LECTURE 2: SHELL SCRIPTS

2.0 INTRODUCTION ON SHELL SCRIPTS

2.0.1 What are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter.

2.0.2 What is Shell Scripting?

Shell scripting is writing a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script, which can be stored and executed anytime. This reduces the effort required by the end user.

2.0.3 What is Linux Shell Scripting used for?

Shell scripts allow us to program commands in chains and have the system execute them as a scripted event, just like batch files. They also allow for far more useful functions, such as command substitution. You can invoke a command, like date, and use it's output as part of a file-naming scheme.

2.0.4 Language is Shell Scripting?

Common scripting languages include: Shell scripts - sh, bash, **csh**, tcsh. Other scripting languages - TCL, Perl, Python.

2.0.5 Where do we use Shell Scripting?

A few examples of applications shell scripts can be used for include:

- Automating the code compiling process.
- Running a program or creating a program environment.
- Completing batch.
- Manipulating files.

- Linking existing programs together.
- Executing routine backups.
- Monitoring a system.

2.0.6 Is Shell Scripting a Programming Language?

A shell script is a computer program designed to be run by the Unix shell, a command-line interpreter. The various dialects of shell scripts are considered to be scripting languages.

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

2.1 WRITING A SCRIPT

A shell script is a file that contains ASCII text. A *text editor* is used to create a shell script. A text editor is a program, like a word processor, that reads and writes ASCII text files. There are many, many text editors available for your Linux system, both for the command line environment and the GUI environment. Some common text editors are shown in figure 2.1 below:

Name	Description	Interface
vi, vim	The granddaddy of Unix text editors, vi, is infamous for its difficult, non-intuitive command structure. On the bright side, vi is powerful, lightweight, and fast. Learning vi is a Unix rite of passage, since it is universally available on Unix-like systems. On most Linux distributions, an enhanced version of the traditional vi editor called vim is used.	command line
Emacs	The true giant in the world of text editors is Emacs by Richard Stallman. Emacs contains (or can be made to contain) every feature ever conceived for a text editor. It should be noted that vi and Emacs fans fight bitter religious wars over which is better.	command line
nano	nano is a free clone of the text editor supplied with the pine email program. nano is very easy to use but is very short on features. I recommend nano for first-time users who need a command line editor.	command line
gedit	gedit is the editor supplied with the Gnome desktop environment.	graphical
kwrite	kwrite is the "advanced editor" supplied with KDE. It has syntax highlighting, a helpful feature for programmers and script writers.	graphical

Figure 2.1: Text Editor

QUESTION ONE

Write “MTE 201 Script Writing” program.

Solution

```
#!/bin/bash
# First class writing script
echo "MTE 201 Script Writing"
```

2.1.1 Analysis on the Above Script

The first line of the script is important. This is a special clue, called a *shebang*, given to the shell indicating what program is used to interpret the script. In this case, it is “/bin/bash”. Other scripting languages such as ‘Perl’, ‘awk’, ‘tcl’, ‘Tk’ and ‘python’ also use this mechanism.

The second line is a comment. Everything that appears after a "#" symbol is ignored by **bash**. As your scripts become bigger and more complicated, comments become vital. They are

used by programmers to explain what is going on so that others can figure it out. The last line is the **echo** command. This command simply prints its arguments on the display.

2.1.2 Setting Permissions

The next thing we have to do is give the shell permission to execute your script. This is done with the **chmod** command as follows:

daniel@daniel-HP-Pavilion-g6-Notebook-PC:~$ chmod 755 First class writing script

The "755" will give you read, write and execute permission. Everybody else will get only read and execute permission.

Note: if you want your script to be private (i.e. only you can read and execute), '700' is used instead.

2.1.3 Putting It in Your Path

The script will run when the expression below is typed:

daniel@daniel-HP-Pavilion-g6-Notebook-PC:~$./First class writing script

When the script is run with the above command, "MTE 201 Script Writing" is displayed.

Note: When you type in the name of a command, the system does not search the entire computer to find where the program is located. The shell does know. Here's how: the shell maintains a list of directories where executable files (programs) are kept, and only searches the directories in that list. If it does not find the program after searching each directory in the list, it will issue the famous command not found error message.

This list of directories is called your **path**. You can view the list of directories with the following command:

daniel@daniel-HP-Pavilion-g6-Notebook-PC:~$ echo $PATH

This will return a colon separated list of directories that will be searched if a specific path name is not given when a command is attempted. To execute your new script, a pathname ("./") to the file is specified.

2.1.4 Directories

Directories to your path can be added with the following command, where directory is the name of the directory you want to add:

daniel@daniel-HP-Pavilion-g6-Notebook-PC:~$ export PATH=$PATH:directory

A better way would be to edit your .bash_profile or .profile file (depending on your distribution) to include the above command. That way, it would be done automatically every time you log in.

