Linux: File System Commands - Part II

Written by Mayakkannan Subhas (MK)

Introduction

In the previous chapter, **Linux; File System Commands - Part I**, we have discussed the importance of file system and its use in storing, retrieving files and the metadata. We have also discussed about absolute and relative paths, the way to access files stored in the system and the first set of commands to create files, directories and links (shortcuts) and commands to list files and directories and get information about different type of files.

In this chapter, we will look into **wildcards**, a feature provided by shell to create patterns to match file and directories and more commands to perform operations such as copy, move, rename and delete files and directories, list contents of files and directories, get file usage statics and display used and free space of the disks attached to the system.

Points to remember

1. Linux treats everything that is stored in the system as **file**
2. A **directory** is a special type of file that has references to other files and directories stored in it
3. File names that start with a `.` are called hidden files and we need options to display these files
4. The file system has a single entry point `/` aka the **root directory**
5. There are two type of path names to refer files and directories; **absolute** and **relative** paths
6. Most of the commands are silent when executed successfully. We need to enable the verbose option `-v` or `--verbose` to get message about the activity

Wildcards

Wildcards are the shell feature that allows us to create patterns that can be used as argument in place where files or directories are expected. The pattern may match zero or more files thus helping us reduce the typing and avoid mistakes

Wildcard	Description
`*`	represents zero or more characters
`?`	represent any one character
`[abc]`	any one of the listed characters
`[^abc]`	any one **ASCII** character other than the listed characters
`[!abc]`	same as `[^abc]`
`[a-z]`	any one of the lowercase alphabet; `-` represents range
`[^a-z]`	any one character other than the lowercase alphabet
`[!a-z]`	same as `[^a-z]`
`[[class]]`	matches a group of characters referred by the **class** name

Common character groups: `[[class]]`

> The class name is a placeholder for a single character from the character set

|Class|Description| | `[:alpha:]` |upper and lowercase alphabet| | `[:lower:]` |lowercase alphabet| | `[:upper:]` |uppercase alphabet| | `[:digit:]` |numbers: 0-9| | `[:xdigit:]` |hexadecimal numbers: 0-9A-Fa-f| | `[:alnum:]` |alphabets and numbers 0-9A-Za-z| | `[:space:]` |space, tab, newline, carriage return and vertical tab|

Wildcard Examples:

- `*.txt` : any file that ends with `.txt` that has **Zero or more characters prefix**; `.txt` , `a.txt` , `sample_file.txt` ,...
- `a?.txt` : any file that starts with an `a` followed by **only one character** and ends with `.txt` ; `a1.txt` , `az.txt` , `a@.txt` ,...
- `[aeiou][0-9].txt` : any file that starts with a **lowercase vowel** followed by a **number**; `a1.txt` , `z9.txt` , ...

- `[aeiou][^has 0-9].txt` : any file that starts with a **lowercase vowel** followed by a **non-number** character; `aX.txt` , `zZ.txt` , ...
- `[[:xdigit:]][:xdigit:]][[:digit:]]` : files that are **three character long**, first 2 chars are **hexadecimal digits** `0-9A-Fa-f` and one **number** `0-9` ; `ff9` , `ef3` , `bc4`

Brace Expansion: `{...}`

There is another special character set called the *brace expansion* that is used as wildcard. This is not used to create patterns to match existing files and directories, instead it can be used to generate combination of text patterns.

Type	Description
`A{op1,op2,op3}Z`	creates 3 strings with prefix + {each option} + suffix
`{1..10}`	generates number between 1-10, both inclusive
`{01..100}`	generated `001 002 003 ... 100` in latest BASH versions;

> if **brace expansion** is used in commands that expects actual files and directories as arguments, the commands will produce file not found error for text generated that are not the name of existing files; `rmdir` command for example

Brace Expansion Examples:

- `echo sample.{csv,txt,dat}` will display `sample.txt` , `sample.csv` and `sample.dat`
- `touch sample.{csv,txt,dat}` will create there empty files named `sample.txt` , `sample.csv` and `sample.dat`
- `echo {a,x,z}{1..3}` will display `a1 a2 a3 x1 x2 x3 z1 z2 z3`
- `mkdir -pv 20{20,21}/{01..12}/{01..31}` creates nested directories; `2020` and `2021` as top-level directories and each directory contains 12 directories for months `01-12` and each month has 31 directories for days `01-31` . This command creates around **750+** directories in one go

> if the **Bash version** is less than 4.x `echo $BASH_VERSION` then the month and date may not have leading zeroes for values `1-9`

Escaping and Quoting

Since the **shell** has special meanings for certain characters such as the ones used in wildcards, we need a mechanism to pass these characters as-is without special meanings. We need a way to decide when to use these characters to represent the special meanings and when to use as-it-is. There are two ways to accomplish this

Escaping

Unix supports escape-sequences like `\t`, `\n` for TAB and newline. Here adding backslash character (`\`) in front of `t` provides the special meaning TAB. We can use the backslash to accomplish the opposite; that is, adding a backslash in front of the special character to inform the shell to treat the character as-it-is.

