Cornell Eta Kappa Nu
Linux and Git Workshop

School of Electrical and Computer Engineering
Cornell University

revision: 2015-11-11-21-55

Welcome to the HKN Tutorial for Linux and Git! Between Linux and Git, you will find two extremely useful skills that will be useful in both your academic and professional careers. What you see in this document will serve as a basic introduction to Linux and Git, and you are certainly encouraged to explore further, as a short tutorial like this barely scratches the surface of how powerful Linux and Git are. Linux and Git skills are heavily sought after in both academia and industry, and by the end of this tutorial, hopefully you will be comfortable with both.

To follow along with the tutorial, type the commands without the `%` character. In addition to working through the commands in the tutorial, you should also try the more open-ended tasks marked with the ★ symbol.

1. Linux

1.1. Introduction

Linux is an open-source operating system, modelled on UNIX, named after Linus Torvalds. Basic Linux knowledge is becoming more and more of an essential skill for engineers. When learned in-depth, Linux enables the user to enhance their productivity to a ridiculous level. We will give a brief, but brisk introduction to Linux for beginners during this section.

1.2. The Linux Command Line

In this section, we introduce the basics of working at the Linux command line. Please note that this Linux tutorial is obviously not comprehensive and cannot replace the extensive amount of documentation available online or elsewhere. The goal here is to gain comfort with the Linux command line, which can be daunting if you are used to only working with GUIs, like that of Windows or Mac OS.

The shell is the original Linux user interface which is a text-based command-line interpreter. The default shell once you log into is Bash. While there are other shells such as `sh`, `csh`, and `tcsh`, for this course we will always be assuming you are using Bash. As mentioned above, we use the `%` character to indicate commands that should be entered at the Linux command line, but you should not include the actual % character when typing in the commands on your own. This will lead to a bad time.

Initial Tutorial Setup

In order to keep a clean predictable flow of commands through this tutorial, and make sure it's machine-independent, we will first run a series of commands. Please follow them *exactly* for everything in this tutorial to make sense. In the following code, replace `<netid>` with your actual NetID.

```
% cd
% mkdir -p <netid>
% cd <netid>
% MYDIR=${PWD}
```

We will explain the meaning of all of these commands as the tutorial continues.

Hello World

As in every new computer task, we must begin with the obligatory "Hello, World" example. To display the message "Hello, World" we will use the `echo` command. The `echo` command simply "echos" its input to the console.

```
% echo "Hello, World"
```

The string we provide to the `echo` command is called a *command line argument*. We use command line arguments to tell commands what they should operate on. Although simple, the `echo` command can very useful for creating simple text files, displaying environment variables, and general debugging.

★ *To-Do On Your Own:* Experiment with using the `echo` command to display different messages.

Manual Pages

You can learn more about any Linux command by using the `man` command. Try using this to learn more about the `echo` command.

```
% man echo
```

You can use the up/down keys to scroll the manual one line at a time, the space bar to scroll down one page at a time, and the `q` key to quit viewing the manual. You can even learn about the `man` command itself by using `man man`. As you follow the tutorial, feel free to use the `man` command to learn more about the commands we cover.

★ *To-Do On Your Own:* Use the `man` command to learn more about the `passwd` command.

stdin, stdout, and stderr

Linux uses "streams" to interact with the user. This is how Linux can take input into a program, as well as provide output to the user, both for feedback and for errors. There are three streams by default in Linux, aptly named `stdin, stdout, and stderr`. One can visualize these streams' behavior with the figure below:

When you use `echo`, the words you type in are "echo"ed back from `stdin` to `stdout`. While we will not explore these streams in detail in this tutorial, they are worth learning about, as they can be very powerful tools in development.

★ *To-Do On Your Own:* Learn about what the differences among `stdin`, `stdout` and `stderr` are, and how you might be able to use them to make life easier.

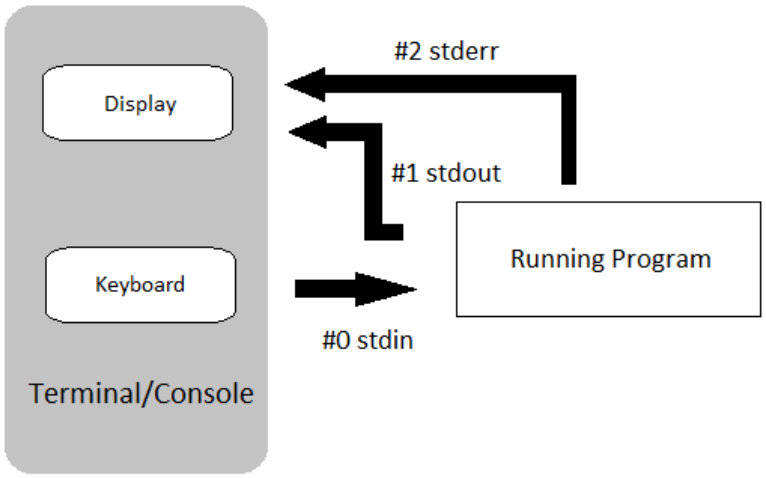

Figure 1: Visualization of Linux Streams

Create and View Files

We can use the `echo` command and a feature called *command output redirection* to create simple text files. This "redirection" effectively takes what `echo` sends to the `stdout` stream, and "redirects" it into a file instead, using the > operator. The following commands will create a new file named `hkn-linux.txt` that simply contains the text “Cornell University HKN”.

```
% cd ${MYDIR}
% echo "Cornell University HKN" > hkn-linux.txt
```

When you’re using a command line without a GUI, you might need a quick way to see the contents of a file. There are a few ways to do this. For short files, we can use the `cat` command to quickly display all contents of a file.

```
% cat hkn-linux.txt
```

For larger files, `cat` will output the entire file to the console, which can end up being a nuisance. For these situations, we can use the `less` command to show one screen-full of text at a time. You can use the up/down keys to scroll the file one line at a time, the space bar to scroll down one page at a time, and the `q` key to quit viewing the file.

```
% less hkn-linux.txt
```

Directory Commands and Moving around the Filesystem

Obviously, having all files in a single location would be hard to manage effectively. We can use *directories* (also called folders) to logically organize our files, just like one can use physical folders to organize physical pieces of paper. You are probably already used to doing this on your own computer. In Linux, the mechanism for organizing files and directories is called the *file system*. When you first log into `amdpool`, you will be in your *home directory*. This is your own private space that you

can use to work on anything you want. You can use the `pwd` command to print the directory in which you are currently working, which is known as the *current working directory*.

```
% pwd
```

When you're on a command line, you might also want to know what files are in the current working directory. You can use the `ls` command to list the filenames of the files you have created.

```
% ls
```

We can provide *command line options* to the `ls` command to modify the command's behavior. For example, we can we can use the `-l` (i.e., a dash followed by the letter l) command line option to provide a longer listing with more information about each file.

```
% ls -l
```

You should see the newly created `hkn-linux.txt` file along with some additional directories or folders. Use the following commands to create a few more files using the `echo` command and command output redirection, and then list the files again.

```
% cd ${MYDIR}
% echo "Eta" > hkn-linux-word1.txt
% echo "Kappa" > hkn-linux-word2.txt
% ls
```

★ *To-Do On Your Own:* Create a new file named `hkn-linux-word3.txt` which contains the word "Nu", then use `cat` and `less` to verify the file contents.

