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

CMPE012/L: COMPUTER SYSTEMS AND ASSEMBLY PROGRAMMING

GIT INTRODUCTION AND BASIC SHELL COMMANDS

INTRODUCTION:

Git is an industry standard tool for version control, and you should know how to use it as part of your engineering education (for example, the entire Microsoft Windows development is version controlled using Git). You will be using Git for version control on your labs, as well as the method to turn in all of your lab assignments. This document assumes no prior knowledge of terminals, SSH, Git, and the submission commands. If you do have knowledge of terminals and UNIX commands, skip directly to the last section for reading on how to submit using Git.

Note that we are using our own SOE GITLAB server at https://gitlab.soe.ucsc.edu/gitlab/; this is not to be confused with the commercial GITLAB service (you will note the slug on the top left of the browser page indicating that you are on the SOE server).

GIT BASH:

You will largely interact with the Git server through a Bash shell terminal that is available as part of the Git software installed on the lab computers. Note that Git is a free download to your own computers from https://git-scm.com/ and is available cross platform. It is already installed on all lab computers.

Bash is a command line interface to the operating system, and uses the standard set of UNIX commands for interacting with files and folders. 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.

To launch the Git Bash window, either right click in the explorer window or use the windows key and type “Git Bash” and launch the program. A terminal window will appear and lists your current directory along with the $ bash prompt. 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. A return or enter triggers the command to run.

BASIC BASH COMMANDS:

The Git Bash terminal implements most of the UNIX terminal commands. An explanation of the full set of terminal commands is beyond this document, but many tutorials exist online and can be easily found using your standard search tools. A small subset of the commands are, however, quite useful and have examples and explanations below. Also note that pressing the TAB key will trigger Bash to attempt to autocomplete the next command (for instance it will pattern match all files that match the rest of what you have typed). It will fill in until it cannot decide which is the next part of the pattern, and expects you to type in the next letter. If you type TAB again, it will list all possibilities which match the pattern.

Note that most commands have a help included in Git Bash, using the ‘--help’ argument.

pwd – print working directory. This command is used to print out your current working directory (i.e.: which directory you are currently in). For example: $pwd

cd – change directory. This command is used to change the working directory. 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 /c/Users/USERNAME). Additionally the '.' symbol refers to the current working directory and the '..' symbol refers to the parent directory.

For example to change to your root directory: `$ cd ~`
To move to a sister directory: `$ cd ../lab1`

ls – list files. This command is used to list the files in a directory. 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. This can be useful. Note that listing the files of a different directory does not change your directory (you need a cd command to do that).

For example to list the files in your current working directory: `$ls`
To list the files in your home directory: `$ls –hal ~`
To list files in the sister directory: `$ls ../lab1`

cp – copy files. This command is used to copy files using `cp source destination`. If no path is included then the current working directory is used. Note that cp leaves the original file intact and creates a copy.

For example to copy the README.txt file to a new file: `$cp README.txt OLDREADME.txt`
This can be verified to have worked using an ls command. If you are copying to another directory and want the same name, you can use path/. to indicate using the same name.

mv – move files. This command is used to move files using `mv source destination`. If no path is included then the current working directory is used. Note that mv can be used to rename a file as well.

For example to rename the OLDREADME.txt file to a new file: `$mv OLDREADME.txt README.txt`
This can be verified to have worked using an ls command. If you are moving to another directory and want the same name, you can use path/. to indicate using the same name. Note that Bash assumes you know what you are doing and will overwrite another file without warning.

mkdir – make directory. This command is used to create directories or folders. If no path is included then the directory is created under the existing working directory.

For example, to create a temp subdirectory: `$mkdir temp`
Again this can be verified using the ls command.

rm/rmdir – remove file or directory. The rm command deletes a file or files (rmdir deletes a directory). Note that you can use wildcards to specify more than one file. Also rmdir will not allow you to remove a directory if there are still files in it.

For example, to delete all files in the temp directory: `$rm ./temp/*.*` (be careful with this)
To delete the (now empty) temp directory: `$rmdir temp`

touch – updates the access date/time of a file to the current time, creates an empty file if it does not exist. The touch command updates the access time of an existing file, or creates a new empty file if it does not exist.

