Linux Operating System and Programming
TOPIC # 14

Essential Shell Programming

In the simplest case, a script is nothing more than a list of system commands stored in a file. At the very least, this saves the effort of retyping that particular sequence of commands each time it is invoked. Moreover, we may write script to automate some task or for re-usability purpose. It is not complied with a separate executable file as with a C program but each statement is loaded into memory when it is to be executed . Hence shell scripts run slower than the programs written in high-level language. .sh is used as an extension for shell scripts.

Interpreter line / Sha-bang

The sha-bang (#!) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated. The #! is actually a two-byte magic number, a special marker that designates a file type, or in this case an executable shell script (type man magic for more details on this fascinating topic). Immediately following the sha-bang is a path name. This is the path to the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility. This command interpreter then executes the commands in the script, starting at the top (the line following the sha-bang line), and ignoring comments.

Commenting Statements in Shell Script

'#' sign is used to comment lines in shell script. The first line 'Sha-bang' is exception here. It starts with '#' but its not a comment.

Invoking the script

1) Change the file permission to be executable and run it:
   chmod +x myscript.sh
   ./myscript.sh

2) Give script name as parameter to any shell utility:

   sh myscript.sh
   Here, script need not to be executable. Moreover, sha-bang/interpreter line will be ignored as we are specifically using particular shell type here.

```
[jpandya@JMP ~]$   whereis sh
sh: /bin/sh /usr/share/man/man1p/sh.1p.gz /usr/share/man/man1/sh.1.gz
[jpandya@JMP ~]$ ll /bin/sh
lrwxrwxrwx 1 root root 4 Jul 27  2012 /bin/sh -> bash
[jpandya@JMP ~]$ ll /bin/bash
-rwxr-xr-x 1 root root 735004 Jan 22  2009 /bin/bash
[jpandya@JMP ~]$
```

3) Make entry in crontab and let it be scheduled.
4) Part of another shell script.

Making the shell script interactive using read statement:

read reads information from standard input and stores into variable.

The read statement is the shell's internal tool for making scripts interactive (i.e. taking input from the user). It is used with one or more variables. Inputs supplied with the standard input are read into these variables. For instance, the use of statement like
read name
causes the script to pause at that point to take input from the keyboard. Whatever is entered by you will be stored in the variable name.

read -s password
reads silently and hence does not echo on the screen. Similar to stty -echo and stty echo.

```
[jpandya@JMP shellscripts]$ cat readit.sh
#!/bin/bash
# Understanding read

#reads all inputs until enter key into the one and only variable specified
echo "Enter one or more input separated by space, tab. See that read var1 is executed."
read var1
echo $var1


#reads all inputs into variables specified. If exact input then all variables get values.
echo "Enter exactly three input separated by space(s) or tab. See that read var1 var2 var3 is executed."
read var1 var2 var3
echo $var1
echo $var2
echo $var3

#If more number of input are provided than the total variables specified, than last is assigned all remaining input.
echo "Enter exactly five input separated by space(s) or tab. See that read var1 var2 var3 is executed."
read var1 var2 var3
echo $var1
echo $var2
echo $var3

#If less number of input than total variables than remaining variables stay unassigned.
echo "Enter exactly three input separated by space(s) or tab. See that read var1 var2 var3 var4 var5 is executed."
read var1 var2 var3 var4 var5
echo $var1
echo $var2
echo $var3
echo $var4
echo $var5
```

Output:

```
[jpandya@JMP shellscripts]$ ./readit.sh
Enter one or more input separated by space, tab. See that read var1 is executed.
hi there how are you?
hi there how are you?
Enter exactly three input separated by space(s) or tab. See that read var1 var2 var3 is executed.
hi there        how
```

```
hi
there
how
Enter exactly five input separated by space(s) or tab. See that read var1 var2 var3 is executed.
hi there how are you
hi
there
how are you
Enter exactly three input separated by space(s) or tab. See that read var1 var2 var3 var4 var5 is executed.
hi there how
hi
there
how
```

Shell Positional Parameters / Using Command Line Arguments

Shell scripts also accept arguments from the command line. Therefore e they can be run non interactively and be used with redirection and pipelines. The arguments are assigned to special shell variables. Represented by $1, $2, etc; similar to C command arguments argv[0], argv[1], etc. The following table lists the different shell parameters.

