Module III : Shell Programming

Basics of shell programming

A shell is a command line interpreter. It allows us to interact with it by entering commands from the keyboard, execute the commands and display its output on the monitor. The interaction is text – based. This type of interface is called Command Line Interface or CLI.

To determine the current active shell, use the command: echo $0

To find all available shells in the system, use the command: cat /etc/shells

The shell script is a computer program containing variables, control structures, commands, functions, etc. it is designed to run by the Linux Shell.

Types of Shell

There are many types of Shells in Linux

1. Bourne Shell (sh) - It is the original UNIX shell. It is portable, compact in size, requires minimal resources and executes rapidly. The prompt for this shell is '$' it is the base of other shells like POSIX shell Korn Shell & Bourne Again Shell.
2. BASH Shell (bash) - It is a popular shell, now found as the default on most Linux systems. It is free ware shell.
3. Korn Shell (ksh) - It is compatible with Bourn shell. It includes most of the features of Bourne Shell. It provides better performance.
4. C Shell (csh) - It uses the syntax of the C programming language. It provide interactive features such as command alias and history to make it more suitable for interactive applications. But the script language of the C shell is not compatible with Bourne, Korn and BASH Shells.
5. Tenex C Shell (tcsh) - It is an extended version of C Shell. It has programmable command line completion, command line editing, and few other features.
6. Z Shell (zsh) - It is an extended Bourne Shell with a large number of improvements. It includes some features of bash, ksh and tcsh. It has some unique features like file name generation.

Shell Name	Developed By	Where	Remark
Bourne Shell	Stephen Bourne	AT & T Bell labs for Unix system in 1977	Original UNIX Shell
BASH Shell	Brain Fox	As part of GNU project in 1988	Most common Shell in Linux
Korn Shell	David G Korn	AT & T Bell labs in 1983	Better performance
C Shell	Bill Joy	University of California	Similar to C Programming language
Tenex C Shell	Ken Greer	Cornegie Millon University	Enhanced version of C Shell
Z Shell	Paul Falstad	While studying at Princeton University in 1990	Powerful scripting language

Example

1. The following script uses the read command which takes the input from the keyboard and assigns it to the variable 'PERSON' and prints it on the screen.

```
echo "What is your name ? "
read PERSON
echo " Hello , $ PERSON"
```

2. To create a script containing 2 commands

```
# Two commands
pwd
ls
```

To execute it, type the following in the command prompt.

```
sh filename
```

Shell Programming

Rules for naming variables

- Begin with alphanumeric characters or under score (_) followed by one or more alphanumeric characters.
- Don't use special characters.
- Values are case sensitive.
- Null variables can be defined as $v="".
- Don't put space on either sides of the equal (=) sign.

Arithmetic operators

1. + - addition $ a + $ b
2. - - subtraction $ a - $ b
3. * - Multiplication $ a * $ b
4. / - Division $ a \ / $ b
5. % - remainder after Division

Relational Operators

1. – eq - Equals
2. – ne - not equal
3. – gt - greater than
4. – ge - greater than or equal
5. – lt - less than
6. – le - less than or equal

Logical operators

1. – a - AND
2. – o - OR

Quotes

There are 3 types of quotes

1. " - double quotes
2. ' - single quotes
3. ` - back quotes

Control Structures:

1. if - Simple if is used for decision making in shell script. If the given condition is true, then it will execute the code inside the block.

Syntax:

```
if [ condition ]
then
   statements
fi
```

eg:

```
# Check number is 1
    echo " enter a number "
    read no
    if [ $ no – eq 1 ]
    then
       echo " number 1"
    fi
```

2. if … else - it is also used for decision making. If the condition is true, then it will execute the statements in the true block; otherwise it will execute the false block.

Syntax:

```
if [ condition ]
then
    statements
else
    statements
fi
```

eg:

```
# Check number is positive or not
echo " enter a number "
read no
if [ $ no –gt 0 ]
then
   echo " number is positive "
else
   echo " number is negative "
fi
```

3. if … elif - It is possible to create compound conditional statements using els .. if (elif). If the 1st condition is true, then the true part is executed. Otherwise, the 2nd condition is checked. If the 2nd condition is true, the true part of elif is executed.

Syntax:

```
if [ condition ]
then
   statements
elif [ condition ]
then
   statements
fi
```

eg: Check + ve or - ve

```
echo " enter a number "
read n
if [ $ n –gt 0 ]
then
   echo " Positive "
elif [ $ n –lt 0 ]
then
   echo " Negarive "
fi
```

4. If .. elif … else - if the condition1 is true, then the true block is executed. If the condition 2 is true, then the true block of elif is to be executed. Otherwise the else block is to be executed.

