Improve command line productivity

Linux #redhat #if-then #loops #script

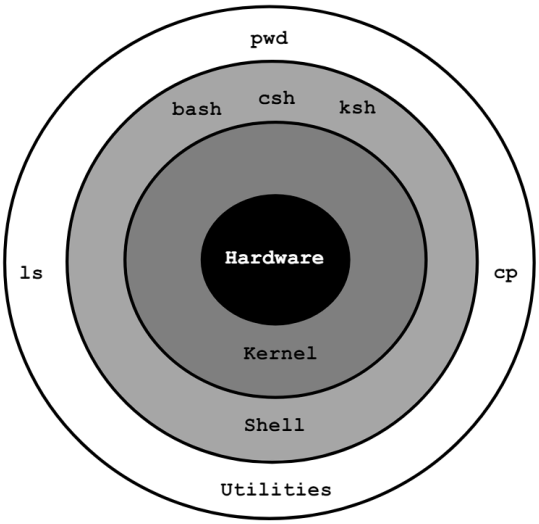
Linux shell scripting is a strong tool for systems administrator to automate daily repetitive tasks

In this note:

- What is a shell
- Shell scripting
- Basic shell scripts
- Input and Output of scripts
- If then scripts
- For loop scripts

Introduction to Shell

- What is a Shell?
  - Its like a container
    - Any kind of container that holds your environment, information inside of it.
  - Interface between users and Kernel/OS
    - When you start up a GUI, you execute commands by double clicking on a program. So that GUI interface is your shell.
    - In Linux when you don't have a GUI, you are given a terminal shell or CLI command, this is your shell.
  - CLI (Command Line Interface) is a Shell
- Find your Shell
  - `echo &0`
    - This command will tell you what shell you are in.
  - Available Shells "`cat /etc/shells`"
    - Listing of all your shells in your Linux distribution.
  - Your Shell? `/etc/passwd`
- Windows GUI is a shell
- Linux KDE GUI is a shell
- Linux sh, bash etc. is shell



Example using `echo $0` command:

```
[user@localhost ~]$ echo $0
```

Output:

```
-bash
```

- It tells you you are using bash

Example using `cat` command:

```
[user@localhost ~]$ cat /etc/shells
```

Output:

```
/bin/sh
/bin/bash
/usr/bin/sh
/usr/bin/bash
/bin/ksh
/bin/rksh
/usr/bin/ksh
/usr/bin/rksh
```

- These are the available shells that are installed in your distribution

Example using `cat` command:

```
[user@localhost ~]$ cat /etc/passwd
```

Output:

```
...
mmarin:x:1000:1000:Maximus Marin:/home/mmarin:/bin/bash
spiderman:x:1001:1001::/home/spiderman:/bin/bash
ironman:x:1002:1002:Ironman Character:/home/ironman:/bin/bash
babubutt:x:1003:1003::/home/babubutt:/bin/bash
named:x:25:25:Named:/var/named:/sbin/nologin
```

- It tells the shell used by each user, for example in the case of the user "mmarin" the shell is `/bin/bash` (it's at the end).
- We can change that shell to a different shell if you want to.

Shell Scripting

- What is a Shell Script?
  - A shell script is an executable file containing multiple shell commands that are executed sequentially (the first command is always executed first). The file can contain:
    - Shell (`#!/bin/bash`)
      - The `!` sign is referred as "bang" in Linux
      - A shell script always starts with the shell is gonna run in
    - Comments (`# comments`)
      - You can tell what each line does with comments
    - Commands (`echo`, `cp`, `grep`, etc.)
    - Statements (`if`, `while`, `for`, etc.)
- Shell script should have executable permissions (e.g. `-rwx r-x r-x`)
  - When you create a script, a script is an empty file. Everything in Linux is just a fil, a flat file, until you change it to an executable or link.
  - Remember that the third bit in a file is always executable (`x`)
- Shell script has to be called from absolute path (e.g. `/home/userdir/script.bash`)
  - If it is not in absolute path you can still run it if you are in the current location, then all you have to do is put `./` in the name of your script.

Shell Script - Basic Scripts

- Output to screen using "`echo`"
- Creating tasks
  - Telling your id, current location, your files/directories, system info
  - Creating files or directories
  - Output to file "`>`"
- Filters/Text processors through scripts (`cut`, `awk`, `grep`, `sort`, `uniq`, `wc`)

Start writing a script

- Create a directory to save your scripts to in your user home directory. (`mkdir myscripts`)

Example using `vi` command:

```
[user@localhost myscripts]$ vi output-screen
```

- Creating a file called "output-screen" and opening it with `vi` file editor.
- Note we are inside our previously created "myscripts" directory.

