CSE 303, Spring 2009
Homework 2: Retro Grade-It (`bash` scripting)
50 points
Due Friday, April 17, 2009, 11:30 PM

This assignment focuses on `bash` shell scripting and more sophisticated use of common Unix commands. There are two parts to this assignment: a simpler `.bash_profile` login script and a larger grading script.

Turn in two files, one named `.bash_profile` and one named `gradeit.sh`, from the Homework section of the course web site. You will also want the various support files from the Homework section of the course web site. Some support files are `.tar.gz` archives. To extract the files from such an archive, place it into a folder on your Linux/Unix machine, and type `tar -xzvf` ***`filename`***`.tar.gz` .)

In terms of grading, most of the points will come from the correctness of your scripts. Some points will also come from the style, design, and "internal correctness" of your script file. If you use an unnecessarily clunky solution to solve a problem that can be solved in a simpler way, you may lose partial points.

Note: The answers to all questions in this assignment can be found entirely using commands shown in the lecture notes from the first week. You may use other commands if you like, but you should constrain yourself to those from lecture or from the *Linux Pocket Guide* textbook. Ask the instructor if you are unsure whether a particular command is allowed.

Part 1 of 2: Login Script (`.bash_profile`):

As we have discussed in class, `.bash_profile` is a script that runs every time you log in to a Bash shell. For the first part of this assignment, you should create a `.bash_profile` file in your home directory that performs the following:

1. Creates an alias so that when you type `attu`, you will connect to `attu.cs.washington.edu` with SSH. (This alias isn't very useful if you're testing your script on `attu` itself, but set it up anyway.) Also create at least 2 more aliases of your own choosing and put them into your bash profile.
2. Sets it so that when you create files, they are given read/write permission to you (the owner), but no permissions to anyone else. This will keep your homework from being seen by others. (See the `umask` command.)
3. Sets your prompt to a string that includes your user name and the short name of the computer you are on (such as your computer name you set in Ubuntu, or `attu1`, or `attu3`), in exactly the following format: `stepp@attu2$`
4. Lists which of your friends are online and what they are doing. Define a file named `buddylist.txt` in your home directory whose contents are the CSE NetIDs of two or more of your friends in CSE, one ID per line. The file can have as many lines as you like so long as it is at least two. If you don't know anybody in CSE, you may use `stepp` and `reges`. Now modify your `.bash_profile` script to display which of those users are logged in to the system and what they're doing, in sorted order by NetID. For example, if your `buddylist.txt` contains:

```
stepp
bensonl
ln
```

Then a possible output from your `.bash_profile` might be:

```
ln       pts/5    :pts/6:S.1       21:09    2:20m  0.34s  0.28s vim poisson.ml
ln       pts/6    207.108.209.70   20:33    2:15m  0.06s  0.02s screen
ln       pts/8    :pts/6:S.0       20:33    2:15m 11.23s  0.08s bash ./getml.sh
stepp    pts/3    67.160.45.86     00:10    7:46   0.02s  0.04s banner ^_^
stepp    pts/10   67.160.45.86     00:20    8:34   0.02s  0.00s cowsay
```

(Hint: You can get this to work using commands we've already seen, but you will need to read the `man` pages.)

Note that your command should work regardless of the number of user names in `buddylist.txt`. You should not assume that `buddylist.txt` contains exactly 3 names as shown above, etc.

(continued on next page)

Part 2 of 2, Grading Script (`gradeit.sh`):

For the second part of this assignment, you will write a script `gradeit.sh` that "grades" solutions to our last homework assignment (the `commands` file). The basic idea is that you're given a set of students' `command` files, and your script will run each of them one at a time, examining the output to see if it matches the expected output for the assignment. You will also examine the source code to see whether it has enough comments. (The description below is not exactly how you'll actually be graded on your HW1; it's just an exercise.) The output from a run of your program will look like this:

```
$ ./gradeit.sh 50
Retro Grade-It, 1970s version
Grading HW2 with a max score of 50

Processing bensonl ...
bensonl has correct output
bensonl has 7 lines with comments
bensonl has earned a score of 50 / 50

Processing oterod ...
oterod did not turn in the assignment
oterod has earned a score of 0 / 50

Processing reges ...
reges has incorrect output (8 lines do not match)
reges has 3 lines with comments
reges has earned a score of 42 / 50

Processing saptre ...
saptre has correct output
saptre has 2 lines with comments
saptre has earned a score of 45 / 50

Processing stepp ...
stepp has incorrect output (37 lines do not match)
stepp has 1 lines with comments
stepp has earned a score of 8 / 50

Processing toddk4 ...
toddk4 has incorrect output (73 lines do not match)
toddk4 has 5 lines with comments
toddk4 has earned a score of 0 / 50
```

To get started, download the following support files from the course web site:

- **`students.tar.gz`** : This file contains a set of fake student solutions to examine. Extract this file to your HW2 directory with the `tar` command. It will create a set of student folders and files such as:

  ```
  students/bensonl/commands
  students/reges/commands
  students/oterod/CoMaNdZ
  students/toddk4/commands
  ```

  Each student's directory is supposed to contain a `commands` file representing that student's solution to HW1 (though some students have not turned in the file properly, so it may not exist or have the wrong name).

  Note that you are not targeting the specific students shown above; your code should not specifically mention names like `stepp` or `bensonl`. Your code should process all students in the `students` directory.

- **`expected.txt`** : This file contains the expected HW1 output. Download this file into your HW2 folder. This is the text you'll compare to each student's output to see whether it is correct.

- **`testfiles.tar.gz`** : This file contains a set of testing files that your script will extract into each student's directory while "grading" the student. Download this to your HW2 directory, but don't extract its contents yet. (Your script will extract it for each student.) You may assume that no file inside this archive has any spaces in its name or contains the word `commands` as part of its name.