$1, $2, ... positional parameters representing command line arguments
$# Total Argument count. This does not count the program name itself.
$0 name of the program/command
$* Complete set of parameters as one big string
“$@” Each quoted string is treated as a sequence argument

```
cat positional.sh
# Talking about positional parameters
#       $1, $2, .... positional parameters representing command line arguments
#       $# Total Argument count. This does not count the program name itself.
#       $0 name of the program/command
#       $* Complete set of parameters as one big string
#       "$@" Each quoted string is treated as a sequence argument

clear
if [ $# -eq 0 ];
then
        echo "No arguments provided to the $0 shell script or command.'" [ using $0 ]'
        exit 1
else
        echo "Total arguments provided $#"' [ using $# ]'
        echo "Arguments as one big string is $*"' [ using $* ]'
        echo "Arguments as one big string is $@"' [ using $@ ]'

        echo "All the arguments from the list as below :"
        echo 'Traversing through $* or $@'
        for arg in $* #for arg in "$@"
        do
                echo $arg
```

```
        done

        echo 'Refering $1 till last count of argument'
        for ((i=1; i<=$#; i++))
        do
                echo "At $i is ${!i}"
        done

        echo "Accessing last parameter directly ${!#}"

        echo "Have a nice day!!"
        exit 0
fi
```

IF Statements

The if statement makes two way decisions based on the result of a condition. The following forms of if are available in the shell:

Form 1	Form 2	Form 3
if command is successful then execute commands fi	if command is successful then execute commands else execute commands fi	if command is successful then execute commands elif command is successful then... else... fi

ifform1.sh

```
#!/bin/bash
#1st form------ if-then-fi statement
#example of copying two files
#
echo -e "Enter the source file and target file  name : \n"
read source target
if cp $source $target
then
        echo "file copied succesfully"
fi
```

ifform2.sh

```
#!/bin/bash
# 2nd form------ if-then-else-fi statement
#example of comparing two numbers for equality
#
echo -e "Enter a number \c"
read no1
echo -e "Enter another number \c"
read no2
if [ $no1 -eq $no2 ];
```

```
then
	echo "Both numbers are equal "
else
	echo "Both numbers are not equal"
fi
echo "Have a nice day."

[jpandya@JMP shellscripts]$ chmod +x ifform2.sh
[jpandya@JMP shellscripts]$ ./ifform2.sh
Enter a number 10
Enter another number 10
Both numbers are equal
Have a nice day.
[jpandya@JMP shellscripts]$ ./ifform2.sh
Enter a number 10
Enter another number 20
Both numbers are not equal
Have a nice day.
```

ifform3.sh

```
#!/bin/bash
# echo calling a script with its hardlink to work differently]
# i.e. this file name is call.sh and let's say there are date.sh, cal.sh, ls.sh hard links of this original file.
# And when user runs this shell script using sh date.sh or sh cal.sh or sh ls.sh
# via those hard link program behaves accordingly based on the name how it is executed. Note that $0 # is the
application name. The limitation is if you run ./date.sh $0 also stores ./ and it does not match.


if test "$0" == "date.sh"
then
	date
elif test "$0" == "cal.sh"
then
	cal
elif test "$0" == "ls.sh"
then
	ls
else
	echo "sorry"
fi
```

Exit status of command/shell script and null check using Swap.sh

```
#!/bin/bash
# Shell script to swap two numbers

#echo 'Enter two numbers separated by whitespace'
#read n1 n2
echo 'Enter n1'
read n1
```

```
echo 'Enter n2'
read n2

#if [ -z "$n1" ] || [ -z "$n2" ];
if [ -z "$n1" -o -z "$n2" ]; # -z for null check and -o for logical OR
then
        echo "Either n1 or n2 is not read properly";
        exit 1 #Failure
else
        echo 'You entered n1 as '$n1' and n2 as '$n2
        buffer=$n1
        n1=$n2
        n2=$buffer
        echo 'After swaping now n1 is '$n1' and n2 is '$n2
        exit 0 # Success
fi
```

