CSC207 Lab 1 — Unix Shell and Git

To earn lab marks, you must arrive on time and actively participate for the entire session. Make sure that your TA checks you off on the attendance sheet.

1 Overview

This week, you are going to work with Git using a UNIX command line shell. Before you start, look over the Useful Unix and Git commands; you will be using them throughout the lab.

2 Choose a driver and navigator

In these labs, you are encouraged to work in pairs. You and your partner together will be able to figure out problems better than you would individually. Find yourself a partner. If you have trouble finding one, let your TA help you. This partnership is only for today's lab, unless you want to continue it.

We strongly advise you to form a partnership with a classmate who has a similar level of background. For example, if this is your first time working in a Linux environment, we suggest that you partner with someone who is also new to Linux.

In all the labs, we will use the terms *driver* and *navigator*. Here are the definitions of the two roles:

- **Driver:** Types at the keyboard. Focuses on the immediate task at hand.
- **Navigator:** Thinks ahead and watches for mistakes.

In lab handouts, we'll often refer to you as `s1` and `s2`, and `s1` will be the first driver.

Tip: If you are new to the computer science teaching labs, you should be `s1` to get practice logging in while `s2` helps.

3 Log in and get things set up

`s1` drives and `s2` navigates.

You can't both be logged in to the same machine at once, so let `s1` be the person to log in. If necessary, `s2` should offer assistance with this, but here is a very important rule for this and all other labs: **Make sure neither of you learns the other's password while you work through this process!**

Also, during this lab, `s2` does *not* log in and does *not* check out a repository. Both of you work *in `s1`'s account on `s1`'s git repository.* If you are `s2` and wish to try this exercise in your own account, you can do so after the lab.

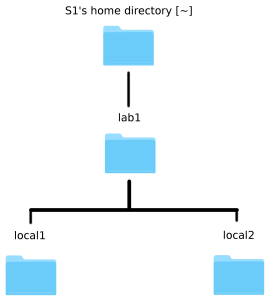
After logging in, open a new terminal window.

You need to use some basic Unix commands in order to complete this lab. These commands are listed on the second handout and the navigator should look them up.

1. Change to `s1`'s home directory. (You are probably already there — find out by typing `pwd`.)
2. Create a directory called `lab1`. (Using `mkdir`.)

3. Create two directories named `local1` and `local2` inside `lab1`. These two directories will be used later for checking out local copies of `s1`'s repository.

At this point, the directory structure should be as follows (there may be other files and directories in `s1`'s home directory that are not shown below):



4 Git (`git` Setup)

Git wants to know your preferred editor, since it will put you in that editor when the time comes to type in a log message. Tell Git to use `nedit` as the editor. We will also tell git who `s1` is. `s2` should repeat this step on their account later. issue the following commands, replacing the relevant information:

```
git config --global core.editor nedit
git config --global user.name "Your Name Here"
git config --global user.email your@email.here
```

Each of you has a git repository that we created for use in this lab.
For subsequent steps, you'll need to know the URL for `s1`'s repository. It will have this form:

```
https://markus.teach.cs.toronto.edu/git/csc207-2017-05/s1username
```

Now you are ready to use Git.

1. Make the directory `lab1/local1/` your current directory. Change your directory to `lab1/local1/`. (Hint: use the `cd` command.)
2. Now, run `git clone [URL]` to get a local copy of the repository and also a working copy in the `s1username` directory. `s1`'s `teach.cs` password will be required.
3. Before doing anything else, run `git status`. You should see this message:

   ```
   fatal: Not a git repository (or any of the parent directories): .git
   ```

   This is because you are currently in `lab1/local1/`, rather than in the repository you just cloned.
4. Change directory to that `s1username` directory. This is the top directory in your repository. Run `git status` again. You should see this:

   ```
   On branch master
   Your branch is up-to-date with 'origin/master'.
   nothing to commit, working tree clean
   ```

