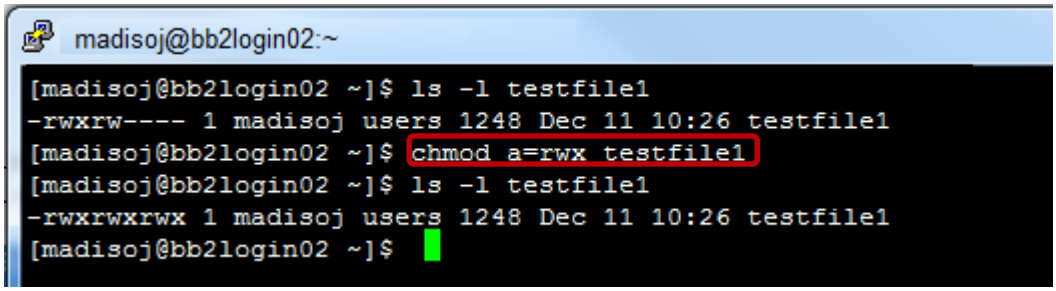
Before: (access is irrelevant)	User	Group	Others
	rwx	rw-	rw-

To modify access mode for testfile1 which will give everyone (user, group, and other) read, write, and execute, type:

chmod a=rwx testfile1

After:	User	Group	Others
	rw-	rw-	---

WARNING: Giving 'write access' to other (everyone) is a very risky thing to do!


madisoj@bb2login02:~
[madisoj@bb2login02 ~]$ ls -l testfile1
-rwxrw---- 1 madisoj users 1248 Dec 11 10:26 testfile1
[madisoj@bb2login02 ~]$ chmod a=rwx testfile1
[madisoj@bb2login02 ~]$ ls -l testfile1
-rwxrwxrwx 1 madisoj users 1248 Dec 11 10:26 testfile1
[madisoj@bb2login02 ~]$

Chmod by numbers

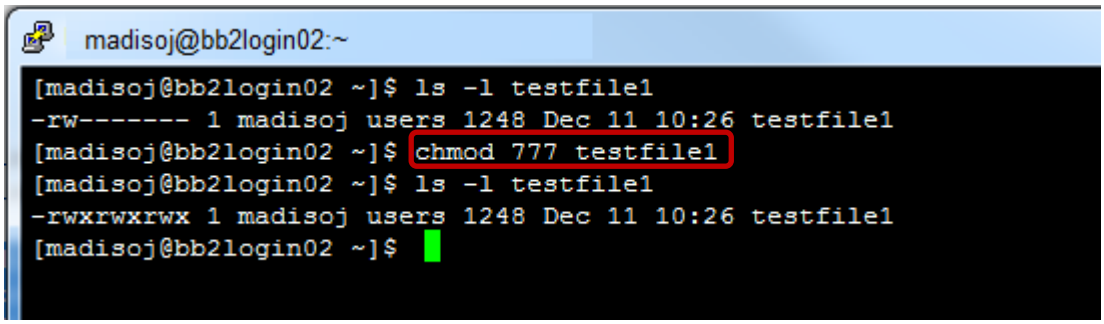
File access can be set using numbers rather than letters. The access is specified by three digits, the first being for the User, the second being for the Group, and the third for Other. Each digit is calculated by adding the numbers associated with read, write, and execute access. The associated numbers are as follows:

4	Read
2	Write
1	eXecute (or search in the case of a directory)

Chmod will then take the numbers as an argument instead of letters as above. In this way, a particular mode can be set instead by adding and subtracting from specific modes. For example:

