Assignment #1

Due Date 1: Friday, 17 May, 2019, 5:00 pm
Due Date 2: Friday, 24 May, 2019, 5:00 pm
Questions 1 and 2 are due on Due Date 1; the remainder of the assignment is due on Due Date 2.
If you joined the course within two days of Due Date 1, then the entire assignment is due on Due Date 2.

1. Provide a Linux command line to accomplish each of the following tasks. Your answer in each subquestion should consist of a single command or pipeline of commands, with no separating semicolons (;). (Please verify before submitting that your solution consists of a single line. Use `wc` for this.) Before beginning this question, familiarize yourself with the commands outlined on the Linux handout. Keep in mind that some commands have options not listed on the sheet, so you may need to examine some man pages. Note that some tasks refer to a file `myfile.txt`. No `myfile.txt` is given. You should create your own for testing.

   (a) Print the 10th through 25th words (including the 10th and 25th words) in `/usr/share/dict/words`. You may take advantage of the fact that the words in this file are each on a separate line. Place your command pipeline in the file `a1q1a.txt`.

   (b) Print the (non-hidden) contents of the current directory in reverse order. Place your command pipeline in the file `a1q1b.txt`.

   (c) Print the number of lines in the text file `myfile.txt` that do *not* contain the string `cs246`. Place your command pipeline in the file `a1q1c.txt`.

   (d) Print the first line that contains the string `cs246` from the text file `myfile.txt`. Place your command pipeline in the file `a1q1d.txt`.

   (e) Print the number of lines in the text file `myfile.txt` that contain the string `linux.student.cs.uwaterloo.ca` where each letter could be either uppercase or lowercase. Place your command pipeline in the file `a1q1e.txt`.

   (f) Print all (non-hidden) files in any *subdirectory* of the current directory that end with `.c` (immediate subdirectories only, not subdirectories of subdirectories). Do not use `find`. Place your command pipeline in the file `a1q1f.txt`.

   (g) Out of the first 20 lines of `myfile.txt`, how many lines contain at least one digit? Place the command pipeline that prints this number in the file `a1q1g.txt`.

   (h) Print all (non-hidden) files in the current directory that start with `a`, contain at least one `b`, and end with `.c`. Place your command pipeline in the file `a1q1h.txt`.

   (i) Print a listing, in long form, of all non-hidden entries (files, directories, etc.) in the current directory that are executable by at least one of owner, group, other (the other permission bits could be anything). Do not attempt to solve this problem with `find`. Place your command pipeline in the file `a1q1i.txt`.

   (j) Before attempting this subquestion, do some reading (either skim the man page or have a look on the Web) on the `awk` utility. In particular, be sure you understand the effect of the command

   ```
   awk '{print $1}' < myfile.txt
   ```

   Give a Linux pipeline that gives a sorted, duplicate-free list of userids currently signed on to the (school) machine the command is running on. Place your command pipeline in the file `a1q1j.txt`.

2. For each of the following text search criteria, provide a regular expression that matches the criterion, suitable for use with `egrep`. Your answer in each case should be a text file that contains **just the regular expression**, on a single line (use `wc` to verify this). If your pattern contains special characters, enclose it in quotes.

   (a) Lines that contain both cs246 and cs247.
   Place your answer in the file `a1q2a.txt`.

   (b) Lines that contain nothing but a single occurrence of laughter, where laughter is defined as a string of the form `Hahahahahahahahahahahahahaha!`, with arbitrarily many `ha`'s.
   Place your answer in the file `a1q2b.txt`.

   (c) Lines that contain nothing but a single occurrence of generalized laughter, which is like ordinary laughter, except that there can be arbitrarily many (but at least one) `a`'s between each pair of consecutive `h`'s. (For example: `Haahahaaaa!`) Place your answer in the file `a1q2c.txt`.

   (d) Lines that contain at least one `a` and at least two `b`'s.
   Place your answer in the file `a1q2d.txt`.