File editor:

```
#!/bin/bash
~
~
~
~
:
```

- Hit “I” to enter the INSERT mode and start writing.
- Type `#!/bin/bash`. A shell script always starts with the shell is gonna run in.

Script to print something to the screen:
File editor:

```
#!/bin/bash

echo Hello World!
~
~
:
```

- Use the `echo` command to print something to the screen.
- Once typed your commands you can just simply Escape and right quit out of this file.
- Now the file is created, you can verify its contents by doing `cat output-screem`

Running a simple script (Output to screen)

- If we try to run the previous script named "output-screen" by running `./output-screen` it won't let us because we do not have executable permissions.

To give executable permission to our script:
Example using `chmod` command:

```
[user@localhost myscripts]$ chmod a+x output-screen
```

- Remember the `a` option stands for "all" and specifies that changes will be done for ALL levels (user, group, others).

Output from `ls -l`:

```
total 4
-rwxr-xr-x. 1 mmarin mmarin 31 Jun 28 12:25 output-screen
```

- It is now highlighted in green. This means it is an executable file.
- Now we have executable right on everyone.

Running the script from relative path

```
[user@localhost myscripts]$ ./output-screen
```

- To run the script you just have to type the path of the executable file.
- Remember the `.` sign is used to refer to the relative path (current path).
- The name of the executable file has to be specified.

Output:

```
Hello World!
```

- This is your first script.

Running the script from absolute path

```
[user@localhost ~]$ /home/mmarin/myscripts/output-screen
```

Output:

```
Hello World!
```

Defining Tasks

File editor:

```
#!/bin/bash
# Define small tasks
# This script is written by Maximus Marin
# Date written: 2024

whoami
echo
pwd
echo
hostname
echo
ls -ltr
echo
```

- If the `echo` command in a script is empty it will output just an empty line.

Output from `./run-commands`:

```
mmarin

/home/mmarin/myscripts

linuxtest

total 8
```

```
-rwxr-xr-x. 1 mmarin mmarin  31 Jun 28 12:25 output-screen
-rwxr-xr-x. 1 mmarin mmarin 146 Jun 28 12:43 run-commands
```

Defining variables

File editor:

```
#!/bin/bash
# Example of defining variables

a=Max
b=Marin
c='Linux class'

echo "My first name is $a"
echo "My surname is $b"
echo "My surname is $c"
```

- To call a variable inside a string you must use the `$` sign followed by the variable name.
- If you just type `a` as in the name of the variable, the variable will not be called, it will just output a letter "a". You need the `$` sign to call the variable.
- The "=" sign cannot be spaced from the variable name and its definition. This is called declaring a variable, and it should be all joined together, like in `a=Max`
- Using single quotes for strings will not call a variable.
  - For example: `echo 'My first name is $a'` will output only `My first name is $a`
- Using double quotes for strings will call a variable.
  - For example: `echo "My first name is $a"` will output `My first name is Max`

Output from `./variable-command`:

```
My first name is Imran
My surname is Afzal
My surname is Linux class
```

Input/Output

- Writing a script where the script will wait for the user input.
- Create script to take input from the user
  - `read`
  - `echo`

File editor:

```
#!/bin/bash
# Author
# Date
# Description

a=`hostname`
```

```
echo Hello, my server name is $a
echo
echo What is your name?
read namecont
echo
echo Hello $namecont
echo
```

- The `read` command reads whatever the user types at that moment, and saves it into a variable specified in the script next to the "read" keyword. In this case the variable that will save the inputs from the user is `namecont`.
- Note we are using the `hostname` command to refer to the hostname of the Linux machine and we are assign that command a variable with the name `a`. In this case we have to use ticks [`] (sign below the Esc key) to specify to run this command first and assign whatever the command outputs to that variable.
- If we just declare `a=hostname` it will just output "hostname" to the screen, not the actual hostname of the machine.

