Shell Scripting

COSC301 Laboratory Manual

Important

For this lab, you should do your work in Client1, as it contains some required software components. Also you need to install a few more packages using **sudo apt install libtext-csv-perl libnet-patricia-perl**

Note: this will require NAT to be enabled in the network tab of the VM settings.

As you will be writing scripts, it is important that you have a good editor. The editor you use is a matter of personal preference, but it's often important in a server environment to be able to use a powerful editor that is fast, uses minimal resources and is widely available.

1. Effective Use of an Editor (Vim)

Log in to Client-1 and open a terminal window. Then run the command **vimtutor** to start the in-built tutorial. Follow the instructions in the tutorial to learn the basics of the Vim editor. One of the reasons we want to teach you **vim** here is because you already have some exposure to other editors, and want to train your mind to be open to different approaches, and not to fear things because they are different or unfamiliar. You will also find it incredibly useful for making fast edits to the configuration files you will deal with during this paper.

The remainder of this lab will be fairly simple, but there will be some essential commands that we shall need to cover. After that has been completed, you will work through the examples given in the lecture to appreciate how a pipeline is built up, then we'll let you develop your own simple script.

You will likely find this lab very foundational, but also enormously empowering, once you begin to appreciate everything that scripting could do for you.

2. Some Useful Commands

There are many commands that you might find useful for scripting, but if we were to give them to you all at once, you would be overwhelmed. So we'll give you some commands that are used very commonly, along with a brief description. You can refer to their manual pages to understand them better. Many of these commands are demonstrated in the lecture handouts. You can get more help by using the **man** command.

cat
: Concatenate files and print on the standard output. Commonly this is just used to output a (single) file to stdout, and thus introduce it into a pipeline, although that is commonly unnecessary.

echo
: Display a line of text. Rather like a simple 'print' command.

printf
: Format and print data. Much like C's printf(3) function.

head
Output only the first n lines of the input.

tail
Output only the last n lines of the input.

tr
Translate (eg. change to upper-case) or delete characters.

cut
Remove sections from each line of input. Not very sophisticated; **sed** offers more power.

sort
Sort lines of text, possibly as numbers.

uniq
Remove duplicate lines from sorted input.

grep
Print lines from input that match a pattern.

wc
Print the number of bytes, words, and lines in input.

tee
Read from standard input and write to standard output and files. If you think of the plumbing analogy, this provides a 'T' junction. It can be very useful when inspecting what is going through a particular part of a pipeline.

ps
Report process status. Using **ps -eo pid,command** can be useful in scripts, as using a custom output format can be easier to parse. Supports a huge number of other options as well, most of which are not useful for scripting purposes.

kill
Send a signal to a process. Signals are not necessarily fatal, but are a primitive form of inter-process communication.

xargs
Build and execute command lines from standard input. Very useful in conjunction with **find**.

find
Search for files (or directories, etc.) in a directory hierarchy.

find needs plenty of examples to show how best to use it. The manual page for find(1) should contain a section showing various examples. We shall revisit **find** later in this section.

mktemp
Make temporary file name (unique).

du
Estimate file space usage. Note that this program can be particularly annoying at times.

Here is an example you can put into a file called `big5`, that shows the largest five entries in the current directory. It's useful for figuring out where all the disk space is being used up.

```
$ du -k -d1 | grep -v '^[0-9]*[[:space:]]*\.$' | sort -rn | head -5
Your output will look different, you might not have any.
42080   ./Lectures
7020    ./Labbook
…
```

We should mention that there are GUI tools that are much more enlightening about where disk-space is being used, such as the Disk Usage Analyzer application in Ubuntu.

basename
Strip directory, and optionally a named suffix, from filenames.

```
$ basename "/path/to/foo.txt"
foo.txt
```

dirname
Strip non-directory suffix from file name.

```
$ dirname "/path/to/foo.txt"
/path/to
```

date
Print or set the system date and time.

```
$ date "+Today: %y/%m/%d"
Today: 07/04/18
```

sleep
Delay for a specified amount of time, in seconds. Some systems may have **usleep** available, for sub-second intervals.

getopt
Parse command options (enhanced).

An example was provided with the lecture notes.

2.1. find, `-prune` and `-print`

find can become a very confusing command to understand if you don't specify `-print` as the operation to perform upon match. Note that if you do not specify an operation, such as `-print` or `-exec` then you will get something that appears to be the same as `-print`, *but it is not*. You will notice a (very confusing) difference if you, for example, use the `-prune` option to omit directories from your search. Here is an example, which could take quite a while to figure out, of how to use `-prune` correctly. The reason it took a long time figuring out was because one could be previously under the misapprehension that **find** behaves as if `-print` was the default operation.

