Linux Shell Programming

Introduction

- Shell is an environment in which we can run our commands, programs, and shell scripts.
- It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.
- - kernal = core of a computer's operating system
- - terminal = text input/output environment
- - shell = command line interpreter
- - shell scripts = commands written in a file

Shells Available in Unix

- C shell (csh)
- TC shell (tcsh)
- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again Shell (bash)

Bourne Again Shell (bash)

Bash is the default shell in most Linux distributions.Bash offers functional improvements over sh for both programming and interactive use.

# Comments.Lines beginning with a # (with the exception of #!) will not be executed.

```
 # This line is a comment.
```

; Command separator. Permits putting two or more commands on the same line.

```
 echo hello ; echo world
```

;; Terminator in a case option

```
case "$variable" in
 abc) echo "\$variable = abc" ;;
 xyz) echo "\$variable = xyz" ;;
esac
```

`  command substitution. The `command` construct makes available the output of command for assignment to a variable

$ Variable substitution (contents of a variable)

$(variable)

“ “ Weak quotes

echo “hello world”

‘ ‘ Strong quotes

echo ‘hello ‘world’ ’

\ Single character quote

The help command provides information on built-in commands.
simply entering ‘help’ at the terminal prompt will show a complete list of the built-in commands available.
enter ‘help’ followed by the command you wish to learn more about.

```
help pwd
```

With the ‘-d’ option, the help command will only return a short description of the specified command.

```
help -d pwd
```

With the -s option, help will return a short usage synopsis for the specified command.

```
help -s pwd
```

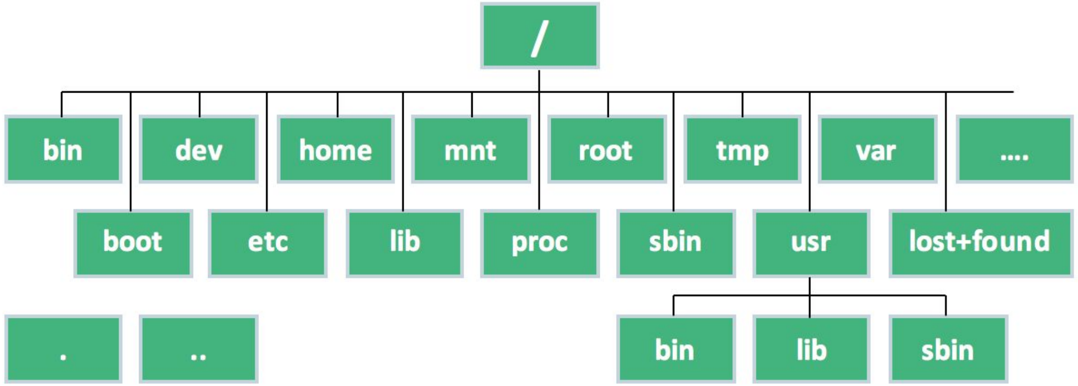
The -m option formats the help command output as a pseudo-manpage.

