コース: ALX Software Engineering

Bash Notebook

作家: Anrich Tait

Abstract

My notes on the bash shell, this will also include some useful information regarding Unix systems. Most of my learning in this regard was in relation to my software engineering studies.

Contents

Chapter 1

Shell Basics

1.1 Objectives

1. What is the shell?
2. Navigation.
3. Looking around
4. A guided tour
5. Manipulating files
6. Working with commands
7. Reading man pages
8. Keyboard shortcuts
9. LTS
10. Shebang

1.2 Resources

1. What is the shell?
2. Navigation

3. Looking around
4. A guided tour
5. Manipulating files
6. Working with commands
7. Reading man pages
8. Keyboard shortcuts
9. LTS
10. Shebang

Man Pages

cd, ls, pwd, less, file, ln, cp, mv, rm, mkdir, type, which, help, man

1.3 Notes

1.3.1 What is the shell?

The shell is a command line interface (CLI) that takes commands and passes them to this operating system.

Bash is the most common example of a shell program, others include: ksh, tcsh and zsh.

Most interactions with the shell are done through a terminal like gnome, alacritty or kitty.

NOTE: Make sure that the last symbol of your shell prompt is not #. If it is this means that you are in sudo (super user) mode and this can be dangerous.

1.3.2 Navigation

Nothing much here for me to learn, maybe important to note thoguh:
Important facts about file names:

1. File names that begin with a period character are hidden. This only means that ls will not list them unless we say ls -a. When your account was created, several hidden files were placed in your home directory to configure things for your account. Later on we will take a closer look at some of these files to see how you can customize our environment. In addition, some applications will place their configuration and settings files in your home directory as hidden files.

2. File names in Linux, like Unix, are case sensitive. The file names ”File1” and ”file1” refer to different files.

3. Linux has no concept of a ”file extension” like Windows systems. You may name files any way you like. However, while Linux itself does not care about file extensions, many application programs do.

4. Though Linux supports long file names which may contain embedded spaces and punctuation characters, limit the punctuation characters to period, dash, and underscore. Most importantly, do not embed spaces in file names. If

you want to represent spaces between words in a file name, use underscore characters. You will thank yourself later.

1.3.3 Looking around

A closer look at the long format: ls -l

```
-rw-------   1 me       me           576 Apr 17  2019 weather.txt
drwxr-xr-x   6 me       me          1024 Oct  9  2019 web_page
-rw-rw-r--   1 me       me        276480 Feb 11 20:41 web_site.tar
-rw-------   1 me       me          5743 Dec 16  2018 xmas_file.txt

----------     -------  -------  -------- ------------ -------------
    |             |        |         |         |             |
    |             |        |         |         |         File Name
    |             |        |         |         |
    |             |        |         |         +---  Modification Time
    |             |        |         |
    |             |        |         +--------------  Size (in bytes)
    |             |        |
    |             |        +------------------------       Group
    |             |
    |             +---------------------------------       Owner
    |
    +-----------------------------------------------  File Permissions
```

- File name: Name of file or directory
- Modification time: Last time the file was modified
- Size: size of the file in bytes
- Group: Group that has file permissions other than the file owner
- Owner: The user that owns the file.
- File Permissions: A representaion of the file's access permissions. The first character is the file type, "-" indicates a regular or ordinary file. A "d" indicates a directory. The second character represents the read, write and execution rights of the file owner. The third represents the rights of the file's group. The last represents the permissions granted to everyone else.

Directory	Description
/	The root directory where the file system begins.
/boot	Linux kernel and bootloader directory. The kernel is a file called vmlinuz.
/etc	Configuration files, important locations include: /etc/passwd - user info, /etc/fstab - mounted devices(disk drives), /etc/hosts - Network host names and IP adresses, / - system service scripts run at boot time.
/bin, /usr/bin	Most of the system programs.
/usr	folders/files that support user applications
/usr/local	Used for installation of software (user). Most commononly in /usr/local/bin
/var	Files that change as the system runs, including logs and spools.
/lib	The shared libraries
home	user files

The list goes on and on, do research to find specific files.

1.3.4 Manipulating Files

Commands to know/understand:

1. cp: copy files and directories
2. mv: move or rename files and directories
3. rm: remove files and directories
4. mkdir: make directories

Wildcards: Wildcards allow the user to specify groups of filenames. This makes mass file manipulation much easier.