Output from `./input-script`:

```
Hello, my server name is linuxtest

What is your name?
Milo Milays

Hello Milo Milays
```

- In this example script the user typed "Milo Milays" when prompted by the `read` command.
- Think of this: When your are installing a program or an update, now you know that when you going through the vizard and it asks you questions, do you want to install that? You say yes, no, put in your organization name, put it this. All the information that the system gathers from you, it actually plugs that information inside wherever your variable is defined. So that's just a simple example of that

If-then Scripts

- If then statement
  - If this happens = do this
  - Otherwise = do that

Check the variable

File editor:

```
#!/bin/bash

count=100
if [ $count -eq 100 ]
then
        echo Count is 100
else
        echo Sorry the count is NOT 100
fi
```

- Note that the condition next to the "`if`" keyword has to be boxed in square brackets `[]`.
- Note the equal sign is represented with `-eq`.
- Note the keyword "`fi`" ends the statement. (fi is the opposite of if and it is telling the script to exit out or that its work is done).

Output from `./ifthen-script`:

```
Count is 100
```

- As the condition is true, the script will execute whatever the "`then`" branch (true) will say, in this case is an `echo` command.
- If we change the number to anything else than 100, it will choose and execute the "`else`" branch.

Check if a file `error.txt` exist

File editor:

```
#!/bin/bash

clear
if [ -e /home/mmarin/error.txt ]
        then
        echo "File exist"
        else
        echo "File does not exist"
fi
```

- Note the script first clears out the screen with a `clear` command.
- Note the `-e` keyword stands for "exists" and checks if the file specified next to it exists or not.
- This script will loke for a file named `error.txt` in the location `/home/mmarin/error.txt`, if the file is found it will output "File exist", if not, it will output "File does not exist"

Output from `./ifthen-new`:

```
File does not exist
[mmarin@linuxtest myscripts]$
```

- You can do `touch /home/mmarin/errir.txt` to create this file to test your script.

Check the response and then output

File editor:

```
#!/bin/bash

clear
echo
echo "What is your name?"
echo
```

```
read a
echo

echo Hello $a sir
echo

echo "Do you like working in IT? (y/n)"
read Like
echo

if [ "$Like" == y ]
then
echo You are cool

elif [ "$Like" == n ]
then
echo You should try IT, it’s a good field
echo
fi
```

- Note we can use the `elif` keyword for an optional condition in case the first one is not met.

Other if statements

If the output is either Monday or Tuesday

- `if [ “$a” = Monday ] || [ “$a” = Tuesday ]`

Test if the error.txt file exist and its size is greater than zero

- `if test -s error.txt`

If input is equal to zero (0)

- `if [ $? -eq 0 ]`

If file is there

- `if [ -e /export/home/filename ]`

If variable does not match

- `if [ "$a" != "" ]`

If file not equal to zero (0)

- `if [ error_code != "0" ]`

Comparisons:

- `-eq` equal to for numbers
- == equal to for letters
- `-ne` not equal to
- !== not equal to for letters
- `-lt` less than

- `-le` less than or equal to
- `-gt` greater than
- `-ge` greater than or equal to

File Operations:

- `-s` file exists and is not empty
- `-f` file exists and is not a directory
- `-d` directory exists
- `-x` file is executable
- `-w` file is writable
- `-r` file is readable

For Loop Scripts

- For loops
  - Keep running until specified number of variable
    - e.g: variable = 10 then run the script 10 times
    - e.g: variable = green, blue, red (then run the script 3 times for each color.)
  - It simplifies repetitive tasks

Simple for loop

File editor:

```
#!/bin/bash

for i in 1 2 3 4 5
do
echo "Welcome $i times"
done
```

- The letter `i` is a variable and it will hold the numbers 1, 2, 3, 4, 5, in different instances depending on the iteration number of the for loop.
- The `do` keyword specifies to literally "do (execute)" the following commands in the loop.
- The `done` keyword says the end of the for loop.
  - Anything inside the `do`-`done` space will be run the number of times the for loop specifies.

