Linux Shell Programming

Marin Iuliana

What is Shell?

- provides an interface to the LINUX system
- environment in which we can run our commands, programs, and shell scripts
- the prompt, $, which is called command prompt, is issued by the shell
- command prompt can be customized using the environment variable PS1
- In LINUX there are two major types of shells:
  - The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
  - The C shell. If you are using a C-type shell, the default prompt is the % character.

Borne Shell

- Is preceded by a pound sign, #
- Shell scripts and functions are both interpreted. This means they are not compiled.
- Note all the scripts have the .sh extension
- Alert the system that a shell script is being started. This is done using the shebang construct. (*It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.)*

#!/bin/sh (you put this in a script ending with .sh)

- This tells the system that the commands that follow are to be executed by the Bourne shell.
- There are again various subcategories for Bourne Shell which are listed as follows:
- Bourne shell (sh) Korn shell (ksh) Bourne Again shell (bash) POSIX shell (sh).

Let’s play!

- Create a script file

```
touch script.sh
```

- Write the following:

```
#!/bin/bash
# fun
pwd
ls
```

- save the above content and make this script executable

```
$chmod +x script.sh
```

- Execute the script using:

```
$./script.sh
```

Using Variables

- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.
- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_). By convention, Linux Shell variables would have their names in UPPERCASE.
- The reason you cannot use other characters such as !,*, or - is that these characters have a special meaning for the shell.
- Example:

```
#!/bin/sh
VAR=“Good night!“
echo $VAR
```

- The shell provides a way to mark variables as read-only by using the readonly command

```
readonly VAR
```

- Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

```
unset VAR
```

Variable Types

- **When a shell is running, three main types of variables are present:**
- **Local Variables: A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.**
- **Environment Variables: An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.**
- **Shell Variables: A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.**

Special Parameters $* and $@

- **There are special parameters that allow accessing all of the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".**
- **Both the parameter specifies all command-line arguments but the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.**

```
#!/bin/sh
for VAR in $*
do
    echo $VAR
done
```

- **Test with:**

```
$./test.sh Have a nice day
Have
a
nice
day
```

Arrays

```
#!/bin/sh
NAME[0]=“Ionel“
NAME[1]=“Radu“
NAME[2]=“Vasile“
echo "First Index: ${NAME[0]}“
echo "Second Index: ${NAME[1]}“
```

- **You can access all the items in an array in one of the following ways:**

```
${array_name[*]}
${array_name[@]}
```

Arithmetic Operations

```
#!/bin/sh
val=`expr 2 + 2`
echo "Total value : $val"
```

NOTE: There must be spaces between operators and expressions

- **It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.**
- **All the arithmetical calculations are done using long integers.**

Arithmetic Operations

Assume variable a holds 10 and variable b holds 20

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

Relational Operators

Assume variable a holds 10 and variable b holds 20

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example:

[$a <= $b] is correct where as [$a <= $b] is incorrect.

Boolean Operators

Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

Assume variable a holds "abc" and variable b holds "efg" then:

Operator	**Description**	**Example**
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-n $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[str $a] is not false.

File Test Operators:

Assume a variable file holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on:

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-c $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

Decision Making

- The if...else statements

if…fi

if …else …fi

If … elif …else…fi

- The case...esac statement is very similar to switch

Shell Loops

- **while**
- **for**
- **until**
- **select**

```
#!/bin/sh
a=0
while [ "$a" -lt 10 ]    # this is loop1
do
    b="$a"
    while [ "$b" -ge 0 ]  # this is loop2
    do
        echo -n "$b "
        b=`expr $b - 1`
    done
    echo
    a=`expr $a + 1`
done
```

- **Here -n option lets echo to avoid printing a new line character.**

Loop Control

- break
- continue

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
    echo $a
    if [ $a -eq 5 ]
    then
        break
    fi
    a=`expr $a + 1`
done
```

```
#!/bin/sh
NUMS="1 2 3 4 5 6 7“
for NUM in $NUMS
do
        Q=`expr $NUM % 2`
        if [ $Q -eq 0 ]
        then
                echo "Number is an even
number!!"
                continue
        fi
        echo "Found odd number"
done
```

Shell Substitutions

- **The shell performs substitution when it encounters an expression that contains one or more special characters.**
- **"\n" is substituted by a new line**

```
#!/bin/sh
a=10
echo -e "Value of a is $a \n“
```

- **Here -e option enables interpretation of backslash escapes.**

Escape sequences

Escape	Description
\\	backslash
\a	alert (BEL)
\b	backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

You can use -e option to disable interpretation of backslash escapes (default).
You can use -n option to disable insertion of new line.

Variable Substitution

Form	Description
${var}	Substitue the value of *var*.
${var:-word}	If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change.
${var:=word}	If *var* is null or unset, *var* is set to the value of **word**.
${var:?message}	If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly.
${var:+word}	If *var* is set, *word* is substituted for var. The value of *var* does not change.

```
#!/bin/sh
echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"
```

Mechanisms

- **The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell:**

```
#!/bin/sh
echo "I have \$1000“
```

Output: I have $1000

Ex:

```
DATE=`date`
echo "Current Date: $DATE“
```

Will display the current date.

I/O Redirections

- **who command redirects complete output of the command in users file.**

$ who > users

- **Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check *users* file then it would have complete content:**
- **$ cat users**
- **If a command has its output redirected to a file and the file already contains some data, that data will be lost.**

```
$ echo line 1 > users
$ cat users
line 1
```

- **You can use >> operator to append the output in an existing file.**

Redirections

- **As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.**
- **To count the number of lines in the file *users:***

```
$ wc -l users
2 users
```

- **Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the wc command from the file *users*:**

```
$ wc -l < users
2
```

- **The << operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.**

```
$wc -l << EOF
    content
    EOF
1
```

Shell Functions

```
#!/bin/sh
# Define your function here
Hello () {
    echo "Hello World $1 $2“
    Hau
    return 10
}
Hau(){
    echo “Bau!”
}
# Invoke your function
Hello Ionel Popescu
ret=$?
echo “Value is $ret”
```

- **Execute script: $./test.sh**

Output: Hello World Ionel Popescu

```
Output: Hello World Ionel Popescu
    Bau!
    Value is 10
```

Applications

- Check if a number is prime.
- Print on the screen the first N prime numbers where N is read from the keyboard.
- Print on the screen the first N terms of the Fibonnaci sequence.

Homework

- Cows and Bulls game

Bulls: number of digits placed correctly

1	2	3	4
1	3	2	4

2 bulls

2 cows

Homework

- Same request, but generate the string to which the comparison is done randomly.

- Check if a string is a palindrom or not.

- For a given IP with its associated mask compute:

a) Network and Broadcast address

b) The first and the last available address for stations

c) The total no of station addresses available