Most Linux distributions encourage a practice in which each user has a specific directory for the programs he/she personally uses. This directory is called bin and is a subdirectory of your home directory. If you do not already have one, create it with the following command:

daniel@daniel-HP-Pavilion-g6-Notebook-PC:~$ mkdir bin

Move your script into your new bin directory and you're all set. Now you just have to type:

daniel@daniel-HP-Pavilion-g6-Notebook-PC:~$ hello_world

and your script will run. On some distributions, most notably Ubuntu, you will need to open a new terminal session before your newly created bin directory will be recognized.

2.2 VARIABLES

Variables are areas of memory that can be used to store information and are referred to by a name. It could be described also as a character string to which a value is assigned. The value assigned could be a number, text, file name, device or any other type of data.

2.2.1 How to Create a Variable Name

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character "_".

2.2.2 Rule for Variable Name Creation

To choose the names for variables, these rules must be observed. They are:-

- Names must start with a letter
- Embedded space is not allowed. The use of underscore is recommended.
- Punctuation marks cannot be used.

By convention, Unix shell variables will have their names in UPPERCASE. Below are some examples:

- TT_A
- VAR_1
- VAR_100

2.2.3 Invalid variable names:

- 1_VAR
- _TT_A
- VAR_A!

Note: The reason why other characters such as !, * or – are not used is because they have special meanings for the shell.

To create a variable, a line in the script is put that contains the name of the variable followed immediately by an equal sign ("="). No spaces are allowed. After the equal sign, assign the information you wish to store.

Example:

variable_name = variable_value

VAR_1 = 21

The above example defines the variable VAR_1 and assigns the value "21" to it. Variables of this type are called **scalar variables**. Scalar variables hold only one value at a time.

2.2.4 Accessing Values

To access the stored value in the variable, the command below is typed.

Example one

```
#!/bin/sh
VAR_1 = 21
 echo $VAR_1
```

Result:

The value **21** will be produced.

Example two

```
#!/bin/sh
VAR_2 = "Test One"
 echo $VAR_2
```

Result:

Test One

2.2.5 Constants

As the name variable suggests, the content of a variable is subject to change. This means that it is expected that during the execution of your script, a variable may have its content modified by something you do.

On the other hand, there may be values that, once set, should never be changed. These are called constants. This is brought up because it is a common idea in programming. Most programming languages have special facilities to support values that are not allowed to change. Bash also has these facilities. Instead, if a value is intended to be a constant, it is given an uppercase name to remind the programmer that it should be considered a constant. It is worthy of note that environment variables are usually considered constants since they are rarely changed. Like constants, environment variables are given uppercase names by convention.

2.2.6 Read Only Variables

Shell provides a way to mark variables as **read-only** by using the read-only command. After a variable is marked read-only, its value cannot be changed.

Example

```
#!/bin/sh
VAR_2 = “Test One”
Readonly VAR_2
 VAR_2 = “ONE”
```

Result:

```
/bin/sh: VAR_2: This variable is read only.
```

2.2.7 Unsetting Variable

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. It should be noted that once a variable is unset, the stored value cannot be accessed. The following syntax to unset a defined variable is shown below.

unset variable_name

```
#!/bin/sh
VAR_2 = “Test One”
unset VAR_2
echo $VAR_2
```

The above example does not print anything. This shows that **unset** command cannot be used to unset **read-only** variables.

2.3 VARIABLE TYPES

When a shell is running, three main types of variables are present:-

Local Variable:-A local variable 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 the command prompt.

Environment Variables: the environment variable is available to any primary process of the shell. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.

At the start of a shell session, some variables are already set by the startup file. To see all the variables that are in your environment, use the **printenv command**. One variable in your environment contains the host name for your system.

Shell variables: this 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 local variables and others are environment variables.

2.4 UNIX: SPECIAL VARIABLES

Certain non-alphanumerical characters are reserved for specific functions. For this reason, their use as variable names should be avoided. The characters are:-

S/N	Variable and Description
1	$0 The filename of the current script
2	$n These variables correspond to the arguments with which a script was invoked. “n” here is a positive decimal number corresponding to an argument (the first argument is $1, the second argument is $2 and so on).
3	$# This shows the number of arguments supplied to a script
4	$* All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1, $2.

5	$@ All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1, $2.
6	$? This gives an exit status of the last command executed
7	$$ This is for the process number of the current shell. The process ID under which the shell script is executing is shown.
8	$! The process number of the last background command is shown

Note: $* and $@ will both act the same way unless they are enclosed in double quotes "". Both parameters specify the command-line arguments. While "$*" special parameter takes the entire list as one argument with spaces between, "$@" special parameter takes the entire list and separates it into separate arguments.