Setting up your UNIX Environment for Large Projects

It's time to get started with working on large projects with UNIX! Many of you have probably worked on a Windows for previous smaller, locally-contained projects. However, with larger projects that may also require logging into other servers, this may be quite difficult. This is because testing via puTTY, the SSH-client commonly used for Windows, may also be unreliable. Not only that, editing becomes impossible without the internet, and working with file stored on another server can be slow if you are not using a good internet connection. In this document, we are going to try to help you get set up with your own personal UNIX environment that will allow you to work quickly and efficiently offline, and also have the ability to work with non-local servers.

Table of Contents

Installing the UNIX environment, Ubuntu

UNIX is just your typical terminal, but there are virtual machine environments that allow you to interact with that terminal in more user-friendly ways. One such environment is the Ubuntu framework, which can be installed via Windows. It is a graphical interface to the UNIX terminal, allowing you to use the terminal, but still access many other programs you would want to use in Windows, like Eclipse/Emacs/Vim, Firefox/Chrome, Spotify, PDF Viewers, etc. It can be downloaded http://www.ubuntu.com/download/desktop/windows-installer.

Note: If you already have an Ubuntu, you can skip this page. However, if you would like to replace your current copy of Ubuntu with Ubuntu 12.04 LTS (the version that we are recommending), feel free to back up all of your files with an external drive or Dropbox, uninstall Ubuntu via the Windows Control Panel, and follow the rest of the instructions on this page.

On the web page, you will notice the option to choose your release of Ubuntu before downloading. We recommend installing Ubuntu 12.04.1 LTS. LTS stands for "Long Term Support", meaning that this version of Ubuntu will continue to be patched and upgraded for much longer than normal releases. 12.04.1 is the newest LTS version, and we recommend it over 12.10 unless your computer's hardware is new and graphics are compatible with 12.10, and if you don't mind continually uninstalling and reinstalling your Ubuntu (the software for upgrading from version to version can be buggy at times and may necessitate an uninstall and fresh reinstall of Ubuntu).

Once you have downloaded the Ubuntu installer, run it. It will eventually ask you to set a few custom settings for your Ubuntu, including a username and password. On the left side of this settings panel, there will be a box designating how much space is being allotted for your Ubuntu. You should max this out if you can at 30 GB so that you can run larger programs (like the memory-hogging Eclipse) without causing your memory to fill up too quickly. Then click "OK" after selecting your username and password, and wait for installation to finish.

Now that Ubuntu is now installed, whenever you start your computer, you will be prompted to start Ubuntu or Windows. This does mean you cannot access your Windows files from Ubuntu and vice versa, though.

Why does my Windows 8 take so long to load now?
This is because your default settings on Windows 8 cause the computer to load Windows 8 first, and then give you a choice of Ubuntu or Windows 8. After you make your choice, they reload Windows if you choose Windows 8. They also mean that you have to experience extra wait time for Windows to load even if you want to go to Ubuntu. You can change this by making Ubuntu your default OS choice on that screen. From now on, only basic settings on the computer will load before you make your choice, and loading will get that much faster.

Setting Up Your Ubuntu

Setting up your new Ubuntu requires knowing what kind of software you want. Here are a few highly suggested programs for you to add on to your Ubuntu installation, and how to install them.

Emacs
Sometimes, you just need a quick editor that actually can format your stuff, unlike gedit, the default text editor on Ubuntu (it sucks don't use it). You can get this by typing this in your terminal:

```
sudo apt-get install emacs
```

Vim
The text editor that has a slightly higher learning-curve than Emacs, but has easier-to-hit shortcuts. You can get this by typing this in your terminal:

```
sudo apt-get install vim
```

Eclipse
You may as well install Eclipse Juno if you want the popular Java editor, Eclipse, on your computer. The installation is a little more complicated, so the instructions are here:
http://blog.markloiseau.com/2012/07/install-eclipse-juno-and-android-sdk-on-ubuntu/

Subversion
This may be necessary for setting up some school repository business on your local machine. Version control like Subversion allows you to store files (and previous versions of those files) online, and also allows for teams to edit the same files together. This or Git will be required for your projects. To get svn (short for subversion), type:

```
sudo apt-get install subversion
```