   (e) Lines consisting of a definition of a single C variable of type `int`, without initialization, optionally preceded by `unsigned`, and optionally followed by any single line `//` comment. Example:

   ```
   int  varname;  //  comment
   ```

   You may assume that all of the whitespace in the line consists of space characters (no tabs). You may also assume that `varname` will not be a C keyword (i.e., you do not have to try to check for this with your regular expression). Place your answer in the file `a1q2e.txt`.

3. Write a Bash script called `findGreater` that is executed as:

   ```
   ./findGreater myword file1 file2
   ```

   where `myword` is a sequence of non-whitespace characters, and `file1` and `file2` are names of text files in the current directory. The script prints the name of the file that contains a larger number of lines with the occurrence of the word `myword`. If the files have the same number of lines that contain occurrences of the provided word, the script prints `file1` followed by a single space and then `file2`. In all cases, the script produces a single line of output to standard output. You may assume that the user will call this script correctly; no error checking is needed. Using the provided `file1.txt` and `file2.txt`, the following shows some example executions of the script (lines not starting with a $ are the output produced by the command executed in the previous line):

   ```
   $ ./findGreater thousand file1.txt file2.txt
   file2.txt
   $ ./findGreater be file1.txt file2.txt
   file1.txt file2.txt
   $ ./findGreater Hello file1.txt file2.txt
   file1.txt file2.txt
   $ ./findGreater thought file1.txt file2.txt
   file1.txt
   $ ./findGreater thought file2.txt file2.txt
   file2.txt file2.txt
   ```

4. Scripts are great for automating things that must happen with some frequency. Since this course uses git to make course material available to students, we can write a script to automate the "`pulling`" of any new material that might have been pushed to the course repository. In this question, you will set up a `crontab` job to execute nightly. The job will execute a bash script that pulls any new material.

   Write a bash shell script called `updateRepo` that will do a `git pull` on the repository. Remember, that the `pull` command will only work if you are *within the git repository directory*. Make sure that you change the file permissions on `updateRepo` to be executable. Test your script by executing it, as in:

```
./updateRepo
```

Once you are satisfied that it is working correctly, you will create a local `crontab` job to execute `updateRepo` nightly. Read `http://www.adminschoice.com/crontab-quick-reference` for a description of how to set up a `cron` command and some examples.

In order to prevent the servers from being overloaded, the time you specify for your crontab job execution will be decided by the starting character of your userid and the following table:

First letter of userid	*Start time*
a-d	10:00pm
e-h	10:15pm
i-l	10:30pm
m-p	10:45pm
q-t	11:00pm
u-w	11:15pm
x-z	11:30pm

As an example, if your userid is "jsmith", your crontab job will start at 10:30pm.

Use the command `"crontab -e"` to create your local crontab job to run `updateRepo` every night at the time specified by the table during the months May to June. When the `crontab` job runs, you will receive an email with the results of running `updateRepo`. (For test purposes, set the time to something in the next few minutes to see if it works.)

Note: You should also create a file `README.txt` file that contains the name of the machine you were logged into when you set up your crontab job. You can do this via `echo $HOSTNAME > README.txt`. While you need not submit this file, you will need to know the exact machine you created the crontab job on so that you can remove it once the term is over.

Submit a `zip` file named `cron.zip` containing the following three files:

(a) Your `updateRepo` shell script.

(b) A file named `crontab.txt` that contains the command you entered in your `crontab` file.

(c) A file named `email.txt` which contains a copy of the email that you received when the cron job executed.

Testing tools

The scripts you write in the following questions will be useful every time you write a program. Be sure to complete them!

In this course, you will be responsible for your own testing. As you fix bugs and refine your code, you will very often need to rerun old tests; to check that existing bugs have been fixed, and to ensure that no new bugs have been introduced. This task is greatly simplified if you take the time to create a formal test suite, and build tools to automate your testing.