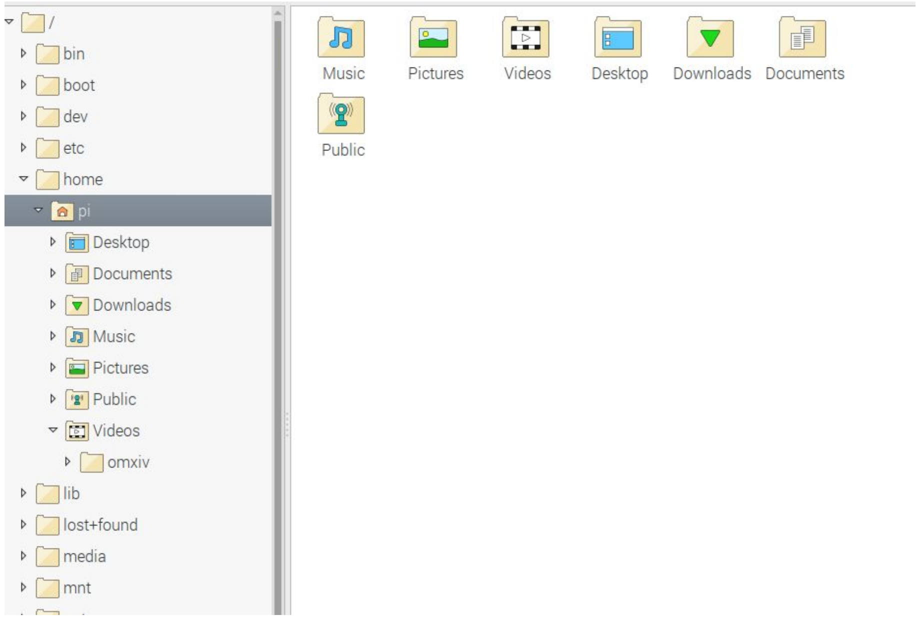
```
help -m pwd
```

The man command shows detailed manuals for each command. These are referred to as “man pages.”

It provides a detailed view of the command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUES, ERRORS, FILES, VERSIONS, EXAMPLES, AUTHORS ..etc

```
man pwd
```


/
bin
dev
home
mnt
root
tmp
var
....
boot
etc
lib
proc
sbin
usr
lost+found
.
..
bin
lib
sbin


/
bin
boot
dev
etc
home
pi
Desktop
Documents
Downloads
Music
Pictures
Public
Videos
omxiv
lib
lost+found
media
mnt
Music
Pictures
Videos
Desktop
Downloads
Documents
Public

★ pwd - prints the complete path of the current working directory.

pwd -L : Prints the symbolic path.

pwd -P : Prints the actual path.

★ cd - change the current working directory.

```
cd [directory]
```

★ ls - list of the names of all files in the current working directory / specified directory.

```
ls
ls /etc
```

★ file - displays the type of a file.

```
file [option] [filename]
```

file -b filename : used to display just file type in brief mode.

file directoryname/* : used to display all files filetypes in particular directory.

★ cat – read and concatenate data from the file and gives their content as output.
cat filename read single filename
cat file1 file2 show contents of file1 and file2
cat -n file show contents of fille with preceding line number
cat -s file suppress repeated empty lines in output
★ cp - copy fiiles or directories from source to destination
cp Src_file Dest_file copy the contents of src_file to the dest_file
cp Src_file1 Src_file2 Src_file3 Dest_directory copies each source file to the destination directory
cp -R Src_directory Dest_directory copies all files of the source directory to the destination directory, creating any files or directories needed
★ mv - move fiiles or directories from source to destination
mv Src_file Dest_file move the file src_file to the dest_file
mv -i Src_file Dest_directory ask the user for confirmation before moving a file that would overwrite an existing file,
mv -f Src_directory Dest_directory overwrite the destination file forcefully and delete the source file.

- ★ mkdir - create multiple directories at once as well as set the permissions for the directories

mkdir foldername

mkdir -m 777 foldername used to set the file modes

- ★ rmdir - remove empty directories from the filesystem

rmdir -p directory remove directory if it is empty

rmdir -p directory remove all child and parent empty directories

rmdir directorylist remove all directories

- ★ whereis - locates source/binary and manuals sections for specified files.

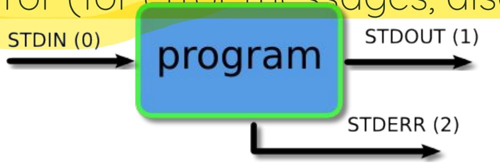
whereis perl List the directories where the perl source files, documentation, and binaries are stored.

whereis -b perl List only perl binaries are stored.

whereis -s perl List only perl sources are stored.

whereis -m perl List only perl manuals are stored.

Every program we run on the command line automatically has three data streams connected to it.
STDIN (0) - Standard input (data fed into the program)
STDOUT (1) - Standard output (data printed by the program, defaults to the terminal)
STDERR (2) - Standard error (for error messages, also defaults to the terminal)



Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.

Pipes allow you to funnel the output from one command into another where it will be used as the input.