Wildcard	Meaning
*	Matches any characters
?	Matches any single character
characters	Matches any character that is part of the set characters.
!characters	Matches any character that is not a member of the set characters

Here are some examples of patterns and what they match.

Pattern	Matches
*	All filenames
g*	All filenames that begin with the character ”g”
b*.txt	All filenames that begin with ”b” and end with ”.txt”
Data???	Any filename that begins with the characters ”data” followed by any other characters.

1.4 Working with commands

Commands to know/understand

1. type: Display information about command type
2. which: Locate a command
3. help: Display reference page for shell builtin
4. man: Display an on-line command reference

What are ”Commands”

Commands fall into 4 categories:

1. Executable programs: programs that can be executed
2. A command built into the shell: or ”shell builtins. Like, ”cd”
3. A shell function: Miniature shell scripts.
4. An alias: Commands that the user defines, built from other commands.

Chapter 2

Shell permissions

2.1 Resources

1. Permissions

man or help pages:

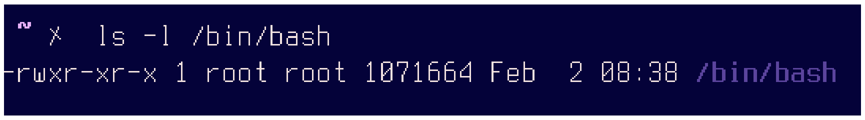
1. chmod - modify file access rights
2. sudo - enter super user mode or execute a command as such
3. su - temporarily enter sudo mode
4. chown - change file ownership
5. chgrp - change a file's group ownership
6. id - print effective user and group IDs
7. groups - display current group names
8. whoami - print effective username
9. adduser - ?
10. useradd - create a new user or update default new user info
11. addgroup - ?

2.2 Permissions Notes

Each file/directory is assigned access rights for the owner of the file, members of a group of related users and everybody outside of the afore-mentioned groups.

Rights can be assigned to read, write and execute a file.

As an example the ls command will be used to look at the 'bash' program located in the /bin directory:

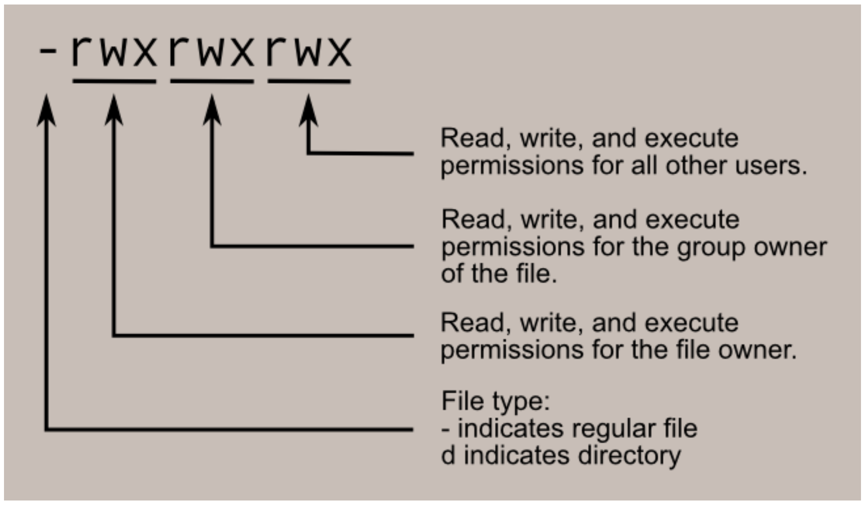


```
~ ✗  ls -l /bin/bash
-rwxr-xr-x 1 root root 1071664 Feb  2 08:38 /bin/bash
```

Here we can see:

1. The file "/bin/bash" is owned by user 'root'
2. The superuser has the right to read, write and execute
3. The file is owned by the group "root"
4. Members of the group 'root' can also read and execute
5. Everybody else can read and execute the file

Below is graphic showing what each portion of the first listing represents:

