GACRC

Introduction to Linux Basics Part II

Georgia Advanced Computing Resource Center
University of Georgia
Suchitra Pakala
pakala@uga.edu

HOW DOES LINUX WORK?

- Variables in Shell
- Shell Arithmetic
- I/O and Redirection
  - Redirecting output, more, less, cat
- Piping, Sorting, Pattern Matching, Searching
- Decision making
  - If condition
- Loops
  - For loop
  - While loop

Variables in Shell

- In Linux (Shell), there are two types of variable:
- **System variables**: Created and maintained by Linux itself, this type of variable defined in CAPITAL LETTERS.
- **User defined variables (UDV)** : Created and maintained by user, this type of variable defined in lower letters.

System Variable	Meaning
BASH=/bin/bash	shell name
BASH_VERSION=1.14.7(1)	shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/pakala	home directory
OSTYPE=Linux	Operating System type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	path settings
PWD=/home/students/Common	current working directory
SHELL=/bin/bash	shell name
USERNAME=pakala	User name who is currently login to this PC

How to define User defined variables (UDV)

- *Syntax:* variable name=value
- '**value**' is assigned to given '**variable name**'
- Value must be on right side = sign
- Examples:

```
$ no=10     # this is fine
```

```
$ 10=no     # this is NOT Fine
            # value must be on right side of = sign
```

```
$ n=10      #to define variable n having value 10
```

```
$ vech=Bus   #to define variable vech having value Bus
```

Rules for Naming variable name

- **Don't put spaces on either side of the equal sign when assigning value to variable**
  - Example: the following variable declaration there will be no error

```
$ no=10    # No error
```

  - But there will be problem for any of the following variable declaration:

```
$ no =10
```

```
$ no= 10
```

```
$ no = 10
```

- **Variables are case-sensitive**

```
$ no=10    #will print 10
$ No=11    #will print 11
$ NO=20    #to print value 20,we need to use $echo $NO
```

- **You can define "NULL" variable**

```
$ tech=
$ tech=""     #nothing will be shown as variable has no value
```

- **Do not use ?,* etc, to name your variable names**

echo Command

- echo command is used to display text or value of variable.
  - echo [options] [string, variables...]
  - Options:

    -n Do not output the trailing new line.

    -e Enable interpretation of the following backslash escaped characters in the strings:

    \a alert (bell)

    \b backspace

    \c suppress trailing new line

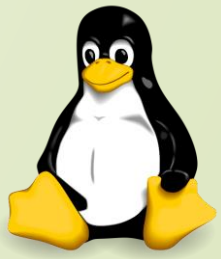
    \n new line

    \r carriage return

    \t horizontal tab

    \\ backslash

```
$ echo -e "An apple a day keeps away \a\tdoctor\n"
```



How to print or access value of UDV (User defined variables)

- To print or access UDV:
  - *Syntax:* $variablename

```
$ fruit=mango
$ n=25
$ echo $fruit
$ echo $n
```

Shell Arithmetic

- To perform arithmetic operations.
- *Syntax:* expr op1 math-operator op2

```
$ expr 3 + 3
$ expr 5 - 1
$ expr 20 / 2
$ expr 20 % 3
$ expr 3 \* 2
$ echo `expr 7 + 3`
```

8 More about Quotes

- There are three types of quotes:

Quotes	Name	Meaning
"	Double Quotes	"Double Quotes" - Anything enclosed in double quotes removed meaning of that characters (except \ and $).
'	Single quotes	'Single quotes' - Enclosed in single quotes remains unchanged.
`	Back quote	`Back quote` - To execute command

```
$ echo "Today is date" #cannot print message with today's date
```

```
$ echo "Today is `date`" # will print today's date
```

Quoting Examples

```
$ FRUIT=apples
$ echo 'I like $FRUIT'          # $ is disabled by ' '
$ I like $FRUIT
$ echo "I like $FRUIT"          # $ is not disabled by " "
$ I like apples
$ echo "I like \$FRUIT"         # $ is disabled forcedly by preceding \
$ I like $FRUIT
$ echo '`pwd`'                  # ` is disabled by ' '
$ `pwd`
$ echo "`pwd`"                  # ` is not disabled by " "
$ /home/gacrc-instruction/pakala
```

