UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE13/L: INTRODUCTION TO PROGRAMMING IN C

Git and Unix

Introduction

This class uses Git on the Unix cluster in the SOE. This document assumes no prior knowledge of terminals, SSH, Git, and the submission commands. If you do have knowledge of terminals and SSH, skip directly to the last section for reading on how to submit using Git.

Terminals

The virtual terminal is a command line interface to the operating system that you are on through an interface called a shell. This interface allows the user to type commands to run or interact with most software on the computer, which POSIX systems such as Mac OS X and GNU/Linux offering more capabilities by default.

This documentation will only discuss the Bash shell as that is the default on the Unix cluster that you will be using. The '$' symbol at the start of the examples refers to the prompt given to you by the Bash shell and everything after that is what you will enter.

The shell is all about file handling and processing input and output between different commands. The shell has a concept of a “current working directory”, which is where all commands execute relative to by default. This is like how your file browser works, where it generally displays the contents of a single directory at a time. Spaces are used to separate the various commands, arguments, and parameters entered.

Changing directories

To do this you can use the change directory command, ‘cd’. The path you want to change to follows the ‘cd’ command by a space. The '~' symbol has a special meaning when used as a path and refers to your user directory (usually located at /home/USERNAME). Additionally the '.' symbol refers to the current working directory and the '..' symbol refers to the parent directory. Also useful is ‘pwd’ which prints the working directory.

To enter a sibling directory of the current directory.

```
$ cd ../cmpe12
```

To go back to your home directory:

```
$ cd ~
```

List files

To list the files in a directory, the list files command 'ls' is used. It takes a list of paths to list the files at, with the current directory being the default. Specifying the additional argument of '-hal' turns on human-readable file sizes, shows hidden files, and uses a one-file-per-line list with file details. Can be useful.

To list all files in your home directory:

```
$ ls ~
```

To list files in a subdirectory:

```
$ ls -hal Spring14/cmpe12/lab1
```

Copying files

The 'cp' command does this.

Copy the provided header file as the basis for a new source file (make sure all header-specific code is removed!):

```
$ cp Leds.h Leds.c
```

Making directories

Use the mkdir command. It takes a single argument: a path to make. Note that it can only make one level of directories at a time, so if you want a cmpe13 directory inside a courses folder and the courses folder doesn't already exist, you'll need to make the courses folder first:

```
$ mkdir courses/cmpe12
mkdir: cannot create directory `courses/cmpe13': No such file or directory
$ mkdir courses
$ mkdir courses/cmpe12
```

Deleting files/folders

To delete files the ‘rm’ command is used. Arguments are a list of file names. Note that it only works for files. For deleting directories, use the ‘rmdir’ command on an empty folder.

Running programs

Programs on the shell are run similar to the `cd` command introduced previously. They always look like, where arguments are optional):

```
$ PROGRAM [ARGUMENT0 ARGUMENT1 …]
```

PROGRAM is going to be a path, or if the program is in a special location, merely the name of the program. Arguments are space-delimited and come after the program name as the following examples show.

Most programs provide their own documentation. Some programs output this help when run with no arguments. Almost all programs will output their documentation when given a --help command. Finally some programs provide help via the ‘man’ program which takes a single argument, the name of the function (if you use ‘man’, the arrow keys scroll and typing ‘q’ and then ENTER will quit).

```
$ tar –help
$ tar -h
$ man tar
```

To pull down a file from the website:

```
$ wget http://classes.soe.ucsc.edu/cmpe012/Spring14/Labs/Lab1/Lab1.zip
```

To unzip a zip-file (where the file is specified as the first argument to the unzip program):

```
$ unzip Lab1.zip
```

To zip up files into a zip-file for emailing or saving:

```
$ zip lab0_done.zip part3.c
```

SSH

SSH stands for Secure Shell and is a way to securely access a terminal on a remote machine. To login you need to know your username and password to the remote machine and have an IP address or URL to the machine.

For accessing the Unix cluster at UCSC, use your CruzID Blue login credentials and the URL 'unix.ucsc.edu'.

On Windows, the PuTTY program provides an SSH client and terminal. On most other operating systems, a standard terminal program will work where you can run the `ssh` command (search through provided applications for 'terminal' or look in the System Tools section of the main menu).

From the Bash shell on GNU/Linux or Mac:

```
$ ssh USERNAME@unix.ucsc.edu
```

To use PuTTY on Windows (which is already available on the lab computers), just launch it from the start menu. Enter the following:

For address: unix.ucsc.edu
For port: 22

Once it starts it will prompt you for your username and password.

