CSC B07 Assignment 2A
Due Date: November 10th @ 11:59pm

Introduction:

In Assignment1, you and your partner used CRC cards to design a set of classes for maintaining a mock file system and interacting with it in a program that operates like a Unix shell. You also did some minimal programming in order to get used to sharing code using an SVN repository.

In Assignment2, you will be working in a team of 4. If someone in your team drops the course, this assignment is also doable in teams of 3. The design requirements for Assignment2A are quite similar to those of Assignment1, although we have modified the commands a bit. There will be two parts to Assignment2. This is Assignment2A or the first part of Assignment2. Read this document carefully. I understand some of the workings of the commands may be vague, however, please use my office hours to get your requirements understood clearly. Let us meet on Teams, and I want you all to take advantage of this, so that I can work with you on your design and help you understand the requirements.

Do not create commands that I have not asked for and do not add new functionality to existing commands that have not been asked for. For instance mkdir takes in as input a single DIR parameter. Do not have mkdir take in a list of DIR to create. If you are not sure about this, it is your responsibility to ask me and get this clarified. This is important, and you will be penalized if you fail to obey this.

Please make sure to get familiar with the readMe files that I have placed in your repos.

Commands:

In all of the following commands, there may be any amount of whitespace (1 or more spaces and tabs) between the parts of a command.

For example (all the **four** variations are correct and valid):

```
/#: mv someFile1 someFile2
/#: mv someFile1 someFile2
/#: mv someFile1 someFile2
/#: mv someFile1 someFile2
```

Furthermore, your program must not crash. Square brackets indicate an optional argument.

Clarification: … indicates a list.

Clarification: every *PATH*, *FILE*, and *DIR* may either be *relative path* or a *full path (absolute path).*

exit

Quit the program

mkdir *DIR1 DIR2*

This command takes in two arguments only. Create directories, each of which may be relative to the current directory or may be a full path. If creating DIR1 results in any kind of error, do not proceed with creating DIR 2. However, if DIR1 was created successfully, and DIR2 creation results in an error, then give back an error specific to DIR2.

cd *DIR*

Change directory to DIR, which may be relative to the current directory or may be a full path. As with Unix, **..** means a parent directory and a **.** means the current directory. The directory must be /, the forward slash. The foot of the file system is a single slash: /.

ls *[PATH ...]*

If no paths are given, print the contents (file or directory) of the current directory, with a new line following each of the content (file or directory).

Otherwise, for each path p, the order listed:

- If p specifies a file, print p
- If p specifies a directory, print p, a colon, then the contents of that directory, then an extra new line.
- If p does not exist, print a suitable message.

pwd

Print the current working directory (including the whole path).

pushd *DIR*

Saves the current working directory by pushing onto *directory stack* and then changes the new current working directory to DIR. The push must be consistent as per the LIFO behavior of a stack. The pushd command saves the old current working directory in *directory stack* so that it can be returned to at any time (via the popd command). The size of the *directory stack* is dynamic and dependent on the pushd and the popd commands.

popd

Remove the top entry from the *directory stack,* and cd into it. The removal must be consistent as per the LIFO behavior of a stack. The popd command removes the top most directory from the *directory stack* and makes it the current working directory. If there is no directory onto the stack, then give appropriate error message.

history [number]

This command will print out recent commands, one command per line. i.e.

```
1. cd ..
2. mkdir textFolder
3. echo "Hello World"
4. fsjhdfks
5. history
```

The above output from history has two columns. The first column is numbered such that the line with the highest number is the most recent command. The most recent command is `history`. The second column contains the actual command. **Note:** Your output should also contain as output any syntactical errors typed by the user as seen on line 4.

We can truncate the output by specifying a number (>=0) after the command. For instance, if we want to only see the last 3 commands typed, we can type the following on the command line:

```
history 3
```

And the output will be as follows:

```
4. fsjhdfks
5. history
6. history 3
```

cat FILE1 [FILE2 ...]

Display the contents of FILE1 and other files (i.e. File2 ....) concatenated in the shell. You may want to use three line breaks to separate the contents of one file from the other file.

echo STRING [> OUTFILE]

If OUTFILE is not provided, print STRING on the shell. Otherwise, put STRING into file OUTFILE. STRING is a string of characters surrounded by double quotation marks. This creates a new file if OUTFILE does not exists and erases the old contents if OUTFILE already exists. In either case, the only thing in OUTFILE should be STRING.

echo STRING >> OUTFILE

Like the previous command, but appends instead of overwrites.

man CMD [CMD2 ...]

Print documentation for CMD (s)

==========

Task1:

Argue with your team until you have worked out a new design. Update your CRC cards and commit them or explain why no changes were necessary. Use the same format as last time (i.e. in Assignment1).

Task2:

Write the shell program implementing each of the above mentioned commands follow the SCRUM software development process. Make sure that, by the deadline of Assignment2A, the code you commit has been tested and fully documented. ***Remember that you should try to check in code that can be compiled and executed into your SVN repository.***

In order to do well for this Assignment, you MUST strictly adhere to the SCRUM software development process. Follow the instructions carefully that are mentioned in the **grading scheme** and what I talked about in lecture regarding Assignment2.

We will schedule your demo sometime in sprint 2. More details on this will be provided shortly.