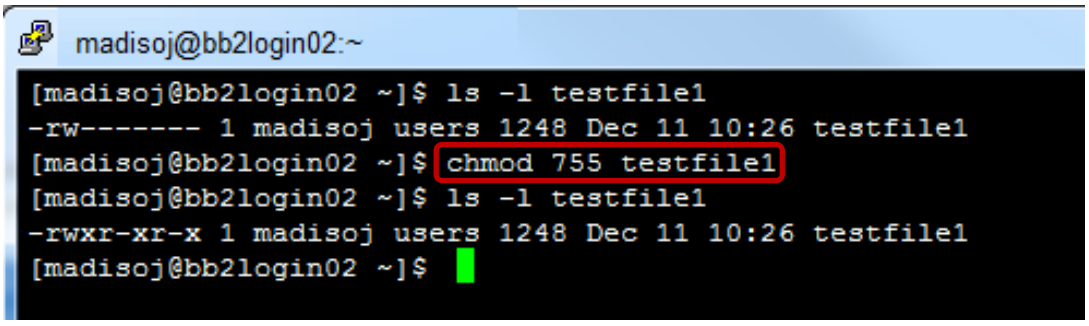
chmod 777 testfile1

would set read, write, and execute modes for user, group, and other. You would no normally wish to give others write access.



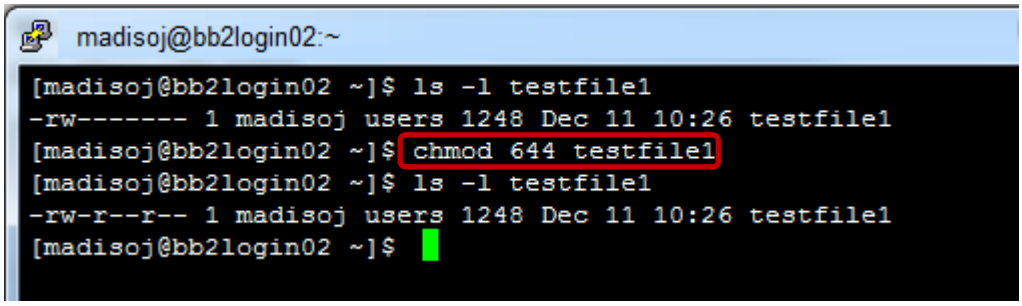
chmod 755 testfile1

would set read, write, and execute for user, and just ready and execute for group and other.



chmod 644 testfile1

would set read and write for user, and just read for group and other.



Part 2: Strings, Patterns, Regular Expressions, and grep

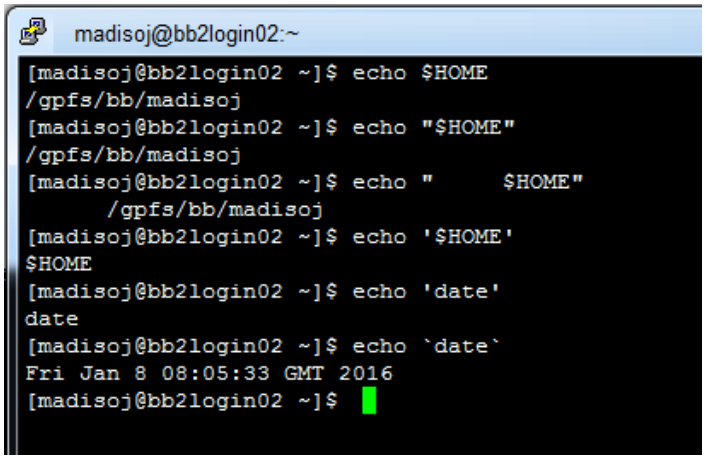
There is a detailed treatment of strings, patterns, and regular expressions in the appendix. This section covers a basic description with a few examples.

Strings

These are sequences of ASCII characters, not just letters. These are taken as arguments to various commands. As special characters and spaces can be included in strings, it is sometimes necessary to quote them, as spaces would normally separate different arguments and you may or may not wish to preserve special characters.

There are three types of quotes recognised by UNIX. Single quotes pas the literal string without further processing. Double quotes allow the substitution of variables, and back quotes are used when the output of a command is to be used as a string.

The differences are shown using the ***echo*** command.



Patterns

Patterns match one or more strings and may contain wildcards (* ?) or ranges of characters ([a-z], [1-9]). Note that quotes will inhibit the expansion of wildcards before the string is passed to the command.