For example, to update the access time of your README.txt file: `$touch README.txt`

To create a new empty file named DeadBeef.txt: `$touch DeadBeef.txt`
This can be verified using the ls –hal command

cat/head/tail – reads a file and dumps it out to terminal. Cat concatenates the file and dumps it to the terminal window; the head command reads from the top of a file and the tail command reads from the bottom. Default for head/tail is 10 lines but can be changed with the '-n' argument. This can be useful to quickly check the contents of a file.

For example, to output the entire README.txt file: `$cat README.txt`
For example, to output the first 10 lines of your README.txt file: `$head README.txt`
To output the first 25 lines of DeadBeef.txt: `$head –n 25 DeadBeef.txt`

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 CMPE-12/L 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.

In the Git Bash terminal, 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 will notice some of the commands are very similar to the terminal commands above. 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. These are both free and excellent resources. 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

While this guide is based off using Git on the Git Bash terminal 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).

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. The repo will be created in the working directory of the Git Bash terminal (which you can check using pwd).

For example, to clone the a test repo for file checking and put it into a folder called testCI:

```
$ git clone git@gitlab.soe.ucsc.edu:classes/testing/elm.git testCI
Cloning into 'testCI'...
remote: Counting objects: 73, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 73 (delta 30), reused 57 (delta 23)
Receiving objects: 100% (73/73), 6.38 KiB | 0 bytes/s, done.
Resolving deltas: 100% (30/30), done.
Checking connectivity... done.
```

Note that when you use the GITLAB server, Git will ask you for your username (CruzID@ucsc.edu) and your Blue password to clone the repo. You can get the https://gitlab address from the GITLAB webpage. Then it will copy all of the files into the appropriate directory. 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
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Copying a file into the repo will be shown as a file that is not tracked. This will be shown in RED in the Git Bash terminal, and you will need to add it using the git add command. For instance, copying a new file called BoilerPlatePiazza.txt into the repo and again running git status results in:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        BoilerPlatePiazza.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Note that git tells you what is going on (and it is very important to read the messages it gives back when you perform an operation). In order to add the untracked file you will need to use the git add command (and then you can check it with a git status).

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

```
$ git add BoilerPlatePiazza.txt
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   BoilerPlatePiazza.txt
```

Now the new file has been added to the GIT repo, so that git is tracking the file. However this change had not been committed to the repo. To do this, we will use the git commit command.

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 changes of adding in the personal responsibility document. The '–am' flag indicates to automatically add modified files and include a message that git will include when you check the repo status.

```
$ git commit -am "Added the BoilerPlatePiazza document"
[master 1a94247] Added the BoilerPlatePiazza document
 1 file changed, 9 insertions(+)
 create mode 100644 BoilerPlatePiazza.txt
```

This has created a snapshot of the current files in your local directory. You will be able to return to this exact point in the repo if you need to. But you are still vulnerable to data loss because the data only exists on your local copy. To synchronize with the server, you need to push the changes to the server using a git push command. Again, using the git status command will tell you what is going on:

```
$ git status
On branch master
```

```
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

The message is telling you that your work is ahead of the server by 1 commit, and that you should push (note that there is no limit to the number of commits you do before pushing, but it is a good idea to push often). Using the git push command synchronizes the files from your local copy to the server so that you can always pull them back down to any computer.

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
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 674 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitlab.soe.ucsc.edu:classes/testing/elm.git
   7a45776..1a94247  master -> master
```

Now if we modify the BoilerPlatePiazza.txt file to have my name in it (using any text editor of my choice and saving it when done) and rerun the git status command:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/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:   BoilerPlatePiazza.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Git shows the presence of a modified file, which has not been staged for a commit. In order to commit the changes, run the git commit –am command with an appropriate message. And then push it to the server.

```
$ git commit -am "Added my name to the end of BoilerPlatePiazza.txt"
[master 32dd5ae] Added my name to the end of BoilerPlatePiazza.txt
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 326 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@gitlab.soe.ucsc.edu:classes/testing/elm.git
   1a94247..32dd5ae  master -> master
```

If you are working on two different machines (or have more than one person working on a project) then you will need to use the git pull command to resynchronize with the server. This is the opposite of push and is used to bring your local directory to match what is on the server.