A directory path is a list of nested directory names; it describes a "path" to get to a specific file or directory. So the above path indicates that there is a toplevel directory named `home`, that contains a directory with your NetID. This is the directory path to your home directory. As an aside, it's important to notice that Linux uses a forward slash (`/`) to separate directories, while Windows uses a back slash (`\`) for the same purpose, and mixing up the two will lead to bad results.

So how do we move around the file system? What if we want to make a new folder? We can use the `mkdir` command to make new directories. The following command will make a new directory named `hkn` within your home directory.

```
% cd ${MYDIR}
% mkdir hkn
```

We can use the `cd` command to change our current working directory. The following command will change the current working directory to be the newly created `hkn` directory, before displaying the current working directory with the `pwd` command.

Please note that you where the tutorial says "..." in a file path, you will see your username and any subdirectory your home folder might be under.

```
% cd hkn
% pwd
/home/.../<netid>/hkn
```

Use the `mkdir`, `cd`, and `pwd` commands to make another directory.

```
% mkdir linux
% cd linux
% pwd
/home/.../<netid>/hkn/linux
```

We sometimes say that `linux` is a subdirectory or a child directory of the `hkn` directory. We might also say that the `hkn` directory is the parent directory of the `linux` directory.

There are some important shortcuts that we can use with the `cd` command to simplify navigating the file system. The special directory named `.` (i.e., one dot) always refers to the current working directory. The special directory named `..` (i.e., two dots) always refers to the parent of the current working directory. The special directory named `~` (i.e., a tilde character) always refers to your home directory. With no arguments, `cd` simply returns us to our home directory. The special directory named `/` (e.g., single forward slash) always refers to the highest-level root directory. The following commands illustrate how to navigate up and down the directory hierarchy we have just created.

```
% pwd
/home/.../<netid>/hkn/linux
% cd .
% pwd
/home/.../<netid>/hkn/linux
% cd ..
% pwd
/home/.../<netid>/hkn
% cd ..
% pwd
/home/.../<netid>
% cd hkn/linux
% pwd
/home/.../<netid>/hkn/linux
% cd
% pwd
/home/.../
% cd /
% pwd
/
% cd ~/<netid>/
% pwd
/home/.../<netid>
```

Notice how we can use the `cd` command to change the working directory to another arbitrary directory by simply using a directory path (e.g., `hkn/linux`). These are called *relative paths* because the path is relative to your current working directory. You can also use an *absolute path* which always starts with the *root directory* to concretely specify a directory irrespective of the current working directory. A relative path is analogous to directions to reach a destination from your current location (e.g., How do I get to the coffee shop from my current location?), while an absolute path is analogous to directions to reach a destination from a centralized location (e.g., How do I get to the coffee shop from the center of town?).

```
% pwd
  /home/.../<netid>
% cd /home/.../<netid>/hkn/linux
% pwd
  /home/.../<netid>/hkn/linux
```

One more useful shortcut to note: The `cd` command with no command line arguments always changes the current working directory to your home directory. We are operating below the home directory in this tutorial, so we won't be using this feature today.

★ *To-Do On Your Own:* Try creating more directories and files in your home directory.

★ *To-Do On Your Own:* Explore the `tree` command. What does it do?

Copy, Move, and Remove Files and Directories

So how do you cut, copy, and paste with Linux? To copy, we use the `cp` command. The first argument is the name of the file you want to copy, and the second argument is the new name to give to the copy. The following commands will make two copies of the files we created in the previous section.

```
% cd ${MYDIR}/hkn/linux
% echo "Hello!" > hkn-linux.txt
% cp hkn-linux.txt hkn-linux-a.txt
% cp hkn-linux.txt hkn-linux-b.txt
% ls
  hkn-linux.txt hkn-linux-a.txt hkn-linux-b.txt
```

We can also copy one or more files into a subdirectory by using multiple source files and a final destination directory as the arguments to the `cp` command. We can use the `-r` command line option to enable the `cp` command to recursively copy an entire directory.

```
% mkdir dirA
% cp hkn-linux.txt dirA
% cp hkn-linux-a.txt hkn-linux-b.txt dirA
% tree
% mkdir dirD
% cp -r dirA dirD
% tree
```

If we want to move a file or directory, we can use the `mv` command. As with the `cp` command, the first argument is the name of the file you want to move and the second argument is the new name of the file. In Linux, "moving" can be synonymous with renaming. For example, to rename a file, you can execute the following commands.

```
% mv hkn-linux.txt hkn-linux-c.txt
% ls
```

Again, similar to the `cp` command, we can also move one or more files into a subdirectory by using multiple source files and a final destination directory as the arguments to the `mv` command.

```
% mkdir dirB
% mv hkn-linux-a.txt dirB
% mv hkn-linux-b.txt hkn-linux-c.txt dirB
% tree
```

We do not need to use the `-r` command line option to move an entire directory at once.

```
% mkdir dirE
% mv dirD dirE
% tree
```

We can use the `rm` command to remove files. The following command removes a file from within the `hkn/linux` subdirectory.

```
% cd ${MYDIR}/hkn/linux
% ls
% echo "I am going to be deleted." > removeMe.txt
% ls
% rm removeMe.txt
% ls
```

To clean up, we might want to remove the files we created in your home directory earlier in this tutorial. We can use the `-r` command line option with the `rm` command to remove entire directories, but please be careful because it is relatively easy to permanently delete many files at once.

```
% cd ${MYDIR}/hkn/linux
% rm -r dirA dirB
% rm -r *
```

★ *To-Do On Your Own:* Creating additional directories and files within the `hkn/linux` subdirectory, and then use the `cp`, `mv`, and `rm` commands to copy, move, and remove the newly created directories and files. Use the `ls` and `tree` commands to display your file and directory organization.

Using `wget` to Download Files

We can use the `wget` command to download files from the internet. For now, this is a useful way to retrieve a text file that we can use in the following examples.

```
% cd ${MYDIR}/hkn/linux
% wget http://hkn.ece.cornell.edu/overview.txt
% cat overview.txt
```

Using `grep` to Search Files

We can use the `grep` command to search and display lines of a file that contain a particular pattern. The `grep` command can be useful for quickly searching the contents of the source files in your lab assignment. The command takes the pattern and the files to search as command line arguments. The following command searches for the word "of" in the `overview.txt` file downloaded in the previous section.

```
% cd ${MYDIR}/hkn/linux
% grep "HKN" overview.txt
```

You should see a few lines within the file that contain the word "of". We can use the `--line-number` and `--color` command line options with the `grep` command to display the line number of each match and to highlight the matched word.

```
% cd ${MYDIR}/hkn/linux
% grep --line-number --color "HKN" overview.txt
```

We can use the `-r` command line option to recursively search all files within a given directory hierarchy. In the following example, we create a subdirectory, copy the `overview.txt` file, and illustrate how we can use the `grep` command to recursively search for the word "of".

```
% cd ${MYDIR}/hkn/linux
% mkdir dirA
% cp overview.txt dirA
% grep -r --line-number --color "HKN" .
```

Notice how we specify a directory as a command line argument (in this case the special `.` directory) to search the current working directory. You should see the three lines from both copies of the `overview.txt` file. The `grep` command also shows which file contains the match.