Examples using the command ***ls***:

ls *	List all files
ls ????	List all files with four characters
ls [0-9]*	List all files starting with a number
ls `*`	List files named *

Regular expressions

These are more advanced forms of patterns used by some UNIX commands and contain even more special characters. For example, **^** matches start of line and **$** matches end of line. Note that the ***** has a different meaning than in patterns; it means zero or more repetitions of an expression.

Examples:

fred	fred anywhere on the line
^fred	fred at the start of a line
fred$	fred at the end of a line
.	any character
.*	any character repeated zero or more times

grep

This command finds all occurrences of a regular expression in a file or from the standard input piped to it and displays all lines containing the regular expression. Regular expressions are explained in Appendix I. These are used by a variety of programs to represent particular character strings.

The grep command can be useful when trying to find a given word or expression in a file or group of files. For example, in the file **terminal.txt**, to find every line with the word window in it you would type:

grep "window" terminal.txt

The lines containing the word window would then be displayed. By default, the search is case sensitive. If you did not wish the match to be case sensitive, you would use the ***-i*** flag.

grep -i "window" terminal.txt

The command can be combined with pipes, wildcards, and other commands. For example,

grep -i "window" *.txt

would find the word **window** in any file with a **.txt** extension.

The regular expression can be used to search for a string at the start or end of a line. For example:

grep -i "^window" *.txt

grep -i "window$" *.txt

If you wish to display a list of full details of directories in the current directory, you could pipe the output of ls –l into grep. For example:

ls -l | grep "^d"

As ^ in a regular expression matches beginning of line, this displays only lines starting with a **d**.

Part 3: More about shells

The Korn shell supports command line recall and edit. This can save a lot of typing, as it gives you the ability to recall previous command lines, alter them, and reissue them. Two modes are supported, Emacs mode and vi mode because of the style of the editor that is copied. In order to enable Emacs mode, the built-in set command is used.

set -o emacs

This is normally done in your .profile file (this is discussed later).

The basic functions supported are as follows:

Ctrl + P	Recall previous line
Ctrl + N	Recall next line (only useful after one or more Ctrl + P)
Ctrl + B	Move back one character
Ctrl + F	More forward one character

There are others which are documented in the ksh manual pages.

ksh Programming

Shells can be used to write programs. The simplest programs are files which contain a sequence of commands and allow the passing of parameters from the command line. This is sufficient for many uses. However, the Korn shell has many features which allow complex programs to be written, such as conditional statements and loops. This course does not cover these, but they are mentioned here in case they are of interest. Full details are contained in the manual pages for the Korn shell.

A simple example program is given below:

#!/bin/ksh	*Sets the shell to be used*
#simple demonstration program	*Comment denoted by #*
#	*Comment denoted by #*
echo "What is your name?"	*Prompts for user name*
read name	*Reads user input*
echo "Hello, $name"	*Types Hello, followed by the user name from previous line*

The program is in a file called prog1.

An example of running it is given below:

prog1

What is your name?

Chris

Hello, Chris

Note that before the program can be run in this way, you must make sure that you have read and executed access to the file and that "working directory" is on your PATH (see below).

The other basic feature which is very useful is the passing of parameters from the command line. These are available as variables called $1, $2, $3, etc. All parameters together are available in a variable called $*.

For example, the program

#!/bin/ksh

echo "Hello, $1."

would print Hello, followed by the first parameter on the command line. The file in this example is called prog2. An example of running it follows:

prog2 Chris

Hello, Chris.

.profile

This is a special shell program which is run every time you login, and it resides in your home directory. Some parameters are set in the system profile, which is a special profile run when everyone logs in. A typical .profile is given below:

```
# ACS ksh .profile
# Last modified: 18-Feb-94 JWH
#
# This is a file of commands which is read in by your shell when
# it first starts up. You may edit it as you wish.
#
PATH=$PATH:$HOME/bin:.
export PATH
#
umask 077
set -o ignoreeof
set -o emacs
#
```

The first few lines start with **#** and are comments. Normally, the first line selects the shell to be used with information selecting the shell; however, this is not done in the .profile as this is run within the login shell, which is set for each user in a system.

