Unix and Git Commands

1 Terminals

Many of you may not be familiar with using a shell. A *shell* is a way for a user (you) to interact with the computer. In our case, this will be done through a terminal session, so both will be used interchangeably.

Becoming familiar with a terminal is a useful skill which will carry beyond your stay here at WWU. An experienced (and somewhat stubborn) programmer may even be able to complete an entire assignment without touching the mouse. A better reason to study the use of a shell is it allows execution of many commands much faster (once you get the hang of it) than with a GUI (graphical user interface), and further allows execution of some commands which simply aren't possible any other way.

Here is an example prompt:

```
pollars@linux-04:~$
```

A *prompt* is the computer's way of saying "I'm ready, let's execute commands!" In the future, we often denote the prompt as simply a `$` for brevity.

Each part represents a different piece of information. `pollars` of course represents my username. `linux-04` represents the machine I'm currently logged on to. The colon `:` represents that there is a switch to talking about the files on that machine. Here, the *current working directory* is simply tilde (`~`). A tilde in Unix speak means the home directory. The home directory for everyone on the WWU computers is `/home/username` where `username` is your universal username that western assigned you. This is the working directory you start in when you log on and is so important several tools exist for making it easy to access, tilde being one of them. Another is that the change directory command with no arguments will take you to your home directory. So if you're getting lost, simply type `cd`.

Another useful command if you're lost is the `man` command. If you are confused about any command, type `man thatcommand` (e.g. `man ls`). I'm of the belief that about half of getting good with Unix is knowing how to decipher the manual pages.

The colon filesystem labeling becomes important to remember in the case of commands like `scp` where to specify a source file, you must give the full address of that file. For example, my copy of the floatlet may be accessed (with the correct credentials) at

```
pollars@linux-04.cs.wwu.edu:/home/pollars/ta/f14/cs247/floatlet/
```

In the case of the labs here at the WWU computer science department, any machine capable of running Linux will be able to access your files. That is, the labs CF162, CF164, CF416, CF405, (and others) will all have the same data.

1.1 Hidden files and .bashrc

When you type the `ls` command you will see most of the files in your current directory. In Unix, a directory is also considered a file. In fact, pretty much everything is considered a file. Files can start with anything, and contain any character except the null character

(\0) and the slash (/). These are reserved for ending strings and describing subdirectories, respectively. But the rest is fair game, though it is generally a lot easier to perform commands if you stick to the alphabet, numbers, and occasionally .'s -'s or _'s.

Any file which starts with a period is considered hidden and doesn't show up with the `ls` commmand unless you use the `-a` flag. A *flag* is an additional, usually optional, argument to a command that gives further specification. You can think of `-a` as "all" in the context of `ls`. Try doing this to your home directory. There probably are a lot of files.

One of particular interest is the `.bashrc` file. This may not exist on your clean home directories, but by the end of the quarter you will probably have some useful stuff in there. A `.bashrc` is a *script* that runs every time you log in. For example, in my script I have

```
alias ls='ls --color'
```

which colors files based on their type (directories, executables, etc.).

Now, I will demonstrate a somewhat complicated command and explain each part.

```
pollars@linux-04:~/ta/f14/cs247$ echo 'alias ls="ls --color"' >> ~/.bashrc
pollars@linux-04:~/ta/f14/cs247$ source ~/.bashrc
```

This is a bit complex, but each part is simple. The tilde as we recall means the home directory. `echo` simply repeats the text. Try echoing various characters and see what happens. For example, try `echo ${PATH}`.

Back to the commands: the single quotes represent informally "copy this text exactly." What this means is that nothing fancy will happen. Compare the result of `echo '${PATH}'` to the one without single quotes. As a rule of thumb, use single quotes when you want to talk about something *exactly* as you typed.

The `>>` command means "append the output of this command to the file specified, and if no such file exists, create it." This is a very succinct way of editing a file to contain the lines we want at the end. The `source` command executes the .bashrc commands so that you don't have to log out and log back in to your shell to get the changes updated. (Take that, Windows!)

2 CSCI 247 Lab 1

We'll be performing a lot of one-time-only setup to prepare for this lab and all others that we'll be doing throughout the course, so hang on tight. Much of this has been documented somewhere in support site `https://support.cs.wwu.edu`, so where possible, this document will refer you there.