Output from `./abc`:

```
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

Simple for loop

File editor:

```
#!/bin/bash
# This script will output Max with different actions

for i in eat run jump play
do
echo See Max $i
done
```

- The `i` container will take a new value on each iteration=, the first value will be "eat", the next one is "run", and so on until there is no other word.

Output from `./xyz`:

```
See Max eat
See Max run
See Max jump
See Max play
```

For loop to create 5 files named 1-5

File editor:

```
#!/bin/bash

for i in {1..5}
do
         touch $i
done
```

- You can do the reverse action by replacing the `touch` command with a `rm` command
- Note the use of brackets and ".."

Specify days in for loop

File editor:

```
#!/bin/bash

i=1
for day in Mon Tue Wed Thu Fri
do
```

```
        echo "Weekday $((i++)) : $day"
done
```

- Note the `i++` statement, which adds 1 to the `i` variable on every iteration.

List all users one by one from `/etc/passwd` file

File editor:

```
#!/bin/bash

i=1
for username in `awk -F: '{print $1}' /etc/passwd`
do
        echo "Username $((i++)) : $username"
done
```

- `awk` or `gawk` is used for pattern scanning and processing language.

`grep` and `egrep`

- **What is grep?**
  - The grep command which stands for “global regular expression print” processes text line by line and prints any lines which match a specified pattern
  - In simple words is a "search" feature in Linux that allows you to search a specific keyword in a file or an output from a command.
- **Check version, and check help**
  - `grep --version`
    - It will give you the version of the `grep` service
  - `grep --help`
    - This will give you all the information about the `grep` command and its options.
- **Search for a keyword from a file**
  - `grep [keyword] [file]`
    - Replace the `[keyword]` with the actual term, or pattern you want to look for.
    - Replace `[file]` with the absolute or relative path of the file in which you want to look for the term.

Example using `grep` command:

```
[user@localhost ~]$ grep mmarin /etc/passwd
```

- Looking for the user "mmarin" in the `/etc/passwd` file

Output:

```
mmarin:x:1000:1000:Maximus Marin:/home/mmarin:/bin/bash
```

- It will give only one line because there is only one user in `/etc/passwd` that has the word "mmarin".

- **Search for a keyword and count**
  - `grep -c [keyword] [file]`
    - What if you have a file that has thousands of words that matches your criteria and you wanted to know how many times it shows up.
    - This command will output only the number of occurrences. Not the line where they appear.
- **Search for a keyword ignore case-sensitive**
  - `grep -i [keyword] [file]`
    - This is the option that you should always use.
    - Tell the system to ignore any lowercase/uppercase from the keyword and from that file.
- **Display the matched lines and their line numbers**
  - `grep -n [keyword] [file]`
    - It will give you the number of line in the file where the keyword exists, as well as the actual line in the file that contains the keyword.

Example using `grep` command:

```
[user@localhost seinfield]$ grep -n Seinfield seinfield-characters
```

Output:

```
1:Jerry Seinfield
8:Morty Seinfield
```

- **Display everything but keyword**
  - `grep -v [keyword] [file]`
    - If you want to get all the lines that do NOT match the keyword.
    - It will output only the lines that do not contain the keyword.
- **Search for a keyword and then only give the 1st field**
  - `grep [keyword] [file] | awk '{print $1}'`
    - What if you only wanted to print the first column from an output.

Example using `grep` command:

```
[user@localhost seinfield]$ grep -i Seinfield seinfield-characters | awk `{print $1}`
```

- The `awk` command with its option specifies to print only the first word of the line.

Output:

```
Jerry
Morty
```

Example using `grep` command:

```
[user@localhost seinfield]$ grep -i Seinfield seinfield-characters | awk `{print $1}` | cut -c1-3
```

- The `cut` command removes elements from the output.
- The `-c` option of the `cut` command stands for "characters" and specifies to cut characters.
- The `1-3` is to specify the range that wont be cut from the output. (`1-3` means to keep only the first 3 characters from the output, so it is inclusive)

Output:

```
Jer
Mor
```

- **Search for any keyword in any output of a command**
  - `ls -l | grep [keyword]`
    - Pipe any command with the `grep` command to search for a keyword in the outputs of that command.

Example using `ls` command:

```
[user@localhost ~]$ ls -l | grep -i desktop
```

Output:

```
drwxr-xr-x. 2 mmarin mmarin    6 Jun  3 12:42 Desktop
```

- **Search for 2 keywords**
  - `egrep -i "[keyword1]|[keyword2]" [file]`

- A little more powerful than the normal `grep` command
- Note the double quote `"`. You have to put them in order to work.
- Note the `|` sign separating the two keywords.

Example using `egrep` command:

```
[user@localhost seinfield]$ egrep -i "Seinfield|Costanza" seinfield-characters
```

- Searching for 2 keywords ("Seinfield" and "Costanza") with the `egrep` command.

Output:

```
Jerry Seinfield
George Costanza
Frank Costanza
Estelle Costanza
Morty Seinfield
```
