SoC_Week2

A Unix Journey: Ctrl+Alt+Delight

May 2024

1 Der Mat Kar Padh Le Time se

In the assignment for last week, we introduced ways to write `for` loops in command, but we also mentioned that it's part of scripting. Well, here we are in the next phase of the project, and this week, we will be learning bash(or equivalent) scripting.
So far, the familiar command-line shell was taking a single command, and we have very little room to run multiple commands using pipelines, `&&`, and `&`, etc. We need to write multiple commands to automate a task, like in last week's assignment. We tried blending the scripting techniques, and it was really messy to write it. Now, you can guess the work of scripting, right? If we see, carefully, that in writing `for` loop, we used "`;`," which indeed represents a command terminator(equivalent to breaking the line because, in scripting, each line typically represents a command) in scripting. So, we were writing a script, nothing like a single command, similarly using "`;`", we can convert a script into this form, but why do it clumsy way? Note that as such, `for` is not a command; rather, it's just a control structure that shell uses to manage the execution flow, much like what is done by `&&`. In contrast, a command is something that interacts with OS to perform some task, and in that sense `for` is very far away.

With that in mind, one can think of a script, which is just some set of commands, that can be written as one line in the terminal using "`;`" but it uses a new line to separate the commands, making it more straightforward to write and understand. And by now, because you will find this a straightforward topic to understand, always keep in mind that everything that will be used is just writing commands separated by "`;`". For the rest of the material, we will first introduce the notation of variables, conditional statements, loops(more than a `for`), arrays, functions, etc.

2 Basic

A Script, as said, is a **sequence** of commands. We write this sequence of commands in a file with the extension "`.sh`". For comments, we can use "`#`." And you run the "`script.sh`" using

```
$ bash script.sh
```

And bash shell works as an interpreter that runs through line by line(because it's a sequence of commands).

So far, so good, but the scripting isn't restricted here, although we will only discuss this part. Given an interpreter, we can have a script for it. How? In the above case, to run the "`script.sh`" we just mentioned the interpreter to be used that was bash, but this can be done more broadly. First, we need to make the shell or terminal(which tells the OS) aware of which interpreter to use for the script without mentioning it explicitly, and this is achieved by mentioning the location of an interpreter in the first line of the script, using what is called a *shebang* or *hashbang* denoted by "`#!`", like in the above script.sh, we will add a newline at the top as "`#!/bin/bash`"; this will tell the shell needs to pick the bash interpreter for this script. And now, to run the script, we need to give the current user the execute permission, which can be done using

```
$ chmod +x script.sh
```

then we can run the script.sh using

```
$ ./script.sh
```

Note if we mention the interpreter using *shebang* in the script, we don't need to mention it again while running it. Similarly, we do it for Python by adding this as a first line into the script.py(the extension can be .sh, but to make it clear, we write .py, but not required; we mean that once you have used *shebang*, then there is no specific need of particular extension)

```
#!/usr/bin/env python
```

then run the following command to run the script.py

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

More on Shebang: When we run a script, control passes to OS, and then the operating system reads the first line of the script to determine which interpreter to use. If the first line starts with `#!`, the operating system executes the script and uses the interpreter specified after the `#!`. The shebang is not ignored by the operating system when executing a script. However, it is ignored by the interpreter when reading and executing the script(because a comment starts with "`#`"). This is because the operating system only uses the shebang to determine which interpreter to use and is not part of the script's instructions.

3 Tools In Bash Scripting

There is too much discussion on the scripting; let's start writing scripts in bash. Well, this reference will first introduce the different syntax in scripting and then be followed by a few examples, so lessgo :)

3.1 Variables

In bash, variables don't need any declaration, mainly because every variable is treated as a character string. But, depending on the context, it can behave like an integer, for example, in an arithmetic

expression.
If you want to define a variable, say, "`demo_var`" with the value "`isthisavariable`", then you can write

```
demo_var=isthisavariable
```

As last week's assignment mentioned, you can use `$demo_var` to access the variable's value(try printing using echo).
Note that there should be before and after "`=`"; otherwise, your shell might interpret them as a command(you will see the reason when we will talk about functions). The next thing to note is that variable names can include alphabets, digits, and underscores; it can be started with alphabets and underscores only; it is case sensitive.

