CS246—Assignment 1 (Spring 2015)

Due Date 1: Wednesday, May 20, 04:55pm
Due Date 2: Wednesday, May 27, 04:55pm

Questions 1 and 2 are due on Due Date 1; the remainder of the assignment is due on Due Date 2.

1. Provide a Unix command line to accomplish each of the following tasks. Your answer in each subquestion should consist of a single pipeline of commands, with no separating semicolons (;). Before beginning this question, familiarize yourself with the Unix commands outlined on the Unix handout. Keep in mind that some commands have options not listed on the sheet, so you may need to examine some man pages. Do not attempt to solve these tasks with `find`.

   (a) Print the (non-hidden) contents of the current directory in reverse chronological order using time modified (most recently modified last).
   Place your command pipeline in the file `a1q1a.txt`.

   (b) Print the last 5 lines that contain the string `cs246` from the text file `myfile.txt`. If less than 5 lines contain the string, print all lines.
   Place your command pipeline in the file `a1q1b.txt`.

   (c) Print the number of lines in the text file `myfile.txt` that contain the string `student.cs.uwaterloo.ca` where each letter could be either uppercase or lowercase.
   Place your command pipeline in the file `a1q1c.txt`.

   (d) Print all (non-hidden) files in any non-hidden *subdirectory* of the current directory that end with `.h` (immediate subdirectories only, not subdirectories of subdirectories).
   Place your command pipeline in the file `a1q1d.txt`.

   (e) Print a listing, in long form, of all non-hidden entries (files, directories, etc.) in the current directory that are writable by at least one of owner, group, other (the other permission bits could be anything). Place your command pipeline in the file `a1q1e.txt`.

   (f) Print the first 10 lines from `program.c` that start with a C #include directive, followed by at least one whitespace (where whitespace is the space character not tab) and followed by a filename that is wrapped in double quotes. We make the simplifying assumption that the filename within quotes is any non-zero sequence of all possible characters excluding double quote. Examples:
   `#include "prog.h"` : is valid.
   `#include  "bla"` : is valid despite extra spaces between #include and filename
   `#include "bla bla`: is not valid due to missing closing quote
   `#include ""` : is invalid, filename should be a non-zero sequence of characters
   `#include "ab"c"` : is invalid filename cannot contain "
   Place your command pipeline in the file `a1q1f.txt`.

   (g) For this question, assume that file `myfile.txt` contains the result of executing the `history` command. Using `myfile.txt`, print the 10th to 15th lines listed (both inclusive). Note, the first line of output may not be numbered one. Place your command pipeline in the file `a1q1g.txt`.

   (h) For this question, assume that file `myfile.txt` contains the result of executing the `history` command. Using `myfile.txt`, print the number of times you have executed

each unique command in your history sorted from most to least frequent. The count and command should be separated by a single space. Leading spaces are fine. Place your command pipeline in the file `a1q1h.txt`.

(i) In file `abc.txt`, find all lines containing all vowels (`aeiou`) at least once, in any order. Place your command pipeline in the file `a1q1i.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. If your pattern contains special characters, enclose it in quotes.

   (a) Lines that contain both cs246 and cs247 in any order (both the `c` and `s` must be in lower case). Place your answer in the file `a1q2a.txt`.

   (b) Lines that contain an occurrence of `<h1>`, followed eventually by an occurrence of `</h1>`. Place your answer in the file `a1q2b.txt`.

   (c) Lines that contain 2015 but no other numbers. Place your answer in the file `a1q2c.txt`.

   (d) Lines that contain nothing but a single occurence of laughter, where laughter is defined as a string of the form `Hahahahahahahahahahahahaha!`, with arbitrarily many `ha`'s. Place your answer in the file `a1q2d.txt`.

   (e) 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 `a1q2e.txt`.

   (f) Lines whose every even character is a digit with the first character in the line considered character one. Place your answer in the file `a1q2f.txt`.

   (g) Print lines containing the string `red` or `blue` surrounded by a matched set of round parenthesis i.e. using the '(' and ')' parenthesis. Example: `Hello (red) World!` Place your answer in the file `a1q2g.txt`.

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

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

   You may assume that all of the whitespace in the line consists of space characters (no tabs) and that an arbitrary amount of whitespace is allowed as per rules of the C language. 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 `a1q2h.txt`.

3. You've been hired as a coop student by the `UW School of Computer Science` to observe the school's administrative processes and find ways to streamline as an objective outsider. You note that every term, the CS course notes sold at `Media Services` are retrieved and stored in Sharepoint. The problem is that the archival site is a flat file space[1], so every file must have a unique name in order to not overwrite already existing files. The files that Media Services provide are named in the form "CS [0-9][0-9][0-9].pdf" (note the space after CS), and administrative staff have to manually rename each of the files to the form "[WSF][0-9][0-9]_CS[0-9][0-9][0-9]_Notes.pdf" before uploading them. For example, the file "CS 116.pdf" from Winter 2015 would be renamed "W15_CS116_Notes.pdf". As a recent graduate of `CS246`, you know that time could be saved by automating the process using a shell script.

[1] http://whatis.techtarget.com/definition/flat-file-system

Write a bash shell script named `renameCourseNotes` that performs the following actions:

(a) Use the `date` command to determine the appropriate term. Assume that the script will be run before the end of the term in which the notes were sold, to keep things simple. If you look at the manual page for `date`, you will find the section on user-defined format strings such as '+%y' useful. Store this information in a variable that can be used as part of the renaming process.

(b) Loop through all of the files from Media Services (and only them). In other words, do not rename any other files unless their name follows the described format.

(c) Rename the selected files to be in the appropriate form. There are several ways to do it, using either features specific to bash, or utilities such as `sed/tr`. Note that the space between the subject "CS" and the course catalog number must be removed as part of the process.