You can also use the git bash program also available on the lab computers. This will provide a bash interface without having to log in to the unix server. You should be able to perform the same operations either way.

Copying files

Files in your home directory (the X: drive on Windows lab computers) is shared with the Unix cluster computers, so no file transfer is required if you only use the lab computers.

If you use a separate computer, you will want to copy files back and forth between computers. You can do this most easily by using SFTP with the easiest option for all platforms to be FileZilla.

To use FileZilla, you just enter the same information as required to do SSH:

Address: unix.ucsc.edu
Port: 22
Username: SOE_USERNAME
Password: SOE_PASSWORD

After you log in you will see your directory on the Unix cluster on the right and your home directory on the local computer on the left. Simply drag files back and forth to

copy them. You can also drag files from a file browser into the right-hand pane to copy files, so you don't always have to browse to the files on the left.

Git

Writing software is a very iterative process. Often you need to try things out, and sometimes things don't work the way you have anticipated. This requires you to go back to your working copy and start over. This gets even more complicated when more than one person is working on a software project simultaneously. This is addressed by version control software which tracks changes and allows you to roll back, branch, merge, and other functions. Many version control systems exist, but the one we are going to use in CMPE12 is Git.

Git is a popular version control system that has some very nice features when working on software projects, and is widely used in the software industry (these are real tools used every day in the real world). When used properly, Git will allow you to always roll back changes as needed, and ensure that code cannot be lost. The basic unit that Git works on is called a *repository* (or repo), and you will be interfacing to one for the labs in this class.

On the Unix terminals, all Git commands start with 'git' followed by the actual command. The first one you should type is: `'git help'` which will print out a basic help file on using Git. You can get more information on any additional command by using: `'git help xxx'`, where xxx is the command you are interested in (e.g.: `'git help add'` will print out a document on the git add command). More information on Git can be found at https://git-scm.com/book/en/v2 and https://git-scm.com/docs.

Git installation

While this guide is based off using Git on the unix terminals or Git Bash on the lab computers, Git can very easily be installed on your own personal systems. The Git client can be downloaded from https://git-scm.com/. Launching the *Git Bash* application will provide access to all of the commands used in this document (note that Git Bash creates a Bash shell for Git on your machine).

First Time Setup

For this class we are using GitLab with HTTPS Authentication. While SSH keys are supported by GitLab they are not covered in the scope of this guide but feel free to use them if you already know how to do so. The first step is to sign up for a GitLab account by following the guide at https://git.soe.ucsc.edu/~git/gitlab/index.html. Note that we can only make your repository after you have done so and there will be a delay. Plan Ahead.

To use Git we also need to configure some user settings. Run the following commands replacing the name, email and text editor with your information so that Git knows your identity.

```
$ git config --global user.name "Max Dunne"
$ git config --global user.email mdunne@soe.ucsc.edu
$ git config --global core.editor (cli editor of choice)
```

Note: if you don't know a good command line editor, just use emacs or vim.

There are many good resources on using Git, however the SOE GitLab server also has a quick cheat sheet online that might be useful: https://git.soe.ucsc.edu/~git/gitlab/using_gitlab.html

Also note that the free online Git book has a very good chapter on the basics of how a repo works and general workflow: https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository

As a handy reference for using the more common commands, these are the ones you will most likely use.

git clone

To start work on a lab we must first get a local copy of the repository. Entering the command below will create a repository in the current directory with similar output. You will need to supply your own information rather than the given ones.

```
$ git clone https://gitlab.soe.ucsc.edu/gitlab/cmpe012/testproject.git
Cloning into 'testproject'...
Username for 'https://gitlab.soe.ucsc.edu':
Password for 'https://mdunne@gitlab.soe.ucsc.edu':remote: Counting
objects: 6681, done.
remote: Compressing objects: 100% (6395/6395), done.
remote: Total 6681 (delta 4526), reused 400 (delta 237)
Receiving objects: 100% (6681/6681), 76.70 MiB | 1.98 MiB/s, done.
Resolving deltas: 100% (4526/4526), done.
```

```
Checking connectivity... done.
```

Git Clone is only used to instantiate a new copy of the repository. If the repository needs to be updated see the pull command.

git status

Status is used to determine the current state of the repository. For instance, calling it on an empty repository gives the output below.