Questions 5 to 7 enable you to automate this testing process. In particular,

- In Q5, you will create a script that will repeatedly execute **a pre-compiled correct solution** to a problem (provided by the course staff) with different arguments to obtain the expected output. In CS136, the expected output was stored in `.expect` files and had to be manually created. In CS246, we use the extension `.out`.
- In Q6, you will create a script that will repeatedly execute **your solution** to a problem with different arguments. The script will compare your program's output to the expected output and let you know which tests pass/fail.
- In Q7, you will generalize the scripts written in Q5 and Q6 so that programs that expect input from standard input can be executed by redirecting input from test files with extension `.in` (same as in CS136).

In summary, once you have implemented these scripts, you would have implemented for yourself the IO-Test functionality that Seashell supports.

5. Write a Bash script called `produceOutputs` that is invoked as follows:

```
./produceOutputs suite-file program
```

The argument `suite-file` is the name of a file containing a list of filename stems (more details below), and the argument `program` is the path of a program to be run.

The `produceOutputs` script runs `program` on each test in the test suite and, for each test, creates a file that contains the output produced for that test.

The file `suite-file` contains a list of stems, from which we construct the names of files containing the command line arguments used by each test. Stems will not contain spaces. For example, suppose our suite file is called `suite.txt` and contains the following entries:

```
test1 test2
reallyBigTest
```

Then our test suite consists of three tests. The first one (`test1`) will use the file `test1.args`. The second one (`test2`) will use the file `test2.args`. The last one (`reallyBigTest`) will use the file `reallyBigTest.args`.

A sample run of `produceOutputs` would be as follows:

```
./produceOutputs suite.txt ./myprogram
```

The script will then run `./myprogram` three times, once for each test specified in `suite.txt`:

- The first time, it will run `./myprogram` with command line arguments provided to the program from `test1.args`. The results, captured from standard output, will be stored in `test1.out`.
- The second time, it will run `./myprogram` with command line arguments provided to the program from `test2.args`. The results, captured from standard output, will be stored in `test2.out`.
- The third time, it will run `./myprogram` with command line arguments provided to the program from `reallyBigTest.args`. The results, captured from standard output, will be stored in `reallyBigTest.out`.

In this course, we use files with an extension of `.out` to represent output files. Their purpose is the same as the `.expect` files that you created/used while using Seashell's IO-Test functionality in CS136.

Note that if the test suite contains a stem but a corresponding `.args` file is not present, the program is run without providing any command line arguments.

Your script must also check for incorrect number of command line arguments to `produceOutputs`. If such an error condition arises, print an informative error message to standard error and abort the script with a nonzero exit status.

6. Create a Bash script called `runSuite` that is invoked as follows:

```
./runSuite suite-file program
```

The argument `suite-file` is the name of a file containing a list of filename stems (more details below), and the argument `program` is the path of the program to be run.

In summary, the `runSuite` script runs `program` on each test in the test suite (as specified by `suite-file`) and reports on any tests whose output does not match the expected output.

The file `suite-file` contains a list of stems, from which we construct the names of files containing the command line arguments and expected output of each test. Stems will not contain spaces. For example, suppose our suite file is called `suite.txt` and contains the following entries:

```
test1 test2
reallyBigTest
```

Then our test suite consists of three tests. The first one (`test1`) will use the file `test1.args` to hold its command line arguments, and assumes that `test1.out` holds its expected output. The second one (`test2`) will use the file `test2.args` to hold its command line arguments, and assumes that `test2.out` holds its expected output. The last one (`reallyBigTest`) will use the file `reallyBigTest.args` to hold its command line arguments, and assumes that `reallyBigTest.out` hold its expected output.

A sample run of `runSuite` would be as follows:

```
./runSuite suite.txt ./myprogram
```

The script will then run `./myprogram` three times, once for each test specified in `suite.txt`:

- The first time, it will run `./myprogram` with command line arguments provided to the program from `test1.args`. The results, captured from standard output, will be compared with `test1.out`.
- The second time, it will run `./myprogram` with command line arguments provided to the program from `test2.args`. The results, captured from standard output, will be compared with `test2.out`.
- The third time, it will run `./myprogram` with command line arguments provided to the program from `reallyBigTest.args`. The results, captured from standard output, will be compared with `reallyBigTest.out`.

