Département Informatique
R1.04 – TP Admin System
Responsables : X.Roirand, F. Lesueur, N. Le Sommer, N. Delomez
Durée : 180mn machine

Le but de ce TP va être de découvrir comment créer des scripts sous Linux. Le but des scripts en administration système est de regrouper des commandes en une seule, de facilement modifier le comportement d'un groupe de commandes en rajoutant des options, de facilement effecter un groupe d'actions en une seule, et d'afficher lisiblement des informations permettant de savoir si un script s'est bien passé ou pas.

Prénom Nom :..... Date :.....
Groupe :.....

Ce TP6 a un énoncé en anglais, vous pouvez y répondre en anglais ou en francais. Tout ce qui sera sur fond bleu signifiera qu'il faut répondre à une question. Le reste est de l'information ou des élements qui vont vous permettre de répondre aux questions suivantes. Il y a beaucoup d'informations à lire, je vous encourage à les lire entièrement pour vous faciliter ensuite le TP.

Bash Introduction

Bash Shell Definition

Bash

Bash is a command language interpreter. It is widely available on various operating systems and is a default command interpreter on most GNU/Linux systems. The name is an acronym for the '**B**ourne-**A**gain **SH**ell'.

Shell

Shell is a macro processor which allows for an interactive or non-interactive command execution.

Scripting

Scripting allows for an automatic commands execution that would otherwise be executed interactively one-by-one.

Bash Shell Script Basics

Do not despair if you have not understood any of the above **Bash Shell Scripting** definitions. It is perfectly normal, in fact, this is precisely why you are reading this Bash Scripting tutorial.

In case you did not know, Bash Scripting is a must skill for any Linux system administration job even though it may not be implicitly requested by the employer.

What is Shell

Most likely, you are at the moment sitting in front of your computer, have a terminal window opened and wondering : "What should I do with this thing?"

Well, the terminal window in front of you contains shell, and shell allows you by use of commands to interact with your computer, hence retrieve or store data, process information and various other simple or even extremely complex tasks.

Try it now! Use your keyboard and type some commands such as `date`, `cal`, `pwd` or `ls` followed by the `ENTER` key.

Q1) Do a copy / paste ou a screenshot of the entered commands and associated outputs.

What you have just done, was that by use of commands and shell you interacted with your computer to retrieve a current date and time (`date`), looked up a calendar (`cal`), checked the location of your current working directory (`pwd`) and retrieved a list of all files and directories located within (`ls`).

What is Scripting

Now, imagine that the execution of all the above commands is your daily task. Every day you are required to execute all of the above commands

without fail as well as store the observed information. Soon enough this will become an extremely tedious task destined for failure. Thus the obvious notion is to think of some way to execute all given commands together. This is where scripting becomes your salvation.

To see what is meant by scripting, use shell in combination with your favorite text editor eg. vi or nano to create a new file called `task.sh` containing all the above commands :

date

cal

pwd

ls

Each on a separate line. Once ready, make your new file executable using `chmod` command with an option `+x`. Lastly, execute your new script by prefixing its name with `./`.

As you can see, by use of scripting, any shell interaction can be automated and scripted. Furthermore, it is now possible to automatically execute our new shell script `task.sh` daily at any given time by use of cron time-based job scheduler and store the script's output to a file every time it is executed. However, this is a tale for an another day, for now let's just concentrate on a task ahead.

What is Bash

So far we have covered shell and scripting. What about Bash? Where does the bash fit in? As already mentioned, the bash is a default interpreter on many GNU/Linux systems, thus we have been using it even without realising. This is why our previous shell script works even without us defining bash as an interpreter. To see what is your default interpreter execute command `echo $SHELL`:

```
$ echo $SHELL


/bin/bash
```

There are various other shell interpreters available, such as Korn shell, C shell and more. From this reason, it is a good practice to define the shell interpreter to be used explicitly to interpret the script's content.

To define your script's interpreter as Bash, first locate a full path to its executable binary using `which` command, prefix it with a shebang `#!` and insert it as the first line of your script. There are various other techniques how to define shell interpreter, but this is a solid start.