Pipes are unidirectional

The | operator feeds the output from the program on the left to the program on the right.

This will sort the given file and print the unique values only.

```
sort record.txt | uniq
```

By default, stdout and stderr are printed to your terminal.
But we can redirect that output to a file using the > operator:

```
echo hello > new-file
```

echo didn’t print anything to the terminal because we redirected its output to a file named new-file.
it replaces new-file’s contents with the new contents.
If you want to append to the file, you can use the >> operator

```
echo hello again >> new-file
```

- ★ ps - to see which processes you're currently running on your Linux system.
Using the -f option for ps you can gain additional useful information on each process in the listing.
- ★ w - command-line utility that displays information about currently logged in users and what each user is doing. It also gives information about how long the system has been running, the current time, and the system load average.
- ★ id - used to find out user and group names and numeric ID's (UID or group ID) of the current user or any other user in the server.
- ★ free - displays the total amount of free space available along with the amount of memory used and swap memory in the system, and also the buffers used by the kernel.
- ★ clear - used to remove all previous commands and output fromterminal windows

★ echo - display a line of text/string on standard output or a file.

```
 echo [option(s)] [string(s)]
 eg:-
 echo Hello World
 echo The value of variable x = $x
echo -e "Hello \nWorld"
```

★ more - used to view the text files in the command prompt, displaying one screen at a time in case the file is large. The more command also allows the user do scroll up and down through the page.

```
more [-options] [-num] [+/pattern] [+linenum] [file_name]
```

[-options]: any option that you want to use in order to change the way the file is displayed. (-d, -l, -f, -p, -c, -s, -u)
[-num]: type the number of lines that you want to display per screen.
[+/pattern]: replace the pattern with any string that you want to find in the text file.
[+linenum]: use the line number from where you want to start displaying the text content.
[file_name]: name of the file containing the text that you want to display on the screen.

Ownership of Linux files

Every file and directory on your Unix/Linux system is assigned 3 types of owner,
User : owner of the file. By default, the person who created a file becomes its owner.
Group : group can contain multiple users. All users belonging to a group will have the same access permissions to the file.
Other : This person has neither created the file, nor he belongs to a usergroup who could own the file. Practically, it means everybody else.

Permissions

Every file and directory in your UNIX/Linux system has following 3 permissions.
Read : authority to open and read a file.
Write: authority to modify the contents of a file.
Execute : In Unix/Linux, you cannot run a program unless the execute permission is set.

using chmod command, we can set permissions (read, write, execute) on a file/directory for the owner, group and the world.

chmod permission file

There are 2 ways to use the command

★ Absolute mode

★ Symbolic mode

Absolute Mode

file permissions are not represented as characters but a three-digit octal number.

```
chmod 764 filename
```

'764' absolute code says the following:

- Owner can read, write and execute
- Usergroup can read and write
- World can only read

This is shown as

```
chmod -rwxrw-r- filename
```

Number	Permission Type	Symbol
0	No permission	---
1	Execute	--x
2	Write	-w-
3	Write + Execute	-wx
4	Read	r--
5	Read+Execute	r-x
6	Read+Write	rw-
7	Read+Write+Execute	rwx

Symbolic Mode

n the symbolic mode, you can modify permissions of a specific owner.
It makes use of mathematical symbols to modify the file permissions..

```
chmod o=rwx filename
```

setting others read+write+execute permissions

```
chmod g+x filename
```

adding execute permission to group

```
chmod u-r filename
```

removing read permission of user

Operator	Description
+	No permission
-	Execute
=	Write

User	Description
u	user
g	group
o	other

Lines beginning with a # (with the exception of #!) are comments and will not be executed.

```
# This line is a comment
```

Comments may also occur following the end of a command.

```
echo “a comment will follow” # this is a comment
```

Comments may also follow whitespace at the beginning of a line.

```
      # This is also a comment
```

a quoted or an escaped # in an echo statement does not begin a comment.

```
echo “hello #This is not a comment”
```

A variable is a name assigned to a location or set of locations in computer memory holding an item of data.

If variable1 is the name of a variable, then $variable1 is a reference to its value.