2.1 Overview

In this lab, we will be creating a public/private key pair that we can use to enable secure access in the CS domain. We will create a repository to host our lab solutions. We will configure our repository to allow secure access to ourselves, allow the professor and teaching assistants (but *not* other students!) access to the repository, associate the public portion of

our cryptographic key to the repository, and then activate the repository to get it ready for hosting our code files. We will clone the repo into a local working folder, add the initial lab files, and push them to our repository. In future labs, the professor may push the lab files directory into your repository, so having it configured to allow this is important. We will learn how to organize these checkins and annotate them for clarity.

2.2 Generating SSL Keys

SSH, which stands for "secure shell", is a mechanism for allowing automatic logins from one computer to another via cryptographic security. You are again referred to the support site at `https://support.cs.wwu.edu/index.php/Setting_up_SSH_keys_in_Unix_Linux`. Follow all four steps in this support page. You will note that the fourth step, "`ssh-agent` and `ssh-add`", will take you to a separate, brief page.

2.3 Phabricator

Phabricator is an open-source collection of web applications to help organizations that deal with development, testing, auditing, or automation of code to do their jobs faster and better. In this class, we will be using only one of the web applications contained in Phabricator.

2.3.1 Diffusion

Diffusion is a Phabricator web application to host repositories. A repository (often abbreviated "repo") is a container for code pertaining to one or more related projects. These repositories may be cloned by other users for use or modification in their projects, which makes repositories one of the basic units of open-source software exchange between individuals. Repositories are also very useful for maintaining an up-to-date archive of project code for a group of individuals working on the same project, or a single individual who may wish to maintain a master location for code that can be accessed from multiple locations.

2.3.2 Creating a Phabricator repository

The support site contains a complete walkthrough for performing this step and is available at `https://support.cs.wwu.edu/index.php/Creating_a_Phabricator_repository`. Follow all the steps in the order they are presented in the support page, but please follow these lab-specific directives:

1. Feel free to ignore any steps that are specific to **SVN** (short for "Subversion", a distinct repository type. In this lab, we will be using **Git** repositories.

2. Please use the following naming policy for your repository unless otherwise directed by your professor or teaching assistant:

   (a) For **Name**, use `cs247`, followed by `f` for Fall quarter, `w` for Winter quarter, `s` for Spring quarter, or `su` for Summer quarter, followed by the last two digits of the year. For instance, for Winter quarter 2015, you would use the name `cs247w15`.

(b) For **Callsign**, use your entire last name followed by your first initial, all in uppercase. For instance, if your name were Semolina Pilchard, you would enter `PILCHARDS` as your repository callsign.

3. When you get to the section entitled "**Repository Policies**", be sure to make your repository visible to your professor, the teaching assistant for your lab, and yourself, and allow your professor rights to push to your repository, since future lab assignments may be distributed by pushing them directly into your repository to check that the policy is correctly configured to permit it.

2.3.3 Managing Phabricator credentials

The support site contains a complete walkthrough for performing this step and is available at `https://support.cs.wwu.edu/index.php/Managing_Phabricator_credentials`. Follow only the section pertaining to **Linux** unless otherwise directed.

3 Git

Git is one of many types of repository. As we said earlier, the repository holds a collection of code files for a single project. How do you interact with the repository? Think of the repository as a parking garage, and Git as a valet. As a guest of the hotel, you are discouraged from going into the garage to get your car yourself. Instead, you have the valet retrieve and park your car for you. Git is the valet for our repository; we will be telling it to retrieve and store our files for us.

We begin by creating a local working copy of this repository, a process called cloning. Once we have a local working copy, we will add the lab files and push the newly-added files into the repository. In the immediate future, we will also modify, remove, and rename files as well, and we will see that `git` has easy-to-use commands for these operations as well.