Let's assume that our files may have spaces between them, which could prove problematic as spaces separate arguments in the Unix command-line. **find** and **xargs** have the ability to separate arguments using an ASCII NUL character, so we shall use that as well.

In this example, we want to remove certain files from a Subversion working directory, but we *never* want to change anything inside any directory called `.svn` because that is private to Subversion. This is perhaps one of the most common use-cases for wanting to use `-prune`, and it will be useful to you later in your later studies, which is why I'm showing you this now. The way to remove a file under a Subversion repository is with the command **svn remove `filename ...`**.

The particular files we wish to remove all start with `._`, which were put there by my Mac and we accidentally imported them. We could do a similar thing with, for example, LaTeX temporary files.

```
This is an example, we don't expect you to run this command.
$ find . -name .svn -prune -or -name ._\* -print0 \
>  | xargs -0 svn remove --
```

We've also added `--` to the end of the **svn remove** command, as this should cause **svn remove** to not treat as an option any filename which might otherwise be seen as an option. Note that this is unnecessary in this case as we know all results are going to start with `._` and never with `-` or `--`, but in principle this is a very good thing to do.

3. Understand the Examples from the Lecture

Now that you've acquainted yourself briefly with the commands listed above, go through the examples from the lecture slides, and understand how they work. It will be useful for you to build up each stage of the pipeline at a time, to see how the output of each command is transformed by the next.

Make sure you ask about those things you do not understand. It may be useful to talk with your peers first. Ensure you understand the example that looks through the web-server logs. You'll be doing a similar thing in this lab.

4. Self-assessment

There is a video available in the "resources" folder demonstrating how to get started for the assessment. It shows such techniques such as: creating a place to put your scripts; modifying your `PATH` variable so you can easily run your script; as well as copying and modifying the resources for this self-assessment.

Note

You should watch the video before creating the following script. You will know how to create the script already after watching the video. All the required resources for creating the script are in the "resources" folder, which should be shared with Client1 as below.

Click on "cosc301-client1 [Running]", at the bottom of the Settings window, you will see Shared folders. Click it.

Click on the folder icon with a green plus icon; this will add a new entry. In the Folder Path drop-down box, select Other... and navigate to the **K:\COSC301\** directory. Click on resources and then click on Select Folder and OK.

Now follow the commands below.

```
$ sudo mount -t vboxsf resources /mnt

$ mkdir scripting
$ cp /mnt/scripting/* scripting
$ cd scripting
```

1. You will create a script that is passed Apache server logs as input, and creates a summary showing each unique client's IP address, and whether it is a local, domestic, or international client. By itself, it's not all that useful, but it can be useful if we were to include other data such as the amount transferred; this could be useful if you are charged different rates depending on where the traffic is coming from or going to. This is a basic application of *geolocation*: trying to find out which country a particular request is coming from. Since we're only interested in determining the scope of a client (local, national, internation), we might call this *geoscoping* instead). This is not a particularly accurate business and it's important to keep any geo-location data up-to-date; weekly updates are suggested.

To help you, we have created two auxiliary Perl scripts in the above directory. The first, called **geolocation-country-prefixes**, converts the IP ranges from the provided Comma-Separated-Values (CSV) formatted geolocation database into a list of CIDR prefixes for a particular country. The geolocation database we're using originally came from IP-to-Country.com. You will only need to use this script once in order to provide the reference data which we shall use over and over again. You need to use the script as below to create `domestics.txt` from `ip-to-country.csv`.

```
$ ./geolocation-country-prefixes country-code < ip-to-country.csv > domestics.txt
```

Tip

The `country-code` is the two of three-letter ISO country code or name for your country. For example: NZ, NZL and "NEW ZEALAND" is New Zealand while OM, OMN or "OMAN" is Oman.

The generated `domestics.txt` will be used by the the second script, called **ip_classify**, which is used to answer the question of whether a particular IP address (or rather, a list of them) are local, domestic or international. The rest of the task therefore is to extract the data we want out of the web-server log files, presenting it in some suitable format, removing duplicates, and presenting the output to **ip_classify**.

ip_classify is given a list of addresses on stdin, and writes out a list of <address,classification> pairs. This script is very fast, because it uses a data-structure called a Patricia Trie[1], which is the same kind of data-structure used for routing table lookups.