For example, if we pass the `*` as argument to the `echo` command, the shell expands the `*` to file list in the current working directory and passes it to `echo` that makes echo print all the files. If we escape `*` as `\*` then `echo` will treat `*` as `*` itself.

```
$ echo *
files      hfs

$ echo \*
*

# not what we wanted
$ echo 5 * review
5 files hfs reviews

$ echo 5 \* reviews
5 * reviews

$ echo a?*
a file with spaces.txt another file with spaces.txt
$ echo 'a?*'a?*
```

Quoting

There are times where we need to keep space separated words a one single entity instead of list of words since **shell** uses **space** as a delimiter to split commands, options and arguments. . For example, we want to pass a sentence to some script as a single argument, we need to wrap the sentence with something to keep the text intact. We can accomplish this using the **quotes**. In fact, Unix systems support there type of quotes.

1. Double quotes or weak quotes
2. Single quotes or strong quotes
3. Backticks or command substitution

Double quotes The shell interpolates the wildcards and special characters such as `$`, `!`, `\` and escape sequences `\t`, `\n` etc.. Since it does not keep all the characters as-they-are, the double quote is also called *weak quotes*.

The `$` symbol is used as a prefix in **shell variables**. If a variable is enclosed inside double-quotes, the value will be substituted. We can escape the `$` as `\$` to prevent the interpolation of variables.

```
echo "\$USER: $USER"
$USER: mbose

# !! refers to the last command executed
#    !! will be substituted with the last command
#    Shell will display the expansion before running the command
$ echo "!!"
echo "echo "\$USER: $USER""
echo $USER: mbose
```

In order to create file with special characters like spaces, `*` in its name, we either need to wrap it inside single or double quotes. The same rules apply when we use these file names as arguments.

```
$ touch "a file with spaces.txt"
$ touch another\ file\ with\ spaces.txt
$ ls
```

```
'a file with spaces.txt'  'another file with spaces.txt'

$ file a\ file\ with\ spaces.txt
a file with spaces.txt: empty
$ file 'a file with spaces.txt'a file with spaces.txt: empty
```

Single quotes

Wrapping the text using *single quotes* also known as *strong quote* suppresses the special meaning of any wildcard, special characters. This is the safest way to quote text. The only restriction is that we cannot have a single-quote inside the text that is wrapped with a pair of single quotes.

> The **GNU Bash Manual** states that 'a single quote may not occur between single quotes, even when escaped with a slash `\'` it will cause error

```
$ echo 'Hello $USER'
Hello $USER

$ echo "Hello $USER"
Hello mbose
```

Backticks There is another special set of quotes that is rarely used called *back-ticks* that is found under the **ESC** key. This set of quotes are used for **command substitution**, that is, the content inside the backticks will be treated and executed as command and the outcome will be replaced by the actual command. The *backticks* quoted string can be nested within double quotes without losing its special meaning.

> The old way of enabling command substitution is by wrapping the command with a pair of``. However it is recommended to use `$()` instead of *backticks*

```
$ echo "Today is date"
Today is date

$ echo "Today is `date`"
```

```
Today is Wed Jul 14 21:18:16 IST 2021

# use $(..) instead of ``
$ echo "Today is $(date)"
Today is Wed Jul 14 21:18:16 IST 2021
```

File System Commands: Part II

#	Name	Description
1	`cp`	copy files and directories
2	`mv`	rename or move files and directories
3	`rm`	remove files and directories with contents
4	`unlink`	remove single file. a simple version of `rm`
5	`ls`	list files and directories
6	`dir`	list files and directories; same as `ls -C`
7	`du`	display the disk usage info of a file or a directory
8	`df`	display free and used space from available hard disks

`cp` : copy files and directories

The `cp` command is used to **copy** one or more files and directories. This command can be used to take backups of existing files. The backup will have the timestamps of the copy operations by default. We have options to copy the metadata of the original file along with the contents.