3.1 Cloning our new repository

At this point, we have created and activated our repository. It's time to clone the repository to a local working directory. In the terminal, type

```
$ cd ~
```

to make sure that when we clone our repository, the local working directory is created in our home directory. Now, if you go the details page for the repository in Phabricator, it will display a command that can be used have `git` clone your repository over either SSH or HTTP. Locate the option for **Clone (SSH).** For our imaginary friend Semolina Pilchard, this string would look like

```
$ git clone ssh://vcs@forge.cs.wwu.edu/diffusion/PILCHARDS/cs247w15
```

Copy and paste this string into the terminal window. The terminal will respond with something like

```
Cloning into 'csci247w15'...
Enter passphrase for key '/home/pilchas/.ssh/id_rsa':
warning: You appear to have cloned an empty repository.
Checking connectivity... done
```

If you get this warning, that's a good sign! It means that you have successfully cloned your repository into a local working directory. There are a few things that could cause this step to fail, including but not limited to:

1. Desynchronized domain passwords. Did you change your password at the beginning of this lab? If so, log off and log back in and retry this command.

2. Incorrect SSH passphrase. Did you associate a passphrase with your SSH key? If so, verify that you entered it correctly.

3. Many other things. Consult your professor or lab assistant.

3.2 Adding new files to our repository

Assuming that you successfully cloned an empty repository, we'll now add the initial lab files and update the repository. From Canvas, download the initial lab in compressed (`.tgz`) format and copy it into our newly-created local working directory (`cs247w15` for our friend Semolina; your name may vary). From there, you can extract the files it contains by either:

1. If you prefer the graphical user interface, right-click the `floatlet.tgz` file and click "Extract here". A new folder, `floatlet`, containing the compressed files, will be created in the working directory.

2. If you prefer the command line, type

   ```
   $ tar xvf floatlet.tgz
   ```

   for the same results.

3.2.1 The three-step process of updating a repository

Imagine your household on the day before trash day. All of the trash cans are full and need to go out to the curb for pickup, but you don't want to go out to the curb multiple times because you're lazy and smart. Instead, you collect all of the contents of the kitchen trash cans into one big trash bag and put it by the door. Then you dump all of the garage trash cans into another bag and put it by the door too, then all of the upstairs trash cans into yet another bag and put it by the door. When you're done with the entire house, you grab all of the trash bags by the door and drag them all out to the curb at once.

`git` uses the same three-step process. Adding an individual bit of trash to a garbage bag is like adding ((`git add`); in Git parlance, this is called "staging") an individual file. When

you close and tie up a trash bag and put it by the door, this is like a commit (`git commit`): it's ready to go out to the curb, but maybe not quite yet; maybe you have other bags that still need to go to the curb. So you can have multiple pending commits, each one consisting of multiple file operations. Lastly, when all of trash cans in the house have been collected into bags waiting by the door, you can make a single trip to the curb (`git push`) to update the repo. At any time, you can use `status` to see what is and is not scheduled for inclusion and what has and has not changed.

3.2.2 Step 1 of 3: Adding new content

The files we want to add to our repository reside in our working folder. We need to tell Git to prepare them for inclusion into the repo. The command for this is `git add` followed by one or more file or directory names. Let's add the entire `floatlet` directory:

```
$ git add floatlet
$
```

Git is the strong and silent type; it didn't say anything, but did it do what we asked? Let's find out.

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

new file:   floatlet/Makefile
new file:   floatlet/README
new file:   floatlet/add_test.c
new file:   floatlet/add_test.dat
new file:   floatlet/add_test.sh
new file:   floatlet/floatlet.c
new file:   floatlet/floatlet.h
new file:   floatlet/print_test.c
new file:   floatlet/print_test.dat
new file:   floatlet/print_test.sh

Untracked files:
  (use "git add <file>..." to include in what will be committed)

floatlet.tgz
$
```

Groovy. This is Git telling us that it has scheduled the `floatlet` directory, and all of the files it contains, to be added to the repo on the next commit. Notice that these are *changes to be committed*; `git` didn't actually modify the repository yet.

3.2.3 Step 2 of 3: Committing a set of changes

The files we added in Step 1 are all related; they're all part of an initial checkin. Once we have scheduled files to be added to the repo, we can "package up" all of these related files to be updated in a single commit, and we can associate with this commit a comment that makes it clear to ourselves, the professor, and anybody grading your work just what you've been up to. Let's commit the files we've added, but use the `-m` switch to provide a message for posterity. (If you're new to coding, you'll be appalled at how quickly you forget what you checked in and why you changed it. You will thank yourself for leaving detailed commit messages.)