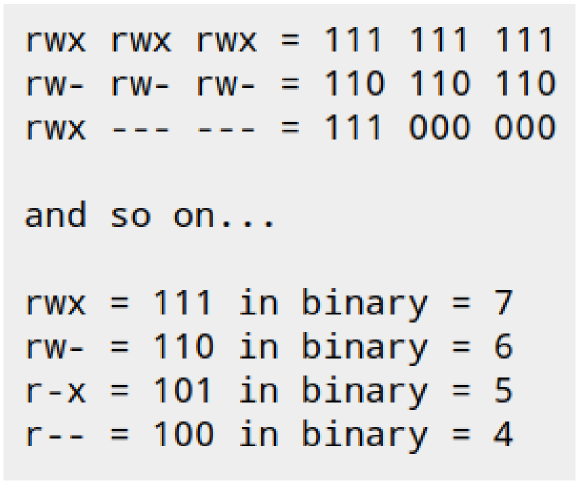

- rwx rwx rwx
Read, write, and execute permissions for all other users.
Read, write, and execute permissions for the group owner of the file.
Read, write, and execute permissions for the file owner.
File type:
- indicates regular file
d indicates directory

2.3 Look at all the pretty commands

2.3.1 chmod

Used to change the permissions of a file or directory. To use it specify the desired permission settings and the file or files that are to be modified.
There are two ways to do this, the following method uses the octal notation method.

Think of the permission settings as a series of bits (like a computer):



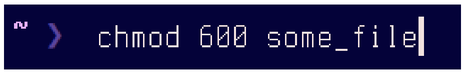
```
rwx rwx rwx = 111 111 111
rw- rw- rw- = 110 110 110
rwx --- --- = 111 000 000

and so on...

rwx = 111 in binary = 7
rw- = 110 in binary = 6
r-x = 101 in binary = 5
r-- = 100 in binary = 4
```

No we can represent each of the three sets of permissions, (owner, group and other) as a single digit to create a convenient way of expressions the permission settings.

For example to set the permissions of a file to have read and write permission for the owner, but wanted to keep the file private from others we would use:



```
~ ❯  chmod 600 some_file
```

File Permissions:
Below is a table of common settings for files, the ones starting with '7' are used with programs since they enable execution. The rest are other kinds of files:

1. 777 - (rwxrwxrwx) No restrictions on permissions, All groups can do everythin. (BE WARNED)
2. 755 - (rwxr-xr-x) File owner can read, write and exec, All others may read and exec.
3. 700 - (rwx——) File woner can read, write and exec. Nobody else can do anything.
4. 666 - (rw-rw-rw-) All users may read and write the file.
5. 644 - (rw-r–r–) File owner may read and write, others can only read.
6. 600 - (rw——-) Owner can read and write a file. All other have no rights.

Directory Permissions:
The chmod command can also be used to change directory permissions.
In this case octal notation is also used but the r, w and x meanings are different:

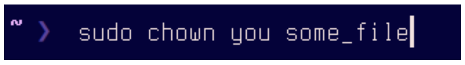
1. r - Allows the contents of the directory to be listed if the x attribute is also set
2. w - Allows files within the directory to be created, deleted or renamed if the x attribute is also set
3. x - Allows a directory to be entered (cd dir)

Here are some common settings for directories:

1. 777 - (rwxrwxrwx) No restrictions on permissions. Anybody may list files, create new files in the directory and delete files in the directory. Generally not a good setting.
2. 755 - (rwxr-xr-x) The directory owner has full access. All others may list the directory, but cannot create files nor delete them.
3. 700 - (rwx——) The directory owner has full access. Nobody else has any rights. This setting is useful for directories that only the owner may use and must keep private.

2.3.2 chown

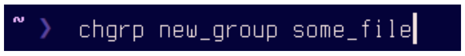
Change file ownership.
Example:
Change the owner of some_file from ”me” to ”you”:



```
~ ❯  sudo chown you some_file
```

2.3.3 chgrp

Change the group ownership of a file or directory.
Example:



```
~ ❯  chgrp new_group some_file
```

Chapter 3

Shell, I/O Redirections and filters

3.1 Learning objectives

1. Understanding Shell, I/O Redirection
2. What are special characters and what to do with white spaces, single quotes, double quotes, backslash, comment, pipe, command seperator, tilde.
3. How to concatenate files and print on the standard output
4. How to reverse a string
5. How to remove sections from each line of files
6. what is the /etc/passwd file and it's format
7. what is the /etc/shadow file and what is it's format

3.2 Resources and man pages

Links:

1. Shell I/O Redirection
2. Special characters

man pages

1. echo

2. cat
3. head
4. tail
5. find
6. wc
7. sort
8. uniq
9. grep
10. tr
11. rev
12. out
13. passwd (man 5 passwd)

3.3 I/O Redirection

I/O redirection is another way of saying input and output redirection, specifically how a user can redirect a command's output to a file, device and other commands.