From now, all our scripts will include shell interpreter definition `#!/bin/bash`.

Q2) Create a small script shell file which does something simple and copy / paste content of the script file and his execution output here. Notice that the first line should be #!bin/bash

File Names and Permissions

Next, let's briefly discuss file permissions and filenames. You may have already noticed that in order to execute shell script the file needs to be made executable by use of `chmod +x FILENAME` command. By default, any newly created files are not executable regardless of its file extension suffix.

In fact, the file extension on GNU/Linux systems mostly does not have any meaning apart from the fact, that upon the execution of `ls` command to list all files and directories it is immediately clear that file with extension `.sh` is plausibly a shell script and file with `.jpg` is likely to be a lossy compressed image.

On GNU/Linux systems a `file` command can be used to identify a type of the file. As you can see on the below example, the file extension does not hold any value, and the shell interpreter, in this case, carries more weight.

Script Execution

Next, let's talk about an alternative way on how to run bash scripts. In a highly simplistic view, a bash script is nothing else just a text file containing instructions to be executed in order from top to bottom. How the instructions are interpreted depends on defined shebang or the way the script is executed.

Another way to execute bash scripts is to call bash interpreter explicitly eg. `$ bash date.sh`, hence executing the script without the need to make the shell script executable and without declaring shebang directly within a shell script. By calling bash executable binary explicitly, the content of our file `date.sh` is loaded and interpreted as Bash Shell Script.

Sometimes, it's useful to debug shell interpreter when an error occured, to understand which lines are properly executed and which ones failed. To do so, it's possible to run a shell script in debug mode. Find on internet how to do so, using your preferred search engine. Notice that we don't want to modifiy the shell script, juste start it in a debug mode.

Q3) Put a screenshot or copy / paste of the command you're using for starting your small shell script in debug mode and the associated output.

Relative vs Absolute Path

Lastly, before we program our first official bash shell script, let's briefly discuss shell navigation and the difference between a relative and absolute file path.

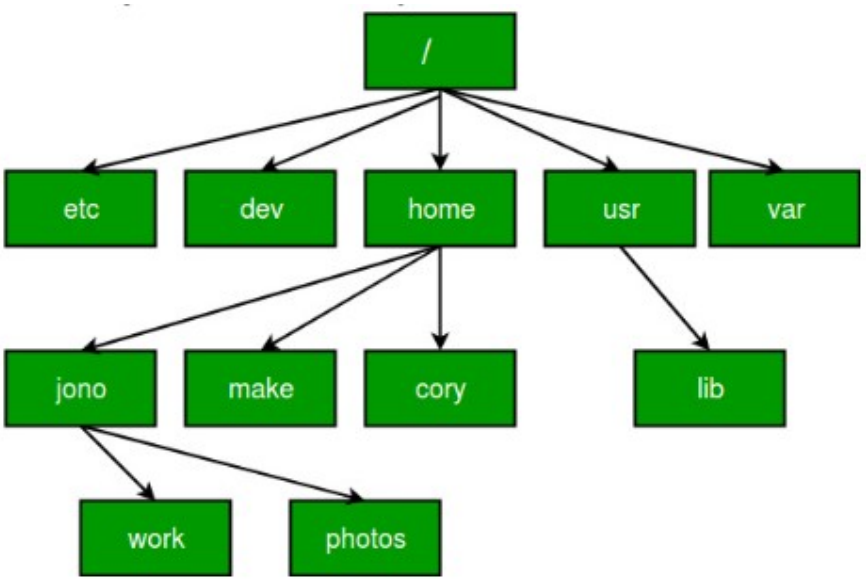
Probably the best analogy to explain a relative vs. absolute file path is to visualise GNU/Linux filesystem as a multiple storey building. The root directory (building's entrance door) indicated by `/` provides the entry to the entire filesystem (building), hence giving access to all directories (levels/rooms) and files (people).

