Activity 1 - Bash Primer

Daniel Schoonwinkel – E&E Postgraduate Networking Course 2019

May 14, 2018

1 Introduction

BASH (Bourne Again Shell) in some form or other installed in most Linux / Unix distributions. Manipulating files and variables using command line arguments and scripts is a valuable skill, looks great on your CV and potentially save you time and repetitive donkey work. BASH is especially useful in scenarios where you only have ssh remote access and no GUI interface (think servers, datacenters, remote IoT, etc.).

In this part of the networking course we will learn to use many frequently used commands and write a few basic shell scripts.

2 Practical examples

In this section you will see examples of how command can be used. Run all of the commands on your own computer to get an idea of the output and what each command does.

2.1 Basic commands

To get a list of files in a directory, get a Terminal (default Ctrl+Alt+T on Ubuntu), type

```
ls
```

and Enter. What do you see?

To get information about how to use a command, type the command and –help (or -h for some commands), like so:

```
ls --help
```

This gives usage instructions and other commandline arguments that can be used after the command. For example:

```
ls -a -l
```

The above command can also be written as:

```
ls -al
```

To get even more information about how a specific command works type:

```
man ls
```

This brings up the manual of given command. Use arrow keys to scroll up and down. Type q to quit.

Try all of these command on your own:

```
ls, find, uname −a, ps −e, df, ifconfig, history
```

2.1.1 File structure

In the Unix file structure, all entries are treated as files (including folders). This could sound strange, but it actually simplifies handing folders and files, because they are treated the same. Furthermore, useful system runtime values can also be stored as files (more on that later).

Navigating the file system is achieved with the change directory (cd) command. Run the command:

```
cd ∼/
```

This will change the directory to the current user's home directory. The ∼ symbol represents the home directory.

Run the following command:

```
pwd
```

What do you see? The output is the **absolute path** to the current user's home directory. You can do a `ls` to get the contents of the directory.

Each `/` represents another folder away from the root of the directory structure. Doing this:

```
cd /
```

will take you to the root (somewhat similar to the C:
directory of Windows.) Doing a `pwd` will also confirm that you are in the highest directory. All **absolute paths** are calculated from here.

Doing a

```
cd /home/yourusername
```

will use the **absolute path** to get to your home directory.

Because absolute paths can become quite cumbersome, we more frequently use relative paths, for example (go back to root (`/` directory first)

```
cd home/
cd yourusername/
pwd
```

Note how both `cd` commands do not have a `/` before them. More useful relative path symbols are the following:

```
cd .
cd ..
```

The first refers to the current directory and the second refers to the directory on up on the hierarchy. If you keep on repeating the second command, you will end up back at the root (`/`) directory. Do this now.

2.2 File and directory manipulation

Git is a version control tool originally created by Linus Torvalds. More details on how Git can be used later, for now, we are going to use it to get the Activity files from the GitHub.com repository. Navigate to your ∼/`Downloads` directory and clone the repository:

```
git clone https://github.com/dschoonwinkel/networkingpgcourse.git
```

In this directory you will see 3 files and the Latex folder (in the Latex folder is the source documentation for this tutorial that you are reading).

The python_populator.py file is a Python script that writes text to "some_text.txt". You can run the python script with

```
python python_populator.py
```

This will start writing to "some_text.txt". You can stop programs on Linux by pressing Ctrl + C. Now use

```
cat some_text.txt
```

to display the contents of the text file. You can use `cat` on any type of text file (even Python scripts and C++ source files) to quickly show the contents of that file.

Now, open another Terminal window, and show them side by side. Start the python script in the one window, and run the `cat` command above repeatedly on the other. You should see that the contents of the text file is growing. You can confirm this by running `ls -l` repeatedly. If you want to see the results of a program repeatedly, you can use this command:

```
watch −n0 cat some_text.txt
```

`watch` calls a command repeatedly at an interval specified (`-n0`, i.e. as frequently as possible). This is very useful for checking the frequently changing status of something. I use it a lot with `ifconfig` for example.

Now stop your program with Ctrl+C. If you wanted to start your python script in the background, i.e. not active in your current terminal allowing you to continue using the terminal, you can accomplish that with the `&` symbol:

```
python python_populator.py &
```

It still seems as if the program is active in the terminal, but if you press Enter, you will see the terminal is ready to accept a command (shown by the $ at the end of the line). Once the script prints something out again, the $ disappears, but can be seen again after you press Enter.

If you press Ctrl+C to stop the program, it will not quit. Why is this? Because the program is running in the background (in a seperate process), so you no longer have direct control of it.

So, how do we stop it? In the other Terminal window that you have open, type the following command:

```
ps −e | grep python
```

You have seen the `ps` command before, showing you the processes running on the machine. The `|` is the piping operator, more on that later. `grep` is a searching tool. Thus the command above can be read as "Show the running processes,

send the results to `grep` and search for all lines containing python". This should show you a list of all programs running with python in their names.

In the list you will see a process ID column (PID). This is the process ID assigned by the operating system to this program. To stop this program, we need to call `kill` on it, for example:

```
kill 5335
```

if the PID was 5335. You should now see that the output has stopped from the first Terminal. If you press Enter on that Terminal it will display a message about the python program that has been terminated.

You could also kill all the running "python" processes with `killall python`. Be warned that this will stop all python processes that you have access to / started, so it could cause unexpected results.

Another way of getting access to the process again from the Terminal is to use the `fg` (foreground) command. Once you have direct control of the process, you can stop it with Ctrl+C. Try this now.