Remember to be extremely careful in the use of your quotation marks since this will have an impact on your code.

Submit your bash shell script, `renameCourseNotes`.

4. As a diligent CS246 student, you have decided that you want to make sure that you do not miss any changes that get made to the materials published in the course git repository. You will set up a crontab job to execute nightly at 11:55pm. The job will invoke a bash script to retrieve any updates to the repository that you have cloned into your local cs246 directory. This means that, instead of manually entering your userid and password, you will have to create a public encryption key to store in your git profile. You can then use the key to retrieve changes. You will need to take the following steps first:

(a) Create a public RSA encryption key using the command:

```
ssh-keygen -t rsa -C "jsmith@uwaterloo.ca"
```

where "jsmith" is replaced by your UW userid. You will be prompted for the directory path in which the file will be created. Use the default location by just pressing RETURN. When prompted for a passphrase, leave it empty by just pressing RETURN. Note: if you end up setting a non-empty passphrase, then the automated pull will not work as the script will ask for the passphrase.

(b) You should now have a `.ssh` directory in your home directory that contains, among other files, a file called `id_rsa.pub`. Use the `cat` command to display the contents of the file to the screen. For example:

```
cat ~/.ssh/id_rsa.pub
```

(c) Copy the contents of the file to your clipboard using the mouse and the copy command appropriate to your operating system.

(d) Open a browser and navigate to `https://git.uwaterloo.ca`.

(e) Login with your Quest userid and password in the LDAP panel.

(f) Click on the profile settings icon that resembles the upper part of a person. This should bring up your profile settings. In the left-hand pane is the SSH Keys option. Click on that link.

(g) This should bring up a screen with two empty boxes. The top one is labelled `Title` and the bottom one is labelled `Key`.

(h) Paste your key into box labelled `Key` and press the `"Add key"` button. If you have been successful, you should see "your-userid@uwaterloo.ca"" as the `Title` and your `Key` reformatted, showing its fingerprint and creation date.

(i) Now we must update the URL for the remote repository to use your SSH Key rather than ask for a password every time we pull. Go back to your UNIX session, change directories to your cs246 repository and execute the following command:

```
git remote set-url origin ssh://gitlab@git.uwaterloo.ca/cs246/1155.git
```

Note: this only has to be done once since we are telling git to use the ssh protocol for authentication. Subsequently, any calls to `"git pull"` will use this url and protocol.

(j) Test that everything is configured correctly by entering the following command from within your cs246 repository:

```
git pull
```

Note: you should no longer have to type in a username/password.

You should either see a message indicating that the repository has been updated, or that no changes needed to be made.

Now, write a bash shell script called `updateRepo` that will do a pull on the repository as described in the previous step. Remember, that the pull command only works 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.

Use the command `"crontab -e"` to create your local crontab job to run `updateRepo` every night at `11:55pm` during the months May to June. When the `crontab` job runs, you should receive email showing the results of running `updateRepo`. (For test purposes, you should try setting the time to be something in the next few minutes, just 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.

5. **Note: the script you write in this question will be useful every time you write a program. Be sure to complete it!** 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 a tool to automate your testing. In this question, you will implement such a tool as a Bash script.

   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 full 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 input and expected output of each test. 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.in` to hold its input, and `test1.out` to store its expected output. The second one (`test2`) will use the file `test2.in` to hold its input, and `test2.out` to store its expected output. The last one (`reallyBigTest`) will use the file `reallyBigTest.in` to hold its input, and `reallyBigTest.out` to store 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 standard input redirected to come from `test1.in`. The results, captured from standard output, will be compared with `test1.out`.
- The second time, it will run `./myprogram` with standard input redirected to come from `test2.in`. The results, captured from standard output, will be compared with `test2.out`.
- The third time, it will run `./myprogram` with standard input redirected to come from `reallyBigTest.in`. The results, captured from standard output, will be compared with `reallyBigTest.out`.

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
Input:
(contents of test2.in)
Expected:
(contents of test2.out)
Actual:
(contents of the actual program output)
```

with the `(contents ...)` lines replaced with actual file contents, as described. **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`.**

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
- missing or unreadable `.in` or `.out` files (for example, the suite file contains an entry `xxx`, but either `xxx.in` or `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.

6. **Note: the script you write in this question will be useful every time you write a program. Be sure to complete it!** In this question, you will start with the `runSuite` script that you created in problem 5, and generalize it. As it is currently written, `runSuite` only works for programs that take their input on stdin; it cannot be used with programs that take parameters on the command line. For this problem, you will enhance `runSuite` so that it can pass command line arguments. The interface to runSuite remains the same:

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

   The format of the suite file remains the same. But now, for each `testname` in the suite file, there will be files `testname.in`, `testname.out`, and an optional third file `testname.args`. If the file `testname.args` is present, then `runSuite` will run `myprogram` with the contents of `testname.args` passed on the command line and the contents of `testname.in` used for input on stdin. If `testname.args` is not present, then the behaviour is identical to problem 5: `myprogram` is run without arguments, and `testname.in` still supplies the input on stdin. The file `testname.out` is used identically to the way it was used in problem 5, and the output of `runSuite` should also be identical to the way it appeared in problem 5. All of the error-checking that was required in problem 5 is required here as well.

   Note: To get this working should require only very small changes to your solution to problem 5.

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`, `a1q2a.txt`, `a1q2b.txt`, `a1q2c.txt`, `a1q2d.txt`, `a1q2e.txt`, `a1q2f.txt`, `a1q2g.txt`, `a1q2h.txt`.

The following files are due at Due Date 2: `renameCourseNotes`, `cron.zip containing updateRepo, crontab.txt and email.txt`, `runSuite`, `runSuite`.