COMP-206 Introduction to Software Systems, Fall 2022

Mini Assignment 2: Bash scripting

Due October 5, 18:00 EST

This is an individual assignment. You need to solve these questions on your own.

You MUST use `mimi.cs.mcgill.ca` **to create the solution to this assignment.** An important objective of the course is to make students practice working completely on a remote system. Therefore, you must not use your Mac command-line, Windows command-line, nor a Linux distro installed locally on your laptop. You can access `mimi.cs.mcgill.ca` from your personal computer using **ssh** or **putty** as seen in class and in Lab A. **If we find evidence that you have been instead using your laptop, etc., to do some parts of your assignment work, you might loose ALL of the assignment points.** This restriction applies only to the editing (coding) and testing of your programs. You are free to (and encouraged to) backup your source code regularly to your personal computer as a precaution against any misadventures. Delays incurred because you accidentally deleted your programs and you had not taken backups will not get any considerations.

All of your solutions should be composed of commands that are executable in `mimi.cs.mcgill.ca`.

For this assignment, you will have to turn in two shell scripts. Instructors/TAs upon their discretion may ask you to demonstrate/explain your solution. No points are awarded for commands that do not execute at all. (Commands that execute, but provide incorrect behavior/output will be given partial marks.) All questions are graded proportionally. This means that if 40% of the question is correct, you will receive 40% of the grade.**TAs WILL NOT modify your scripts in anyways to make it work.**

Please read through the entire assignment before you start working on it. You can loose up to 3 points for not following the instructions in addition to the points lost per questions.

Lab C and D provides some background help for this mini assignment.
Total Points: 20

Ex. 1 — A wrapper script(10 Points)

One of the common uses for shell script is to be a "wrapper" to an existing software or utility to add some extra functionality that the original software does not provide. This facilitates developers to "extend" custom functionality, especially when they do not have access to the original software source or is not feasible to re-write that source.

In this exercise, you will be developing such a wrapper script for a custom program, `namefix`, available at `/home/2013/jdsilv2/206/mini2/namefix` . `namefix` is a simple program that accepts two filenames as arguments (absolute or relative paths). It reads a list of names in the first file, and writes them into the output file after fixing any uppercase/lowercase issues in the name. If the output file already exists, it is overwritten with new contents, otherwise it will be created.
For example,

```
$ cat names.txt
jean-luc picard
Moff gidEon
$ /home/2013/jdsilv2/206/mini2/namefix names.txt namesfixed.txt
$ cat namesfixed.txt
Jean-Luc Picard
Moff Gideon
$
```

However, the program is very rudimentary and can crash or misbehave if executed incorrectly, (below invocation is missing the output file).

```
$ /home/2013/jdsilv2/206/mini2/namefix names.txt
Segmentation fault
$
```

or if it does not have correct permissions for processing the input and output files.

Your are to write a wrapper shell script, `namefix.bash` that will take care of some of the common scenarios that results in the program crashing (by not executing the `namefix` program and instead warning the user about the issue), as well as adding some small functionality.

Make sure that your script starts with a sha-bang to execute it using `bash` and is followed by a small comment section that includes your name, department and email id (format of this comment section is up to you). The script should also have additional comments for the important parts of the code. **-1 point** if not followed.

1. **(4 points)** Your shell script `namefix.bash` should take two input arguments. The first argument is supposed to be the input file and the second argument is **either** the output file **or** the directory in which the output file is to be created.

   Assuming that the script finds no issues (discussed later), it should (and only then) execute the `namefix` program mentioned above (you will have to use the full path to `namefix` as given in the above example) with proper arguments. After that, it should display the contents of the output file.

   ```
   $ ./namefix.bash names.txt namesfixed.txt
   Jean-Luc Picard
   Moff Gideon
   $
   ```

2. **(4 points)**
   Your script should throw an error message for incorrect usage (for example, invoked with incorrect number of arguments). In this case, the script should terminate with code 1.

   ```
   $ ./namefix.bash names.txt
   Usage: namefix.bash <inputfile> <outputfile>
   $ echo $?
   1
   $
   ```

   The script should also terminate (with code 2) with an error message if both the input and output are going to be the same file.
   The script should also check that the input file can be read (i.e. it is present, permissions, etc.) by the program and terminate with an error message if that is not the case (with code 3).
   Similarly, the script should check that the program will be able to write the output (permissions, etc.,) and terminate with an error message if it finds that there will be an issue. In this case, the error code is to be 4. If there are multiple issues, the script only need to indicate one of them. (I.e., you can report an error and terminate at the first issue you detect).

3. **(2 points)** If the output argument given is the name of a directory (existing), then the script should assume that the user want the output file name to be the same as the input file name (just their directory paths being different so that the output is stored in a different directory) and accordingly do rest of the checks, execution, etc. In the below example, assuming `outdir` is actually a directory, the output file would be `outdir/names.txt`.