The read Statement

- *Syntax*: read variable1, variable2,...variableN

```
$ nano hello.sh
#!/bin/bash
# script to read your name from keyboard
#
echo " please enter your name:"
read name
echo " Hello $name, Lets be friends! "
```

```
$ chmod 755 hello.sh
$ ./hello.sh
$ please enter your name:suchitra
$ Hello suchitra, Lets be friends!
```

Wild cards

Wild card	Meaning	Examples	
*	Matches any string or group of characters.	**$ ls ***	Lists all files
		$ ls a*	Lists all files whose first name is starting with letter 'a'
		$ ls *.c	Lists all files having extension .c
		$ ls ut*.c	Lists files having extension .c but file name must begin with 'ut'.
?	Matches any single character.	**$ ls ?**	Lists all files whose names are 1 character long
		$ ls fo?	Lists all files whose names are 3 character long and file name begin with fo
[...]	Matches any one of the enclosed characters	**$ ls [abc]***	Lists all files beginning with letters a,b,c

I/O AND REDIRECTION

- Programs and commands can contain both inputs and outputs
- Input and outputs of a program are called "streams " in Linux
- There are three types of streams
  - **STDIN**: "standard input"-- by default, input from the keyboard
  - **STDOUT**: "standard output"--by default, output sent to the screen
  - **STDERR**: "standard error"--by default, error output sent to the screen

Output Redirection

- To redirect all directory content to output_file:

```
$ ls > my_file
```

- Redirection of this sort will create the named file if it doesn't exist, or else overwrite the named file if it does exist already. You can append the output file instead of rewriting it using a double ">>"

```
$ ls >> my_file
```

I/O AND REDIRECTION

- **Input Redirection**

➢ Input can also be given to a command from a file instead of typing it to the screen like this:

```
$ samplefile < file1
```

- **Error Redirection**

➢ When performing normal redirection, STDERR will not be redirected
➢ Many bash programmers find it useful to redirect only STDERR to a separate file
➢ If the program produces a lot of output, to make it easier to find the errors which are thrown from your program. Using the bash shell, this can be accomplished with "2>"

```
$ samplefile 2> error_file
```

➢ In addition one may merge STDERR to STDOUT with 2>&1

```
$ samplefile > output_file 2>&1
```

Redirecting output, cat , more, less

- list command and > to redirect your output to a file named mylist

```
$ ls -l /etc > mylist
```

- There are three methods for viewing a file from the command prompt: **cat, more and less**
- **cat** shows the contents of the entire file at the terminal, and scrolls automatically

```
$ cat mylist
```

- **more** shows the contents of the file, pausing when it fills the screen.
- Use the spacebar to advance one page at a time

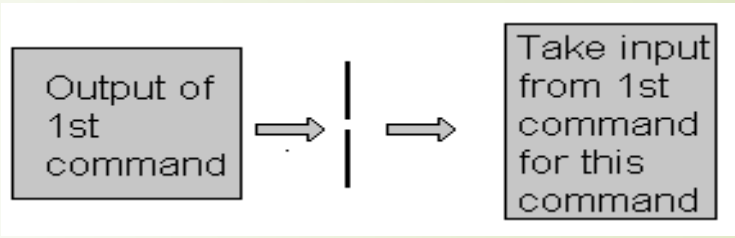
```
$ more mylist
```

- **less** also shows the contents of the file, pausing when it fills the screen.
- Use the spacebar to advance one page at a time, or use the arrow keys to scroll one line at a time (q to quit).
- "g" and "G" will take you to the beginning and end, respectively

```
$ less mylist
```

Piping

- A pipe is a way to connect the output of one program to the input of another program without any temporary file
- Using the pipe operator "|" you can link commands together.
- The pipe will link the standard output from one command to the standard input of another