WARNING: The `cp` command automatically overrides if the file we are copying has another file in the destination with a same name

> The below 3 options works the same for `cp`, `mv` and `rm` commands

Options	Description
`-v`	verbose mode, display messages about the action
`-f`	force operations, if possible

Options	Description
`-i`	interactive mode, ask before performing the action

Other options

Options	Description
`-p`	preserve metadata of source file; owner, timestamp, permissions,..
`--parents`	create full path including directories under the destination
`-R`	recursive copy of directories
`-r`	same as `-R`

Setup Sample Files and Directories for Demo

```
$ mkdir cp_mv_rm
$ cd  cp_mv_rm
$ touch sample_{1..3}.txt
$ mkdir backup
$ tree
.
├── backup
├── sample_1.txt
├── sample_2.txt
└── sample_3.txt
```

Copy Demo

```
# copy file silently
$ cp sample_1.txt backup
$ tree backup/
backup/
└── sample_1.txt

# cp -v: copy with verbose mode on
```

```
$ cp -v sample_2.txt backup/
'sample_2.txt' -> 'backup/sample_2.txt'
```

Preserve file metadata while copying files

```
# cp: with and without preserve metadata
$ cp sample_1.txt s1a.txt
$ cp -p sample_1.txt s1b.txt

# more on `ls -l` later
#  s1b.txt and sample_1.txt has same timestamp: -p effect
#  s1a.txt has the timestamp at the time of copying
$ ls -l
-rw-r--r-- 1 mbose admin    0 Jul 10 15:30 s1a.txt
-rw-r--r-- 1 mbose admin    0 Jul 10 15:24 s1b.txt
-rw-r--r-- 1 mbose admin    0 Jul 10 15:24 sample_1.txt
```

Create source file's directories inside the target before copy

```
# cp --parents : copy full path
$ mkdir -pv data/input
mkdir: created directory 'data'
mkdir: created directory 'data/input'
$ touch data/input/demo.csv
$ tree data
data
└── input
    └── demo.csv

$ cp -v data/input/demo.csv backup
'data/input/demo.csv' -> 'backup/demo.csv'

# this run creates the entire path
$ cp -v --parents data/input/demo.csv backup
data -> backup/data
data/input -> backup/data/input
```

```
'data/input/demo.csv' -> 'backup/data/input/demo.csv'

# the 1st cp command copied demo.csv under backup
# the 2nd cp copied created data/input and copied the file
$ tree
.
├── backup
│   ├── data
│   │   └── input
│   │       └── demo.csv
│   ├── demo.csv
├── data
    └── input
        └── demo.csv
```

Recursive copy

```
$ mkdir latest

# copy fails
$ cp data latest/
cp: -r not specified; omitting directory 'data'
$ tree latest/
latest/

# recursive copy
$ cp -Rv data latest/
'data' -> 'latest/data'
'data/input' -> 'latest/data/input'
'data/input/demo.csv' -> 'latest/data/input/demo.csv'
```

`mv` : move or rename files and directories

The `mv` command can be used either move files from one location to another or rename files

```
$ cd cp_mv_rm
$ tree
.
├── backup
...
├── sample_1.txt
├── sample_2.txt
└── sample_3.txt

# move file sample_3.txt from current dir to backup
$ mv -v sample_3.txt backup/
renamed 'sample_3.txt' -> 'backup/sample_3.txt'
$ tree
.
├── backup
│   └── sample_3.txt
...
├── sample_1.txt
└── sample_2.txt

# renaming files within same directory
$ mv -v sample_1.txt sample_1.csv
renamed 'sample_1.txt' -> 'sample_1.csv'

# we can rename directories too
$ mv -v latest now
renamed 'latest' -> 'now'
```

`rm` : remove files and directories

The `rm` command is used to delete files and directories that are not empty.

Common Options: `-v` , `-f` , `-i` - same as `cp` and `-r` is recursive delete that is used to remove directories with contents by emptying the contents first and finally deleting the directory itself

WARNING 1: Unlike Windows, we do not have **Recycle Bin** in Linux and once deleted the files are gone. Please exercise caution while using `rm`

WARNING 2: Use `rm -r DIR` with caution as it will wipe out the entire contents. Extra caution is needed when the DIR has wildcards

TIPS: Use the `ls` command with recursion option `ls -R` with same argument for `rm -r` first, verify the list before actually running `rm -r`

```
$ cd cp_mv_rm
# remove a file: success
$ rm -v sample_1.csv
removed 'sample_1.csv'

# remove a read-only file: we can say `y` or use -f to delete
$ rm -v sample_2.txt
rm: remove write-protected regular empty file 'sample_2.txt'? n
$ rm -vf sample_2.txt
removed 'sample_2.txt'

# remove directory that are not empty
$ rm -rv data/
removed 'data/input/demo.csv'
removed directory 'data/input'
removed directory 'data/'
```