There is a need for something called *word splitting* and *path expansion*; this was also the reason for making the warm-up question for the last week's assignment.

3.2 Word splitting and Path expansion:

So far, you might've noted this behavior (at least from the assignments). Word splitting refers to splitting a string by shell into different words, separated by space, tab, and newline. For if we try to create a file with the name "`This is a name`" using

```
$ touch This is a name
```

Then what you get is four file with names "This", "is", "a" and "name", and in this cases we either use a double quotes or single quotes(more on these later) or escape using backslash each of the space. This process is typically applied to the results of variable expansions, command substitutions(well, this is nothing but `$(cmd)` thing that we were doing an assignment, output of a command replaces the command itself), and other forms of parameter expansion.

Path expansion, more or less like using regex in the path name, but this applies to all path names. As mentioned in an earlier reference, `ls` command optionally takes a path as an argument, then using path expansion, the shell is able to expand the path name to path names and give you the result. For example you want to list all files in the current directory with the ending "`.txt`", then you can use:

```
$ ls *.txt
```

Formally, we can say that path expansion is a feature in Bash (and other Unix-like shells) that allows for the automatic expansion of patterns in filenames and directory names. This feature uses wildcard characters(some of the special characters used in regex) to match multiple files or directories based on specified patterns.

3.3 Quoting

Whenever we're using a special character with no intention to use it for its special purpose, we need to tell shell, which can be achieved using backslash to escape that character. But there is more easy way to do it because it might be tedious to escape every character in a long name or value of a variable(you can have a declaration of variable like "`x=This is`", you must write "`x=This\is`").

Single Quotes This is very useful while naming anything, like assigning a value to a variable or a name to a file. Single quotes (') preserve the literal value of each character. For example

```
$ s=hello; echo '$s'
```

will output "`$s`" instead of "`hello`", because the "`$`" is no more special character used to access the value of the variable or command substitution. This is why we only use this for naming and all, and it is preferred to use double quotes in general.

Double Quotes Double quotes ("") preserve the literal value of all characters with the exception of `$, ‘` and backslash. Hence, we can still expand the variables and do command substitution (which is nothing but the `$(cmd)`, as to use the output of a command in the arguments of another).

For example When you use `echo $var` without double quotes, the value of the variable `var` is subjected to word splitting; it will be split into multiple words, and each word will be treated as a separate argument to the echo command. When you use `echo "$var"`, the value of the variable `var` is enclosed in double quotes. This means that the value is treated as a single word, even if it contains spaces or other characters that would normally trigger word splitting or path name expansion. E.g.:

```
    var="x   y"
    echo $var
    echo "$var"
output:
    x y
    x   y
```

3.4 Arithmetic

Well, there is no great tool for arithmetic like other languages, but there are some supports. As said earlier, here, variables are treated as strings, so if you do:

```
x=3
y=4
echo "$x+$y"
```

Output will be "`3+4`" not "`7`". We evaluate an arithmetic expression; we have two options; the first one uses a built-in command in Linux systems, "`let`", and can be used for evaluating arithmetic expressions indirectly like:

```
x=3
y=4
let z=x+y
echo $z
```

Or

```
let "x=3" "y=4" "x=x+y"
echo $z
```

But mostly, we prefer the other option(the former option is used only for statements like `x++` in cpp by using "`let x=x+1`"). The other way is very natural; just like expanding a variable, to expand an expression, we can use

```
$((expression))
```

Or

```
$[expression]
```

Other ways are to use "`declare -i`" and "`bc`"(can used for decimal values). E.g:

- 1. The syntax for "`bc`" is through the pipeline, and it is used for decimal things like:

  ```
  n=3/4
  echo "$n" | bc -l
  ```

  where -l is for decimal representation, and it is if used without -l then integer manipulation occurs.