Now you must write a script `gradeit.sh` that gives each student a score on the assignment. Your program should accept the maximum score as a command-line argument. For example, to run it with a max score of 50 points, you would type:

```
$ ./gradeit.sh 50
```

If the user does not pass a value for the maximum points, your script should print the following error message and abort:

```
$ ./gradeit.sh
Usage: ./gradeit.sh MAXPOINTS
```

You do not need to check this argument for validity. If an argument is passed, you may assume it is a positive integer.

If your script is passed an argument, it will examine each student in the `students/` folder, running the student's `commands` file and comparing its output against expected output. The procedure for examining a student is the following:

- Extract the testing files from `testfiles.tar.gz` into the student's folder.
- Run the student's `commands` file, which will examine the newly-extracted test files. (You should run the student's program in such a way that the working directory will be the student's directory containing the test files, since the students wrote their HW1 scripts under that assumption.) You may want to capture the student's output into a file.
- Compare the student's output against the expected output file using `diff`. If the outputs do not match, consider any line of `diff` output containing a `<` or `>` to count as 1 line of unmatched content. Your program should produce output in the following format:

  ```
  bensonl has correct output
  ```

  or:

  ```
  smith has incorrect output (8 lines do not match)
  ```

  For each unmatched line, you should deduct 1 point from the student's score. In other words, if the `diff` output contains 8 lines that have `<` or `>` in them, the student should lose 8 points. If the student has more incorrect lines than there are points in the assignment (such as 60 incorrect lines on a 50-point assignment), the student should receive 0 points. You may assume that the correct output does not contain the `<` or `>` characters.
- Check whether the student has sufficient code comments. Comments are worth 5 points on the assignment. For our purposes, a comment is defined as any line that contains a `#`. A student must have 3 or more lines of comments in his/her `commands` file (including the initial `#!/bin/bash` line). A student that has fewer than 3 lines of comments should lose 5 points from his/her assignment score, down to a minimum of 0. Your script should output how many lines of comments are found, such as:

  ```
  toddk4 has 5 lines with comments
  ```
- Your script should output each student's score on the assignment. If a student `jones` has no differing lines of output and 5 lines of comments in his script, the score output for that student would be:

  ```
  jones has earned a score of 50 / 50
  ```

  If student `davis` has 12 differing lines of output (-12) and 2 lines of comments (-5), the score output would be:

  ```
  davis has earned a score of 33 / 50
  ```
- If the student did not turn in the program or incorrectly named the file, the student gets 0 points on the assignment. If the user `magda` did this, you would output the following message:

  ```
  magda did not turn in the assignment
  ```
- After processing each student, your script must clean up all the test files you had put into the student's folder. To do this you must remove every file or directory other than the student's `commands` file. (If the student turned in a program with the wrong file name, you may delete it.) To do this cleanup, you may want to look into the `find` and `xargs` commands, which make it possible to repeat an action over each of a set of files. Note that the Unix command for removing directories does not work unless the directory is empty; but the command for removing files can be given arguments that will instruct it to remove entire directories and their files at a time.

See the course web site complete sample output runs of the program.

You may want to optionally include your own `commands` solution from HW1 in the data that is processed by this grading script. To do this, convert your `commands` file into a runnable Bash script by putting a proper Bash header at the top of it and making a `students/`***`YOUR_USER_NAME`***`/` directory. You should be able to run your `commands` program and see its output if proper permissions are set. You should make one change to your commands: The answer to problem 3 (listing all songs) should be changed so as not to show the songs in long format.

You may assume that no student's program tries to do anything evil to your computer, such as erasing all your files. You may also assume that students' code will not lock up and get stuck in any sort of infinite loop.

Development Strategy:

This program is hard. Write it in small, testable pieces. Here are some suggested milestones to reach:

- Make your script able to simply output the names of all of the students to be processed.
- Make your script able to simply run each student's `command` program. You can even place the testing files into the student's folder ahead of time rather than having the script do it, while you're still developing your code.
- Some tasks in your script will involve running Unix commands and capturing their output with back-ticks, ` `. Since such commands run silently as their output is being captured, consider simply running the command first without back-ticks to see that the command is running properly and producing results that you expect.
- Some tasks can be postponed until the end, such as cleaning up all the test files and checking for comments.
- Be careful when writing the code to clean up files. Since you'll be removing files, it is possible to delete or overwrite an important file in your directory, even your grading script! Make frequent backups and be cautious about the commands you execute. You may want to `echo` dangerous commands before actually running them.
- Start early! There may not be enough lab/office hours to help you if you're lost on the final day of the assignment.

Each Linux/Unix box can be slightly different; for full credit, your commands must be able to work properly either on `attu`, or on the basement lab computers, or on a fresh Ubuntu installation (with Java installed).

Grading:

The majority of your points will come from the correctness of the behavior of your scripts. Part 1 is graded only on correctness. For Part 2, some points will also be awarded on the style and elegance of your code. You should not use a more complex command or control structure when a more simple one would achieve the same result. You should avoid redundancy as much as possible in the limited context of the shell programming language. If a particular value is important to your program as a whole, you should store that value in a well-named variable near the top of the code and use that variable throughout your code, rather than referring directly to the value throughout your code. (This is analogous to the idea of declaring class constants for important values in Java.)

You should format your code nicely with proper indentation, spacing, and clear variable names. You should place a descriptive comment heading at the top of your program as well as brief comments throughout your script explaining what each part of the script is doing. For reference, our solution to Part 2 is around 65 lines long (40 lines long if you exclude blank lines and comments), though you do not need to match this exactly to get full credit; it is just a rough guideline.