As another example, we will search two special files named `/proc/cpuinfo` and `/proc/meminfo`. These files are present on every modern Linux system, and they contain information about the processor and memory hardware in that system. The following command first uses the `less` command so you can browse the file, and then uses the `grep` command to search for `processor` in the `/proc/cpuinfo` file. Recall that with the `less` command, we use the up/down keys to scroll the file one line at a time, the space bar to scroll down one page at a time, and the `q` key to quit viewing the file.

```
% less /proc/cpuinfo
% grep "processor" /proc/cpuinfo
```

You can probably see that you're on a multicore processor. You can also search to find out which company makes the processors and what clock frequency they are running at:

```
% grep "vendor_id" /proc/cpuinfo
% grep "cpu MHz" /proc/cpuinfo
```

We can find out how much DRAM is in the system by searching for `MemTotal` in the `/proc/meminfo` file.

```
% grep "MemTotal" /proc/meminfo
```

Using `find` to Find Files

We can use the `find` command to recursively search a directory hierarchy for files or directories that match a specified criteria. While the `grep` command is useful for searching file contents, the `find` command is useful for quickly searching the file and directory names in your lab assignments. The `find` command is very powerful, so we will just show a very simple example. First, we create a few new files and directories.

```
% cd ${MYDIR}/hkn/linux
% mkdir -p dirB/dirB_1
% mkdir -p dirB/dirB_2
% mkdir -p dirC/dirC_1
% echo "test" > dirA/file0.txt
% echo "test" > dirA/file1.txt
% echo "test" > dirB/dirB_1/file0.txt
% echo "test" > dirB/dirB_1/file1.txt
% echo "test" > dirB/dirB_2/file0.txt
% tree
```

We will now use the `find` command to find all files named `"file0.txt"`. The `find` command takes one command line argument to specify where we should search and a series of command line options to describe what files and directories we are trying to find. We can also use command line options to describe what action we would like to take when we find the desired files and directories. In this example, we use the `--name` command line option to specify that we are searching for files with a specific name. We can also use more complicated patterns to search for all files with a specific filename prefix or extension.

```
% cd ${MYDIR}/hkn/linux
% find . -name "file0.txt"
```

Notice that we are using the special `.` directory to tell the `find` command to search the current working directory and all subdirectories. The `find` command always searches recursively.

★ *To-Do On Your Own:* Create additional files named `"file2.txt"` in some of the subdirectories we have already created. Use the `"find"` command to search for files named `"file2.txt"`.

Using `tar` to Archive Files

We can use the `tar` command to "pack" files and directories into a simple compressed *archive*, and also to "unpack" these files and directories from the archive. This kind of archive is sometimes called a *tarball*. Most open-source software is distributed in this compressed form. It makes it easy to distribute code among collaborators and it is also useful to create backups of files. We can use the following command to create an archive of our tutorial directory and then remove the tutorial directory.

```
% cd ${MYDIR}/hkn
% tar -czvf linux.tgz linux
% rm -r linux
% ls -l
```

Several command line options listed together as a single option (`-czvf`), where `c` specifies we want to create an archive, `z` specifies we should use "gzip" compression, `v` specifies verbose mode, and `f` specifies we will provide filenames to archive. The first command line argument is the name of the archive to create, and the second command line argument is the directory to archive. We can now extract the contents of the archive to recreate the tutorial directory. We also remove the archive.

```
% cd ${MYDIR}/hkn
% tar -xzvf linux.tgz
% rm linux.tgz
% tree linux
```

Note that we use the `x` command line option with the `tar` command to specify that we intend to extract the archive.

Using `top` to View Running Processes

You can use the `top` command to view what commands are currently running on the Linux system in realtime. This can be useful to see if there are many commands running which are causing the system to be sluggish. When finished you can use the `q` character to quit.

```
% top
```

The first line of the `top` display shows the number of users currently logged into the system, and the *load average*. The load average indicates how "overloaded" the system was over the last one, five, and 15 minutes. If the load average is greater than the number of processors in the system, it means your system will probably be sluggish. You can always try logging into a different server in the cluster.

Environment Variables

As useful as the Bash shell for running commands, it also has some features of programming languages. One aspect of the shell that is similar in spirit to popular programming languages, is the ability to write and read *environment variables*. You have, in fact, been using the MYDIR variable throughout this tutorial! The following commands illustrate how to write an environment variable named `hkn_linux`, and how to read this environment variable using the `echo` command.

```
% hkn_linux="eta kappa nu"
% echo ${hkn_linux}
```

Keep in mind that the names of environment variables can only contain letters, numbers, and underscores. Notice how we use the `${}` syntax to read an environment variable. There are a few built-in environment variables that might be useful:

```
% echo ${HOSTNAME}
% echo ${HOME}
% echo ${PWD}
```

We often use the `HOME` environment variable in directory paths like this:

```
% cd ${HOME}/<netid>/hkn
```

The `PWD` environment variable always holds the current working directory. We can use environment variables as options to commands other than `echo`. A common example is to use an environment variable to "remember" a specific directory location, which we can quickly return to with the `cd` command like this:

```
% cd ${MYDIR}/hkn/linux
% TUT1=${PWD}
% cd
% cd <netid>
% pwd
/home/.../<netid>/
% cd ${TUT1}
% pwd
/home/.../<netid>/hkn/linux
```

★ *To-Do On Your Own:* Create a new environment variable named `hkn_linux` and write it with Electrical and Computer Engineering. Use the `echo` command to display this environment variable. Experiment with creating a new subdirectory within `hkn/linux` and then using an environment variable to "remember" that location.

Command Chaining

We can use the `&&` operator to specify two commands that we want to chaining together. The second command will only execute if the first command succeeds. Below is an example.

```
% cd ${HOME}/hkn/linux
% echo "Hello" > test-chain.txt && cat test-chain.txt
```

★ *To-Do On Your Own:* Create a single-line command that combines creating a new directory with the `mkdir` command and then immediately changes into the directory using the `cd` command.

Command Pipelining

The Bash shell allows you to run multiple commands simultaneously, with the output of one command becoming the input to the next command. We can use this to assemble "pipelines"; we "pipe" the output of one command to another command for further actions using the `|` operator.

The following example uses the `grep` command to search the special `proc/cpuinfo` file for lines containing the word "processor" and then pipes the result to the `wc` command. The `wc` command counts the number of characters, words, or lines of its input. We use the `-l` command line option with the `wc` command to count the number of lines.

```
% grep processor /proc/cpuinfo | wc -l
```

This is a great example of the Linux philosophy of providing many simple commands that can be combined to create more powerful functionality. Essentially the pipeline we have created is a command that tells us the number of processors in our system.

We can create even longer pipelines. The following pipeline will report the number of times you have logged into the system since it was rebooted.

★ *To-Do On Your Own:* Use the `cat` command with the `overview.txt` file and pipe the output to the `grep` command to search for the word "of". While this is not as fast as using `grep` directly on the file, it does illustrate how many commands (e.g., `grep`) can take their input specified as a command line argument or through a pipe.

Aliases, Wildcards, Command History, and Tab Completion

In this section, we describe some miscellaneous features of the Bash shell which can potentially be quite useful in increasing your productivity.

Aliases are a way to create short names for command sequences to make it easier to quickly execute those command sequences in the future. For example, assume that you frequently want to change to a specific directory. We can create an alias to make this process take just two keystrokes.

```
% alias ct="cd ${HOME}/hkn/linux"
% ct
% pwd
/home/academic/<netid>/hkn/linux
```

If you always want this alias to be available whenever you log into the system, you can save it in your `.bashrc` file. The `.bashrc` is a special Bash script that is run on every invocation of a Bash shell.

```
% echo "alias ct=\"cd ${HOME}/hkn/linux\"" >> ${HOME}/.bashrc
```

The reason we have to use a back slash (\) in front of the double quotes is to make sure the `echo` command sees this command line argument as one complete string.