- Here is the bunch of E.g.:

  ```
  echo $((5*2+1))
  echo $[5*2+1]
  let num=5*2+1
  echo $num
  declare -i result
  result=5*2+1
  echo $result
  ```

  The output will same for each line and its `11`.

3.5 Conditionals and Loops

Before we begin, we first need to understand that every command we run in the shell returns a value, 0 when there is no error, and 1 or -1 usually or some other numbers if there is an error. Now, we can discuss the different syntax for conditionals and loops.

- **if-then-else :** The general syntax for conditional is:

```
if cmd1
then
sequence_of_commands
elif cmd2
sequence_of_commands
else
sequence_of_commands
fi
```

Where the condition for "`if`" is checked using the returned value of "`cmd1`" command, similarly for "`elif`" and the scope of "`if`" ends at "`fi`." But the question comes: how do we write conditions like $x \geq 4$? Well, let's first try this :

```
x=1
if $x
then
echo "yes"
fi
```

Then you should expect output as "`yes`", but you will get "`1:  command not found`"; this is because of syntax; anything after the "`if`" must be a command. And we have a command that does what we require now:

```
[ $x -eq 1 ]
```

First, note that you must maintain the space from "`[`" and "`]`" and "`-eq`" means "`==`". The second thing is that the return value of this command is 0 when the condition is true, but "`if`" is executed because a return value 0 is treated as true here. And third is that "`[ $x -eq 1 ]`" is shorthand for the command "`test`". Similarly, suppose you want to check if a file named "thisfile" exists in the current directory or not:

```
test -f "thisfile"
```

Or

```
[ -f thisfile ]
```

Or

```
if test -f "thisfile"
then
echo "Yes"
else
echo "NO"
fi
```

Or

```
if [ -f "thisfile" ]
then
echo "Yes"
else
echo "NO"
fi
```

This work can also be achieved normally using "`&&`" and "`||`", using the fact that:

```
$ cmd1 && cmd2
```

Then `cmd2` is executed only if `cmd1` doesn't fails.

```
$ cmd1 || cmd2
```

Then `cmd2` is executed only if `cmd1` fails. Putting together them, we have:

```
$ [ -f "thisfile" ] && echo "Yes" || echo "NO"
```

Now use the referenced link to get more comparison and options of `test` command.

- **Loops** There are three loops: for,while, and until(opposite to while, where the block is executed till the condition is false). We mention their syntax:

```
for item in list
do
sequence_of_commands
done
```

This is the familiar loop used in the assignments.

```
for (( initialization; condition; increment ))
do
sequence_of_commands
done
```

This is the cpp style for loop, but we use the less here is an eg:

```
for ((i=0; i<5; i++))
do
    echo "$i"
done
```

This will print numbers from 0 to 4. Next, we consider the syntax of `while` and **until** loops. Note that here, we again, by condition, mean a command's return value.

```
while cmd1
do
sequence_of_commands
done
```

And

```
until cmd1
do
sequence_of_commands
done
```

consider an eg to output numbers from 9 to 0:

```
X=10
while [ $X -gt 0 ]
do
let X=X-1
echo $X
done
```

Before we move ahead, we would like to point out the limitations of `test` command; one of them is we can't use regex in comparison to strings like we can't do "$x == at*$"(in regex context). For this, we have a more preferred option; although we were bound to use "`[`" to introduce the `test` command, we can use "`[[`" instead. E.g.:

```
x=\abc"
if [[ $x == a* ]];then
echo "it matches"
fi
```

Here are some differences between them:

- Pattern matching: `[[` allows for more powerful pattern matching compared to `[`. When using `[[`, you can use the `==` and `!=` operators to match a string against a pattern. This is not possible with `[`.
- Logical expressions: `[[` allows for more powerful logical expressions compared to `[`. With `[[`, you can use the `&&` and || operators to combine multiple expressions. This is not possible with `[`.
- Word splitting and path name expansion: When using `[`, word splitting and path name expansion are performed on the arguments. This can lead to unexpected behavior if the arguments contain spaces or special characters. With `[[`, word splitting and path name expansion are not performed, which makes it safer to use with arbitrary arguments.