Syntax:

```
if [ condition1 ]
then
   statements
elif [ condition 2 ]
then
   statements
else
   statements
fi
```

eg: Check + ve or - ve or 0

```
echo " enter a number "
read n
if [ $ n –gt 0 ]
then
   echo " Positive "
```

```
elif   [  $ n –lt  0 ]
then
    echo  " Negarive "
else
    echo  " zero "
fi
```

5. if – elif ladder - It is a series of if statements. Here each if is a part of the else clause of the previous if . Here statements are executed based on the true condition. If none of the condition is true then else block is executed.

Syntax :

```
if [  condition  ]
then
    statements
elif  [  condition  ]
then
    statements
elif  [  condition  ]
then
    statements
.
.
.
else
    statements
fi
```

eg: print day name corresponding to day number

```
echo " Enter a number between 1 and 7 "
read n
if [  $n – eq  1]
then
    echo  " Sunday "
elif [  $n – eq  2 ]
then
    echo  " Monday "
elif [  $n – eq  3 ]
then
    echo  " Tuesday "
elif [  $n – eq  4 ]
then
    echo  " Wednesday "
elif [  $n – eq  5 ]
then
    echo  "Thursday "
elif [  $n – eq  6 ]
then
    echo  " Friday "
elif [  $n – eq  7 ]
then
    echo  "Saturday "
else
    echo  " Not valid "
fi
```

6. Nested if statement : When an if statement contain many elif constructs then we say that it is a nested if statement.

Syntax:

```
if [ condition]
then
    statements
else
```

```
        if [ condition]
        then
            statements
        fi
    fi
```

eg: Check for user name and password

```
echo "Name"
read name
if [ "$name" == "abcd" ]; then
    echo "Password"
    read password
    if [ "$password == "pwd" ]; then
        echo "Hello"
    else
        echo "Wrong password"
    fi
else
    echo "wrong username"
fi
```

7. case - case statements are used to transfer the control to one of multiple statements. Case statements are used instead of if…elif ladder.

Syntax:

```
case exp  in
  pattern1 )  statements  ; ;
  pattern2 )  statements  ; ;
  pattern3 )  statements  ; ;
        .
        .
        .
* )          statements  ; ;
esac
```

The pattern is evaluated and the control is switched to the statements matching any of the patterns. If none of the patterns is matched, then the control goes to statements following " * ".

eg 1: # display the day number of a week

```
echo " Enter a number between 1 and 7 "
read n
case  $n  in
  1) echo  "Sunday " ; ;
  2) echo  "Monday " ; ;
  3) echo  "Tuesday " ; ;
  4) echo  "Wednesday " ; ;
  5) echo  "Thursday " ; ;
  6) echo  "Friday " ; ;
  7) echo  "Saturday " ; ;
  *) echo  "invalid day number " ; ;
esac
```

eg 2: Read a character and display whether it is vowel or not

```
echo " Enter a Character "
read c
case  $c  in
  [ a e i o u A E I O U ] )  echo  "  It is vowel " ; ;
  *)                        echo  " Not a vowel  " ; ;
esac
```

Looping Statements:

1. while - It is used to repeatedly execute a set of statements any number of times based on a condition. (until a condition is occurred)

Syntax :
```
while [ condition ]
do
   statements
done
```

eg:
```
# write a shell script to display integers from 1 to 10
i = 1
while [ $i - lt 10 ]
do
        echo $i
        i = ‘ expr $i + 1 ‘
done
```

2. until - It is used to execute a set of commands repeatedly until a condition is true.

Syntax:
```
until [ condition ]
do
    statements
done
```

eg:
```
# To display numbers from 1 to 10
i = 1
until [ $i - gt 10 ]
do
        echo $i
        i = ‘ expr $i + 1 ‘
done
```

3. for

Syntax (1):
```
for variable_name in list
do
statements
done
```

For each value in the list, the variable_name gets the value and the loop is executed.

eg (i):
```
for no in ( 1 . . 10 )
do
echo $ no
done
```

(ii):
```
# to print the numbers 1 3 5 7 10
for i in 1 3 5 7 10
do
echo $ i
done
```

Note: for, do, done, in are keywords

Syntax (2):
```
for ( ( expn 1 ; expn 2; expn 3 ) )
do
    statements
done
```

This syntax is similar to the syntax of for statement in C++ language. expn1 is executed 1st for initialization. expn2 is executed next for checking condition. If it is true, the loop is executed. Then expn3 is executed for incrementation or decrementation of the loop control variable. Then the loop is repeated if expn 2 is evaluated to true, and so on.

eg:
```
for ( ( i=1 ; I –le 10; i ++ ) )
do
    echo $ i
done
```

Control Statements used in looping statements

1. break - The break statement is used to terminate the execution of the loop and transfer the control to the statement after the end of loop.

Syntax 1): break

Used to exit from the loop

Syntax 2): break n

Used to exit from a nested loop. 'n' specifies the number of loops to be exited.

eg:
```
a=0
while [ $a –lt 10 ]
do
   echo  $a
   if [ $a  -eq  5 ]
   then
      break
   fi
   a= ` expr $a + 1 `
done
```

2. continue - It is used to transfer the control to the next iteration of the loop, ignoring the remaining portion of the current iteration.