Wildcards make it easy to manipulate many files and directories at once. Whenever we specify a file or directory on the command line, we can often use a wildcard instead. In a wildcard, the asterisk (`*`) will match any sequence of characters. The following example illustrates how to list all files that end in the suffix `.txt` and then copies all files that match the wildcard from one directory to another.

```
% cd ${MYDIR}/hkn/linux
% ls *.txt
% cp dirA/file*.txt dirB
% tree
```

The Bash shell keeps a history of everything you do at the command line. You can display the history with the `history` command. To rerun a previous command, you can use the `!` operator and the corresponding command number shown with the `history` command.

```
% history
```

You can pipe the output of the `history` command to the `grep` command to see how you might have done something in the past.

```
% history | grep wc
```

If you press the up arrow key at the command line, the Bash shell will show you the previous command you used. Continuing to press the up/down keys will enable you to step through your history. It is very useful to press the up arrow key once to rerun your last command.

The Bash shell supports tab completion. When you press the tab key twice after entering the beginning of a filename or directory name, Bash will try to automatically complete the filename or directory name. If there is more than one match, Bash will show you all of these matches so you can continue narrowing your search.

1.3. Linux Text Editors

You will need to use a text editor to edit source files in Linux. There are two kinds of text editors: graphical and non-graphical. The non-graphical text editors work by opening files through the command line and then using the keyboard to navigate files, execute commands, etc. The graphical text editors work by providing a GUI so that the user can use a mouse to interact with the editor.

Nano

Nano is a very simple non-graphical text editor installed on `amdpool`. The editor is easy to learn and use. You can start Nano by typing the command `nano` in the terminal and optionally specifying the filename you want to view and edit.

```
% cd ${MYDIR}
% echo "This will be modified soon" > editor-test.txt
% nano editor-test.txt
```

Use the arrow keys to move the cursor position. Notice that the editor specifies most of the useful commands at the bottom of the terminal screen. The symbol ^ indicates the `CONTROL` key. To type any text you want, just move the cursor to the required position and use the keyboard. To save your changes press `CONTROL+O` and press the `<ENTER>` key after specifying the filename you want to save to. You can exit by pressing `CONTROL+X`.

★ *To-Do On Your Own:* Use Nano to make some changes to the `editor-test.txt` text file and then save your edits to your home directory. View the new file using the `cat` command from the command line and then delete the file using the `rm` command.

Emacs and Vim

While `nano` and `gedit` editors are easy to learn, students that anticipate using Linux in the future beyond this course might want to use a more powerful editor such as `emacs`, `vi`, or `vim`. It is beyond the scope of this tutorial to teach you the usage of these editors, but most advanced Linux users use one of these more powerful text editors for development. These two are also the cause of a holy war among Linux users, but just know that both are great for what they do.

1.4. Conclusion

This tutorial hopefully helped you become familiar with Linux and how to use it in a basic manner. There are manymore resources online for learning Linux, and keep in mind that learning to work productively using the Linux operating system can be pay off heavily in both industry and academia.

2. Git

2.1. Introduction

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."
–Linus Torvalds

Now, we will context switch to learning about Git. Git is a method of repository hosting and version control, initially created by Linus Torvalds. Git is used in a variety of contexts, from software and hardware development in industry, to document version control in academia. Git enables rapid, distributed collaboration and iteration on a project. This tutorial covers how to: setup your GitHub account, use Git for basic incremental development yourself and with multiple people, and manage Git branches and GitHub pull requests. We assume for the Git section that you have completed part 1, or have familiarity with Linux already.

2.2. Setting up Your GitHub Account

GitHub is an online service that hosts centralized Git repositories for a growing number of open-source projects. It has many useful features including a web-based source browser, history browser, branch management, merge requests, code review, issue tracking, and even a built-in wiki attached to every repository. If you do not yet have an GitHub account, you can create one here:

- `https://github.com/join`

Your NetID makes a great GitHub ID, or failing that, some format including your last name, such as <firstname><lastname> or <firstinitial><lastname>. Once your account is setup, please make sure you set your full name, and upload a profile photo to GitHub; it is something that future employers might look at, and it's nice if your profile looks professional.

- `https://github.com/settings/profile`

It is infinitely easier to use GitHub with an SSH key pair, as you don't need to input a password every time (just make sure you keep your computer secure...). Follow the directions here to create the pair and associate it with your GitHub account.

- `https://help.github.com/articles/generating-ssh-keys/`

Then, use the following page to upload the public key to GitHub:

- `https://github.com/settings/ssh`

You may see a warning about the authenticity of the host the first time you try accessing GitHub.. Don't worry, this is supposed to happen the first time you access GitHub using your new key. Just enter "yes". The GitHub server should output some text including your GitHub ID. Verify that the GitHub ID is correct, and then you should be all set. There are two good GitHub Guides you might want to take a look at:

- `https://guides.github.com/activities/hello-world`
- `https://guides.github.com/introduction/flow`

GitHub has two integrated tools that students might find useful: an issue tracker and a wiki. Consider using the GitHub issue tracker to track bugs you find in your code or to manage tasks required to complete the lab assignment. You can label issues, comment on issues, and attach them to commits. See the following links for more information about GitHub issues:

