Shell Scripting 2020 fifth week

Format of the learning diary and scripts

The tasks are returned in moodle.

Return the learning diary in **PDF format**. Preamble the document with the following information:

- Name and student number

The learning diary should contain the answers to questions made in the "Put in your answer" parts. Also example output can be put into the answer.

Include the code in week's directory as a tarball with scripts named with the task name as "task#.sh", i.e. for task 2 the name would be "task2.sh".

Deadline for this set of tasks is Monday 7th of December 2020.

If you run into problems, try to read the man pages and try google. If you are still stuck with the task send me a msg in moodle and I will answer during normal office hours (9-17).

Counting in the shell (37)

Counting in the shell

A nice trick that should be mentioned is the command-line calculator bc. It can perform very difficult mathematical operations, at the price of its own quirks regarding syntax. For example, the scale variable is used to truncate result precision, but it doesn't do rounding:

Example for bc

```
$ echo "scale=2; 55575.23923 / 3" | bc

18525.07



$ echo "scale=3; 55575.23923 / 3" | bc

18525.079
```

Perhaps the main feature that bc has but bash doesn't is floating point precision.

So why should we use shell arithmetic at all? It's a performance tradeoff. Shell arithmetic is much faster than invoking bc for every calculation. See the following pointers for help on arithmetic syntax:

- http://www.tldp.org/LDP/abs/html/untyped.html
- section ARITHMETIC EVALUATION of man bash

Put in your answer:

- Write a script average.sh that calculates the arithmetic mean of the input parameters ($1, $2, ...). Use bc, and invoke it only once.

Example average.sh

```
$ average.sh 1 9 6

5.33
```

You could try to count daily averages from the dataset. But this is not required. Maybe you do it just for fun?

```
Hint: getting bc to round values properly can be difficult. Please let us know if you found a solution for rounding. Otherwise don't worry about it. But do print out some decimals.
```

Gone in 10 seconds (38)

Gone in 10 seconds

Rewrite the min-max-temperature.sh script from scratch but faster. The running time for finding the maximum temperature from the whole data set has to be less than 10 seconds. Otherwise you'll get no points from the task.

The output format doesn't matter (we don't have gnuplot dependency now), just as long it contains the maximum temperature (ok it's 57 °C if you didn't know yet). You can even find out both the minimum and maximum on the same run. Of course it would be nice to output the filename containing maximum temperature. Below is an example which outputs a new minimum/maximum whenever one is found.

Put in your answer:

- Present your script and the time measurement.
  NOTE THAT THE SCRIPT TAKES THE WHOLE DATA SET AS THE INPUT

Important hint section.

The biggest impact on performance happens if you run an external command (such as grep or sed) on each of the hp-temps.txt file. Try to process the temperature data as a stream (aka. one big pipeline) as far as possible, and finalize with internal bash command (such as if and read). You have about two options:

1. Save the data as variable (stuff=$(pipeline)) and use that in for loop. For this you would need to force for to split the input only at newlines, not at every whitespace, so that the for loops processes input one line at time. Two sub-choices:
   - Replace all the tab characters in input with something else (maybe a comma ","), or
   - change Bash's built-in IFS variable as newline (\n).
   - And then inside the for loop, you can only use internal BASH to do stuff! Parameter expansion (http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion) might be useful as cut replacement.
2. Process the data stream in a while-read loop (aka pipemill, http://en.wikipedia.org/wiki/Pipeline_(Unix)#Pipemill). This is the standard Bash way to read input line-by-line.
   - However, since the pipemill creates a subshell, variables changed inside it are NOT visible back to parent shell. This does not necessary matter here, since we only need to print out any new maximum (or minimum) found in data set, thus transferring the final MAX/MIN value back to parent is not needed.
   - Or, you can do temporary files, or for maximum fun: process substitution! http://en.wikipedia.org/wiki/Process_substitution
3. Use a find command with the -exec argument (check man find for more info)
4. Combine the find command to the for-loop or pipemill above. Good luck!

