ITF22519: Introduction to Operating Systems

Fall Semester, 2022

Lab10: Shell Programming

Submission Deadline: November 1st, 2022 23:59

You need to get at least 50 points to pass this lab assignment

In this lab, you will do some practice with writing simple shell scripts using the **B**ourne **A**gain **SH**ell (**Bash**) and implementing some of Bash's `built-in` functions. Note that the **Exercises** of this lab assignment may use commands, syntax etc., which you learn in the lectures but not listed in this document.

Before you start, remember to `commit` and `push` your previous lab to your git repository. Then, try to `pull` the new lab to have all necessary materials:

```
$ cd OS2022/labs
$ git pull main main
$ cd lab10
```

1 A Quick and Simple Guide to Bash Scripting

The terms `shell` and `terminal` are often used interchangeably; however, they are in fact different things. The `terminal` provides an interface to type commands into a computer. A `shell` is a computer program which interprets and executed the commands. It is also an interface between the kernel and user and used to access services provided by the Operating System. Besides Bash shell, there are C shell, Korn shell, Bourne shell etc., Here is a 'Hello world' Bash script:

```
#! /bin/bash
#
echo Hello World
```

Make a file called *hello_world.sh* and run it either by typing the following command in your terminal:

```
$ bash hello_world.sh
```

or make the script executable, and run it as such:

```
$ chmod +x hello_world.sh
$ ./hello_world.sh
```

The `chmod` command in Linux is used to change the access mode of the file, in this case is file *hello_world.sh*. The `x` means *executable* mode of *hello_world.sh* (you will learn more about different access modes of a file in the last lab). The command `chmod +x` means "add executable mode to the file *hello_world.sh*". With this command, you have given the *executable* permission to the file *hello_world.sh*. If you do not use this command, you may get a "Permission denied " error message. Note that this command should be used only once; there is no harm to used more than once but it is not necessary. The next command of the script `./hello_world.sh` is to run the script. Note that `./` is to indicate the **PATH** which is current path of the file.

2 Script file format

The followings are some explanations of what each line in the file ***hello_world.sh*** does and how this can be extended.

2.1 The first line

The first line of a script file tells what type of file it is and which program should interpret it. For example, shell scripts that start with

```
#!/bin/bash
```

or

```
#!/bin/sh
```

are shell scripts meant to be interpreted by **Bash** and **SH** respectively. Other scripts can include #!*/bin/python*, #!*/bin/perl*, etc. When a script is executed on the command line, the shell will search for the correct interpreter to start by using this first line of the script. Therefore, with the above example, calling the following command

```
$ ./hello_world.sh
```

is interpreted as

```
$ /bin/bash hello_world.sh
```

2.2 Comments

Any line starting with a "#" notation and not followed by an "!" is considered a comment line and ignored by Bash. Comments can appear anywhere in the file. Note that most interpreters will not accept partial line comments as follow:

```
#!/bin/bash
echo Hello world # print Hello world
```

Instead the correct way to write this for maximum portability would be

```
#!/bin/bash
# print Hello world
echo Hello world
```

2.3 Commands

A command is anything the script is to execute. Script commands are identical to the commands you type on a command line in your terminal. For example the following set of commands:

```
$ git pull origin
```

could be replaced with a single shell script, which will be called ***pull_git.sh***:

```
#!/bin/bash
git pull origin
```

and then executed using the signal command ***pull_git.sh***. (Remember to setup access mode before running). Another example of a command would be the line `echo Hello World` in the script file ***hello_world.sh*** created earlier.

2.3.1 Built-in Commands

So far, commands have been treated as something that will work for every interpreter. This may not be true for all commands that are in the scripts. The set of commands that may not always work for every interpreter are known as **built-in** commands or commands that are built-in to a given interpreter. These built-in commands are potentially different for different interpreters. To check if a command is a built-in or not, simply type into Bash:

```
$ type commandToCheck
```

For example:

```
$ type cd
$ type ls
```

The command `type commandToCheck` will return the type of the command in `commandToCheck`. If a command is a built-in command, `type` will return “built-in”. If a command is an executable, type will return “hashed” etc.,.

2.4 Variables

Variables could be created and initialized with the = sign. To access the variable, prefix its name with a $ sign. Here is a ’Hello World’ example with variable:

```
#!/bin/sh
MY_MESSAGE="Hello World"
echo  $MY_MESSAGE
```

The Shell does not care about types of variables.

```
#!/bin/sh
X=1
echo "X = $X"
X=$((X+1))
echo "X = $X"
```

Another example which reads user name from the standard input (with `read` command) and create a file named `username_file` (with `touch` command). Notice the *`curly brackets`* around the variable:

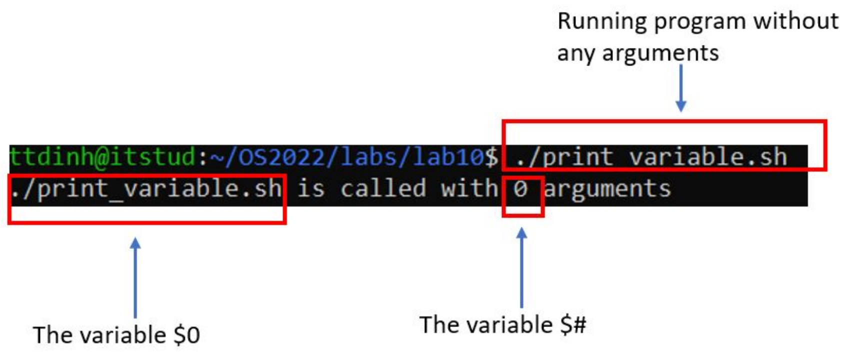
```
#!/bin/sh
echo  "What is your user name?"
read USER_NAME
echo  "Hello $USER_NAME"
echo  "I will create you a file called ${USER_NAME}_file"
touch "${USER_NAME}_file"
```

2.5 Command Line Arguments

Command line arguments can be passed to a shell script just like any C program. The number of command line arguments passed is stored in a variable named $# and each argument is stored in $1, $2,. . . , $n variables. By default, the variable $0 is the name of the program by convention. Here is an example: put following scripts inside a file named *`print_variable.sh`*.

```
#! /bin/bash
echo "$0 was called with $# arguments"
```

The following is the output of the above program:



```
ttdinh@itstud:~/OS2022/labs/lab10$ ./print_variable.sh
./print_variable.sh is called with 0 arguments
```

Figure 1: An illustration of command li ne arguments in Bash shell.

