Lab 9: Using git

Lab sessions Tue Dec 05 to Thu Dec 07

Lab written by Chris Gregg

Learning goals

After this lab, you will have

1. investigated your assignment folders using `git`
2. created a `git` repository with your partner, and worked on the repository together.
3. practiced some `git` commands, and learned how to handle some typical `git` situations.

Share a computer with somebody new. Discuss your favorite part of the guest lecture on Monday.

Get started

Clone the lab9 repo by using the command below.

```
git clone /afs/ir/class/cs107/repos/lab9/shared lab9
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab9) .

Lab exercises

1) Investigate your assignment `git` directories

Both partners should take a turn at the keyboard for this exercise. For the entire quarter, your assignments have been stored in `git` repositories, although most of the interaction with the repositories was handled behind the scenes through the `tools/sanitycheck` and `tools/submit` scripts. Now, you are going to get a chance to investigate your previous assignments using some of the functionality that `git` provides.

- One partner should log into a `myth` machine, and change into the assign2 directory (mywhich / scan_token / etc.), e.g.,

  ```
  $ cd ~/cs107/assign2
  ```

- Run the following commands, and answer the questions below on the checkoff form:

  ```
  $ git log
  (type `q` at the `:` prompt to stop the listing, or the spacebar to conti
  nue to scroll through the messages)
  ```

- Based on the git log, what was the last date that you submitted your assignment?

  ```
  $ git log --grep="submit"
  ```

- Notice that you have a list of all the commits which were generated when you submitted your assignment. If you want to just see a list of the commit messages that say "submit", you can use the following:

  ```
  $ git log | grep submit
  ```

- The difference between the two ways to run grep are that the first is an internal `git` command, and the second simply uses the Linux grep program to find only the lines that match the message.

- How many times did you submit your assignment?

  ```
  $ git log | grep sanity
  ```

- How many times did you run sanitycheck? Too many to count? Try:

  ```
  git log | grep sanity | wc -l
  ```

- Make a change in an editor to the `code.c` file. Then type:

```
$ git status
```

- Notice the `modified: code.c` line that tells you that your file has been modified.

```
$ git commit –m "This is a commit message, and I've made a change to cod
e.c" code.c
$ git push
$ git log
```

- Note that the log now has your latest commit. Does it also show your latest push?
- Perform another `git log` and choose a commit that is a day or two before your final submission. The long hex number after "commit" is the *hash*. Type the following, and replace the sample hash below with the commit hash you chose (use `q` to quit from the listing, or the spacebar to continue to scroll):

```
$ git diff b5d5335265b41614dfda5a7b02f0d2164bec4e42
```

- The lines that begin with `–` are lines that were in the older version, and the lines that begin with `+` are lines that have been added since then.
- Now type (again, replace the sample hash with your hash):

```
$ git checkout b5d5335265b41614dfda5a7b02f0d2164bec4e42
```

- You have now "checked out" an older revision of your code. You can view the files in an editor, but you probably shouldn't change them in this state. If you do want to use those changes, it is best to make a copy of the file, then revert back to the latest revision, as follows:

```
$ git checkout master
```

- Now you are back to the latest state of your repository.

2) Create and work with a shared `git` repository.

In this part of the lab, you and your partner will both work on the same repository and you will both update files. You will investigate what happens when two users try to make changes to different files in a repository, and when they try to make changes to the same file. You should open two terminals on the lab computer (or one person can be on a laptop and the other on a lab computer, **but they must be logged into the same myth machine, e.g., myth3**), and each person should log in to a myth machine with their own usernames.

One of you should be `student 1` below, and the other should be `student 2`. You need to pick a team name (no spaces, but underscores are okay). Once you have picked a team name, replace `team_name` below with your team name.

Don't worry about all of the details of the `git` commands -- the more you use `git`, the more you will start to learn them (or you'll just look them up online when you need the initialization ones, for instance).

- `student 1` should start in their terminal, and should type the following commands, which create a shared repository and then pushes some initial files into it:

```
samples/create_repo
cd our_repo
git init
git add *
git commit –m "first commit"
git remote add origin /tmp/team_name
git push –u /tmp/team_name master
chmod –R a+rwx /tmp/team_name

(note: if you accidentally forget to type the actual team name when you t
ype the remote add origin command, fix it by typing: git remote set-url o
rigin /tmp/the_actual_team_name)
```

- `student 2` should type the following to clone the repository:

```
mkdir -p ~/cs107/lab9 && cd ~/cs107/lab9
git clone /tmp/team_name our_repo
```

- `student 2` should continue by making a change to `file1.txt` with an editor, and then typing:

```
git status
git commit -m "updated file 1" file1.txt
git push
```

- `student 1` should then type:

```
git pull
git log
```

- `student 1` should edit `file2.txt` and make some change, and then continue:

```
git commit -m "changed file 2" file2.txt
git push
```

- `student 2` should edit file 1 (again) and then type:

```
git commit -m "added another change to file1" file1.txt
git push
```

- at this point, `student 2` should get an error! This happens when a user tries to push to a repository when another user has already pushed, so the changes aren't synchronized between the two users. In this case, `student 2` can simply pull and then push, and `student2` will be required to add a "merge" message in an editor:

```
git pull
git push
```

- Now, `student 2` should modify `README.md` and then:

```
git commit -m "updated readme file" README.md
git push
```

- `student 1` should *also* modify `README.md` , and then:

```
git commit -m "added some things to the readme" README.md
git push
```

- This results in another error, and `student 2` should attempt to pull:

```
git pull
```

- At this point, there is a "merge" error! This happens when two repository users make modifications to the same file, and `git` cannot figure out how to non-destructively merge them into one file. `Git` modifies the file in question, and adds `<<<<<< HEAD` and `>>>>>>> hash` comments to deliniate where the merge issues are. If there were a lot of changes, this can be ugly to manually merge, but you have to do the best you can. `student 1` should clean things up, and then do the following:

```
git add README.md
git commit -m "fixed up the readme in a merge"
```

Answer the following questions on the checkoff sheet:

- Why is it always wise to `git pull` before a `git push` ?
- Given the annoyance with the merge conflict, what do you imagine is a typical agreement between team members working on the same repository?

3) Learn the `git bisect` command, and how to get help

There are actually very few `git` commands that a typical user needs on a regular basis, and you have seen most of them today. However, `git` has many more features that are useful and powerful, and `git` has become a workhorse of the programming world.

If you need help when using `git` the first place to go is simply to type

```
git help
```

Sometimes, a bug is introduced into a repository that goes unnoticed, and it would be nice to be able to track down exactly when the bug occurred. If you can test your code, you should be able to determine if the code has the bug or not, but how do you find the location in the repository where the bug first occurred? One way to do it is to use the `git bisect` command. Type the following, and read the documentation on the command:

```
man git-bisect
```

Next, perform another clone, as follows:

```
cd ~/cs107/lab9
git clone /afs/ir/class/cs107/samples/midtermQ3b midtermQuestion3
cd midtermQuestion3

make
./find_min_tester red green blue
```

You should see that there is a problem! The correct answer should be `Min first char of my arguments is b` but it isn't! The repository has hundreds of commit messages, and the error was introduced many revisions ago. Use the `git bisect` command to find it.

What you do recall is that you know that the program was correct when you fixed a pesky return value, so you should start with the following:

```
git log --grep="return"
```

The first commit was a good working version. Use what you learned by reading the `man git-bisect` page to find the location when the bug was introduced, and then put the hash and the commit message from that commit into your lab checkoff form.

Hints:

- To denote the farthest back good commit you know (what you found with the grep command above), you type `git bisect good commit_hash` and replace the commit hash withe one you found above. So, you would start as follows:

  ```
  git bisect start
  git bisect bad
  git bisect good hash_number_from_above
  ```

- How do you test the program when you get to a commit? First, make it, then run the test script:

  ```
  make
  ./find_min_tester red green blue
  ```

- If you get the correct answer (`b`) then it is good, if you get an incorrect answer, it is bad. Every time you type `git bisect bad` or `git bisect good` , you will get another version, which you can check as above. Eventually, you will get to the version that was the original place where the code broke.
- What was the commit hash and commit message for the commit where the bug was introduced?
- After you finish observing where you made the mistake, you probably want to type `git bisect reset` to go back to the newest version (where you could fix the bug).

4) Review for Final Exam with your Lab TA

Since the midterm exam, we have covered three main topics:

- Floating point numbers
- Assembly Code
- Heap allocation

Those three topics will comprise most of the final exam, although you may see some questions from the beginning of the quarter, as well (*suggestion: go re-do the midterm exam from scratch, and see how you do!*). We will have an exam page online mid-week, but as always, studying your

labs and previous assignments is the first step.

Check off with TA

Before you leave, be sure to submit your checkoff sheet (in the browser) and have lab TA come by and confirm so you will be properly credited for labIf you don't finish everything before lab is over, we strongly encourage you to finish the remainder on your own. Double-check your progress with self check (/class/cs107/selfcheck.html#lab9).