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
Username for 'https://gitlab.soe.ucsc.edu': elkaim@ucsc.edu
Already up-to-date.
```

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

Note that there are quite a few more commands and many more sophisticated things can be done in Git. These become extremely useful when working with multiple people on complicated projects. Again, the tutorials referenced above are free and quite good.

GIT WORKFLOW:

The standard workflow for Git typically consists of the following steps:

1. Clone the repository if you have not already.
2. Work on the lab using whatever tools appropriate.
3. Make commits when milestones are reached with useful commit messages. The messages are important, be descriptive. Do this often.
4. Push back to the server when session is done or whenever 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).
5. Repeat as necessary.

Again, the idea here is to commit early and often. Every time you make a significant change, commit. Pushing often ensures that you don't lose work for any reason. Get in the habit of committing often. It is also extremely useful to be able to "roll back" to a point where things were working, now that they are not. This is quite handy, but requires you to have commits to track down where you introduced the bug into your code.

Lastly, your commits will help you in the case of academic misconduct, as the teaching staff will be able to see the incremental changes in your work so we can see you developing the code (as opposed to having nothing exist in the repo and then have the entire code appear in a single commit 5 minutes before the assignment is due).

Bottom line: commit and push often.

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 pulled to be graded) we will use a web interface to find your commit ID.

You will be submitting your commit ID using a google form, which requires your first and last name, and your commit ID. There are several ways to get the commit ID, but the easiest will be to use the web interface one the GITLAB server. The overview page will have your last commit on the main page, and an icon next to it to copy the commit ID to the clipboard.

You can also click on the "history" button on the GITLAB webpage to see all of your commits, and grab the corresponding commit ID (with the same icon) to the version you would like to turn in. Again, you may submit the google form as many times as you like. We will take the last one you submit (and take the time you submitted it as your turn in time).

VALIDATING YOUR LAB SUBMITTAL:

There are many way to verify that you have submitted the lab correctly. To confirm that the files match perfectly you can clone your repo to a new directory and verify the files are actually there (and that the contents are what you expect). This is a common problem in which not all files are added to the repo and this is a quick check to ensure that you meet the minimum submission requirements of the lab (and also that all your work is there so that you get points for the work you did).

You can also go to the GITLAB web interface and view your files there. If your files are on the GITLAB web interface, you have successfully pushed them. Make sure to grab the corresponding commit ID.

As a courtesy, we have created a validation webpage: https://git.soe.ucsc.edu/~git/cgi-bin/repo_check.py that will validate your repo to ensure you have the correct files in the repo. It requires your CruzID, Blue password, and the commit ID (taken from the GITLAB webpage) to run. Select "CMPE 12, Winter 2018" from the pulldown menu for which class, and hit the "Verify repo requirements" button.

If you have included all of the required files for the minimum submission requirements, then you will get a message that looks like:

Congratulations! Your repository (at commit ID 70ca3223451ddc6af10ef7524a4607658707df8e) passed the verification check, and meets the minimumrequirements for this assignment. This verification is only of minimum requirements specified in the assignment, and is not a full test of your code!

If you have failed to include the correct files, you will get a message that looks like:

Unfortunately, your repository (at commit ID `09ebd45b5f5fc26ac72531a4189536a828e74d86`) didn't pass the verification check.

The repo cloned successfully, but verification failed!

```
==================================================
Present files:
Missing files: lab0/CMPE012_PersonalResponsibility.pdf
lab0/CMPE012_Syllabus.pdf
==================================================
FAILED: repository DOES NOT minimum requirements because files were missing!
==================================================
```

Use the commit ID that passes on your Google form submission (or alternately validate your google form submission using the validation website above). Again, this does not check for anything inside the files, only that you have the correctly named files in the repo in the correct folders. You can still fail the lab, but not because you failed to meet the minimum submission requirements.

FINAL NOTES:

Using Git imposes some complexity in the programming course at the beginning; this is regrettable, as you are already busy learning computer systems and assembly. 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.