2.6 Conditions

The syntax for condition is:

```
if [ ... ]; then
       # if-code
else
       # else-code
fi
```

An example of multiple if else conditions is shown below. Notice the spaces around the square brackets as well as the quote around the variables:

```
#!/bin/sh
echo "Enter a number"
read X
if [ "$X" -lt "10" ]; then
    echo "The number $X is less than 10"
else
    if [ "$X" -eq "10" ]; then
         echo "The number $X is equal to 10"
    else
         echo "The number $X is greater than 10"
    fi
fi
```

You can find a list of useful comparisons on this link: shell.

2.7 Further Information

There are far more capabilities to Bash scripting than discussed here. Examples include conditional if statements, loops, and math. Excellent resources to learn more are:

- `https://www.shellscript.sh/`
- `http://linuxcommand.org/lc3_writing_shell_scripts.php`
- `https://www.shellscript.sh/quickref.html`

3 How to Set and Use Environment Variables

When a shell is started, it has to keep track of a lot of settings for resource access and properties. How it keeps track of all of this is through what is called an **environment**. This is a list of variables that hold all sorts of information for the correct execution of the shell. An interesting property of the environment is that any child shell or process of the shell will inherit the variables when started from the parent shell. To list all environment variables that the shell has access to, the following command is used:

```
$ printenv
```

For better readability use:

```
$ printenv | less
```

The output should show some familiar variables, such as `PATH`, `SHELL`, and `HOME`. In the event that printing out the environment variables are insufficiently interesting, creation of personalized environment variables can be performed as well. To create a new environment variable, use the following command:

```
$ export VAR_TEST=valueForVar
```

This command will place `VAR_TEST` in the list of environment variables with the value of `valueForVar`. Note that `valueForVar` will be interpreted as a string, regardless of what its value is. To use a variable (or more specifically, expand it), place a $ before the variable name, as such:

```
$ echo $PATH
```

This would expand the `PATH` variable. An example that some students may be familiar with in this regard is appending a directory to the `PATH`:

```
$ export PATH=$PATH:path/to/new/executable/directory
```

This example would add the path `path/to/new/executable/directory`to the `PATH` variable, allowing the user to access the new executable installed in the directory without typing out the full path to it.

Task 1

Make the program `Bash_example.sh` with the content below. Run the script and explain the output.

```
# This is a simple Bash script
# The first line tells what type of file it is
#! /bin/bash

echo Doing something cool
sleep 2

# Changing directory
echo Changing directory
sleep 2
cd ../
# Print working directory
echo Printing working directory:
sleep 2
pwd

# Doing something fun
```

```
echo Updating the access and modification times of each FILE
sleep 2
echo A file argument that does not exist is created empty
sleep 2
touch a b c d e

# Doing something fun. Well, this is comment
echo Creating a new environment variable
sleep 2
export HELLO="Hallo!"

# Printing the environment variable
echo Printing the environment variable
sleep 2
echo $HELLO

# Deleting file system
echo Deleting file system...
sleep 2
rm a b c d e
echo Done!
```

4 Shellshock (Bash bug)

It is estimated that the Shellshock bug has gone undiscovered for nearly 26 years. There are some good explanations of the bug online. For example: Shellshock Code and the Bash Bug - Computerphile. Shellshock Code and the Bash Bug - Computerphile The impact of the Shellshock bug was originally under estimated. The impact of the bug became much more clear as many researchers realized that several programs (including popular web servers such as Apache) make heavy use of environment variables. `http://www.securitysift.com/shellshock-targeting-non-cgi-php/`.

Task 2

What is an environment variable and how could it be used in conjunction with the Shellshock bug to remotely exploit a web server?

5 Exercises

5.1 Exercise 1 (50 pts)

Create a script file ***push.sh*** to push a file into your `lab10` repository in your GitHub.

5.2 Exercise 2 (50 pts)

Create a script file ***copy.sh*** that does the following:

- The script takes ***source_file*** and ***dest_file*** as two input parameters. Then, the command should be: `copy.sh <source_file> <dest_file>`.
- If the number of input parameters is not two, the script should print out usage message and exit.

- If the `source_file` exists, copy it to the `dest_file`, and print out the number of lines in this file.
- If the `source_file` does not exist, print out the error message and exit.

Expected output of the script is showed below:

```
$ ./copy.sh test.txt
Usage: ./copy.sh <source_file> <dest_file>

$ ./copy.sh random_file1 random_file2
The file random_file1 does not exist

$ ./copy.sh test.txt test2.txt
Copying the file test.txt to test2.txt
The file test.txt has 8 lines
```

Hints: To get number of the lines of a file, use `wc -l` together with `awk`:

```
wc -l <file\_name> | awk '{ print $1 }'
```

6 What To Submit

Complete this lab and put your files into the lab10 directory of your repository. Run git `add .` and `git status` to ensure the files have been added and commit the changes by running `git commit -m Commit Message`. Finally, submit your files to GitHub by running `git push`. Check the GitHub website to make sure all files have been submitted.