   ```
   $ ./namefix.bash names.txt outdir
   ```

4. Your script should work in a logically correct manner irrespective of whether one uses absolute or relative path for its input or output arguments. **-2 points** if there is at least one scenario where it does not work.

5. If your script invokes the `namefix` program in a situation that results in it misbehaving/crashing, **-2 points**, in addition to whatever points lost for above questions.

Ex. 2 —   A Script to Identify prime numbers (10 Points)

On the first day of the class we saw a simple C program that I had written, `primechk` that can be used to verify if a number of prime or not. This program is available in `mimi` as `/home/2013/jdsilv2/206/mini2/primechk`

```
$ /home/2013/jdsilv2/206/mini2/primechk 17
The number is a prime number.
$ /home/2013/jdsilv2/206/mini2/primechk 42
The number is not a prime number.
$
```

The program also terminates with a code of 0 if the number is prime and 1 if it is not a prime number.

However, once again the program is not so clever in terms of the inputs with which it can work reliably. Most importantly, it works correctly only with integers above the value `1` and below `1000000000000000000` (i.e. max 18 digit numbers).

The program can produce incorrect output/misbehave for numbers outside of this range or if given invalid input (like characters, floats, etc.).

In this exercise we will build a shell script `primechk.bash`, that while would be still a "wrapper" concept, will also perform some advanced functionality of its own.

The script will be given an input file as an argument, the script will then pass valid numbers from that file (discussed later) to the `primechk` program (use absolute path to that program similar to the previous exercise) to figure out whether it is a prime number or not.

The script has the following intended usage

```
$ ./primechk.bash -f <numbersfile> [-l]
```

Where `-f` is used to indicate that the next argument to the script is the name of a file. `-l` is an optional argument to indicate whether the script should only print the largest prime number in its output. Otherwise it will print all (valid) prime numbers that it encounters in the file in its output.

1. Make sure that your script starts with a sha-bang to execute it using `bash` and is followed by a small comment section that includes your name, department and email id (format of this comment section is up to you). The script should also have additional comments for the important part of the code. **-1 point** if not followed.

2. **(3 points)** Make sure that the usage is correct, if used incorrectly, throw an error message to the user. Remember, in the Unix spirit, there are more than one way of invoking this script. For example below invocation is also correct.

    ```
    $ ./primechk.bash -l -f <numbersfile>
    ```

    However, you do not have to handle.

    ```
    $ ./primechk.bash -lf <numbersfile>
    ```

    Incorrect usage should result in the script terminating with code 1.

3. **(1 points)** Assuming the usage is correct, the script should check if the file name that was given to it is a file that actually exists (you need not check for permissions, etc.). If not, throw an error message and terminate with code 2.

4. **(4 points)** At the next step, you will pass each line to the `primechk` program if that line is a valid number (according to the discussion above). If the `-l` option is not used, print the numbers that were reported as prime numbers. (In the same order as is in the original file).

    ```
    $ cat mynums.txt
    121
    13
    17 joe
      19
    14.2

    Ab3
    29
    ```

```
$ ./primechk.bash -f mynums.txt
13
29
$
```

The script will terminate with code 0 (even if there were no prime numbers to print).

5. **(2 points)** If `-l` is passed as an option, then the script will print ONLY the largest prime number that it found in the file.

```
$ ./primechk.bash -f mynums.txt -l
29
```

It will then terminate with the code 0. **However, in this case**, if it were not able to find any prime numbers in the file, it will report that it was not able to find any prime numbers as well as terminate with code 3.

6. If your script invokes the `primechk` program in a situation that results in it misbehaving/crashing, **-2 points**, in addition to whatever points lost for above questions.

WHAT TO HAND IN

Upload both of your scripts, `namefix.bash` and `primechk.bash`, to MyCourses under the **mini 2** folder. Do not zip the files together. Re-submissions are allowed, but please try to upload all of the files again (and not just the modified ones) so that TAs do not have to go over multiple submissions to find correct files. You are responsible to ensure that you have uploaded the correct files to the correct assignment/course. There are no exemptions. **If you think it is not worth spending 5 minutes of your time to ensure that your submission that is worth 10% of your grade is correct, we won't either. NO Emailing of submissions**. If it is not in MyCourses in the correct submission folder, it does not get graded. **Because you do not know how to resubmit an assignment or run out of time to figure it out is not an excuse for emailing the assignment**. Imagine what will happen if all of you emailed your assignment instead of submitting it in MyCourses!

Late penalty is -20% per day. Even if you are late only by a few minutes it will be rounded up to a day. Maximum of 2 late days are allowed. Even if you only (re)submit a part of the assignment (e.g., one script) late, late penalty is applicable to the entire assignment - No exemptions!.