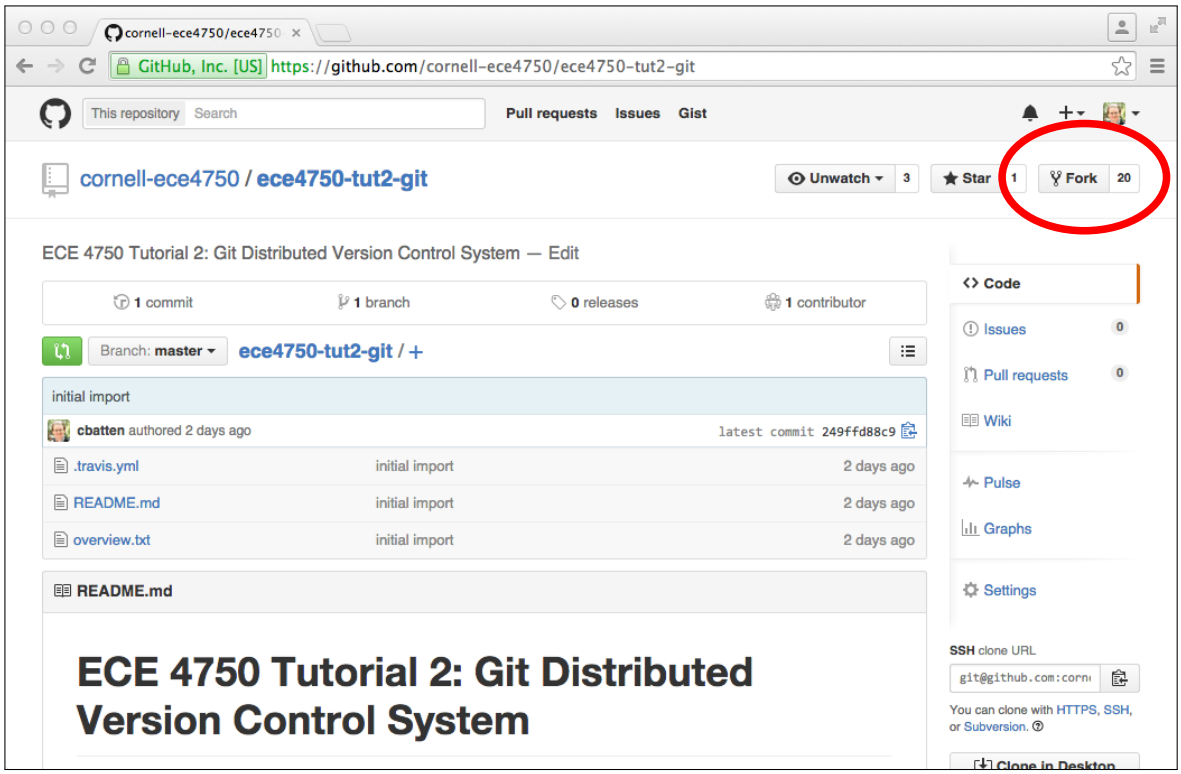

Figure 2: Forking the `ece4750-tut2-git` Repository

- `https://guides.github.com/features/issues`
- `https://help.github.com/articles/about-issues`

Consider using the GitHub per-repository wiki to create task lists, brainstorm design ideas, rapidly collaborate on text for the lab report, or keep useful command/code snippets. See the following links for more information about GitHub wikis:

- `https://guides.github.com/features/wikis`
- `https://help.github.com/articles/about-github-wikis`

In next section, we begin with a basic single-user workflow before demonstrating how Git and Github can be used for effective collaboration among multiple users. We discuss how to resolve conflicts and how to manage branches and pull requests.

2.3. Single-User Workflow

In this subsection, we cover some basic Git commands and illustrate a simple Git workflow, using an example repo from the ECE 4750 course at Cornell, courtesy of Professor Christopher Batten and his students. This will be an initial template, so the first step is to *fork* this tutorial repository. Forking is the process of making a personal copy of someone else's repository on GitHub. Start by going to the GitHub page for the tutorial repository located here:

- `https://github.com/cornell-ece4750/ece4750-tut2-git`

Figure 2 shows the corresponding GitHub page. Click on *Fork* in the upper right-hand corner. If asked where to fork this repository, choose your personal GitHub account. After a few seconds, you should have a brand new repository in your account:

- `https://github.com/<githubid>/ece4750-tut2-git`

Where `<githubid>` is your GitHub ID, not your NetID. Now that you have your own copy of the tutorial repository, the next step is to *clone* this repository to `amdpool` so you can manipulate the content within the repository. We call the repository on GitHub the *remote repository* and we call the repository on `amdpool` the *local repository*. A local repository is a first-class mirror of the remote repository with the entire history of the repository, and thus almost all operations are essentially local requiring no communication with GitHub. The following commands write an environment variable with your GitHub ID, create a subdirectory for this tutorial in your home directory before using the `git clone` command to clone the remote repository and thus create a local repository.

```
% GITHUBID="<githubid>"
% mkdir -p ${MYDIR}/git
% cd ${MYDIR}/git
% git clone https://${GITHUBID}@github.com/${GITHUBID}/ece4750-tut2-git
hkn-git
% cd hkn-git
% GITDIR=${PWD}
```

Where again `<githubid>` is your GitHub ID, not your NetID. The `git clone` command takes two command line arguments. The first argument specifies the remote repository on GitHub you would like to clone, and the second argument specifies the name to give to the new local repository. Note that we created an environment variable with the directory path to the local repository to simplify navigating the file system in the rest of this tutorial.

The repository currently contains two files: a `README` file, and `overview.txt` which contains an overview of ECE 4750. These files are contained within what we call the *working directory*. The repository also includes a special directory named `.git` which contains all of the extra repository metadata. You should never directly manipulate anything within the `.git` directory.

```
% cd ${GITDIR}
% ls -la
```

Let's assume we want to create a new file that contains a list of fruits, and that we want to manage this file using Git version control. First, we create the new file.

```
% cd ${GITDIR}
% echo "apple" > fruit.txt
```

To manage a file using Git, we need to first use the `git add` command to tell Git that it should track this file from now on. We can then use `git commit` to commit our changes to this file into the repository, and `git log` to confirm the result.

```
% cd ${GITDIR}
% git add fruit.txt
% git commit -m "initial fruit list"
% git log
```

The `-m` command line option with the `git commit` command enables you to specify a *commit message* that describes this commit. All commit messages should include a "subject line" which is a single *short* line briefly describing the commit. Many commits will just include a subject line (e.g., the above commit). If you want to include more information in your commit message then skip the `-m` command line option and Git will launch your default editor. You still want to include a subject line at the top of your commit message, but now you can include more information separated from the subject line by a blank line.

Note, you can learn about any Git command and its usage by typing `git help <command>`, where `<command>` should be substituted by the actual name of the command. This would display the output similar to the manual pages for a Linux command, as seen in Tutorial 1. Students are encouraged to learn more about each Git command beyond the details covered in this tutorial.

The `git log` command displays information about the commit history. The beginning of the output from `git log` should look something like this:

```
commit 0e5b2b2c05b5837839554fa047e52e121c8206b1
Author: cb535 <cb535@cornell.edu>
Date:   Sat Aug 18 18:01:17 2015 -0400

initial import of fruit
```

Conceptually, we should think of each commit as a copy of all of the tracked files in the project at the time of the commit. This commit just included changes to one file, but as we add more files each commit will include more and more information. The history of a git repository is just a long sequence of commits that track how the files in the repository have evolved over time. Notice that Git has recorded the name of who made the commit, the date and time of the commit, and the log message. The first line is the commit id which uniquely identifies this commit. Git does not use monotonically increasing revision numbers like other version control systems, but instead uses a 40-digit SHA1 hash as the commit id. This is a hash of *all* the files included as part of this commit (not just the changes). We can refer to a commit by the full hash or by just the first few digits as long as we provide enough digits to unambiguously reference the commit. Now let's add a fruit to our list and commit the change.

```
% cd ${GITDIR}
% echo "mango" >> fruit.txt
% git commit -m "added mango to fruit list"
```

Unfortunately, this doesn't work. The output from `git commit` indicates that there have been no changes since the last commit so there is no need to create a new commit. Git has a concept of an *index* which is different compared to other version control systems. We must "stage" files (really we stage content not files) into the index, and then `git commit` will commit that content into the repository. We can see this with the `git status` command.

```
% cd ${GITDIR}
% git status
```

which should show that `fruit.txt` is modified but not added to the index. We stage files in the index with `git add` like this:

```
% cd ${GITDIR}
% git add fruit.txt
```

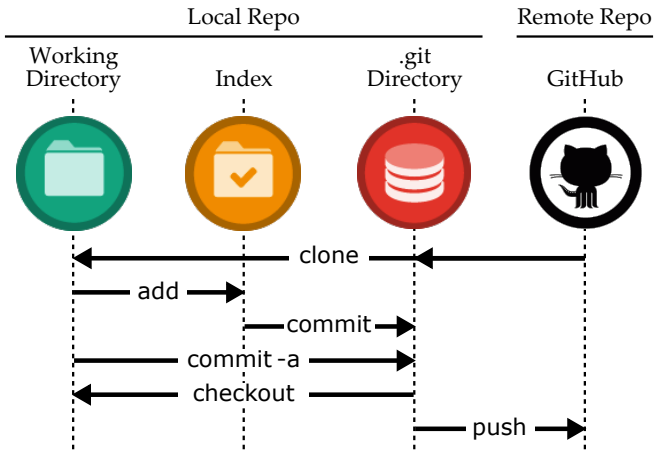

Figure 3: Git and GitHub for Single-User Development

```
% git status
```

Now `git status` should show that the file is modified and also added to the index. Our commit should now complete correctly.

```
% cd ${GITDIR}
% git commit -m "added mango to fruit list"
% git status
```