- *Syntax:* command1 | command2

```
$ ls | more    #output of ls command is given as input to more command

$ who | sort  #output of who command is given as input to sort command which will
               print sorted list of user's

$ who | sort > user_list   # out of sort is redirected to user_list file

$ who | wc -l              #prints number of users who logon to system

$ who | grep suchi         #print if particular user name, if logon or nothing is
                           printed
```

Sorting

- The Linux **sort** command sorts the content of a file or any STDIN, and prints the sorted list to the screen

```
$ cat temp.txt
cherry
apple
x-ray
clock
orange
bananna
```

```
$ sort temp.txt
apple
bananna
cherry
clock
orange
x-ray
```

- To see sorted list in reverse order, use the **-r** option

```
$ sort -r temp.txt
x-ray
orange
clock
cherry
bananna
apple
```

Pattern Matching

- **grep** is another useful search utility
- It searches the named input file for lines that match the given pattern and prints those matching lines
- In the following example, search for instances of the word “World" in the file “sample1”
- If there are no matches, grep will not print anything to the screen

```
$ cat sample1
Welcome to the Linux World.
Linux is free and open source
Software.
```

```
$ grep World sample1
Welcome to the Linux World.
```

Searching

- Finding files on the system and finding a particular text string within a file are very useful.
- searching in **/usr/lib**, looking for files named libmenu.so, and whenever it finds one, prints its full path
- The **find** command is useful for finding where missing libraries are located, so the path may be added to the **LD_LIBRARY_PATH** environment variable

```
$ find /usr/lib -name libmenu.so -print
```

- **grep** command searches for patterns and prints matching lines
- Here, it looks for "score" in the file lincoln.txt

```
$ grep score lincoln.txt
```

- In following example, grep searches input from **ps -ef** (which outputs all processes in full format), and prints out a list of csh users

```
$ ps -ef | grep csh
```

- **More commands on one command line:**
- *Syntax*:command1;command2

```
$ pwd ; ls
$ cd .. ; ls
$ date ; who
```

- **Tilde Expansion (Home Expansion): ~**

```
$ cd ~username    # home directory associated username
$ cd ~            # replaced by $HOME
$ cd ~/           # same as above
```

- **Command Substitution: `command` (` is back quota!)**

```
$ cd `pwd`        # same as cd /home/gacrc-instruction/pakala
```

Decision Making

- bc - Linux calculator program.

Expression	Meaning to us	Your Answer	BC's Response
5 > 12	Is 5 greater than 12	NO	0
5 == 10	Is 5 is equal to 10	NO	0
5 != 2	Is 5 is NOT equal to 2	YES	1
5 == 5	Is 5 is equal to 5	YES	1
1 < 2	Is 1 is less than 2	Yes	1

- In bc, relational expression always returns **true** (1) or **false** (0 - zero).

if condition

- if condition which is used for decision making in shell script
- If given condition is true then command1 is executed.
- *Syntax:*

```
if condition
then
        command1 if condition is true or if exit status of condition is 0(zero)
fi
```

```
#!/bin/bash
#
#Script to print file
#
if cat $1
then
echo -e "\nFile $1, found and successfully echoed"
fi
```

```
$ nano sampledata.sh
$ chmod 755 sampledata.sh
$ ./sampledata.sh sample
Hello!!!!!
Welcome to Linux world....

File sample, found and successfully
echoed
```

- Shell script name is sampledata.sh($0)
- sample (which is $1) is a file
- If sample file exists, it will print sample files content to the screen.