fast-min-max-temperature.sh

```
./fast-min-max-temperature.sh /tmp/lost24/monitor/

NEW MAX: 2011.10.10 04:40 26

NEW MIN: 2011.10.10 04:40 26

...

NEW MIN: 2010.12.20 14:55 4

NEW MAX: 2011.04.24 14:55 57
```

Hipstafy-dropbox (39)

Hipstafy-dropbox

We're going to create a "dropbox" folder, where you can move (drag-n-drop) image files, and those will be automatically hipstafied as in Week 3. For that, we'll create a script named

hipstafy-wait.sh, which waits for new files, and calls hipstafy.sh (from week 3) when a new file arrives.

Create a folder named hipstafy-dropbox. Perhaps to your Desktop, so it's easily available. Note that the scripts should not be here, but in the same directory structure where all your other LinuxFun scripts are.

Enter the just-created directory in terminal. Command "inotifywait .". Then drag-n-drop a file (any file) to the folder. Watch what happens on terminal.

Inotifywait (https://github.com/rvoicilas/inotify-tools/wiki/) is a very powerful tool for watching file and directory changes. With parameter -m it goes to monitor mode, and will not exit until killed. Our plan:

1. Set up inotifywait for directory hipstafy-dropbox, listening new files in monitor mode.
2. When a new file arrives, execute hipstafy.sh from week 3 and generate a hipstafied image into subdirectory hipstafy-dropbox/hipstafied/.
3. Maybe some error checking, like check if the dropped file is an image file?

Try out inotifywait -m. Do a 1) file move and 2) file copy. Inotifywait is listing a lot of events when a new file arrives (and some random events in between, when someone accesses the directory). Figure out which events are the ones you need to listen, and specify the events with multiple -e parameters.

Inotifywait uses a common "event model" in Bash: when a new event occurs, print a line of text. There is no event-handler registration in Bash, so we'll need to "listen" for those text lines. This is done with a while-read pipemill, which also neatly handles queuing the events.

Try out your final hipstafy-dropbox with some summerly images!

Note that ImageMagick (the convert command) will create a temporary output file first into current directory, which is the one where you called convert. If the temp file is created into hipstafy-dropbox, inotifywait fires events on those, and you could end up in a (possibly infinite) error loop. So: call convert in some other directory, maybe in the hipstafy-dropbox/hipstafied -subdirectory, so that convert creates its temp file there.

Put in your answer:

- Present the hipstafy-wait script you created.

Summoning deamons (40)

Summoning deamons

You could set up hipstafy-wait to run forever in, say, Screen, but we'll use this opportunity to teach you another trick. With Bash, it's quite simple to implement minor daemons, i.e., programs that are kept running in the background and serving a specific task until stopped.

Example simple Bash deamon

```
$ tempdir=$(mktemp -d); while [ 1 ]; do echo "Still alive at `date`" >
$tempdir/portal.txt; sleep 1; done & echo $! > $tempdir/true_name.txt && echo
"$tempdir/true_name.txt"
```

The output of the command is the process id (pid) of the job running in the background. We save the pid into the text file true_name.txt using the special variable $! (http://www.tldp.org/LDP/abs/html/refcards.html#AEN22095).

Assuming a relatively recent Ubuntu distribution and Bash binary, the above is perhaps the simplest way to implement a daemon as a shell script. You can verify that it works by running watch cat $tempdir/portal.txt. The single output line keeps getting replaced every ten seconds.

The daemon survives even if you terminate your SSH connection. Note that this will cause you to lose the $tempdir variable, so may want to make a note of it elsewhere. You can verify this by logging off and back in to your favorite Ukko node.

Depending on your X-terminal (e.g., gnome-terminal), the above minor daemon may still be vulnerable to closing the terminal window. We will get back to this topic in a short while.

Now, release the daemon by running kill $tempdir/true_name.txt or directly by the pid returned by the shell. If all else fails, you will have to hunt down the daemon with pgrep -u $USER -fl or similar.