5. Using a text editor, create a file named `myfile.txt`, for example, by running

```
nedit myfile.txt
```

and saving the file. **This will open a new window. Once you close it, the focus will switch back to the terminal window.**

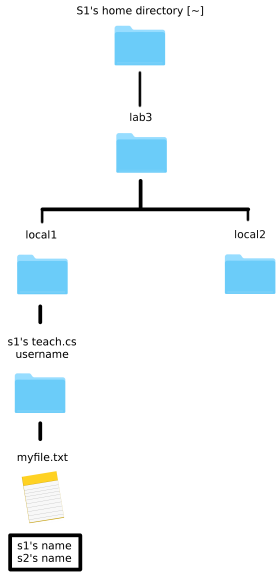
6. Run `git add myfile.txt` and then run `git status`. Read the output and make sure you undersand it.

7. Commit your changes by typing `git commit`, then run `git status`. (Later, you can use the `-m` flag to specify the commit message, but try it without first.)

8. Continue editing `myfile.txt`: put the names of `s1` and `s2` (and anyone else you are working with) one per line in the file. Commit the changes to the repository.

9. Run `git status`.

   The directory structure should now be:



10. Remember that, in Git, commiting changes only affects the local repo. At this point, run a `git push` command to propagate the changes to the remote repo.

5 A second local working copy, and a conflict

You are going to check out a second copy of the repository, create a conflict, and resolve that conflict.

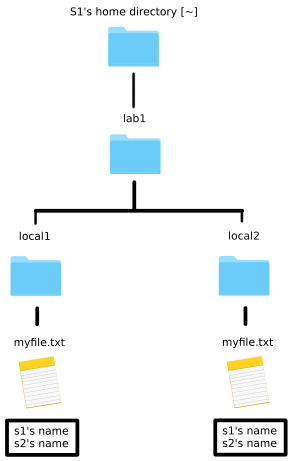
1. **Switch roles: `s2` drives and `s1` navigates.**

2. Open another terminal window. (Note that you are still working within `s1`'s account.)

3. Change to the directory `local2` and clone exactly as you did before.

   Note: we are still in `s1`'s account, and we have two local copies of the repository. The local copies are in sync.

The directory structure should now be:



4. Edit `myfile.txt` under `local2` by adding `s2`'s favourite movie and favourite flavour of ice cream on two new lines at the end of the file, but *leave the group member names alone*. Commit the changes to your local repo, but do **not** push the changes to the remote repo!

   Switch roles: `s1` drives and `s2` navigates.

5. pull to the working copy in `local1`. Since you just changed `myfile.txt` in `local2` and committed the new version to the repository, you should not see any changes in `local1`.

6. return to `local2` and push the changes. Then return to `local1`, and pull again. The changes should now be present.

7. Now, you will create a set of conflicting changes. To do so, edit `myfile.txt` again in `local1`, change the ice cream flavour, and commit and push the changes.

8. Before pulling into `local2`, **switch roles (`s2` drives and `s1` navigates)** and have `s2` change *the same line with the ice cream flavour* in `myfile.txt` inside `local2`. Choose a different flavour, so that there will be a conflict.

9. Try to commit the changes. This should succeed.

10. Now pull. This should succeed but result in a conflict.

    Edit the file `myfile.txt` to remove the conflict. Afterward, commit the changes and then push the result. This should succeed. You have now merged and resolved a conflict.

6 A Final Optional Challenge

1. **Switch roles: `s1` drives and `s2` navigates.**

2. In the terminal window containing `local2`, display the status of the files and directories in this local copy. Use the `status` and `diff` commands to see where the local copy is out of date with the master repository.

3. Look at the revision history of `myfile.txt` using the `git log --follow -p -- myfile.txt` command.

4. What needs to be done now to make sure `local1` and `local2` both contain the latest version of `myfile.txt`? Complete the necessary steps.