Test Command

- test command or [expr] is used to see if an expression is true, and if it is true it returns zero(0), otherwise returns nonzero for false.
  - *Syntax*: test expression OR [expression]

```
#!/bin/bash
#
# Script to see whether argument is positive
#
if test $1 -gt 0
then
echo "$1 number is positive"
fi
```

```
$ chmod 755 test.sh

$ ./test.sh 25
25 number is positive

$ ./test.sh -5
Nothing is printed
```

GACRC

Flow Control

	Test Expression	Description
File testing	-e file	True if file exists
	-d or -f file	True if file exists and is a directory or a regular file
	-r or -w or -x file	True if file exists and is readable or writable or executable
	-s file	True if file exists and has a nonzero size
	file1 -nt or -ot file2	True if file1 is newer or older than file2
String testing	-z or -n string	True if the length of string is zero or nonzero
	str1 == str2	True if the strings are equal
	str1 != str2	True if the strings are not equal
	arg1 OP arg2	OP is one of –eq, -ne, -lt, -le, -gt, or -ge. Arg1 and arg2 may be +/- integers
ARITH testing Logical testing	! expr	True if expr is false
	expr1 -a expr2	True if both expr1 AND expr2 are true
	expr1 -o expr2	True if either expr1 OR expr2 is true

Loops

- **for Loop**:
  - *Syntax:*

    for { variable name } in { list }
    do execute one for each item in the list until the list is finished
    done
  - Example:

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

```
$ chmod 755 forloop.sh
$ ./forloop.sh
welcome 1 times
welcome 2 times
welcome 3 times
welcome 4 times
welcome 5 times
```

While Loop:

- *Syntax:*

```
while [ condition ]
do
        command1
        command2
        --
done
```

```
$ chmod 755 whileloop.sh
$ ./whileloop.sh 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

```
#!/bin/bash
#Script to test while statement
if [ $# -eq 0 ]
then
  echo "Error - Number missing from command
line argument“
  echo “syntax : $0 number”
  echo " Use to print multiplication table for given
number"
exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
 echo "$n * $i = `expr $i \* $n`"
 i=`expr $i + 1`
done
```

Bash Profile

- Why we have those automatically set *shell variables*?
  Configure your working environment on Linux as you wish!
- Example: .bash_profile for interactive login shell

```
if [ -f ~/.bashrc ]; then      # if .bashrc exists and is a regular file, then
        . ~/.bashrc            # run/source it in current shell to
fi                             # make interactive login and non-login shell
                               # to have the same environment

# User specific environment and startup programs
PATH=$PATH:$HOME/bin

export PATH
```

Shell Scripting Examples:

```
#!/bin/bash
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
# if no command line argument
if [ -z $1 ]
then
  rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
# otherwise make first argument as rental
  rental=$1
fi
case $rental in
   "car") echo "For $rental $45 per day”;;
   "van") echo "For $rental $85 per day";;
   "jeep") echo "For $rental $55 per day";;
   *) echo "Sorry, I can not get a $rental for you";;
esac
```

Shell Scripting Examples

- Serial job submission script (zcluster):

```
#!/bin/bash
cd /escratch4/pakala/pakala_Nov_13
export PATH=/usr/local/fastqc/latest:${PATH}
fastqc SRR1369670.fastq -o Output_File
```

- Batch Threaded job submission script (zcluster):

```
#!/bin/bash
cd /escratch4/pakala/pakala_Nov_13
time /usr/local/ncbiblast/latest/bin/blastall -p 2 [options]
```

https://wiki.gacrc.uga.edu/wiki/Running_Jobs_on_zcluster

Linux Command Reference

ls	–> directory listing
cd	–> change directory
pwd	–> show current directory
mkdir dir	–> create a directory dir
rm file	–> delete file
cp file1 file2	–> copy file1 to file2
mv file1 file2	–> rename or move file1 to file2
ln -s file link	–> create symbolic link link to file
touch file	–> create or update file
cat > file	–> places standard input into file
more file	–> output the contents of file
head file	–> output the first 10 lines of file
tail file	–> output the last 10 lines of file
file	–> to determine a file's type
grep pattern files	–> search for pattern in files
ps	–> display your currently active processes
top	–> display all running processes
kill pid	–> kill process id pid
chmod	–> change the permissions of file

- 4 – read (r)
- 2 – write (w)
- 1 – execute (x)

Thank You ☺