So even though Git is tracking `fruit.txt` and knows it has changed, we still must explicitly add the files we want to commit. There is a short cut which uses the `-a` command line option with the `git commit` command. This command line option tells Git to commit any file which has changed and was previously added to the repository.

```
% cd ${GITDIR}
% echo "orange" >> fruit.txt
% git commit -a -m "added orange to fruit list"
% git status
```

Staging files is a useful way to preview what we will commit before we actually do the commit. This helps when we have many changes in our working directory but we don't want to commit them all at once. Instead we might want to break them into smaller, more meaningful commits or we might want to keep working on some of the modified files while committing others.

Figure 3 illustrates how the commands we have used so far create a single-user development workflow. The `git clone` command copies the remote repository to create a local repository which includes both the working directory and the special `.git` directory. The `git add` command adds files to the index from the working directory. The `git commit` command moves files from the index into the special `.git` directory. The `-a` command line option with the `git commit` command can commit files directly from the working directory to the special `.git` directory.

Now that we have made some changes, we can use `git log` to view the history of last few commits and then add another line to the `fruit.txt` file.

```
% cd ${GITDIR}
```

```
% git log
% echo "plum" >> fruit.txt
% cat fruit.txt
```

Imagine you didn't like your changes and want to revert the changes, you would use the `git checkout` command as below.

```
% cd ${GITDIR}
% git checkout fruit.txt
% cat fruit.txt
```

As illustrated in Figure 3, the `git checkout` command resets any a file or directory to the state it was in at the time of the last commit. The output from the `git status` command should look something like this:

```
% cd ${GITDIR}
% git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
(use "git push" to publish your local commits)
nothing to commit, working directory clean
```

The `git status` command is telling us that the local clone of the repository now has more commits than the remote repository on GitHub. If you visit the GitHub page for this repository you will not see any changes. This is a critical difference from other centralized version control systems. In Git, when we use the `git commit` command it only commits these changes to your *local repository*.

If we have done some local work that we are happy with, we can push these changes to the *remote repository* on GitHub using the `git push` command.

```
% cd ${GITDIR}
% git push
% git status
```

Notice that the output of the `git status` command indicates that our local repository is up-to-date with the remote repository on GitHub. Figure 3 shows visually the idea that the `git push` command moves commits from your local repository to the remote repository on GitHub. Visit the GitHub page to verify that our new commits have been pushed to the remote repository:

- `https://github.com/<githubid>/ece4750-tut2-git`

Click on *commits* at the top of the GitHub page to view the log of commits. You can browse who made each commit, what changed in each commit, and the state of the repository at each commit. Return to the main GitHub page for the repository and click on the `fruit.txt` file to view it.

★ *To-Do On Your Own:* Create a new file called `shapes.txt` that includes a list of different shapes. Commit the new file, make some edits, and commit these edits. Use `git status` and `git log` to keep track of your changes. Push your changes to GitHub and browse the updated files on GitHub.

2.4. Multi-User Workflow

Since your tutorial repository is public on GitHub, any other user can also clone this repository. If you would like to collaborate with another GitHub user, you would need to give that user read/write permission. To emulate how collaboration with GitHub works, we will "pretend" to be different users by cloning extra copies of the tutorial repository.

```
% cd ${MYDIR}/git
% git clone https://${GITHUBID}@github.com/${GITHUBID}/ece4750-tut2-git
%alice
% cd alice
% ALICE=${PWD}
% cd ${MYDIR}/git
% git clone git@github.com:${GITHUBID}/ece4750-tut2-git bob
% cd bob
% BOB=${PWD}
```

We can now emulate different users by simply working in these different local repositories: when we work in `ALICE` we will be acting as the user Alice, and when we work in `BOB` we will be acting as the user Bob. Figure 4 illustrates a multi-user development environment: both Alice and Bob have their own separate local repositories (including their own working directories, index, and special `.git` directories), yet they will both communicate with the same centralized remote repository on GitHub.

Let's have Alice add another entry to the `fruit.txt` file, commit her changes to her local repository, and then push those commits to the remote repository on GitHub:

```
% cd ${ALICE}
% echo "banana" >> fruit.txt
% git commit -a -m "ALICE: added banana to fruit list"
% git log --oneline
% git push
% cat fruit.txt
```

If you view the GitHub page for this repository it will appear that you are the one making the commit (remember we are just pretending to be Alice), which is why we used `ALICE:` as a prefix in the commit message.

Now let's assume Bob wants to retrieve the changes that Alice just made to the repository. Bob can use the `git pull` command to pull all new commits from the remote repository into his local repository. The `git pull` command performs two actions, it first fetches all the updates and then merges or applies them to the local project. If there are no conflicts in the file contents, the command executes successfully. If there are conflicts, the command does not merge all the changes and reports the conflicting content. We will learn how to resolve conflicts in Section 2.5.

```
% cd ${BOB}
% git pull
% git log --oneline
% cat fruit.txt
```

Figure 4 shows visually the idea that the `git pull` command moves commits from the remote repository on GitHub to your local repository. Bob's copy of tutorial repository should contain Alice's most

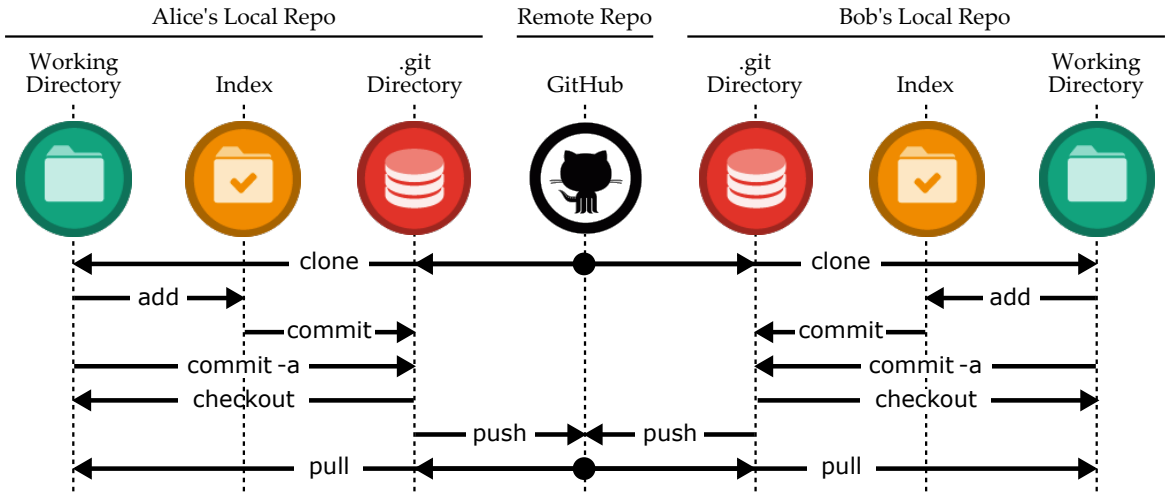

Figure 4: Git and GitHub for Multi-User Development

recent commit and his copy of the `fruits.txt` file should include "banana". Now let's assume Bob also wants to make some changes and push those changes to the remote repository on GitHub:

```
% cd ${BOB}
% echo "peach" >> fruit.txt
% git commit -a -m "BOB: added peach to fruit list"
% git log --oneline
% git push
% cat fruit.txt
```

Similar to before, Alice can now retrieve the changes that Bob just made to the repository using the `git pull` command.

```
% cd ${ALICE}
% git pull
% git log --oneline
% cat fruit.txt
```

This process is at the key to collaborating via GitHub. Each student works locally on his or her part of the lab assignment and periodically pushes/pulls commits to synchronize with the remote repository on GitHub.