Syntax 1): continue

Syntax 2): continue n

Here ' n ' specifies the n^{th} enclosing loop to continue.

Eg: # to display odd numbers from a list
```
NUMS = " 1 2 3 4 5 6 7 "
for  NUM  in  NUMS
do
    Q = ` expr  $ NUM  % 2 `
    if [  $Q  - eq 0  ]
    then
       continue
    fi
    echo  $Q
done
```

Passing Arguments to the Shell Script (Parameter passing & arguments)

Arguments can be passed to the script when it is executed. They are called command line arguments. To pass command line arguments, we can write them after script name separated with space.

eg: sh file.sh 10 20 30

Maximum length of command line parameters are not defined by shell, but by the OS.

Positional Parameters

There are a series of special variables($1, $2,…) that contain the contents of the command line.

$1 - references 1^{st} command line argument.

$2 - references 2^{nd} command line argument.

. . . . . .

. . . . . .

$o - references name of the script.

$2 - references 2^{nd} command line argument.

$* - references all command line arguments.

$@ - references all command line arguments.

$# - references count of command line arguments.

eg 1: sh file.sh 10 20 30

```
echo " Script name : $o"
echo " Total number of arguments: $ #"
echo " Argument list : "
echo " 1. $ 1 "
echo " 2. $ 2 "
echo " 3. $ 3 "
echo " All arguments are : $ * "
```

eg2: # To find largest value accepted from command line arguments.

```
if [ $1 - gt $2 ]
then
        echo " large is $ 1 "
else
        echo " large is $ 2 "
fi
```

To run : sh large 89 23

Shell variables - They are variables used in shell programming. Two types are :

1. user – defined shell variables
2. system defined shell variables

System Defined Shell Variables - These variables are pre-defined variables. All system variables are expressed in uppercase letters. Following are some system defined shell variables.

1. PATH - Describes the directories that are to be searched whenever a command is executed. A colon (:) separates one path from the next in the list.
   eg: PATH = : / bin : / usr / bin : / usr /s3c
   echo $ PATH
2. LOGNAME - Describes the login name
   eg : echo $LOGIN
3. HOME - Describes the path of the user's home directory.
   eg: echo $ HOME

Shell Keywords in Linux

Keywords are the words whose meaning are already been explained to the shell. The keywords can't be used as variables names. Following are some keywords used in Linux

echo	if	while	exit	umask
read	else	do	return	ulimit
set	elif	done	for	trap
unset	fi	until	exec	
shift	case	break	eval	
export	esac	continue	wait	

Automating System Tasks - Tasks (jobs) can be configured to run automatically within a specific period of time, or on a specific date, or when the system load average decreases below a specific value. The automated task utilities are : - cron, anacron, at and batch.

1. **anacron** - cron and ancron are daemons that can schedule execution of recurring tasks at the exact time, day of the month, month, day of the week and week. cron jobs can run as often as every minute. Cron assumes that the system is running continuously. If the system is not on at the time when a job is scheduled then the job is not executed. But anacron remembers the scheduled jobs. If the system is not running at the time when the job is scheduled. The job is then executed as soon as the system is up. Anacron can only run a job once a day.

eg: A simple etc / anacron tab file

SHELL = / bin / sh
PATH = / bin :/ usr /bin : / sbin
MAIL TO = root
RANDOM_DELAY = 30
START_HOURS_RANGE = 16 – 20

Period in days	Delay in Minutes	Job identifier	Command
1	20	Daily job	ls home >/tmp/proc
7	25	Weekly job	/etc/weeklyjob.sh

1st three lines define the variables that configure the environment.

- SHELL - Shell environment used for running jobs
- PATH - Path to execute the programs
- MAILTO -usernames of the users who receive the output of the anacrons jobs by email.

The next 2 variables modify the scheduled time for the defined jobs.

- RANDOM_DELAY – Maximum number of minutes that will be added to the delay. If the time is missed, the scheduled jobs are not executed on the day.

The remaining lines are in the following format:-

- Period in days – Frequency of job execution in days.
- Delay in minutes – number of minutes to wait before executing the job.
- Job identifier – unique name of a particular job.
- Command - the command to be executed.

2. cron - Configuration files for cron jobs are in the etc/crontab directory, which can only be modified by root user. The file contains the following:

The 1st three lines contain the SHELL, PATH and MAILTO variables. In addition, the file can contain HOME variable. The home variable defines the home directory. The remaining lines have the following format:-

Minute hour day month day of week user name command

Minute - integer from 0 -59
Hour - 0 – 23
Day - 1 – 31
Month - 1 – 12
Day of week - 0 – 7
User name - specifies the user
Command - commands to be executed

* can be used to specify all valid values.

Any line begin with # are comments and are not processed.

To create a crontab, use the command:-

crontab –e : to edit

eg: 29 0 * * * / usr / bin /example

To run the command /usr / bin / example at 12.30 everyday.

3. at - Refer scheduling commands.