`unlink` : remove a single file

This is a simple version of `rm` that accepts an actual file name as argument and deletes that file. If argument is a wildcard or directory, `unlink` will error out.

```
$ mkdir data
$ touch a{1..3}.txt
$ ls
a1.txt  a2.txt  a3.txt  data

# delete a1.txt: success
```

```
$ unlink a1.txt
$ ls
a2.txt  a3.txt  data
```

Errors: wildcards and directories are not valid arguments

```
# delete directory: error
$ unlink data
unlink: cannot unlink 'data': Is a directory

# delete multiple files: error
$ unlink a*
unlink: extra operand ‘a3.txt’
```

`du` : display disk usage statistics of files and directories

The `du` command displays the size occupied by a given file or directory passed as argument or current working directory is used as default.

If the argument is a directory, the `du` command displays the size of all its sub-directories and files by default. Options are available to get summary statistics alone

Option	Description
`-s`	display summary only
`-h`	display size in human readable form - K, M,.. for KB, MB,..
`-d N`	consider `N` level of subdirectories only
`-b`	consider actual byte size, usually smaller than used size

```
# size of all directories
$ du
4       ./data/input
8       ./data
4       ./temp/data
8       ./temp
```

```
4       ./config/db
8       ./config
4       ./shortcuts
36      .

# just the summary
$ du
36      .

# actual vs used size of files
# actual size: won't consider holes in a spare file
$ du -b -h sample.txt
1.0K    sample.txt

# used size
$ du -h sample.txt
4.0K    sample.txt
```

Real life use cases

```
# overall size of current user's home directory
$ du -sh ~
540M    /home/mbose

# run as admin: all user's home directories
cd /home
$ sudo du -d 1 -h
400M    ./admin
70M     ./hradmin
30M     ./lost+found
350M    ./mbose
100M    ./mk
150M    ./dbauser
150M    ./awsuser
1G      ./itadmin
2.2G    .
```

`df` : display free and used disk space

The `df` command displays the total, used and available spaces and used percentage of all disks in the system. The most commonly used option is `-h` to print size in human readable format as in KB, MB, GB etc...

```
$ df -h
Filesystem  Size  Used  Avail  Use%  Mounted on
overlay     60G   44G   17G    73%   /
tmpfs       64M   0     64M    0%    /dev
tmpfs       3.9G  0     3.9G   0%    /sys/fs/cgroup
/dev/sda1   60G   44G   17G    73%   /root
/dev/root   2.0G  1.2G  820M   59%   /lib/modules
shm         64M   0     64M    0%    /dev/shm
```

`ls` : list files and directories

The `ls` command is used to list the files and directories of the current working directory by default. It can accept a file, directory or a wildcard. This is one of the most versatile beginner level commands the we will encounter. The `ls` command has probably 40-50 different options to control the way the command list the files.

This very basic information we get from the `ls` command is the file and directory names sorted in alphabetic order. In addition to the name, the options enable the `ls` to control the format of the display and what other metadata about the files can be displayed. Common metadata include

- file type
- permission settings
- owner
- group the owner belongs to
- file size
- modification timestamp

** The permission, owner and group metadata will be discussed in the future chapter on **Permissions**.

Option	Description
`-1`	display one file per row; default is columnar display
`-a`	display hidden files; `--all`
`-A`	display hidden files but discard `.` and `..`
`-F`	append a unique character at the end to identify file types
`-i`	display **inode**, a unique number for each file
`-l`	long list: permission, file type, size, owner etc..
`-t`	sort by modification time, newest first
`-S`	sort by file size, largest first
`-h`	display file size in human-readable form; K, M, G, T etc..
`-r`	reverse sorting, used along with `-S` or `-t`
`-R`	recursively list contents of sub-directories

For this demo, I have created files with different sizes. I have also copied files with different timestamps using `cp --preserve` option.