★ *To-Do On Your Own:* Create a new file called `letters.txt` in Bob's local repository that includes a list of letters from A to M, one per line. Commit the new file, make some edits to add say more letters from say M to Z, and commit these edits. Use `git push` to push these commits to the centralized repository. Switch to Alice's local repository and use `git pull` to pull in the new commits. Verify that all of your files and commits are in both Bob's and Alice's local repositories.

2.5. Resolving Conflicts

Of course the real challenge occurs when both Alice and Bob modify content at the same time. There are two possible scenarios: Alice and Bob modify different content such that it is possible to combine their commits without issue, or Alice and Bob have modified the exact same content resulting in a conflict. We will address how to resolve both scenarios.

Let us assume that Alice wants to add `lemon` to the list and Bob would like to create a new file named `vegetables.txt`. Alice would go ahead and first pull from the central repository to grab any new commits from the remote repository on GitHub. On seeing that there are no new commits, she edits the file, commits, and pushes this new commit.

```
% cd ${ALICE}
% git pull
% echo "lemon" >> fruit.txt
% git commit -a -m "ALICE: added lemon to fruit list"
% git push
```

Since Bob recently pulled from the remote repository on GitHub, let's say he assumes that there have been no new commits. He would then go ahead and create his new file, commit, and attempt to push this new commit.

```
% cd ${BOB}
% echo "spinach"  >  vegetables.txt
% echo "broccoli" >> vegetables.txt
% echo "turnip"   >> vegetables.txt
% git add vegetables.txt
% git commit -m "BOB: initial vegetable list"
% git push
To git@github.com:<githubid>/ece4750-tut2-git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to
'git@github.com:<githubid>/ece4750-tut2-git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

On executing the sequence of commands above, you should notice that Git does not allow Bob to push his changes to the central repository as the version of the central repository has been updated by Alice. You should see a message similar to the one above. Git suggests us to merge the remote commits before pushing the local commits. We can do so by first using the `git pull` command to merge the local commits.

```
% cd ${BOB}
% git pull
```

Git will launch your default text editor because we need to *merge* your local commits and the remote commits. You will need to enter a commit message, although usually the default message provided

by Git is fine. After saving the commit message, we can take a look at the Git history using `git log` to see what happened.

```
% cd ${BOB}
% git log --oneline --graph
*   656965a Merge branch 'master' of
github.com:<githubid>/ece4750-tut2-git
|\
| * 9d943f3 ALICE: added lemon to fruit list
* | 4788dbe BOB: initial vegetable list
|/
* c7cc31e BOB: added peach to fruit list
* dc28bc9 ALICE: added banana to fruit list
```

The `--graph` command line option with the `git log` command will display a visual graph of the commit history. You can see that Bob and Alice worked on two different commits at the same time. Alice worked on commit `9d943f3` while Bob was working on commit `4788dbe`. Bob then merged these two sets of commits using a new commit `656965a`. Bob can now push his changes to the remote repository in GitHub.

```
% cd ${BOB}
% git push
```

GitHub has a nice commit history viewer which shows a similar commit graph as we saw above:

- `https://github.com/<githubid>/ece4750-tut2-git/network`

While this approach is perfectly reasonable, it can lead to a very non-linear and complex commit history. We strongly recommend that you use an alternative called *rebasing*. While merging "merges" sets of commits together, rebasing will apply one set of commits first and then apply a second set of commits on top of the first set of commits. This results in a more traditional linear commit history. Let's reconstruct a similar situation as before where Alice adds another fruit and Bob adds another vegetable at the same time.

```
% cd ${ALICE}
% git pull
% echo "plum" >> fruit.txt
% git commit -a -m "ALICE: added plum to fruit list"
% git push
% cd ${BOB}
% echo "potato" >> vegetables.txt
% git commit -a -m "BOB: added potato to vegetable list"
% git push
To git@github.com:<githubid>/ece4750-tut2-git
! [rejected]        master -> master (fetch first)
```

To rebase, we will use the `--rebase` command line option with the `git pull` command.

```
% cd ${BOB}
% git pull --rebase
% git push
```

```
% git log --oneline --graph
* 0d5fba5 BOB: added potato to vegetable list
* e56ad1b ALICE: added plum to fruit list
*   656965a Merge branch 'master' of
github.com:<githubid>/ece4750-tut2-git
|\
| * 9d943f3 ALICE: added lemon to fruit list
* | 4788dbe BOB: initial vegetable list
|/
* c7cc31e BOB: added peach to fruit list
* dc28bc9 ALICE: added banana to fruit list
```

Study the output from the `git log` command carefully. Notice how instead of creating a new merge commit as before, rebasing has applied Alice's commit `e56ad1b` first and then applied Bob's new commit `0d5fba5` on top of Alice's commit. Rebasing keeps the git history clean and linear.

Sometimes Alice and Bob are editing the exact same lines in the exact same file. In this case, Git does not really know how to resolve this *conflict*. It does not know how to merge or rebase the two sets of commits to create a consistent view of the repository. The user will have to manually resolve the conflict. Let's explore what happens when Alice and Bob want to add a new fruit to the `fruits.txt` file at the exact same time. First, Alice adds `kiwi` and pushes her updates to the remote repository on GitHub.

```
% cd ${ALICE}
% git pull
% echo "kiwi" >> fruit.txt
% git commit -a -m "ALICE: added kiwi to fruit list"
% git push
```

Now Bob adds `date` and tries to push his update to the remote repository on GitHub.

```
% cd ${BOB}
% echo "date" >> fruit.txt
% git commit -a -m "BOB: added date to fruit list"
% git push
To git@github.com:<githubid>/ece4750-tut2-git
! [rejected]        master -> master (fetch first)
```

As before, Bob uses the `--rebase` command line option with the `git pull` command to pull the commits from the remote repository on GitHub.

```
% cd ${BOB}
% git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: BOB: added date to fruit list
Using index info to reconstruct a base tree...
M       fruit.txt
Falling back to patching base and 3-way merge...
Auto-merging fruit.txt
CONFLICT (content): Merge conflict in fruit.txt
Failed to merge in the changes.
```

Git indicates that it was not able to complete the rebase. There is a conflict in the `fruit.txt` file. We can also use the `git status` command to see which files have conflicts. They will be marked as `both modified`:

```
% cd ${BOB}
% git status
```

Git instructs Bob to first resolve the conflict and then use the `--continue` command line option with the `git rebase` command to finish the rebase. If you take a look at the `fruit.txt` file you will see that it now includes conflict markers showing exactly where the conflict occurred.