Output:

```
[jpandya@JMP shellscripts]$ ./swap.sh
Enter n1
10
Enter n2
20
You entered n1 as 10 and n2 as 20
After swaping now n1 is 20 and n2 is 10
[jpandya@JMP shellscripts]$ echo $?
0
[jpandya@JMP shellscripts]$ ./swap.sh
Enter n1
10
Enter n2

Either n1 or n2 is not read properly
[jpandya@JMP shellscripts]$ echo $?
1
[jpandya@JMP shellscripts]$ ./swap.sh
Enter n1

Enter n2
20
Either n1 or n2 is not read properly
[jpandya@JMP shellscripts]$ echo $?
1
[jpandya@JMP shellscripts]$ ./swap.sh
Enter n1

Enter n2

Either n1 or n2 is not read properly
[jpandya@JMP shellscripts]$ echo $?
1
```

[jpandya@JMP shellscripts]$

The logical Operators && and ||

The shell provides two operators that aloe conditional execution, the && and ||.

Usage:

cmd1 && cmd2

cmd1 || cmd2

&& delimits two commands. cmd 2 executed only when cmd1 succeeds.

Example1:

$ grep ‘director’ emp.lst && echo “Pattern found”

Output:

9876 Jai Sharma Director Productions

2356 Rohit Director Sales

Pattern found

Example 2:

$ grep ‘clerk’ emp.lst || echo “Pattern not found”

Output:

Pattern not found

Shorthand for test

[and] can be used instead of test. The following two forms are equivalent

Test $x –eq $y and [$x –eq $y]

Numerical Comparison Operators used by test or [ ]

Test Meaning

-eq Equal to

-ne Not equal to

-gt Greter than

-ge Greater than or equal to

-lt Less than

-le Less than or equal to

String Comparison

Test True if

-z for checking that the value is null or not

s1 = s2 Strings s1 equal s2

s1 != s2 String s1 is not equal to s2

-n stg String stg is not a null string

-z stg String stg is a null string

stg String stg is assigned and not null

s1 == s2 String s1 equal s2 (Korn and Bash only)

friendly.sh

```
#!/bin/bash
# Friendly script which when username and password matched to predefined values displays a welcome message or displays let's be friend message.

ufriend="friend"
```

```
pfriend="dneirf"
echo "\n Hi, This script will show welcome message if you know name and password otherwise it will request to be friend."
echo -e "\n Enter Test Friend Name \c"
read username
echo -e "\n Enter Test Friend Password \c"
read -s password
# -s reads silently and hence does not display on the screen

if [ ! -z $username ] && [ ! -z $password ]; # -n means not null
then
	if [ $username == $ufriend ] && [ $password == $pfriend ];
	then
		echo -e "\n Welcome, you are a old friend and old is gold. !! "
	else
		echo -e "\n Hi, You seem to be a new admission. Let's be friend. "
		echo -e "\n Use username as $ufriend and password as $pfriend going forward."
	fi
else
	echo -e "\n Either username or password is blank"
	exit 1
fi
echo -e "\n Have a nice day !"
```

Output:

```
[jpandya@JMP shellscripts]$ ./friendly.sh
Hi, This script will show welcome message if you know name and password otherwise it will request to be friend.
Enter Test Friend Name jigar
Enter Test Friend Password
Hi, You seem to be a new admission. Let's be friend.
Use username as friend and password as dneirf going forward.
[jpandya@JMP shellscripts]$ ./friendly.sh
Hi, This script will show welcome message if you know name and password otherwise it will request to be friend.
Enter Test Friend Name friend
Enter Test Friend Password
Welcome, you are a old friend and old is gold. !!
[jpandya@JMP shellscripts]$
```

File Tests

-f file	file exists and is a regular file
-r file	file exists and is readable
-w file	file exists and is writable
-x file	file exists and is a executable
-d file	file exists and is directory

