ITF22519: Introduction to Operating Systems

Fall Semester, 2021

Lab10: Shell Programming

Submission Deadline: November 4th, 2021 23:59

In this lab, you will do some practice with writing simple shell scripts using the Bourne Again SHell (Bash) and implementing some of Bash's built-in functions. 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 Introduction2OS/labs
$ git pull upstream 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 the computer. A shell such as Bash is a program which interprets and executed the commands. Here is a 'Hello world' Bash script:

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

Saving this code into a file called ***hello_world.sh*** and run it either by typing the command:

```
$ bash hello_world.sh
```

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

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

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 should tell 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 using this first line. Therefore with the above example, calling

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

is interpreted as

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

2.2 Comments

Any line starting with a # 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 typing commands on the command line. For example the following set of commands:

```
$ git pull upstream main
$ git status
```

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

```
#! /bin/bash
git pull upstream main
git status
```

and then executed using the signal command ***update_git.sh***. Another example of a command would be the line echo "Hello World" in the script file ***hello_world.sh*** created earlier.

Though it is not very important to the use of the Bash shell, this information will be useful for portability purposes of the script and for the implementation of the task for the lab. 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 the interpreter. These built-in commands are potentially different for different interpreters. The other set of commands are aliases, functions, executables, and keywords of which we are mainly interested in executables. To check if a command is a built-in or not, simply type into Bash:

```
$ type commandToCheck
```

This 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'. Use the `type` command to check the type of `cd` and `ls` commands:

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

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. The variable $0 is the name of the program by convention. Here is an example: put following scripts inside *print_variable.sh* and check the output:

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

2.6 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. 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 (100 pts)

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

- (**25 pts**) The script takes *source_file* and *dest_file* as two input parameters. Then, the command should be: `copy.sh <source_file> <dest_file>`.
- (**25 pts**) If the number of input parameters is not two, the script should print out usage message and exit.
- (**25 pts**) If the `source_file` exists, copy it to the `dest_file`, and print out the number of lines in this file.

- (**25 pts**) 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.