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

if we add a file, README.txt for this example, and call status again the output changes to show README.txt in a new section called untracked changes.

```
$ git status
On branch master

Initial commit

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

        README.txt

nothing added to commit but untracked files present (use "git add" to
track)
```

Git will not track any files unless they are added to git, this is done using the **git add** command.

git add

The add command has two main purposes: adding files to git to be tracked; and staging files for commit. Calling *git add README.txt* gives no output but calling *git status* again shows that git is now tracking the file.

```
$ git status
On branch master

Initial commit

Changes to be committed:
```

```
  (use "git rm --cached <file>..." to unstage)

        new file:   README.txt
```

git commit

When the commit command is called a snapshot of the repository at the current moment in time is taken. This should be done often as each commit will allow the code to be rolled back to that point (this is ideal if there is a bug to track down that did not exist before). To allow commits to be differentiated a comment must be made with each commit signified by the *–m* modifier. These comments should give a brief description of what changes have been made to the code. The command below was used to commit the empty README.TXT file.

```
$ git commit -m "committing an empty readme for a test"
```

Calling **git status** again shows that the file has disappeared from the list of changes to be committed. If README.TXT is then modified and *git status* is called yet again it shows that the file is not staged for commit.

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   README.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

The file can be staged by calling git add on the file. Alternatively adding *-a* to the commit string will cause all files tracked by git and modified to be committed.

```
$ git add README.txt
$ git commit -am "Updated the readme with Hello World"
```

Git push

After a commit all files in the repository still only exist in the *local* copy (that is on the host machine). The files are pushed back to the server using the following command.

```
$ git push
```

Git pull

Git **pull** performs the opposite action as push. Instead of pushing files to the server it pulls files to your local copy. **WARNING: This overwrites all of the local files**. It is called with:

```
$ git pull --rebase
```

This is normally used to update your repository after working on a different computer (i.e. switching from a lab computer to a laptop).

Git Workflow

The standard workflow for Git typically consists of the following steps

1. Clone the repository or pull if the repository if work has been done in a different location.
2. Work on the lab as normal.
    a. Make commits when milestones are reached with useful commit messages. The messages are important, be descriptive.
3. Push back to the server when session is done or when you want to make sure you have a backup (you don't need to push every commit, but if you do, then the most you will ever lose is from the latest commit).
4. Repeat as necessary.

Lab Submittal

As there will be one repository for all the labs, lab submittal is merely denoting a specific commit for grading. Graders will clone this specific commit and grade the assignment using this point in time. To do so they will need the commit ID. To do so and to ensure that the files are actually on the git server (the only place they can be graded) we will use a web interface to find our commit ID. (Note that for security reasons this interface is not available off campus without a VPN. Go here for information on setting one up if you need to submit off campus).

Again go to https://git.soe.ucsc.edu/~git/gitlab/using_gitlab.html which has directions on how to get the commit ID. Submit the Commit ID using the Google form associated with the lab. The time you submit this form determines when your lab was turned in.

This ID should match the ID found when running **git log**. A sample output is shown below using the modifier *–n 1* to only show one entry.

```
$ git log -n 1
commit 78629a85de99468250a860d85a173108f3cff9b3
```

```
Author: Max Dunne <mdunne@soe.ucsc.edu>
Date:   Sun Mar 27 14:04:58 2016 -0700

    removed leftover edits from COSMOS and clarified language about
    folder directories
```

We are currently using canvas for grading. We do not grade files on canvas but ones pulled from git.

Validating Your Lab Submittal

As the web interface was used to find the commit ID that particular commit is on the server. To confirm that the files match perfectly you can clone your repo to a new directory and verify the files are actually there. This is a common problem in which not all files are added to the repo and this is a quick check to ensure points are not lost because of it.

Final Notes

Using Git imposes some complexity in the programming course at the beginning; this is regrettable, as you are already busy learning to code. However, learning how to use Git now, and maintaining your Git repos will have innumerable benefits down the line. The first and most obvious is that you can move from one computer to another and instantly have all of your code to work with (you don't have to juggle the files; Git will handle it for you). Committing and pushing often means that your code is safely backed up on the servers—lightning can strike your laptop and all you will have lost is the code you wrote from the time of the last push. This will encourage incremental development which is a key to successful programming. Lastly, the hope is that you will continue to use Git beyond this class, and discover that Git is one of the more useful tools that you have been taught while at UCSC.