To navigate to a room 1 on level 3 we first need to enter the main door `/`, then make our way to level 3 `level3/` and from there enter the `room1`. Hence, the absolute path to this particular room within a building

is `/level3/room1`. From here, if we wish to visit room2 also on level3 we first need to leave our current location that is room1 by entering `../` and then include room's name `room2`. We took a relative path to room2 which in this case is `../room2`. We were already on level 3, so there was no need to leave the entire building and take absolute path via main entrance `/level3/room2`.

Fortunately, GNU/Linux features a simple compass tool to help you navigate throughout the filesystem in the form of `pwd` command. This command, when executed, will always print your current location.

Exercice on relative vs absolute path :



For all questions in Q4, consider the above tree directory as the tree directory used in those questions.

Q4)

Current directory is : /home/jono/work

a) What command you would type for moving to /home, using relative path ?

b) What command you would type for moving to /home, using absolute path ?

c) What command you would type for moving to /home/jono/photos, using relative path ?

d) What command you would type for moving to /home/jono/photos, using absolute path ?

Current directory is : /var

e) What command you would type for moving to /usr/lib, using relative path ?

f) What command you would type for moving to /usr/lib , using absolute path ?

Current directory is : /

g) What command you would type for listing /home/make directory, using relative path ?

h) What command you would type for listing /home/make directory, using absolute path ?

Quick Tip:

Execute **cd** command without any arguments to instantly navigate to your user home directory from any location. Execute **cd -** to toggle between your last two visited locations. In what directory you end up after executing **cd ~** and **cd .** commands

Navigation through GNU/Linux filesystem is a simple and yet to many a very confusing topic. Familiarise yourself with GNU/Linux filesystem navigation before you move on to next sections of this tutorial.

Hello World Bash Shell Script

Now, it is time to write our first, most basic bash shell script. The whole purpose of this script is nothing else but print "Hello World" using `echo` command to the terminal output. Using any text editor create a new file named `hello-world.sh` containing the below code:

```
#!/bin/bash

echo "Hello World"
```

Copy the above content in the file named hello-world.sh

Once ready, make your script executable with the `chmod` command and execute it using relative path `./hello-world.sh`:

Q5) Copy / paste the command and associated output of your hello-world.sh execution.

Simple Backup Bash Shell Script

Let's discuss a command line execution and how GNU/Linux commands fit into the shell script creation process in more detail.

Any command which can be successfully executed directly via bash shell terminal can be in the same form used as part of bash shell script. In fact, there is no difference between command execution directly via terminal or within a shell script apart from the fact that the shell script offers non-interactive execution of multiple commands as a single process.

Quick Tip:
Regardless of the script complexity, do not attempt to write your entire script in one go. Slowly develop your script by testing each core line by executing it first on the terminal command line. When successful, transfer it to your shell script.

Additionally, most commands accept so called options and arguments. Command options are used to modify command's behaviour to produce alternative output results and are prefixed by `-`. Arguments may specify command's execution target such as file, directory, text and more.

Each command comes with a manual page which can be used to learn about its function as well as what options and arguments each specific command accepts.

Use `man` command to display manual page of any desired command. For example to display a manual page for the `ls` command execute `man ls`. To quit from manual page press `q` key.

Although our first "Hello World" shell script requires a solid understanding of the file creation, editing and script execution, its usability can be clearly questioned.

The next example offers more practical application as it can be used to backup our user home directory. To create the backup script, on **Line 2** we will be using `tar` command with various options `-czf` in order to create a compressed tar ball of entire user home directory `/home/linuxconfig/`. Insert the following code into a new file called `backup.sh`, make the script executable and run it:

NOTE: your home directory may be different, use your home directory's full path instead of /home/linuxconfig

```
#!/bin/bash

tar -czf /tmp/myhome_directory.tar.gz /home/linuxconfig
```

Quick Tip:
Enter **man tar** command to learn more about all **tar** command line options used within the previous **backup.sh** script.

Q6) Try to run the tar command without – option prefix, does it work ?

Q7) Does it work ? If so why, if not why ?

Q8) What is the option to let tar command follows symbolic links when creating the tarball ?

