SHELL SCRIPTING / PROGRAMMING

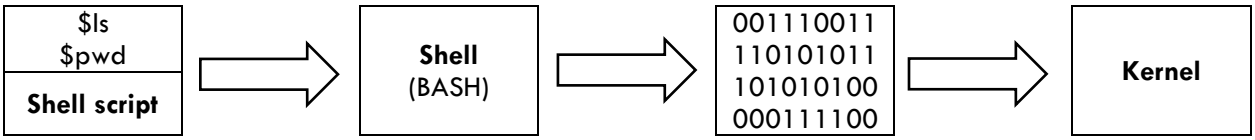
CONCEPTS

Kernel is the heart of an operating system. It manages computer resources and *runs different programs*.

Shell is a *command language interpreter* that (1) accepts commands from the standard input device (keyboard) or from a file, (2) translates them into native binary language and (2) passes these commands to the kernel for actual execution.

Shell script or **shell program** is a *collection of commands* that are stored in a *file*. The shell can read this file and act on the commands as if they were typed at the keyboard.

In short, this is what happens when a command or a shell script is being executed:



RATIONALE

- Shell script can take input from user or file and output them on screen or another file
- Useful to create our own commands
- Saves lots of time
- Automates some task of day-to-day life
- Automates system administration tasks

WRITING, SETTING PERMISSION, AND EXECUTING A SHELL SCRIPT

To write a shell script, you can use a command line or a GUI editor.

Command line editors: **`vi`**, **`vim`**, **`nano`**, **`pico`**

GUI editors: **`gedit`**, **`kwrite`**

To set proper permission to execute a shell script you need to issue the following command:

```
chmod 755 script-name
```

To run or execute a shell script CHOOSE ONE of the following ways:

`./script-name`	Assuming you are in the **same directory** as the shell script
`bash script-name`	
`sh script-name`	

SAMPLE SHELL SCRIPT

```
1  #!/bin/bash
2  #A simple script for printing 'Hello Word'
3  clear
4  echo "Hello World"
5  exit 0
```

VARIABLES

1. ***System variables*** – created and maintained by Linux itself, uses UPPER-CASE LETTERS,
   `printenv`: command to list all system or environment variables and their respective values

SYSTEM VARIABLE	SAMPLE VALUE	MEANING
SHELL	/bin/bash	Shell name
OSTYPE	linux	Type of OS
LOGNAME	joman	Logging name
PWD	/home/joman/Desktop	Current working directory
USERNAME	juandelacruz	Username of currently logged-in user
PATH	/usr/bin:/sbin:/bin	Path settings and directories

2. ***User-defined*** variables – created and maintained by the user, user lower-case letters

   SYNTAX: **`variable_name=value`**

   NAMING CONVENTIONS:
   1. It must start with a letter {a-z, A-Z}.
   2. It must not contain embed spaces. Use underscore instead.
   3. Don't use punctuation marks.
   4. Don't use a name that is already a word understood by bash. These are called *reserved words* and should not be used as variable names. If you use one of these words, bash will get confused. To see a list of reserved words, use the help command.
   5. It is case-sensitive.

   EXAMPLES:

```
x=10            nickname=joman
no=7.5          fullname="juan dela cruz"
y=-4            course=""
age=
```

   ACCESSING THE VALUES OF VARIABLES

   SYNTAX:

```
$SYSTEMVARIABLE
$variable_name
${variable_name}
```

   EXAMPLES:

```
$USERNAME
$nickname
${fullname}
```

SUBSTITUTIONS AND ASSIGNMENT OPERATOR

SOURCE:

```
1  #!/bin/bash
2
3  title="Shell Scripting"
4  today=$(date +"%x %r %Z")   #inside the parentheses are actual linux command
5
6  echo $HOME      #system variable
7  echo $title     #user-defined variable
8  echo $today     #user-defined variable
```

OUTPUT:

```
/home/user
Shell Scripting
Friday, 09 August, 2013 01:21:49 PM PHT PHT
```

NOTE: Actual linux commands can be enclosed inside `$(linux-command-here)` and the output can be assigned to a variable as shown in line number 4.

SINGLE QUOTES, DOUBLE QUOTES AND ESCAPE CHARACTER

SOURCE:

```
1  #!/bin/bash
2
3  var1="this is some text"
4  var2='this is some text'
5
6  echo $var1
7  echo $var2
8
9  echo "My hostname is $HOSTNAME"
10 echo 'My hostname is $HOSTNAME'
11
12 echo "My hostname is \$HOSTNAME"
13 echo 'My hostname is $HOSTNAME'
```

OUTPUT:

```
this is some text
this is some text
My hostname is Ubuntu32-VirtualBox
My hostname is $HOSTNAME
My hostname is $HOSTNAME
My hostname is \$HOSTNAME
```

Single Quotes

Does not evaluate or substitute actual values of variables
Prints the string as is, does not recognize escape character '\'

Double Quotes

Evaluates or substitutes actual values of variable
Recognizes the escape character '\'

BASIC INPUT/OUTPUT

Getting input from the user:

```
read variable_name
```

Printing output to the screen:

```
echo "output"
```

NOTE: Variables in shell scripts are ***loosely-typed.***

SOURCE:

```
1  #!/bin/bash
2
3  echo "Enter your name: "
4  read name
5
6  echo "Nice meeting you $name!"
```

BASIC ARITHMETIC

SYNTAX:

```
$((arithmetic-expressions))
```

ARITHMETIC OPERATORS:

```
+ - * / % ** ( )
```

NOTE: Extra spaces inside the double parentheses are allowed. Grouping of expressions using parentheses are also permitted. Variables need not have '$' when used inside `$(( ))`.