```
% cd ${BOB}
% cat fruit.txt
apple
mango
orange
banana
peach
lemon
plum
<<<<<<< 7f8ec0fb9d7c705e3caf9034f3b96b0c8e7cad92
kiwi
=======
date
>>>>>>> BOB: added date to fruit list
```

This shows that the commit from the remote repository on GitHub has `kiwi` as the last line in the file, while the last commit to the local repository has `date` on the last line in the file. To resolve the conflict we can directly edit this file so that it reflects how we want to merge. We can choose one fruit over the other, choose to include neither fruit, or choose to include both fruit. Edit the file using your favorite text editor to remove the lines with markers <<<<, === and >>>> so that the file includes both fruit.

```
% cd ${BOB}
% cat fruit.txt
apple
mango
orange
banana
```

```
peach
lemon
plum
kiwi
date
```

The next step is critical and easy to forget. We need to use the `git add` command to add any files that we have fixed! In this case we need to use the `git add` command for the `fruit.txt` file.

```
% cd ${BOB}
% git status
% git add fruit.txt
% git status
```

Notice how the output from the `git status` command has changed to indicate that we have fixed the conflict in the `fruit.txt` file. Since we have resolved all conflicts, we can now continue the rebase:

```
% cd ${BOB}
% git rebase --continue
% git push
% git log --oneline --graph
% cat fruit.txt
```

Resolving conflicts is tedious, so to avoid conflicts you should communicate with your group members which student is going to be executing which files. Try to avoid having multiple students working on the same file at the same time, or at least avoid having multiple students working on the same lines of the same file at the same time.

★ *To-Do On Your Own:* Experiment with having both Alice and Bob edit the same lines in the `overview.txt` file at the same time. Try to force a conflict, and then carefully resolve the conflict.

2.6. Branches and Pull Requests

In this subsection, we describe branches and pull requests which are slightly more advanced topics but tremendously useful. Students could probably skim this subsection initially, and then revisit this information later in the semester. Branches and pull requests enable different students to work on different aspects at the project at the same time while keeping their commits separated in the remote repository on GitHub. So far, all of our work has been on the *master* branch. The master branch is the primary default branch. Creating additional branches can enable one student to work on a new feature while also fixing bugs on the master branch, or branches can enable students to experiment with some more advanced ideas but easily revert back to the "stable" master branch.

Let's say that Alice wants to work on a new list of animals in Alice and Bob's shared repository, but she wants to keep her work separate from the primary work they are focusing on. Alice can create a branch called `alice-animals` and commit her new ideas on that branch. It is usually good practice to prefix branch names with your NetID to ensure that branch names are unique. The following commands will first display the branches in the local repository using the `git branch` command before creating a new branch called `alice-animals`.

```
% cd ${ALICE}
% git branch
% git checkout -b alice-animals
% git branch
% git status
```

The `git branch` command uses an asterisk (`*`) to indicate the current branch. The `git status` command also indicates the current branch. Alice can now create a new file and commit her changes to this new branch.

```
% cd ${ALICE}
% git branch
% echo "cow"  > animals.txt
% echo "pig" >> animals.txt
% echo "dog" >> animals.txt
% git add animals.txt
% git commit -m "ALICE: initial animal list"
% git log --oneline --graph --decorate
```

The `--decorate` command line option with the `git log` command will show which commits are on which branch. It should be clear that the `alice-animals` branch is one commit ahead of the `master` branch. Pushing this branch to the remote repository on GitHub requires a slightly more complicated syntax. We need to specify which branch to push to which remote repository:

```
% cd ${ALICE}
% git push -u origin alice-animals
% cat animals.txt
```

The name `origin` refers to the remote repository that the local repository was originally cloned from (i.e., the remote repository on GitHub). You can now see this new branch on GitHub here:

- `https://github.com/<githubid>/ece4750-tut2-git/branches`

You can browse the commits and source code in the `alice-animals` just like the `master` branch. If Bob wants to checkout Alice's new branch, he needs to use a slightly different syntax.

```
% cd ${BOB}
% git pull --rebase
% git checkout --track origin/alice-animals
% git branch
% cat animals.txt
```

Alice and Bob can switch back to the `master` branch using the `git checkout` command.

```
% cd ${ALICE}
% git checkout master
% git branch
% ls
% cd ${BOB}
% git checkout master
% git branch
% ls
```

The `git branch` command should indicate that both Alice and Bob are now on the `master` branch, and there should no longer be an `animals.txt` file in the working directory. One strength of Git is that it makes it very easy to switch back and forth between branches.

Once Alice has worked on her new branch, she might be ready to merge that branch back into the `master` branch so it becomes part of the primary project. GitHub has a nice feature called *pull requests* that simply this process. To create a pull request, Alice would first go to the branch page on GitHub for this repository.

- `https://github.com/<githubid>/ece4750-tut2-git/branches`

She then just needs to click on *New pull request* next to her branch. *You must carefully select the base fork!* If you simply choose the default you will try to merge your branch into the repository that is part of the `cornell-ece4750` GitHub organization. Click on *base fork* and select *<githubid>/ece4750-tut2-git*. Alice can leave a comment about what this new branch does. Other students can use the pull request page on GitHub to comment on and monitor the new branch.

- `https://github.com/<githubid>/ece4750-tut2-git/pull/1`

Users can continue to develop and work on the branch until it is ready to be merged into `master`. When the pull request is ready to be accepted, a user simply clicks on *Merge pull request* on the GitHub pull request page. When this is finished the Git history for this example would look like this:

```
% cd ${ALICE}
% git pull
% git log --oneline --graph
*   f77c7f2 Merge pull request #1 from cbatten/alice-animals
|\
| * fe471e9 ALICE: initial animal list
* | 80765f3 BOB: added date to fruit list
|/
* 7393cac ALICE: added kiwi to fruit list
* 4c1fff6 ALICE: added kiwi to fruit list
* 1982bea BOB: added potato to vegetable list
* 5fd4d2e ALICE: added plum to fruit list
*   eaab790 Merge branch 'master' of github.com:cbatten/ece4750-tut2-git
|\
| * e44ab18 ALICE: added lemon to fruit list
* | 4a053a6 BOB: initial vegetable list
|/
* 2a431b0 BOB: added peach to fruit list
* 98ad45a ALICE: added banana to fruit list
```

★ *To-Do On Your Own:* Have Bob create his own branch for development, and then create a new file named `states.txt` with the names of states. Have Bob commit his changes to a new branch and push this branch to the remote repository on GitHub. Finally, have Alice pull this new branch into her local repository.

2.7. Conclusion

This tutorial hopefully helped you become familiar with Git and how to use it for version control. Git is used heavily in both industry and academia, and will be an invaluable skill to market yourself as having. There are many other commands you might want to use for "rolling-back" to previous versions, managing branches, and tagging specific commits. You are definitely encouraged to read more on Git and GitHub.

Acknowledgments

This tutorial was originally developed as separate Linux and Git tutorials for the ECE 4750 Computer Architecture course at Cornell University by the painstaking work of Shreesha Srinath, Christopher Torng, and Christopher Batten. With permission from Professor Batten, Cornell's HKN Chapter forked and modified these tutorials for the Linux/Git workshop during Fall 2015. For the original tutorials, please refer to the ECE 4750 webpage. We thank Professor Batten, Shreesha, and Chris for letting us use these fantastic tutorials.