This technique works only conditional to multiple things which we can't take for granted when writing proper daemons. It is clearly a tradeoff: daemonizing a Bash process works in some cases, but for it to work in all cases, there are multiple details that must be taken care off.

- Many shells issue a SIGHUP signal to running processes when the user terminates the session by logging off.
- A shell program running in the background normally has no access to stdin/stdout/stderr. Trying to access any of them results in getting killed by the operating system.
- The pid of the background process must be stored somewhere for eventual easy termination.
- The rest of the topics go beyond this course.

For our next exercises, cases 1-3 are enough. The rest of the issues are left as advanced topics, with the understanding that if you will encounter these, you should switch from a shell script to a more advanced programming language immediately.

SIGHUP

Whether the shell issues a SIGHUP signal to all running processes is handled by the shopt huponexit on my system. huponexit is a Bash-specific shell option and in my case, it is off by default. This is the reason that the above example works in Ukko. In your case, and specially using other shells that Bash, the option might be enabled by default or not exist at all.

The traditional way of avoiding SIGHUP is to prepend a command or script with the nohup command. This method has the added benefit that it specifies the command exactly.

In most cases, use nohup. If in trouble, rewrite your command as a function first. If the minor daemon above was killed by closing your terminal window previously, nohup should render it invulnerable.

Advanced: The trap example below shows how to capture other signals:

- Proper use of the trap Bash builtin: http://linuxboxadmin.com/articles/tools-and-utilities/a-bourne-again-daemon.html
- Why an exiting shell does not send HUP to child processes: http://en.wikipedia.org/wiki/Nohup#Existing_jobs.2C_processes

File descriptors

The minor daemon we introduced works fine because its output is redirected to a file, and it never reads stdin. In the general case, preparing a daemon for execution requires us to close stdin/stderr/stdout, commonly called the file descriptors. In fact, we must close all file descriptors, for a process may have opened others than the standard three.

Usually the programmer knows about this, however. In our case it is sufficient to redirect all output streams to files > log.out 2> log.err, depending on how you want the output to be processed. Other options are /dev/null and 2>&1, which appends stderr to stdout.

Advanced: Check the following links for more info on how to close & redirect all relevant file descriptors:

- Full-fledged example that closely follows Stevens' Advanced Programming in the UNIX Environment: see http://blog.n01se.net/?p=145
- SSH problems if stdin/stdout/stderr remains open: http://www.zsh.org/mla/users/2005/msg00152.html

PID

We used the file true_name.txt to store the pid of the background process. Usually, daemon pids are stored under /var/run/, but you will not have write access to this directory on the Ukko nodes (sadly).

In our examples, it is sufficient to redirect $! to a designated file, called the pidfile. Note that $! is replaced every time a process is executed in the background, so save it as early as possible in your script.

A nice trick to check a daemon's status is to issue a kill -0 to the pid contained in the pidfile. If the daemon is still alive, the result $? will be 0, otherwise nonzero.

Advanced: pidfiles are problematic in at least two ways: one problem is that process numbers loop, and the second is that a race condition exists if two copies of the daemonized script are run concurrently. Check these links for additional information:

- UUIDs vs PIDs: http://www.howtoforge.com/simple-bash-script-to-work-as-a-daemon
- Bullet-proof use of lock files: http://sysadvent.blogspot.com/2008/12/day-9-lock-file-practices.html

Put in your answer:

Your task is to extend the hipstafy-wait script to a medium-sized daemon. The hipstafy-wait should be wrapped around with a daemon interface hipstafy-daemon.sh that accepts the minimum arguments of start | stop | status | restart.

1. start will background execute your earlier implemented hipstafy-wait and save its pid into a pidfile
2. stop will stop the treadmill and clean up temporary files, if any
3. status will check whether the daemon indicated by your pidfile is still alive
4. restart will first stop and then start the daemon

Output should be printed to a logfile. You may separate stderr into its own file.

You do not need to trap other signals than the SIGHUP sent by the shell. Use the nohup approach, just in case.

- Present your script and snippets of output showing the daemon interface and proper usage.