The output that you see on screen is called stdout. If you want to save that output to a file, you need to redirect it to a filename. For example:

```
python python_populator.py > output1.txt
```

You will now see that there is no output in the Terminal window, because it is being sent to a file. `output1.txt` is created and overwritten every time this command is run. To append the output to an existing file, use >> instead of >.

Other useful commands for file / directory manipulation are `cp, mv, rm`, copy, move and remove respectively. **Beware that using rm will completely remove a file / folder, with no chance of recovery.**

Copying / removing directories and their subdirectories requires the `-r` recursive flag, e.g. `cp -r , rm -r`.

2.3 Piping commands

Now that you have a good idea of how files work on Linux, let us get to more interesting commands:

The piping operator `|` sends the stdout of one program to the stdin of another program.

A couple of examples will help us illustrate how the piping operator can be used. In each case, run the first command seperately, then use the piping operator to see the final result. For best results, ensure that you have about 20 lines of text in some_text.txt.

```
ls | grep text
find . | grep text
cat some_text.txt | grep 1
cat some_text.txt | less  #This requires 'q' to quit.
ls | head −n 2
cat some_text.txt | tail
```

Try all of the commands above and any combination that you think could be interesting. Be aware that not all commands accept input from stdin, and some do not accept it in the way that you would expect. Always check the man

pages to know how programs work. `less` is a simple interactive text viewer. It displays the text, instead of dumping it to the Terminal. `head` shows the top lines 'n' of the input, can be used with files as well. `tail` is the opposite of head, showing the last 'n' lines of a file. Default is 10.

2.4 Networking commands

In this section, we will be discussing basic networking commands, and using `ssh` to get access to a remote machine.

Firstly, ensure that you are connected to the RaspberryPi Wifi. Your IP should be in the 192.168.2.* range. You can check your IP on Linux by using the `ifconfig` command.

Firstly, we will do a `ping` to see if you can communicate to the Raspberry Pi.

```
ping 192.168.2.1
```

A `ping` sends out an Internet Control Message Protocol (ICMP) packet that requests the destination to reply with ICMP packet. If a ping is successful, it is very likely that we will be able to communicate with the target. Most computers, servers and routers are set up to reply to `ping`s, making it a very reliable tool for checking connectivity. If you are connected to a hotspot with Internet connection, you can

```
ping www.google.co.za
```

to check connectivity to Google. You will see that the domain name (www.google.com) is resolved to an IP address. This is called domain name lookup and is provided by the Domain Name Service (DNS). More on this protoocol later in the course.

Now that you have pinged the Raspberry Pi, try logging in remotely using `ssh`:

```
ssh guest@192.168.2.1
```

The password is raspberry.

This should show another commandline, very similar to the one that you were using, but with a different username, for example, you would see

```
pi@craftberrypi41: ~$
```

Most commands are universal between Linux distributions, so you would be able to perform the commands mentioned above on the Raspberry Pi commandline as well.

Now, lets use our access to the Raspberry Pi to play around with networking.

Start with the following command on the Raspberry Pi:

```
nc -l 9999
```

This is starts a `netcat` session listening on TCP port 9999.

Whatever is received on that TCP port will be printed to stdout.

Now on your PC, in another Terminal, ensure that you have a valid connection to the Raspberry Pi. Then, use this command to connect to the TCP port 9999 on the Raspberry Pi.

```
nc 192.168.2.1 9999
```

Type some text on your PC's commandline and press Enter, you should see it on the Raspberry Pi's terminal.

Note: `nc` accepts only one TCP connection, so to further experiment, you will need to stop both sides of the connection with Ctrl+C.

Now, we will be using `nc` to transfer a file. `nc` accepts input to its stdin, and can therefore be used with the `|` operator. For example:

```
cat some_text.txt | nc 192.168.2.1 9999
```

The contents of some_text.txt should then be sent to the terminal on the Raspberry Pi. If you wanted to save this text, you could redirect the output of `nc` on the Raspberry Pi terminal to a file, e.g. :

```
nc -l 9999 > received_text.txt
```

This can be used to transfer any type of file over a TCP connection.

Another way of transferring files over the network would be `scp`, a copy utility using the secure shell layer. Using `scp` is left as an exercise to the reader. For more advanced and efficient copying, also have a look at `rsync`.

2.5 Cmdline editing and scripting

Most Linux commandlines support "word processor-like" text editing directly from the commandline with (amongst others) `vi / vim and nano`. My personal preference is `nano`, but not all Linux distributions have `nano` built-in, with `vi` being the most common. Try editing some_text.txt with both. Note: to edit `vi / vim`, press i. To exit, press Escape, and type `:wq!` for (write, quit, yes-overwrite). To exit `nano`, you will need to press Ctrl+X and confirm writing changes with 'y'. As you will see, `nano` is much easier to understand for a beginner, although true `vim` skills can put you in a higher social status of true commandline guru's.

Now that you can edit from the commandline, we will have a look at writing BASH scripts. Scripts are essentially a list of commands performed after one another, and are great for simplifying routine or repetetive tasks. Scripts are not compiled and will simply run from start to finish, even if one of the commands in the script failed. Error checking should be done with "if" statements in the script to ensure that the correct result is obtained.

Run the example_script.bash as follows:

```
bash example_script.bash 1
```

If you look at the script, you should notice 3 things:

1. The syntax of the `if` statement.
2. The `echo` command: similar to `print` command from python / `printf` from C.
3. The commandline argument used in the `if`: $1. $2 represents the second commandline argument etc.

Other useful commands : `du -sch $(ls), sudo shutdown -h now, sudo reboot`

3 References

List of sources:

- http://tldp.org/LDP/abs/html/writingscripts.html