-s file file exists and has a size greater than zero
-e file file exists (Korn and Bash only)
-L file file exists and is a symbolink link
f1 -nt f2 f1 is newer than f2
f1 -ot f2 f1 is older than f2
f1 -ef f2 f1 is linked to f2

filetest.sh

```
echo "enter a file name :"
read f_name

if [ ! -z "$f_name" ]
then
      if [ -f "$f_name" ]
      then
            if [ -r "$f_name" -a -w  "$f_name" -a -x "$f_name" ] ; then
                  echo " file is having all the permissions "
            else
                  echo " read write and execute permission denied "
            fi
      else
          echo "file not exist"
      fi
fi
```

expr: Computation and String Handling

The Broune shell uses expr command to perform computations. This command combines the following two functions:
-Performs arithmetic operations on integers
-Manipulates strings
Computation:
expr can perform the four basic arithmetic operations (+, -, *, /), as well as modulus (%) functions.
Examples:

```
$ x=3 y=5
$ expr 3+5
8
$ expr $x-$y
-2
$ expr 3 \* 5       Note:\ is used to prevent the shell from interpreting * as metacharacter
15
$ expr $y/$x
1
$ expr 13%5
3
```

expr is also used with command substitution to assign a variable.
Example1:

```
$ x=6 y=2 : z=`expr $x+$y`
$ echo $z
```

```
8
Example2:
$ x=5
$ x=`expr $x+1`
$ echo $x
6
```

There are more operators like power and more.

```
[jpandya@JMP ~]$ echo $((2**5))
32
[jpandya@JMP ~]$ echo 2^5 | bc
32
[jpandya@JMP ~]$
```

String Handling:
expr is also used to handle strings. For manipulating strings, expr uses two expressions separated by a colon (:). The string to be worked upon is closed on the lef t of the colon and a regular expression is placed on its right. Depending on the composition of the expression expr can perform the following three functions:

1. Determine the length of the string.
2. Extract the substring.
3. Locate the position of a character in a string.

```
[jpandya@JMP ~]$ expr "hitherehowareyou" : '.*'
16
[jpandya@JMP ~]$ expr "hitherehowareyou" : '..\(.....\)'
there
[jpandya@JMP ~]$ expr "hitherehowareyou" : '[^t]*t'
3
[jpandya@JMP ~]$
```

Sources of list:

List from variables

List from command substitution

List from wildcards

List from positional parameters

The case CONDITIONAL

basiccalc.sh using while

```
ans=yes
```

```
while [ $ans == "yes" ]
do
        echo "Enter two values"
        read a b
        echo "Enter operator"
        read op
        case $op in
         plus|+|pl)
                        x=`expr $a + $b`
                        echo "$x" ;;
         sub|-|su)
                        x=`expr $a - $b`
                        echo "$x" ;;
         div|/)
                        x=`expr $a / $b`
                        echo "$x" ;;
         mul|"*")
                        x=`expr $a \* $b`
                        echo "$x" ;;
          *)
                        echo "sry" ;;
        esac

        echo "Enter yes to continue "
        read ans
done
```

identifykey.sh

```
#!/bin/bash
# Testing ranges of characters.
*
echo; echo "Hit a key, then hit return."
read Keypressshellscripts]$ vi ./lowercase_file_loop.sh

case "$Keypress" in
  [[:lower:]]   ) echo "Lowercase letter";;
  #[a-z]   ) echo "Lowercase letter";;
  [[:upper:]]   ) echo "Uppercase letter";;
  #[A-Z]   ) echo "Uppercase letter";;
  [0-9]         ) echo "Digit";;
  *             ) echo "Punctuation, whitespace, or other";;
esac
```

Following are actually regular expressions:

- Any alphabetical character, regardless of case [:alpha:]
- Any numerical character [:digit:]
- Any alphabetical or numerical character [:alnum:]
- Space or tab characters [:blank:]
- Hexadecimal characters; any number or A–F or a–f [:xdigit:]

Any punctuation symbol [:punct:]
Any printable character (not control characters) [:print:]
Any whitespace character [:space:]
Exclude whitespace characters [:graph:]
Any uppercase letter [:upper:]
Any lowercase letter [:lower:]
Control characters [:cntrl:]