COMMANDS ALLOWED

You may use ONLY the following commands, but any option provided by the following commands, even the ones that have not been discussed in class. Depending on your solution approach, you may not have to use all of these commands.

```
$( )      $(( ))    $[ ]      [ ]     [[ ]]
basename  bc        break     case    cat
cd        continue  dirname   echo    exit
export    expr      for       grep    if
ls        pwd       set       shift   while
```

You can also use redirection, logical and/or/negation/comparison/math operators and any check operators (such as checking if something is a file) as required with the `[[ ]]` operator. You can also use shell variables and use/manipulate them as needed. You may use the concept of shell functions, but it might be an overkill for this assignment. You may not use any commands that are not listed here or explicitly mentioned in the assignment description. **Using commands not allowed will result in 3 points deduction per such command.**

ADDITIONAL RESTRICTIONS

- Your scripts should not produce any messages/errors in the output unless you explicity produce it using an echo statement (following the message examples given in the question - also see the example tester output provided for different messages). Violating this would result in **2 points** deduction per such message occurence.

- Your scripts should not take more than 10 seconds to run through the entire tester script / should not "hang", etc. This usually is a result of some logical error in your code. Any unfinished test cases will not receive points. To get an idea, a very naive solution for this assignment runs under 1s with the provided tester script.

- Your scripts must not create any files internally (other than what it is asked to produce as the final output file, etc.), even as temporary output storage that the script deletes by itself. Doing this will result in **3 points** deduction on the assignment score.

- DO NOT Edit/Save files in your local laptop, not even editing comments in the scripts. This can interfere with the fileformat and it might not run on `mimi` when TAs try to execute them. This will result in a 0. No Exemptions !!. TAs will not modify your script to make it work.

ASSUMPTIONS

- You can assume that that no file or directory names will contain any white space characters and will contain only alpha-numeric characters, underscore (_) and period (.) characters.

MINITESTER

A tester script, `mini2tester.bash`, is provided with the assignment (along with an example output) so that you can test how your scripts are behaving.

It is recommended that you <u>first</u> run your scripts yourself, test each of the options and arguments using the examples above and also your own test cases. Once you are fairly confident that your script is working, you can test it using the tester script. **DO NOT use the tester script before you have tested all of the options by yourself. If your script has too many issues, the output of the tester script could be overwhelming and discouraging.** TAs will NOT help you because you are not passing the tester scripts. It means you did not do your own individual testings properly enough.

When you are ready, in order to run the tester, put the tester script in the same folder as your scripts for this assignment and run

```
$ ./mini2tester.bash
```

The idea is that the tester's output should be very similar or even identical to that of the example output provided, except for directory names.

TAs will be testing using a similar tester, with possibly different directory/file names but similar test case principles and a few extra test cases. **Remember, the tester script given to you may not cover all the test cases that is stated in this assignment description.** You are responsible to test different scenarios discussed here.

QUESTIONS?

If you have questions, post them on the Ed discussion board and tag it under mini 2, but <u>do not post major parts of the assignment code.</u> Though small parts of code are acceptable, we do not want you sharing your solutions (or large parts of them) on the discussion board. If your question cannot be answered without sharing significant amounts of code, please make a private question on the discussion board or utilize TA/Instructors office hours.

Please remember that TA support is limited to giving any necessary clarification about the nature of the question or providing general advice on how to go about identifying the problem in your code. You are expected to know how to develop a high level logic, look up some syntax/options and most importantly, debug your own code. Lab D covers a lot of useful debugging techniques. We are not testing your TA's programming skills, but yours. Do not go to office hours to get your assignment "done" by the TAs.

Emailing TAs and Instructors for assignment clarifications, etc., is not allowed. TAs and instructors may convert private posts to public if they are not personal in nature and the broader student community can benefit from the information (to avoid repeat questions). Also check the pinned post "Mini 2 General Clarifications" before you post a new question. If it is already discussed there, it will not get a response. You can email your TA only if you need clarification on the assignment grade feedback that you received.

HINTS

Solving this assignment requires you exploring and learning a few small things on your own (in the spirit of the Unix culture). Some information is provided below to help you narrow down your search. You may not need all of these ideas - it largely depends on how you plan to solve the questions. You are not obliged to use them.

- Wondering how to extract the names of directory/file from a path? explore the commands `basename` and `dirname` to learn what they can do before starting to work on your assignment.

- How do you check if two arguments are pointing to the same directory? Just checking the arguments passed may not work as one could be absolute path and the other could be a relative path. There is an option you can use in the conditional operator (similar to the options used to check if something is a file, etc.). Explore to find that solution (not in the slides).

- What option can you pass to `grep` to print only those lines that DO NOT match a pattern?

- Some times we are not interested in a message (whether its output or error) displayed by a command, only its intended outcome. Explore how the special file `/dev/null` can be used to discard unwanted messages.