```
$ git commit -m "This is my first checkin."
[master (root-commit) 2ca05f2] This is my first checkin.
 Committer: Semolina Pilchard <pilchas@some.workstation>

 10 files changed, 687 insertions(+)
 create mode 100644 floatlet/Makefile
 create mode 100644 floatlet/README
 create mode 100644 floatlet/add_test.c
 create mode 100644 floatlet/add_test.dat
 create mode 100755 floatlet/add_test.sh
 create mode 100644 floatlet/floatlet.c
 create mode 100644 floatlet/floatlet.h
 create mode 100644 floatlet/print_test.c
 create mode 100644 floatlet/print_test.dat
 create mode 100755 floatlet/print_test.sh
$
```

But we're not quite done yet!

3.2.4 Step 3 of 3: Sending all staged commits to your repository

You might have thought that `commit` had a sense of finality about it, but it's not quite the last word. As we said above, it might be the case that you now want to start on a new set of related changes; in fact, it is good practice to group related changes into a single commit. However, if you're new at this, I would suggest pushing your changes to the repository early and often, so let's follow up every `commit` with a `push`, which is in fact the final word in the process.

With Git, the first push is sometimes a doozy. Our new repository is completely empty and because it lacks an even branching structure (technical consideration), Git often gets upset on the first push:

```
$ git push
...
No refs in common and none specified; doing nothing.
Perhaps you should specify a branch such as 'master'.
```

```
fatal: The remote end hung up unexpectedly
error: failed to push some refs to 'ssh://vcs@forge.cs.wwu.edu/diffusion/PILCHARDS/cs247
```

Blah, blah, blah. To get around this error, we have to mutter the arcane incantation

```
$ git push origin master
Counting objects: 13, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (13/13), 5.97 KiB | 0 bytes/s, done.
Total 13 (delta 1), reused 0 (delta 0)
To ssh://vcs@forge.cs.wwu.edu/diffusion/PILCHARDS/cs247w15.git
 * [new branch]      master -> master
```

This incantation is also a one-time-only thing on the first push to an empty repository. Future pushes can just say `git push`.

4 Working Remotely

If you are running Linux on your home computer, things are much easier. Even easier if you know how to use a text editor which works in a terminal such as emacs or vim. We'll start with the easiest and work our way up.

4.1 Working remotely on Linux

You'll be connecting via the command `ssh`. There is the option -Y which allows X-windows to run over an ssh connection. Simply put, you can open up windows on your home computer running programs on the machines here on the lab.

You can connect to any lab computer. However, it is recommended that you choose the following "computer" to connect to:

```
linux.cs.wwu.edu
```

This is preferable to, say, `cf416-05.cs.wwu.edu`, for example, because the computers at the linux address are in some closet somewhere and are not touched by students. That is, no one will randomly turn off the computer and boot you off.

For example, Semolina Pilchard would connect using the command

```
ssh -Y pilchas@linux.cs.wwu.edu
```

Occasionally, you might get the following message:

```
Warning: the ECDSA host key for 'linux.cs.wwu.edu' differs from the key
for the IP address '140.160.137.172'
Offending key for IP in /home/pichas/.ssh/known_hosts:12
Matching host key in /home/pichas/.ssh/known_hosts:21
Are you sure you want to continue connecting (yes/no)? yes
```

"Yes" is the correct response here. Every quarter the lab computers get re-imaged and the ECDSA host key changes. If you connect to the same computer every time, you will only get this warning once.

Now, if you wanted to run a program, you would simply type `emacs` to bring up an emacs instance on your home computer. Note that you must have emacs (or any other computer you have) installed on your home computer. To do this in Ubuntu, for example, type

```
sudo apt-get install emacs23
```

4.2 Working in Windows

The simplest method for working remotely in a Windows environment is to download PuTTY (`http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`). Then, follow the instructions at `https://support.cs.wwu.edu/index.php/VNC_and_tunneling_through_SSH`. If you do not want to set up tunnelling over SSH, you can just use the terminal interface to get work done. Vim works in a terminal screen, and emacs can be run in a terminal mode by adding the `-nw` (no window) flag. If you do not use either of those editors, the editor `nano` could also be used, but keep in mind, nano is *not* a powerful editor and is only useful when doing very simple editing. These three options are listed below:

```
$ nano myfile.txt
$ vim myfile.txt
$ emacs -nw myfile.txt
```

5 Helpful Commands