Git
A slightly more powerful version of version control is called git. This software allows for committing software to "save" changes before pushing them into the online repository. This can be installed with:

```
sudo apt-get install git
```

To set the git commit message editor to Vim, type:

```
git config --global core.editor vim
```

Otherwise, type this if you have Emacs installed (you should have at least Emacs or Vim on your computer at this point):

```
git config --global core.editor emacs
```

Gitk
A GUI for git that allows for users to view changes so far in the git repository, and undo changes cleanly. It can be installed with:

```
sudo apt-get install gitk
```

Google Chrome
Some people just prefer using Chrome to Firefox. This isn't as easy as the other programs to set up, but can be done with these commands (in order):

```
wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub | sudo apt-key add -
sudo sh -c 'echo "deb http://dl.google.com/linux/chrome/deb/ stable main" $>>$
/etc/apt/sources.list.d/google.list'
sudo apt-get update
sudo apt-get install google-chrome-stable
```

Spotify For Linux
You can still listen to your music without downloading all of it to your Ubuntu by downloading Spotify. Run these commands in order (after installing emacs):

```
sudo emacs /etc/apt/sources.list
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 94558F59
sudo apt-get update
sudo apt-get install spotify-client
```

Dropbox
Dropbox is very useful if you are worried about losing your files should Ubuntu bug out for some reason (this does happen from time to time, and is a legitimate concern). This can be installed by going to your software manager, and searching for Dropbox, and installing it from there. You can ignore any warnings about the software not being downloaded from the Dropbox website and thus not being as stable or updated.

CCSM
CCSM is a software that allows you to tweak the settings on your Ubuntu at the administrator level. This is a very powerful tool, and should be used with extreme caution. Some useful tweaks will be discussed in the document “Getting Off the Ground with Compiz”, but it is important to note here that unsafe usage of CCSM can actually cause Ubuntu to crash on startup. If this happens, the only thing you can do is to uninstall Ubuntu from your Windows, and re-install it. Here is how CCSM can be downloaded:

```
sudo apt-get install compizconfig-settings-manager
```

Basic UNIX Commands Every Programmer Should Know

- ls – Displays the contents of a folder
  - la -a – Displays all of the contents of a folder – even its hidden files
  - ls -l – Displays all of the non-hidden contents of a folder – along with their file permissions and other details
- cp – Copies a file from location to another (2 arguments must be given). If the second argument is a file, that second argument is overridden by a new copy of the first argument. If the second argument is a folder, a copy of the first argument is put into that folder.
  - cp -R – Recursive copy, meaning that if the first argument is a folder, all of the contents of that folder are copied along with it.
- rm – Removes a file
  - rm -r – Recursively removes files from the target. Thus, if the argument is a folder, all of its contents are removed, too.
  - rm -f – Force the remove. This may be necessary for some folders
- mv – Moves a file (2 arguments must be given). If the second argument is a file, the file is renamed into the second argument. If the second argument is a folder, the file is not renamed, but into the folder.
- ssh – handles SSH login protocols. It takes in an address like cs9e-1bm@star.cs.berkeley.edu, and tries to log into that address. This may require users to provide login credentials.
- scp – takes in a file or folder, and copies it into a non-local path. This is done by concatenating a non-local address with a colon, followed by the non-local path, like in the following example: scp testFile cs9e-1bm@star.cs.berkeley.edu:newPath/for/myTestFile/
  - scp -r – Recursively copies all of a folder's contents so that all of the folder's contents are copied to the nonlocal address, as well.

UNIX Wildcards

Sometimes, when trying to work all files in a folder, or perhaps only the files that start with “a”, or copying multiple files to another folder, it is very annoying to work with each file, one at a time. This is one instance when wildcards can be useful.

Wildcards can be described as UNIX shorthand – they are used to represent sets of strings so that you can copy, remove, or do any kind of work with multiple files at once. We will not go into detail about how to use them, and will only discuss the use of the ‘*’ character and the tab character in regular expressions, since this is probably all you will need for typical scripting and terminal use.