Note that if the test suite contains a stem but a corresponding `.args` file is not present, the program is run without providing any command line arguments.

If the output of a given test case differs from the expected output, print the following to standard output (assuming test `test2` failed):

```
Test failed: test2
Args:
(contents of test2.args, if it exists)
Expected:
(contents of test2.out)
Actual:
(contents of the actual program output)
```

with the `(contents ...)` lines replaced with actual file contents, as described. The literal output `Args:` should appear, even if the corresponding file does not exist.

Follow these output specifications *very carefully*. You will lose a lot of marks if your output does not match them. If you need to create temporary files, create them in `/tmp`, and use the `mktemp` command to prevent name duplications. **Also be sure to delete any temporary files you create in `/tmp`.**

Note: Do **NOT** attempt to compare outputs by storing them in shell variables, and then comparing the shell variables. This is a very bad idea, and it does not scale well to programs that produce large outputs. We reserve the right to deduct marks (on this and all assignments) for such bad solutions.

You can get most of the marks for this question by fulfilling the above requirements. For full marks, your script must also check for the following error conditions:

- incorrect number of command line arguments to `runSuite`
- missing or unreadable `.out` files (for example, the suite file contains an entry `xxx`, but `xxx.out` doesn't exist or is unreadable).

If such an error condition arises, print an informative error message to standard error and abort the script with a nonzero exit status.

7. In this question, you will generalize the `produceOutputs` and `runSuite` scripts that you created in problems 5 and 6. As they are currently written, these scripts cannot be used with programs that take input from standard input. For this problem, you will extend `produceOutputs` and `runSuite` so that, in addition to (optionally) passing command line arguments to the program being executed, the program can also be (optionally) provided input from standard input. The interface to the scripts remains the same:

```
./produceOutputs suite.txt ./myprogram
./runSuite suite.txt ./myprogram
```

The format of the suite file remains the same. But now, for each `testname` in the suite file, there might be an optional `testname.in`. If the file `testname.in` is present, then the script (`produceOutputs` or `runSuite`) will run `myprogram` with the contents of `testname.args` passed on the command line as before and the contents of `testname.in` used for input on stdin. If `testname.in` is not present, then the behaviour is almost identical to problem 5/6 (see below for a difference): `myprogram` is run with command line arguments coming from `testname.args` with nothing supplied as input on stdin.

The output of `runSuite` is changed to now also show the input provided to a test if the test failed. Assuming test `test2` from Q6 failed, the output generated by the updated `runSuite` is as follows:

```
Test failed: test2
Args:
(contents of test2.args, if it exists)
Input:
(contents of test2.in, if it exists)
Expected:
(contents of test2.out)
Actual:
(contents of the actual program output)
```

with the `(contents ...)` lines replaced with actual file contents, as described. The literal output `Args:` and `Input:` should appear, even if the corresponding files do not exist.

All error-checking that was required in problems 5 and 6 is required here as well.

(a) Modify `produceOutputs` to handle input from standard input

(b) Modify `runSuite` to handle input from standard input

Note: To get this working requires minor changes to your solution to problems 5 and 6.

Submission:
The following files are due at Due Date 1: `a1q1a.txt`, `a1q1b.txt`, `a1q1c.txt`, `a1q1d.txt`, `a1q1e.txt`, `a1q1f.txt`, `a1q1g.txt`, `a1q1h.txt`, `a1q1i.txt`, `a1q1j.txt`, `a1q2a.txt`, `a1q2b.txt`, `a1q2c.txt`, `a1q2d.txt`, `a1q2e.txt`.

The following files are due at Due Date 2: `findGreater`, `cron.zip`, `produceOutputs`, `runSuite`, `produceOutputs`, `runSuite`.