Note that case statement can use regular expression which you learn in grep utility.

basiccalc2.sh using until construct

```
ans=y
until [ $ans == "n" ]
do
        echo "Enter two values"
        read a b
        echo "Enter operator"
        read op

        case $op in

        +) x=`expr $a + $b`
          echo "$x" ;;
        -) x=`expr $a - $b`
          echo "$x" ;;
        /) x=`expr $a / $b`
          echo "$x" ;;
        "*") x=`expr $a \* $b`
          echo "$x" ;;
        *)echo "sry" ;;
        esac
    echo "Enter n to stop "
    read ans
done
```

case can also specify the same action for more than one pattern . For instance to test a user response for both y and Y (or n and N).
Example:

```
Echo “Do you wish to continue? [y/n]: \c”
Read ans
Case “$ans” in
    Y | y );;
    N | n) exit ;;
esac
```

Wild-Cards: case uses them:
case has a superb string matching feature that uses wild-cards. It uses the filename

matching metacharacters *, ? and character class (to match only strings and not files in the current directory).

Example:

```
Case “$ans” in
      [Yy] [eE]* );;                    Matches YES, yes, Yes, yEs, etc
      [Nn] [oO]) exit ;;                 Matches no, NO, No, nO
            *) echo “Invalid Input”
esac
```

for: Looping with a list

```
for variable in list
do
        commands
done

for i in {1..10}
do
        echo $i
done

for ((i=1;i<=10;i++)
do
        echo $i
done
```

Set and Shift

The set statement assigns positional parameters $1, $2 and so on, to its arguments. This is used for picking up individual fields from the output of a program.

Shift: Shifting Arguments Left

Shift transfers the contents of positional parameters to its immediate lower numbered one. This is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes S2 and so on.

You can use shift command to parse the command line (args) option. For example consider the following simple shell script:

```
#!/bin/bash
#where to use shift command?
echo "Program to learn shift "
while  test $1  ;
do
      if [ "$1" = "-b" ]
      then
```

```
        ob=$2
    case $ob in
        16) basesystem="Hex";;
        8) basesystem="Oct";;
        2) basesystem="Bin";;
        *) basesystem="Unknown";;
    esac
    shift 2
    elif [ "$1" = "-n" ]
    then
        num=$2
        shift 2
    else
        echo "Program $0 does not recognize option $1"
        exit 1
    fi
done
output=`echo "obase=$ob;ibase=10; $num;" | bc`
echo "$num Decimal number = $output in $basesystem number system(base=$ob)"
```

Save and run the above shell script as follows:

$ chmod +x convert
$ ./convert -b 16 -n 500
500 Decimal number = 1F4 in Hex number system(base=16)
$ ./convert -b 8 -n 500
500 Decimal number = 764 in Oct number system(base=8)
$ ./convert -b 2 -n 500
500 Decimal number = 111110100 in bin number system(base=2)
$ ./convert -b 2 -v 500
Program ./convert does not recognize option -v
$ ./convert -t 2 -v 500
Program ./convert does not recognize option -t
$ **./convert -b 4 -n 500**
500 Decimal number = 13310 in Unknown number system(base=4)
$ ./convert -n 500 -b 16
500 Decimal number = 1F4 in Hex number system(base=16)

The Here Document (<<)

The shell uses the << symbol to read data from the same file containing the script. This is

referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.

myftp.sh

```
#!/bin/bash
# For downloading test file from ftp using "Here Document" concept

ftp -n ftp.ddu.ac.in <<BLOCKK
quote USER anonymous
quote PASS anonymous
cd pub
mget ebooklist.txt
bye
BLOCKK
grep java ebooklist.txt
echo "File downloaded successfully."
```

trap: interrupting a Program

Normally, the shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk. The trap statement lets you do the things you want to do when a script receives a signal. The trap statement is normally placed at the beginning of the shell script and uses two lists:

trap ‘command_list’ signal_list

When a script is sent any of the signals in signal_list, trap executes the commands in command_list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals – the ones used with the kill command.