‘*’ stands for all possible letters – and as many of them as you may need. What this means is that the command:

```
rm apt*
```

can remove all files in the current folder that start with the string “apt”. Similarly, all .class files in the current directory can be removed with the following command:

```
rm *.class
```

Tab characters are implicitly inserted by typing the “tab” key. These don’t do anything by themselves, but they are extremely useful in autocompleting commands, file names, and folder names. This means that if you have one really long file name like “extremelyLongFileName”, you don’t necessarily have to type the whole file name. Instead, you can type “tab” after enough characters have been typed already to make this unique. This means that if you have 2 files in your directory – “extremelyLongFileName” and “extraFile”, you can press the tab key after you have typed “extre” to autogenerate the string for “extremelyLongFileName”. Similarly, typing “extra” followed by a tab character will create the string “extraFile”.

You can also complete aliases and commands with these wildcards. You can see this by typing “emac” followed by a tab character – UNIX will complete your command for you.

NOTE: If you’re pressing tab, but nothing is showing up, it is because there is either no file/command that starts with the letters that you have typed so far, or because you have multiple files/commands that start with the letters that you have typed so far. You can find out of it is the latter by pressing the tab key twice – if there are conflicting options for your text so far, they will all be displayed in the terminal. You can try this by typing “e” into your terminal, and tapping the tab key a few times. Tapping the tab key does not delete the command so far, either, so you can continue to type your command after you see the conflicts.

GREP

Sometimes, when dealing with large projects, you will realize that you need to look for different instances of a certain variable or method. You may also find yourself in an instance where you need to find where you declared that java.util.concurrent.locks.ReentrantReadWriteLock, especially when you don't remember the name of the variable you used for that rarely-used object. Instances like these can be where grep comes in very useful. In fact, this is why GREP has earned its own section in this document – it is useful enough to set apart from other normal UNIX commands and deserves practice of its own.

GREP can be used in 2 different ways. The first usage is to search a file for a keyword. This is done by calling passing in a regular expression (regex), and passing in any number of files to search.

The second way GREP can be called is by calling it recursively on a folder. This is probably the more useful option for GREP. This is called by raising the –R flag right after the word "GREP", and then finish the call as usual.

Sometimes, though, when calling GREP recursively on a folder, certain ghost/binary files will be matched as well. During this case, GREP will report the matches, which is not helpful for your projects. For this reason, when calling GREP on a folder, it is highly advised to also raise a –I (capital "i") flag. Thus, the calls will always begin like with "grep -IR".

SSH

SSH is a way to log into a remote server. If you have been coding on a Windows, you may have had this blackboxed away for you with the software puTTY. On an Ubuntu, this can't be done for you, but it's not that much harder without puTTY.

On an Ubuntu, you can do this with the command "ssh", and passing in the nonlocal address of the server. Given the settings on the server, though, you may have to give it some login credentials.

Always giving the server your credentials, however, can feel like a waste of time after a while. This really doesn't have to be done, especially with modern RSA technology. When you log into a remote server, you can have your computer automatically pass your public key to the server.

During RSA encryption, your private key is given to the server. It uses the public key to encrypt a message, and gives you the encrypted message. Your computer then takes the encrypted message, and uses your private key to decrypt the message. Your machine then send the decrypted message back to the server, and if it matches the original message, the server allows you in as if you had given it your login credentials in the first place.

Setup for automatic secure login via RSA private-public is fairly easy to setup. It can be done by putting each of the following commands into the terminal (replace "serverNameForYouToReplace" with the server you are trying to log into):

```
ssh-keygen -t rsa
cd .ssh
scp id_rsa.pub serverNameForYouToReplace:.
ssh serverNameForYouToReplace
```

Once you have logged into the nonlocal server via SSH, execute the following commands:

```
cat id_rsa.pub >> .ssh/authorized_keys
rm id_rsa.pub
cat .ssh/id_rsa.pub >> .ssh/authorized_keys
```

Aliasing and Auto-Sourcing Your Own Commands

All machines that run Linux/UNIX are mostly written in C, and when opening a terminal, the machine actually executes one of the 3 files in the home folder of the computer: .bashrc, .bash_profile, .profile (notice that there is a period before each of these filenames). The computer will always prefer the file named .bashrc over the other 2, and will run it instead of the other 2. If it is not present, the computer will try to run .bash_profile. If both .bashrc and .bash_profile are not there, then it will try to run .profile. The method by which the file is executed is special, and we call it "sourcing" of that file. You can always source extra files manually, but one of these 3 files will always automatically be sourced.