PATH=$PATH:$HOME/bin:. and export PATH

These set your PATH environment variable. The directories $HOME/bin and . (your working directory) are added to your PATH variable. Note that $PATH is expanded to the original value of PATH. This method of modifying PATH avoids the possibility of omitting some of the directories needed to find the command. The other method is to specify all directories needed in the final PATH and is not recommended, unless you need to remove directories from the PATH set by the system.

unmask 077

This sets the user file creation mask. The effect of this setting is to ensure that nobody else can read or write files which you create. The access modes can be set for individual files or groups of files

using the **chmod** command, should you need to give other people access to them after their creation.

set –o ignoreeof

Normally, Ctrl + D signifies the end of file. This has the same effect as exit to the shell. This command disables this feature, preventing accidental logouts by use of the Ctrl + D.

set –o emacs

This sets Emacs mode for command line recall and editing.

If you wish to change any settings or add any commands, you may do this by editing your .profile file. For example, if you wish to display the date and time whenever you login, yo could add the command:

echo It is `date`

More redirection

There is one type of redirections which allows the input of several lines to a command. An example is given below.

```
cat <<XXXX
Twinkle twinkle little star,
How I wonder what you are,
Up above the stars so high,
Like a diamond in the sky
XXXX
```

The above example simply displays the verse on the terminal. However, in programming, the pipe could be into a command to send Email. For example, and if necessary, variables could be included in the text. The XXXX is a sequence of characters which are given to indicate end of input. Any sequence of characters can be used, but it is wise to use a sequence unlikely to appear in the command input.

There is a command which can be useful when redirecting standard output. This is **tee** and allows output from a command to be directed to a file and the terminal at the same time. For example:

ls -l | tee dirlist

would direct the output from the ***ls*** command into the file dirlist as well as displaying it on a terminal.

In addition to standard output, commands also produce error output. This is used for error message and is treated differently by the system and is not redirected in the same way as standard output. However, error output can be directed to a file. For example:

ls -l xxxx 2> temperrors

Would put any errors produced by the ***ls*** command into the file called temperrors instead of displaying the errors on the terminal. This feature is particularly useful if you issue a command which you know will produce a lot of error messages and you do not wish to display them at the terminal. An example of this occurs later in the course.

Part 4: Finding Files

find

List will only list files in a directory or those immediately below it. The ***find*** command will find files within a directory hierarchy, wherever they are within the tree. For example, to find the file treasure, the following would be used:

find . -name "treasure" -print

The starting directory in this case is the current directory – all directories below it in the tree are searched. Any directory could be specified at this point. If you wished to search the entire hierarchy, you would use:

find / -name "treasure" -print

Note that this will take some time. If you wish to cancel the command before it has finished, you can do so by typing 

+ 

. You can cancel many other commands in this way. The other problem with running a command like the one above is that you can generate a lot of error messages. In this case, the errors are because you do not have sufficient access to some of the directories in the filestore to examine their contents. If you wish to suppress the error messages, you can use redirection. This is demonstrated in the following example:

find / -name "treasure" -print 2> /dev/null

The file /dev/null is a special device on UNIX and is effectively a "black hole". Any unwanted output can be directed to it and it will be discarded.

If you want to search for a pattern rather than a fixed filename, you need to put the pattern in quotation marks. This is to prevent the shell from expanding the pattern before passing the pattern to the command. If you omit the quotes, the shell would find matches for the pattern in the current directory, and pass them as arguments to the find command. This would almost certainly not produce the desired results.

find . -name "treasure*" -print

In all of the above examples, the *-print* option has been used. The *-print* displays the pathname when it finds the file. If *-print* is missing, the command will find the file and not display anything. This may seem a little bizarre to the uninitiated and volatile fuel for the UNIX basher, but *find* has other options and you do not always want to display the pathname if using some of the other options.