3.3.1 Standard Output Redirection

Most command line program send their results to the standard output, by default the standard output directs it's contents to the display.

One way the user can redirect this standard output is by using the '>' characters.

```
$ ls > file file_list.txt
```

In the above example the ls command is executed and the results are outputted to a file named file_list.txt. Since the ls output was redirected to a file, no output will be displayed in the terminal.

In the above example every time the command is run file_list.txt will be overwritten, we can append the output (add to end of file) by using two '»' characters.

```
$ ls  file_list.txt
```

3.3.2 Standard Input Redirection

Many commands accept input from something called standard input. Standard input can receive input from the Keyboard (via user in terminal) or from files (such as file_list.txt) through redirection.
We can redirect standard intput from a file by using the '<' character:

```
$ sort < file_list.txt
```

In this example the **sort** command is used to process the contents of the file. The 'sort' commands description is as follows: Write sorted concatenation of all FILE(s) to standard output.
The output of this command will be sent to the standard output and displayed on the terminal screen for the user to see. If you wish to send the sorted list to another file it can be done like this:

```
$ sort < file_list.txt > sorted_file_list.txt
```

3.3.3 Pipelines

Pipelines make it possible to feed the output of one command into the input of another. Indeed creating some powerful capabilities. Example:

```
$ ls -l | less
```

This exampke shows how the ls command can be fed into the 'less' command. Essentially creating a scrolling output.
Some other examples to try

1. ls -lt | head: Displays the 10 newest files in the current directory.

2. du | sort -nr : Displays a list of directories and how much space they consume , sorted from the largest to smallest.
3. find . -type f -print | wc -l : Displays the total number of files in the current working directory and all of it's subdirectories.

3.3.4 Filters

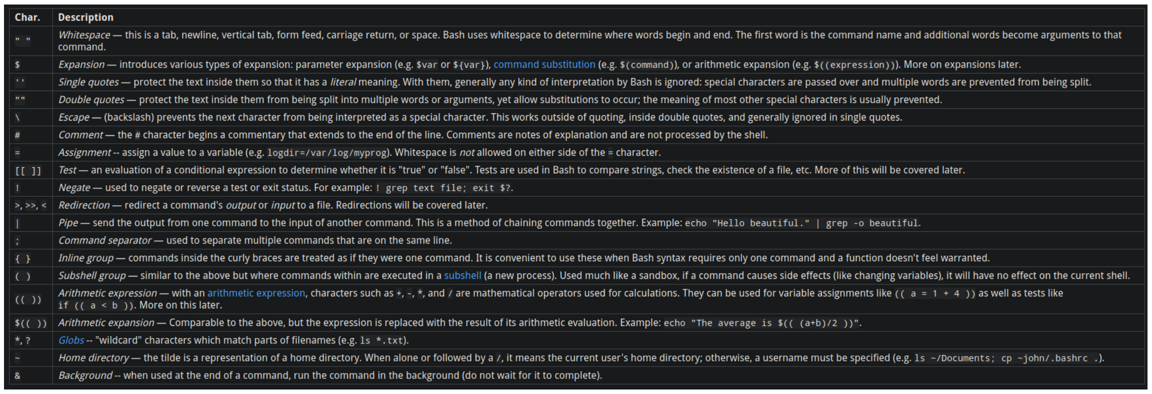
Filters are frequently used with pipes to take input from one command and then output the result in standard output. This has a wide range of capabilities that allow the user to process or search for specific information.
Some common filters include:

1. sort : sorts standard input
2. uniq : removes duplicate lines of data
3. grep : outputs ever line that contains a specified pattern of characters
4. fmt : outputs formatted text
5. pr : splits data into pages with page breaks, headers and footers (in prepartion for printing)
6. head : outputs first few lines of input
7. tail : outputs last few lines of input
8. tr : Translates characters (upper/lower case conversion, change line termination characters)
9. sed : Stream editor, more advanced text translations that 'tr'
10. awk : An entire programming language designed for constructing filters.