Q9) Which value is returned by tar when a fatal error occured ?

Q10) Which option mades the tar command verbose ?

Q11) Add this option to your tar command in your script, then rerun your script again and copy / paste the command and associated output of your hello-world.sh execution.

Variables

Variables are the essence of programming. Variables allow a programmer to store data, alter and reuse them throughout the script.

Create a new script `welcome.sh` with the following content:

```
#!/bin/bash


greeting="Welcome"

user=$(whoami)

day=$(date +%A)


echo "$greeting back $user! Today is $day, which is the best day of the entire week!"

echo "Your Bash shell version is: $BASH_VERSION. Enjoy!"
```

By now you should possess all required skills needed to create a new script, making it executable and running it on the command line.

Q12) Run the above welcom.sh script and copy / paste or add the screenshot of the associated output.

Let's look at the script more closely. First, we have declared a variable `greeting` and assigned a string value `Welcome` to it. The next variable `user` contains a value of user name running a shell session. This is done through a technique called command substitution. Meaning that the output of the `whoami` command will be directly assigned to the user variable. The same goes for our next variable `day` which holds a name of today's day produced by `date +%A` command.

The second part of the script utilises the `echo` command to print a message while substituting variable names now prefixed by `$` sign with their

relevant values. In case you wonder about the last variable used `$BASH_VERSION` know that this is a so called internal variable defined as part of your shell.

Quick Tip:
Never name your private variables using UPPERCASE characters. This is because uppercase variable names are reserved for internal shell variables, and you run a risk of overwriting them. This may lead to the dysfunctional or misbehaving script execution.

Q13) Explain in french the above Quick Tip ?

Variables can also be used directly on the terminal's command line. The following example declares variables `a` and `b` with integer data. Using `echo` command, we can print their values or even perform an arithmetic operation as illustrated by the following example:

```
a=8

b=4

echo $a

4

echo $b

8

echo $[$a + $b]

12
```

Now that we have bash variable introduction behind us we can update our backup script to produce more meaningful output file name by

incorporating a date and time when the backup on our home directory was actually performed.

Furthermore, the script will no longer be bind to a specific user. From now on our `backup.sh` bash script can be run by any user while still backing up a correct user home directory

Notice that in some cases, a variable is used with $<variable_name>, for instance $myvar, but sometimes the variable usage requires to use ${<variable_name>}, this is the case here for the output filename, because the string we wanna obtain contains the variable value plus some other value(s), for instance, we can do :

echo « my variable value is $myvariable »

but if we want to add to the varaible some strings without space, it should use {..} like below :

echo « my variablue value is start_${myvariable}_end »

Q14) Modify the script so now it does :

- Backup home directory of any user (if the user's home location is /home/<username>)
- The user value is taken from a current environment variable indicating the logged user (you may have to find which one it is)
- The backup directory is /tmp/<username>_home_<date_with_year_month_dayofmonth_hour_minute> (remember the {…} here)
- List the backup file

Q15) Modify the script in order to add one comment line per line where you've put a comment in.

You may have already noticed that your modified script introduces one new bash scripting concepts. Your new `backup.sh` script contains comment line. Every line starting with `#` sign except shebang will not be interpreted by bash and will only serve as a programmer's internal note.

Like for lot of scripts, your script can be parametrized. To do so, you have to figure out how to make your script parametrizable, that is, when you call you script, you write the script's name but also additional parameters which can be different any time you start a new script instance. For instance, you can use your script for backuping home of user toto :

./backup.sh toto

And to backup home of user tutu

./backup.sh tutu

Q16) Modify the script in order to be able to use the first parameter as the user to use for backuping the user's home.

Functions

The topic we are going to discuss next is functions. Functions allow a programmer to organize and reuse code, hence increasing the efficiency, execution speed as well as readability of the entire script.

It is possible to avoid using functions and write any script without including a single function in it. However, you are likely to end up with a chunky, inefficient and hard to troubleshoot code.

Quick Tip:
The moment you notice that your script contains two lines of the same code, you may consider to enact a function instead.