If this component of the system were to be done using just shell commands, you would find a vast performance drop. This highlights one useful thing about shell scripting: we can use a selection of different tools in order to fulfill different criteria for different sub-tasks. Because the auxiliary script uses some extra Perl modules (libraries), you will need to do this assessment on Client1 which has the particular modules already available.

[1]"Trie" is pronounced "try" and is short for "retrieval"

Edit your copy of **ip_classify** in order to tell it where to find its resources. Create `domestics.txt` using **geolocation-country-prefixes**.

We should give a proper name for the script. This task is left up to you (indeed, it is part of the assessment). **You must choose a suitable name!** How do we choose a suitable name? We shall offer you some advice: first, use the principle of Huffman encoding: commands that are referred to very commonly should have short names (eg. **ls**), while commands used infrequently should have longer, more descriptive and preferably structured names. The structuring helps when you make use of Tab completion: put the most significant theme at the beginning. For example, suppose we have a number of scripts that do things related to the running of a web server; we might have scripts that generate various reports and we might have scripts that manage the various sites etc. Therefore, it would be useful to have such scripts begin with **web-report-** or **web-site-**, for example. By typing **web-** and then using Tab completion we get a little menu of the suitable commands.

To help get you started, think of noun and verb phrases that describe what the script does, what it operates on, and what it produces. In our case, we might start with a list such as: web logs, geoscope, classify, report, source address, and client. Also think about what the script *is not*, so as to avoid confusion: its input is web logs, *not* IP addresses.

Warning

Let the following be a lesson in what *not* to do: there are two commands for adding a user to a system, one is interactive and the other is meant for use in scripts. One (we *always* forget which) is called **adduser**, the other **useradd**. If you accidentally use the non-interactive one, you end up creating a user, which you then have to remove and recreate using the other command.

Create your script file and mark it executable using **chmod**.

You will find a copy of some logs in `access_log`; have a look at this file using **less** or a similar tool to become familiar with the format. Remember, all we need to produce from this file is a list of unique IP addresses.

In the shell, which is a good place to develop parts of your script, develop a pipeline that a) outputs only the IP address (hint: **cut** everything into space-delimited fields and output only the first field), b) **sort** the input of IP addresses into a sorted list of unique IP addresses, and c) classifies each line according to **ip_classify**.

Take care of the following:

- Since a call to **cut** will be first in your script, give it whatever arguments you have been given `"$@"` as these could be filenames, or expand to nothing, in which case **cut** will take its input from stdin. This is similar to **cat** in terms of its input: multiple files can be specified *at the end* of the command, or if none is specified input will come from stdin. Many of the Unix toolbox commands (**cut** and **sort** among them) work in this way. See the “Special Parameters” section in bash(1).

- You must name your script suitably.

- You must have a she-bang (`#!`) line.

- Do not assume that you are running in a particular directory. Put required resource files in well-known places. Hard-coding the locations of resources such as configuration files and auxiliary resources can be useful, but hard-code nothing that is particular to a single invocation of the program, such as the location of the input files, or even which log entries you are interested in.

Here are some examples of how your script should be able to be used:

```
One argument
$ your-script ./access.log
…
Multiple arguments
$ your-script ./access.log ./access.log.2
…
Input redirected from file into stdin
$ your-script < ./access.log
…
Input filtered from another command into stdin
$ grep -w 404 ./access.log | your-script
…
```

The output should look like the following.

```
Your numbers will differ.
127.0.0.1 local
139.80.123.34 domestic
139.80.123.36 domestic
139.80.32.2 domestic
```

2. **[Optional challenge]** Add the number of hits from each client, and sort (on the second column/key) the output in descending number of hits. Add column headers and align the output using **column -t**. Output should look something like the following (Hints: You need to use **uniq** and **awk**):

```
IP           HITS   GEOSCOPE
1.2.3.4      123    international
12.3.4.5     113    national
192.168.1.2  12     local
```

5. Last Words

Scripting is an incredibly useful skill, and you end up using it in many places. When building software using **make**, you are using shell commands in the `Makefile`. Shell scripts are frequently used to define complex firewall rule-sets for Unix systems, drive nightly backup and maintenance systems, start up and shutdown Unix systems and services and save a lot of repetition in common workflows.

Shell scripting, however, is not a particularly fast way of automating tasks, although it can be much faster in some cases, but they can save a lot of developer time (remember, CPU time is cheap compared to developer/user time). If you find your shell script is too slow or awkward, you would either introduce more complex processing using a tool such as **awk**, or re-write the script in a higher level scripting language, such as Perl, TCL or Python. Perl is a mainstay tool for many Unix system administrators, although Python has been steadily increasing in popularity for many years, particularly on Linux systems.