Now, let's move to the next topic.

3.6 Command Line Argument

What do you need more? Uhmm! perhaps having a way to pass an argument to the script will be good. This can also be achieved easily if you think "`./script.sh`" as a command, just like other commands, to pass the arguments to the script, say "`hello`", "2" and "23", you can do:

```
$ ./script.sh "hello" 2 23
```

And that's it, but wait a minute, how do I access the argument in the script? Well, for that, see this list:

```
$0 Name of the current shell script
$1-$9 Positional parameters 1 through 9
$# The number of positional parameters
$* All positional parameters, \$*" is one string
$@ All positional parameters, \$@" is a set of strings
$? Return status of the most recently executed command
$$ Process ID of the current process
```

It's easy to get handy with them, but we emphasize on the difference between "`$*`" and "`$@`" with this eg:

```
echo "Using \$*:"
for arg in "$*"
do
echo "[$arg]"
done
```

```
echo "Using \$@:"
for arg in "$@"; do
echo "[$arg]"
done
```

Try running the above script with some arguments and see the difference.

3.7 Arrays

Firstly, different shells might have different indexing of the arrays. To declare an array, you can use:

```
array=("enty1" "entry2" "entry3")
```

Then you access a particular index, say 1^{st}:

```
${array[1]}
```

You can access all the elements of the array as:

```
${array[@]}
```

You append or add a new entry using:

```
array[3]+="entry4"
```

Try adding a new entry to an index that already exists :). You can delete an index using:

```
unset array[3]
```

Optionally, you can look for `declare` command to declare arrays.

3.8 Functions

In bash, functions are treated or considered commands, taking and accessing arguments similar to how command line arguments pass script or any command. Here, the function also returns like a command. Here is the general syntax:

```
demo_func() {
sequence_of_commands
}
```

And suppose you want to call the function we some argument:

```
if demo_funt "These" "are"  "argument"
then
do
echo "Yes"
done
```

You can access the argument in the function much like the command-in-line argument to the script(that using `$*,$1,...`).

3.8.1 Local, Global variables

Normally, in another language, if you declare a variable inside a function, then it is local. Still, in bash, every variable works as global, and you must mention the local ones explicitly. E.g.:

```
demo_func() {
x=2
local y=3
}
demo_func
echo $x
echo $y
```

Here, 2 will be printed as x is a global variable and y is a local variable. Note that you need to call "`demo_func`"; otherwise, the sequence of commands inside that function will not get executed.

3.9 Environment Variables

Environment variables are kind of global variable throughout a shell sessions, in inherited by any child shells or processes and hence they are part of the environment in which a process runs and can be used to pass configuration information to applications and scripts. You can make a variable a environment variable by using "`export`" command.

```
export var="value"
```

You can use the command "`env`" to show the list of environment variables.

Some of variables in your system that describe your environment:

- `SHELL`: what shell you're running
- `USER`: username of the current user
- `PWD`: present working directory
- `PATH`: specifies the directories to be searched to find a command

You can make or change a permanent environment variable, by editing the file `.bashrc` or `.bash_profile`, `.bashrc` commonly used to set environment variables, aliases(they are shortcuts that you can create), and shell options that should be available in all interactive shells, `.bash_profile` used to set environment variables and other settings that should be initialized once at login. Here is the a video to change these files to customize your shell(it's for zsh, but for bash it is similar).

You can explore more on scripting on your own, since this was just an introductory reference. Check this