You can think of the function as a way to the group number of different commands into a single command. This can be extremely useful if the output or calculation you require consists of multiple commands, and it will be expected multiple times throughout the script execution. Functions are defined by using the function keyword and followed by function body enclosed by curly brackets.

Here's a simple script containing one function :

```
# !/bin/bash

function user_details {

        echo « User Name : ${whoami} »

        echo « Home Directory : $HOME »

}
```

Q17) Copy the above content to a script name user_details.sh then execute it. Copy/Paste output or screenshot. Does it work ? Why ?

Q18) Modify the script to make it working and copy/paste the script content (or screenshot).

The following video example defines a simple shell function to be used to print user details and will make two function calls, thus printing user details twice upon a script execution.

Q19) Put the function call before the function definition. Does it work ? Why ?

It is important to point out that the function definition must precede function call, otherwise the script will return `function not found` error.

The user_details.sh scrip introduced yet another technique when writing scripts or any program for that matter, the technique called indentation. The `echo` commands within the `user_details` function definition were deliberately shifted one TAB right which makes our code more readable, easier to troubleshot.

With indentation, it is much clearer to see that both `echo` commands below to `user_details` function definition. There is no general convention on how to indent bash script thus it is up to each individual to choose its own way to indent. Our example used TAB. However, it is perfectly fine to instead a single TAB use 4 spaces, etc.

Having a basic understanding of bash scripting functions up our sleeve, let's add a new feature to our existing backup.sh script. We are going to program two new functions to report a number of directories and files to be included as part of the output compressed the backup file.

Q20) Create two functions in the backup.sh script :

- One, named number_of_files, for listing the number of files that will be included in the backup. You can use, find, |, and wc to write this function, check with man <command> or google some examples if you need.
- One, named number_of_directories, for listing the number of directories that will be included in the backup. You can use, find, |, and wc to write this function, check with man <command> or google some examples if you need.
- Call those two functions before doing the tar
- Copy/paste new backup.sh content or do a screenshot
- Execute it and check that the functions are doing what they've been created for
- Copy/paste output of the script execution or screenshot

The output should be starting like :

```
$ ./backup.sh

Number of files to be included:19

Number of directories to be included:2

[…]
```

Bonus :

Q21) There's a command under linux which display strings as big letters, like this :

Try to find this command and check this is the right one. You can check this using : <command> Question 17 | md5sum, it should be :

ab867f19c98b2a79e2d8c4327613187c

Q22) Sometimes, it's useful to decompose a primer number, find the appropriate unix command to do so. You can check that this is the right one, if you do <command> 154 | md5sum, it should give:

4665931679c4d9feb1f4991fd7f329db

Q23) For the fun, there's an Unix command displaying a cow and say want you expect her to say, find this command and do <command> I love R1.04 | md5sum, it should give :

bdfbca985c61354229eea3d3e79ff78e

Q24) cat command print a file content, find the command which do the same but in reverse order (first line will be the last and last line will be the first), find this command and do <command> /tmp/file.txt | md5sum, where file.txt contains :

line1

line2

line3

It should give :

d41d8cd98f00b204e9800998ecf8427e

Q25) Find the Unix command which display the opposite of a string, find this command and do <command> I love R1.04 | md5sum, it should give :

1d67b3e98f379323ea0b8c31e06bc764

Q26) cat command print a file content, find the command which display a file's content but in random order, find this command and do <command> /tmp/file.txt | md5sum, where file.txt contains :

line1

line2

line3

It should give (repeat the full command 20 times) :

cc3d5ed5fda53dfa81ea6aa951d7e1fe

or

65fbf5d080d3776a1ffdee680802bdcb

or

2b46d482c1414ab43531a2d15d23ea60

or

6f9e7d632ecded3054196272cd97e23c

or

d1deff46e4362ef3bcc9a10c748f2bb0

or

2f1a5d9e338462eb8974b33139017e72

Origine du document : https://linuxconfig.org/bash-scripting-tutorial-for-beginners
Auteur et date du document d'origine : Lubos Rendek (28 May 2020)