Example: To remove all temporary files named after the PID number of the shell:

trap ‘rm $$* ; echo “Program Interrupted” ; exit’ HUP INT TERM

trap is a signal handler. It first removes all files expanded from $$*, echoes a message and finally terminates the script when signals SIGHUP (1), SIGINT (2) or SIGTERM(15) are sent to the shell process running the script.

A script can also be made to ignore the signals by using a null command list.

Example:

trap ‘’ 1 2 15

Nested for loop

pattern_nestedfor.sh

```
/bin/bash

# 1
# 1 2
# 1 2 3
# 1 2 3 4
# 1 2 3 4 5

for ((i=1;i<=5;i++))
do
      for ((j=1;j<=i;j++))
      do
```

```
    echo -n "$j "
    done
    echo  " "
done
```

Functions

test.sh

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World"
}

# Invoke your function
Hello
```

```
$./test.sh
Hello World
$
```

```
Debugging shell script

“set -x” turns on the debugging mode.

“Set +x” turns it off.

The shell prints each statement as it is being executed, affixing a + to each.
```

Objectives to work on Shell Scripts Development

1)

Write a script that upon invocation shows the time and date, lists all logged-in users, and gives the system uptime. The script then saves this information to a logfile, systemusage_mm_dd_yyyy.log format.

2)

Write a shell script to clean up log files. Use variables for specifying log file location, see that only root ($UID 0) shall be able to execute this script, others shall get message "Only root can execute log clean up.".

3)

Write a shell script to display and store to a file, 2 power n series using for loop. Here, n is entered by user. Do not use multiplication operator. In certain logic, shell script shows inappropriate result for n=63, why that happens. correct answer examples, pow(2,62) is 611686018427387904. pow(2,63) is 9223372036854775808.

4)

Write a shell script which reads a keyboard key and identifies the key from below categories and displays the

message on the screen. "Lowercase letter", "Uppercase letter", "Digit", "Punctuation, whitespace, or other". Hint: [[:lower:]] identifies "Lowercase letter" [[:upper:]] identifies "Uppercase letter" [0-9] identifies "Digit" * can identify others. Use case statement.:JMPandya

5)

Take user's birth date as input (DD/MM/YYYY) format and show how many days, months and years he/she is of as of today(system date). You may use expr, backtick(``) and date command with format attribute to do maths.

6)

Write a shell script which takes username as input (read name only if it's not provided as command line argument) and checks out whether the user is logged in currently or not. If the user is not logged in then verify whether the user account exist or not. Display appropriate messages. Hint: use 'who -u' for logged in users and /etc/passwd file for user account.: Zeel Soni (Student)

7)

Using loops create following patterns

```
*
**
***
****
*****
```

```
_ _ _ _ 1
_ _ _ 1 2
_ _ 1 2 3
_ 1 2 3 4
1 2 3 4 5
```

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

Fibonacci series:
1 1 2 3 5 8 13 21 34 ...

8) Write a program to display multiplication table.

9)

Write a shell script to find unknown x and y from two Quadratic equations a1xx+b1y+c1=0 and a2xx+b2y+c2=0. Read a1,b1,c1 and a2,b2,c2 and display x and y. You may use bc(basic calculator) to achieve intermediate results. : Toral Vyas and Kothari Urja (CE Student)

10)

write a shell script to display all factors of input number without using the command "factor". If the input is not a number, display an error message. Please, learn that "factor" is an existing command as well which shows all factors of a number. Also, later see that if you can display uniqe factors. Also, may be prime factors.

[jpandya@JMP ~]$ factor 50
50: 2 5 5
[jpandya@JMP ~]$ factor sdf
factor: `sdf' is not a valid positive integer
[jpandya@JMP ~]$

: Padmaja Amin (CE Student)

11)

Write a shell script to download file /pub/ebooklist.txt from ftp.ddu.ac.in and display all files for java. You may use “Here Document” concept to practice.

Courtesy:
http://www.freeos.com/guides/lsst/ch04sec14.html

Contributors (CE Department, DDU):

- Prof. Jigar M. Pandya