CS2106 Introduction to Operating Systems
Shell Scripting and Process Programming
(Modified from previous semesters)

Introduction

A shell is a program that runs on top of the operating system, and its main task is to interface the user with the operating system (OS). The familiar graphical environment in Windows and MacOS are both examples of graphical shells; they allow a user to interact with the OS using graphical elements like menu bars, context menus, text boxes, clickable links and images, etc.

Bash – Bourne Again Shell – is an example of a command-line shell, just like zsh in MacOS. Users interact with the OS by typing in commands.

Bash Scripting

Bash scripting is an essential skill for anyone who works on servers running unix operating systems like Linux. It can automate many tasks. For example, you can write a Bash script to automatically compile your code, run unit tests if the compilation succeeds, and then push the code to a Github repository if the unit tests pass, while capturing the outputs of each stage to a file for later review.
This section introduces just the basics of shell scripting. For more details please see https://devhints.io/bash

Before we start building a more interesting script, let's go through some basics.

i. A simple example of Hello World Script

Following is simple hello world shell script "hello.sh":

```
#!/bin/bash          # which shell to use to execute the code
echo "Hello world!" # Echo is similar to printf in C.
```

To execute this file, you need to convert it to an executable file by executing the following command:

```
chmod a+x ./hello.sh
```

This command sets the "executable" flag on hello.sh for "all" using the parameter "a+x", thus everyone can execute this file. To execute:

```
./hello.sh
```

Note: echo does not normally process '\n' or other slash escape sequences. To process escape sequences, specify the "-e" option when calling echo. E.g.

```
echo -e "\nHello world.\n"
```

ii. Variables

Variables are very useful in Bash scripts, and can be used to store strings, numeric values, and even the outputs of programs. For example:

```
#!/bin/bash
x=15
y=20
let z=$x-$y      # Method 1
sum=$(($x+$y))   # Method 2
echo "$x - $y = $z"
echo "$x + $y = $sum"
```

NOTE: In your assignment statements (e.g., x=5), it is VERY IMPORTANT that there is NO SPACE between the variable, the '=' and the value. Likewise in the line z=$x-$y, it is VERY IMPORTANT that there are NO SPACES in the statement. If you have spaces, you will get an error message like "x: command not found".

Notice some things about the script above:

(a) You assign to a variable using =, which is expected. However as mentioned earlier, there must not be any spaces in your assignment statement. For example, x=5 is correct, but x = 5 is wrong and will result in an error like "x: command not found"
(b) Use $<var name> to access the value stored in <var name>. For example, we used $x and $y to access the values stored in x and y.
(c) Notice that we can similarly access the values of the variables in the echo statement by using $.

iii. Test Statements

Bash can test for certain conditions using the [[.]] operator. Some things you can do:

Test	Result
`[[ -z <string> ]]`	Tests if <string> is empty (i.e. "").
`[[ -n <string> ]]`	Tests if <string> is not empty.
`[[ <string1> == <string2> ]]`	Tests if <string1> is equal to <string 2>
`[[ <string1> != <string 2> ]]`	Tests if <string1> is not equal to <string 2>
`[[ num1 -eq num2 ]]`	(Numeric) Tests if num1 == num2
`[[ num1 -ne num2 ]]`	(Numeric) Tests if num1 != num2
`[[ num1 -lt num2 ]]`	(Numeric) Tests if num1 < num2
`[[ num1 -le num2 ]]`	(Numeric) Tests if num1 <= num2
`[[ num1 -gt num2 ]]`	(Numeric) Tests if num1 > num2

`[[ num1 -ge num2 ]]`	(Numeric) Tests if num1 >= num2
`[[ -e FILE ]]`	Tests if file exists
`[[ -d FILE ]]`	Tests if file is a directory
`[[ -s FILE ]]`	Tests if file size > 0
`[[ -x FILE ]]`	Tests if file is executable
`[[ FILE1 -nt FILE2 ]]`	Tests if FILE1 is new than FILE2
`[[ FILE1 -ot FILE2 ]]`	Tests if FILE1 is older than FILE2
`[[ FILE1 -ef FILE2 ]]`	Tests if FILE1 is the same as FILE2

Note the spaces after [[and before]]. They ARE important. Example:

```
#!/bin/bash

echo "Enter the first number: "
read NUM1
echo "Enter the second number: "
read NUM2

if [[ NUM1 -eq NUM2 ]];
then
    echo "$NUM1 = $NUM2"
elif [[ NUM1 -gt NUM2 ]];
then
    echo "$NUM1 > $NUM2"
else
    echo "$NUM1 < $NUM2"
fi
```

Some things to note:

- You can read from the keyboard using "read". The syntax is "read <varname>", where <varname> is the variable that we want to store the read data to.
- The "if" statement has an odd syntax; you need a semi-colon after the [[..]]:

```
if [[ NUM1 -eq NUM2 ]];
then
      echo "$NUM1 = $NUM2"
else
      echo "$NUM1 != NUM2"
fi
```

iv. Loops

Bash supports both for-loops and while-loops. The for-loop is similar to Python's. For example, to list all the files in the /etc directory, we could do:

```
for i in /etc/*;
```

```
do
      echo $i
done
```

You can also iterate over a range:

```
for i in {1..5};
do
      echo $i
done
```

The while loop works pretty much the way you'd expect it to. For example:

```
i=0
while [[ $i -le 5 ]];
do
      echo $i
      let i=i+1
done
```

v. Miscellaneous Topics

vi.

Let's look at a few additional features in Bash, which you may find useful.

a) Redirecting Output

You can redirect output to the screen (stdout) to a file. For example, if you wanted to capture the output of "ls" to a file:

```
ls > ls.out
```

Note that you can *append* to an output file by using >> instead of >. For example:

```
ls -l >> ls.out
```

Would append output to ls.out without overwriting previous content.

b) Redirecting Input

You can also redirect input from the keyboard to a file. Suppose you have a file called "talk.c" that take in an input file. You can use the following command to redirect input file to it:

```
gcc talk.c -o talk
./talk < file.txt
```

c) Pipes

Assuming one program prints to the screen and other reads from the keyboard, you can channel the output of the first program to the input of the second using a mechanism called a “pipe”. Following command list all the files in the current directory (`ls`), which then is piped as input to the `wc` (word count) command. This command then counts the number of files present.

```
ls | wc -l
```