The only times a variable appears without the $ prefix is when declared or assigned.

```
variable1=23
echo $variable1
```

Essentially, Bash variables are character strings, but, depending on context, Bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits.

Assignment Operator

= is the all-purpose assignment operator, which works for both arithmetic and string assignments.

```
var=27
category=minerals # No spaces allowed after the "="
```

Logical Operators

! NOT

&& AND

|| OR

Arithmetic Operators

+ plus
- minus
* multiplication
/ division
** exponentiation
% mod (returns the remainder of an integer division operation)
+= plus-equal (increment variable by a constant) [38]
-= minus-equal (decrement variable by a constant)
*= times-equal (multiply variable by a constant)
/= slash-equal (divide variable by a constant)
%= mod-equal (remainder of dividing variable by a constant)

Relational Operators

-eq equal to
-nt not equal to
-gt greater than
-lt lessthan
-ge greater than or equal to
-le less than or equal to
< is less than (within double parentheses) (("$a" < "$b"))
<= is less than or equal to (within double parentheses) (("$a" <= "$b"))
> is greater than (within double parentheses) (("$a" > "$b"))
>= is greater than or equal to (within double parentheses) (("$a" >= "$b"))

cHEck

When referencing a variable, it is generally advisable to enclose its name in **double quotes**. This prevents reinterpretation of all special characters within the quoted string except $, ` (backquote), and \ (escape).
Use double quotes to prevent word splitting.An argument enclosed in double quotes presents itself as a single word, even if it contains whitespace separators.
Within **single quotes**, every special character except ' gets interpreted.
Consider single quotes ("full quoting") to be a stricter method of quoting than double quotes ("partial quoting").
The escape (\) preceding a character tells the shell to interpret that character literally.

read

Reads the value of a variable from stdin, that is, interactively fetches input from the keyboard.

```
 echo "Enter the value of variable 'var1': "
read var1
```

echo

prints (to stdout) an expression or variable

```
 echo Hello
 echo $a
```

An echo requires the -e option to print escaped characters.

Normally, each echo command prints a terminal newline, but the -n option suppresses this.

if can test any command.Reads the value of a variable from stdin, that is, interactively fetches input from the keyboard.

```
 if [ condition ]
then
command1
else
command2
fi
```

elif is a contraction for else if. The effect is to nest an inner if/then construct within an outer one.

```
if [ condition1 ]
then
 command1
elif [ condition2 ]
then
 command4
else
 default-command
fi
```

The case construct is the shell scripting analog to switch in C/C++. It permits branching to one of a number of code blocks, depending on condition tests.

```
 case "$variable" in
 "$condition1" )
command...
 ;;
 "$condition2" )
command...
 ;;
```

A loop is a block of code that iterates [52] a list of commands as long as the loop control condition is true.

```
 for arg in [list]
do
 command(s)...
done
```

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status)..

```
while [ condition ]
do
 command(s)...
done
```

break command terminates the loop (breaks out of it)

```
for innerloop in 1 2 3 4 5
do
if [ "$innerloop" -eq 3 ]
then
break # Try break 2 to see what happens.
fi
done
```

continue causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

```
for inner in 1 2 3 4 5 6 7 8 9 10 # inner loop
do
if [[ "$inner" -eq 7 && "$outer" = "III" ]]
then
continue 2 # Continue at loop on 2nd level, that is "outer loop".
fi
done
```

All-purpose expression evaluator: Concatenates and evaluates the arguments according to the operation given (arguments must be separated by spaces).

```
a=`expr 5 + 3`
_
```

Bash can't handle floating point calculations, and it lacks operators for certain important mathematical functions.

```
echo "scale=2; 2/3" | bc
.66
```

searches a file for a particular pattern of characters and displays all lines that contain that pattern.

case insensitive search

```
grep -i “Word” file.txt
```

display count of matches

```
grep -c “Word” file.txt
```

checking whole word only

```
grep -w “Word” file.txt
```

Bash arrays can hold multiple values at the same time.

```
arrayname[index]=value
```