Setup sample files, directories, links,...

```
# create sample files, dirs, links under "ls_demo"
$ touch sample.txt demo.csv names.txt backup.zip
$ mkdir data temp config
$ ln -s config cfg
$ cp ~/bin/main.sh .

# hidden file
$ touch .gitignore
```

Simple listings: no options, `-1` , `-a` , `-A` and `-F`

Every directory has two entries; `.` and `..` that refers to the current directory and the parent directory respectively. These entries are not visible by default as Unix treats everything that starts with `.` as hidden files. If you recollect, these two are used in relative paths as well.

```
# ls: current dir as arg, list in alphabetical order, columnar
$ ls
backup.zip  cfg        config      data   demo.csv
main.sh     names.txt  sample.txt  temp

# ls -1: one entry per line
backup.zip
cfg
config
data
demo.csv
names.txt
sample.txt
temp

# ls -a: display hidden files
#   . and .. are references to current and parent dirs
$ ls -a
.           ..         backup.zip  cfg       config
data        demo.csv   .gitignore  main.sh   names.txt
sample.txt  temp

# ls -A : same as -a, discards . and ..
$ ls -A
backup.zip  cfg       config      data        demo.csv   .git-
ignore  main.sh   names.txt   sample.txt  temp

# ls -F: mark `/, @, *` suffix for dir, link and executable
$ ls -F
backup.zip  cfg@       config/     data/   demo.csv
main.sh*    names.txt  sample.txt  temp/
```

So far, we have seen just the file names being displayed and the options to control what is included and adding symbols to highlight file types. The following examples will display more than just the file names.

Display inode - an unique number allocated to each file

> The **inode** also known as **index node** is a data structure that stores file attributes and is referred by an unique integer

```
# display inode and file name, one per line
$ ls -1i
131218 backup.zip
131222 cfg
...
...
131214 names.txt
131209 sample.txt
131220 temp
```

Display metadata of files, sorted alphabetically

Col	Description
1	file type (byte 1), Permission (bytes 2-10)
2	number of references, files usually 1, dirs has number of subdirectories including `.` and `..`
3	owner of the file
4	group of the owner
5	file size
6	Month in `mmm` format
7	day of the month
8	time in HH:MM for files less than one year old year in YYYY format for files more than one year old
9	file name

```
# the -l aka long option displays file type, permission
# owner, group, file size, modified timestamp and file name
-rw-r--r-- 1 mbose admin    0 Jul 11 06:24 backup.zip
lrwxrwxrwx 1 mbose admin    6 Jul 11 06:25 cfg -> config
drwxr-xr-x 2 mbose admin 4096 Jul 11 06:24 config
```

```
...
...
-rw-r--r-- 1 mbose admin    0 Jul 11 06:24 sample.txt
drwxr-xr-x 2 mbose admin 4096 Jul 11 06:24 temp
```

Sort by modification timestamp `-t` and file size `-S`

For this demo, we need to use the `-l` option to compare the sorted order.

```
# sort by size descending
$ ls -lS
drwxr-xr-x 2 mbose admin 4096 Jul 11 06:24 config
drwxr-xr-x 3 mbose admin 4096 Jul 11 11:51 data
drwxr-xr-x 2 mbose admin 4096 Jul 11 06:24 temp
lrwxrwxrwx 1 mbose admin   10 Jul 11 11:52 backup -> backup.zip
lrwxrwxrwx 1 mbose admin    6 Jul 11 06:25 cfg -> config
-rw-r--r-- 1 mbose admin    0 Jul 11 06:24 backup.zip

# sort by modification timestamp descending
#   set timestamp manually, current timestamp for backup.zip
$ touch -d "5 months ago" main.sh
$ touch backup.zip

$ ls -lt
-rw-r--r-- 1 mbose admin    0 Jul 11 12:12 backup.zip
lrwxrwxrwx 1 mbose admin   10 Jul 11 11:52 backup -> backup.zip
drwxr-xr-x 3 mbose admin 4096 Jul 11 11:51 data
...
...
-rw-r--r-- 1 mbose admin    0 Jul 11 06:24 sample.txt
-rwxr-xr-x 1 mbose admin    0 Feb 11 12:12 main.sh
-rw-r--r-- 1 mbose admin    0 Jan  9  2021 names.txt
```

Reverse sort `-t`

```
$ ls -ltr
-rw-r--r-- 1 mbose admin    0 Jan  9  2021 names.txt
```

```
-rwxr-xr-x 1 mbose admin    0 Feb 11 12:12 main.sh
-rw-r--r-- 1 mbose admin    0 Jul 11 06:24 sample.txt
-rw-r--r-- 1 mbose admin    0 Jul 11 06:24 demo.csv
...
```

`dir` : list files and directories

The `dir` command performs same actions as the `ls` command and it may be suitable for people who have come from Windows environment. This command is not available in many OS versions; For example, the **Mac systems** doesn't have the `dir` command. If your system has the `dir` command, you can use the `ls` demo to practice on your own.

Summary

In this chapter, we have discussed about the file system and its use in storing and retrieving files and directories. We have also discussed about the absolute and relative paths, wild cards and finally saw practical examples of 15 commands that are used to create and manipulate files and directories, get file lists and usage statistics.