Before continuing, verify which one of these files is present on your computer, and if you are using a nonlocal server, verify which one of these files is present on that server.

The file that is executed when the terminal is started is important because users can use it to define extra commands. In UNIX, it is actually possible to define your own commands (this is called **aliasing**), and even define complex functions (known as **scripts**) in other files. In this document, we will only discuss aliasing, as scripting requires a much greater understanding of the Bash syntax and language.

Aliasing a new command is often used to make processes run more smoothly in UNIX. A good example is to create an alias that does several things. One example is the following:

```
alias pmake="make clean; make"
```

which allows you to clean out all automatically-generated files, and recompile all of your files.

Another example is:

```
alias rgrep="grep -IR"
```

Which makes saves the time of always raising the grep flags.

There are many other things that can be made possible with aliasing, with the only limit being that aliased commands are not allowed to contain spaces. Even SSH can be simplified with aliases:

```
alias 9e="ssh cs9e-1bm@hive13.cs.berkeley.edu"
```

If this command is combined with the RSA-enabled login described in the SSH portion of this document, just typing "9e" into the terminal will automatically log you into the account cs9e-1bm@hive13.cs.berkeley.edu.

What if you want to share aliased commands with others? Sending your .bashrc file isn't the easiest way to do this, since other users may also have their own custom commands. However, if you are using version control like subversion or git, you can share a file containing all of these aliased commands. Fellow project partners can then update their copies of the repository, and manually source the file. This can be done by entering the directory containing that file, and calling "source" with that file as an argument:

```
source fileContainingTheAliases
```

If you would like to use even less keystrokes, a period can be used to represent the word "source":

```
. fileContainingTheAliases
```

Teammates can also, after getting the aliases through version control, source the file through their .bashrc, .bash_profile, or .profile files. This can be done by adding

```
source fileContainingTheAliases
```

to the bottom of the file that is used by the computer.

Here are some other sample commands that have been used to give you ideas for sourcing your own aliases.

```
alias fullClean=”rm -rf testOutputs/*; make clean; make”
alias fc=”fullClean”

alias cleanCheck=”make check; make clean”
alias cc=”cleanCheck”

alias projectFolder=”cd ~/projectFolder/trunk/proj1”
alias home=”pf”

alias up=”svn up”
alias reupdate=”svn up; fc”
alias commit=”svn commit -m “automated commit from ldwu”
alias revert=”svn revert”

alias writer="libreoffice --writer" #text editing for doc files
alias word="writer"
alias officeWriter="writer"
alias officeWord="writer"
alias libreWriter="writer"
alias libreWord="writer"

alias adobe=”evince” #pdf document viewer
alias pdf=”adobe”

alias sl=”ls” #fixing typos
alias emasc=”emacs”

alias emacs=”emacs &” #by default, emacs will always open without clogging your terminal
alias gk=”gitk &” #gitk will always open without clogging terminal
```

If you would like to start in the project folder when opening a terminal or logging into a server, you can also always include this line in the .bashrc, .bash_profile, or .profile files:

```
cd projectDirectory
```

Sometimes, the terminal comes with pre-aliased commands, and you don't want those commands. You can always remove an alias by calling "unalias" on that command. A good example is how some terminals have, by default, an interactive form of removing files, "rm –i". This can be very frustrating if you want to remove many files, so you can put this command in your .bashrc, .bash_profile, or .profile files:

```
Unalias rm
```

If you want to check if a certain command is an alias, you can also always just type "alias" followed by the command name without assigning any value to this alias, like this:

```
alias rm
```

If "rm" is an alias (not a built-in command), then the terminal will display the alias.