SOURCE:

```
1  #!/bin/bash
2
3  x=10
4  y=5
5
6  echo "Sum is $((x+y))"
7  echo "Difference is $((x-y))"
8  echo "Product is $((x*y))"
9  echo "Modulo is $((x%y))"
10 echo "Average is $(( (x+y)/2 ))"
```

OUTPUT:

```
Sum is 15
Difference is 5
Product is 50
Modulo is 0
Average is 7
```

CONDITIONAL STATEMENTS

SYNTAX:

```
#!/bin/bash

# First form
if condition ; then
    commands
fi

# Second form
if condition ; then
    commands
else
    commands
fi

# Third form
if condition ; then
    commands
elif condition ; then
    commands
fi
```

EXAMPLE:

```
#!/bin/bash
x=10
y=5

# First form
if test $x -lt $y; then
    echo "x is less than y"
fi

# Second form
if test $x -lt $y ; then
    echo "x is less than y"
else
    echo "x is not less than y"
fi

# Third form
if test $x -lt $y ; then
    echo "x is less than y"
elif test $x -gt $y ; then
    echo "x is greater than y"
fi
```

SYNTAX for ***CONDITIONS***:

```
#!/bin/bash

# First form
test expression

# Second form
[ expression ]
```

SYNTAX for ***EXPRESSIONS***:

RELATIONAL OPERATORS:

```
#!/bin/bash

operand1=1
operand2=5

#using '[ expression ]'
[ operand1 <operator> operand2 ]

#using 'test expression'
test operand1 <operator> operand2
```

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.

BOOLEAN OPERATORS:

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.

SYNTAX for **CASE STATEMENTS**:

```
#!/bin/bash

echo -n "Enter a number between 1 and 3 inclusive > "
read character
case $character in
    1 ) echo "You entered one."
        ;;
    2 ) echo "You entered two."
        ;;
    3 ) echo "You entered three."
        ;;
    * ) echo "You did not enter a number"
        echo "between 1 and 3."
esac
```

```
echo -n "Type a digit or a letter > "
read character
case $character in
            # Check for letters
    [a-z] | [A-Z] ) echo "You typed the letter $character"
            ;;

            # Check for digits
    [0-9] )     echo "You typed the digit $character"
            ;;

            # Check for anything else
    * )         echo "You did not type a letter or a digit"
esac
```

NOTE: Patterns can be literal text or wildcards. You can have multiple patterns separated by the "**|**" character. Notice the special pattern "*****". This pattern will match anything, so it is used to catch cases that did not match previous patterns.

LOOPS

WHILE-LOOP:

```
#!/bin/bash

number=0
while [ $number -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

FOR-LOOP *(Example 1)*:

```
#!/bin/bash

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 numbe
      continue
   fi
   echo "Found odd number"
done
```

```
#!/bin/bash

number=0
while ((  number < 10 )); do
    echo "Number = $number"
    number=$((number + 1))
done
```

FOR-LOOP *(Example 2)*:

```
#!/bin/bash

count=0
for i in $(ls /bin); do
    count=$((count + 1))
    echo "Word $count ($i) contains $(echo -n $i | wc -c) characters"
done
```

FOR-LOOP *(Example 3)*:

```
#!/bin/bash

for ((  number=0; number < 10; number++ )); do
    echo "Number = $number"
done
```

ARRAYS

SOURCE:

```
#!/bin/bash
x=2

#declaring an array
declare -a Unix=('Debian' 'Red hat' 'Ubunu' 'Suse' 'Fedora')

#declaring an array
fruits=( "apple" "banana" "cherry" )

#accessing individual element
echo "Unix[1] = ${Unix[1]}"
echo "Unix[$x] = ${Unix[$x]}"

#print all elements
echo "Fruit = ${fruits[@]}"
echo "Unix = ${Unix[*]}"

#array length
echo "Fruit-length: ${#fruits[@]}"
echo "Fruit-indices: ${!fruits[@]}"
echo "${fruits[0]}-length: ${#fruits[0]}"
```

OUTPUT:

```
Unix[1] = Red hat
Unix[2] = Ubuntu
Fruits: apple banana cherry
Unix: Debian Red hat Ubuntu Suse Fedora
Fruit-length: 3
Fruit-indices: 0 1 2
apple-length: 5
```

OTHER ARRAY CONSTRUCTS:

`${array[*]}`	All items in the array
`${array[@]}`	
`${!array[*]}`	All indices in the array
`${#array[*]}`	Array length
`${#array[0]}`	Length of the first element

FUNCTIONS

As programs get longer and more complex, they become more difficult to design, code, and maintain. As with any large endeavor, it is often useful to break a single, large task into a number of smaller tasks.

We will begin to break our single monolithic script into a number of separate functions.

SYNTAX:

```
function-name() {
        command
        .
        .
        .
        command
        return value
}
```

SOURCE:

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World $1 $2"
   return 10
}

# Invoke your function
Hello Zara Ali

# Capture value returnd by last command
ret=$?

echo "Return value is $ret"
```

REFERENCES

[1] http://www.tutorialspoint.com/unix/unix-shell.htm

[2] http://linuxcommand.org/writing_shell_scripts.php#contents