Note: Chapter 6 of The Linux Command Line covers this in more detail. I have the book in the extra resources directory.
Alse consider reading more on AWK

3.4 Special Characters

In bash there is a concept of special characters, these are characters that carry out special instructions or have an alternate meaning. They are also known as *meta characters*
Some common examples of special characters:



Char.	Description
`" "`	*Whitespace* — this is a tab, newline, vertical tab, form feed, carriage return, or space. Bash uses whitespace to determine where words begin and end. The first word is the command name and additional words become arguments to that command.
`$`	*Expansion* — introduces various types of expansion: parameter expansion (e.g. `$var` or `${var}`), command substitution (e.g. `$(command)`), or arithmetic expansion (e.g. `$((expression))`). More on expansions later.
`''`	*Single quotes* — protect the text inside them so that it has a *literal* meaning. With them, generally any kind of interpretation by Bash is ignored: special characters are passed over and multiple words are prevented from being split.
`""`	*Double quotes* — protect the text inside them from being split into multiple words or arguments, yet allow substitutions to occur; the meaning of most other special characters is usually prevented.
`\`	*Escape* — (backslash) prevents the next character from being interpreted as a special character. This works outside of quoting, inside double quotes, and generally ignored in single quotes.
`#`	*Comment* — the `#` character begins a commentary that extends to the end of the line. Comments are notes of explanation and are not processed by the shell.
`=`	*Assignment* -- assign a value to a variable (e.g. `logdir=/var/log/myprog`). Whitespace is *not* allowed on either side of the `=` character.
`[[ ]]`	*Test* — an evaluation of a conditional expression to determine whether it is "true" or "false". Tests are used in Bash to compare strings, check the existence of a file, etc. More of this will be covered later.
`!`	*Negate* — used to negate or reverse a test or exit status. For example: `! grep text file; exit $?`.
`>`, `>>`, `<`	*Redirection* — redirect a command's *output* or *input* to a file. Redirections will be covered later.
`|`	*Pipe* — send the output from one command to the input of another command. This is a method of chaining commands together. Example: `echo "Hello beautiful." | grep -o beautiful`.
`;`	*Command separator* — used to separate multiple commands that are on the same line.
`{ }`	*Inline group* — commands inside the curly braces are treated as if they were one command. It is convenient to use these when Bash syntax requires only one command and a function doesn't feel warranted.
`( )`	*Subshell group* — similar to the above but where commands within are executed in a subshell (a new process). Used much like a sandbox, if a command causes side effects (like changing variables), it will have no effect on the current shell.
`(( ))`	*Arithmetic expression* — with an arithmetic expression, characters such as `+`, `-`, `*`, and `/` are mathematical operators used for calculations. They can be used for variable assignments like `(( a = 1 + 4 ))` as well as tests like `if (( a < b ))`. More on this later.
`$(( ))`	*Arithmetic expansion* — Comparable to the above, but the expression is replaced with the result of its arithmetic evaluation. Example: `echo "The average is $(( (a+b)/2 ))"`.
`*`, `?`	*Globs* -- "wildcard" characters which match parts of filenames (e.g. `ls *.txt`).
`~`	*Home directory* — the tilde is a representation of a home directory. When alone or followed by a `/`, it means the current user's home directory; otherwise, a username must be specified (e.g. `ls ~/Documents; cp ~john/.bashrc .`).
`&`	*Background* -- when used at the end of a command, run the command in the background (do not wait for it to complete).

```
$ echo "I␣am␣$LOGNAME"
I am lhunath
$ echo 'I␣am␣$LOGNAME'
I am $LOGNAME
$ # boo
$ echo An open\ \ \ space
An open   space
$ echo "My␣computer␣is␣$(hostname)"
My computer is Lyndir
$ echo boo > file
$ echo $(( 5 + 5 ))
10
$ (( 5 > 0 )) && echo "Five␣is␣greater␣than␣zero."
Five is greater than zero.
```