ECE199JL: Introduction to Computer Engineering **Fall 2012**
Lab 0 **Due: see Section 1 below**

Introduction to the Lab Environment

The goal of this introductory lab exercise is to introduce you to the tools and environment that you will use throughout the coming year as well as in future engineering classes. The Engineering Workstations (EWS) labs serve as the primary academic computing resource for students in the College, and can be accessed either directly or remotely. To ensure that you have a consistent software environment, most of your classes (including this one) will require that you make use of EWS for assignments.

By the end of this lab, you will be able to:

- Navigate the Linux directory structure through the command line interface.
- Add, remove, and edit files using the command line terminal.
- Compile a program.
- Use Subversion to maintain a backup copy of your work.

Please note: **To help you get started on this assignment, all discussion sections will meet in the EWS lab located in 440 DCL on Tuesday 28 August.** You will need to know the password to your Illinois (AD domain) account in order to log in.

1 Timing

This laboratory is broken up into several parts:

- Before discussion section on Tuesday 28 August: please read Section 2.
- In discussion section, we will walk through the first steps of the lab (Section 3.2) to make sure that you are able to complete the lab on your own later.
- By Friday 31 August, you need to complete the survey (Section 3.3), send it to us, and sign up for a meeting slot.
- Before the second discussion section (on Tuesday 4 September), you need to complete Section 3.4.

The lab also contains a number of appendices intended to serve as reference material.

2 Introduction to Linux/Unix

The computers in the lab are running a Linux operating system. An **operating system** is the most basic software running on a computer. It manages the computer's resources (such as memory), peripherals (such as a mouse and a keyboard), and application execution. Microsoft Windows, Mac OS, and Unix are examples of operating systems. Linux is a "Unix-like" operating system. Don't worry if you do not know what Unix is. Just know that Linux and Unix are very similar.

The Linux computers in DCL 440 and DCL 520 as well as in Everitt 252 and Grainger 057 are managed by EWS. When you are logged in to a Linux computer managed by EWS, all of your personal files appear the same, regardless of which physical EWS computer you are sitting at.

Typically when you first log in to a modern operating system, you will be greeted with a graphical desktop environment, known as a **Graphical User Interface** or GUI for short. GUIs allow the user to interact with the computer by manipulating graphical windows and icons using the mouse. For example, you can double-click on a folder icon on your desktop and a window opens up showing the contents of that folder. Generally speaking, you will find documents and folders inside a given folder. These folders can also have folders and documents inside of them.

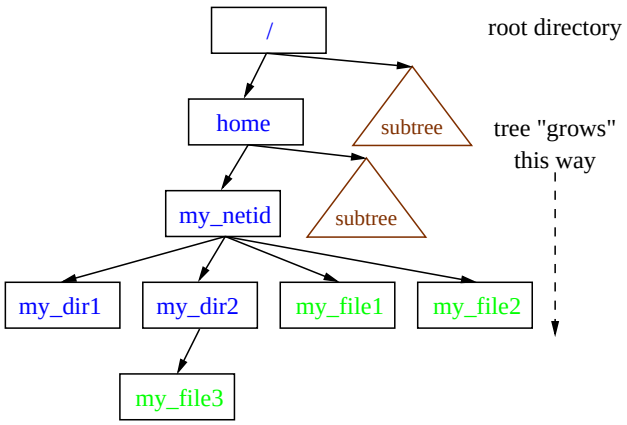

Figure 1: Directory Structure (Tree Representation).

On a Unix system, folders are known as **directories**, and documents are known as **files**. Directories inside of directories are called **subdirectories** and the directory containing a given directory is called the **parent directory**. Files and directories form a directory hierarchy or **directory structure**. You can think of the directory structure as a tree where each folder (directory) is a branch of the tree and the leaves are the documents (files). Figure 1 illustrates this tree structure. The root of the tree is denoted by "/" and the tree grows downwards.

Each file and directory in the directory structure can be reached using that file's/directory's **path**. Understanding paths is your key to navigating the directory structure. There are two types of paths: absolute path and relative path. An **absolute path** starts at the root of the filesystem (starts at `/`) and specifies each directory prior to reaching the destination file/directory. Each step in this **traversal** is separated by a `/`. Figure 2 identifies the absolute path for each directory in a portion of Fig. 1's directory structure.

The **current working directory** is where you currently are in the overall directory structure. This directory is analogous to viewing a folder in a window using the GUI. The window that you are operating in is the current working directory. You can only manipulate contents in that window. You would need to change your current working directory to manipulate files in a different directory. Your **home directory** is the current working directory on startup. By default, EWS sets home directories to be at the path `/home/my_netid` where `my_netid` is your personal NetID.

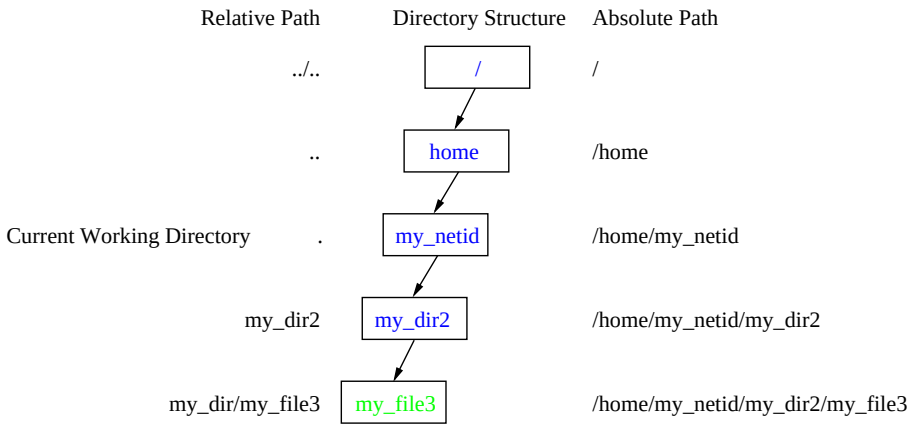

Figure 2: Section of directory structure from Fig. 1 with corresponding absolute and relative paths to each of the directories.

Sometimes, the absolute path is inconvenient to type out fully, especially with larger directory structures. You can use the **relative path** syntax to start the navigation at the current directory instead of at the root of the directory structure. A **syntax** is the method to write or express a concept. To indicate a relative path, you do not start the path with a `/`. Also you use `..` to reference the parent directory. Figure 2 shows the relative path to each of the directories from the current working directory located at `/home/my_netid`.

A GUI is convenient for new/casual users but this convenience comes at the price of control, expressiveness, and speed. The GUI takes away control from the user so the user can't harm the computer unintentionally. Most operating systems have an alternative interface called a **command line interface**. In this interface, instead of telling the system what to do by clicking on icons, you type text commands using the keyboard into a **terminal**, and the computer responds with a textual response to this terminal. You can express advanced actions easily through the command line interface that would take many clicks to do in a GUI, if it were possible to do through the GUI in the first place.

When you open a terminal window, there is a basic program running called the **shell** which reads and interprets the commands you type. The shell breaks down what you type into programs, flags, and arguments. Commands used in command line follow the following syntax:

```
program_name [flags] [arg1] ...
```

The above is called a synopsis. A **synopsis** describes command line syntax in an abstract way. It is not meant to be taken literally; each word is a placeholder for the name of real programs and arguments. [Words in square braces] represent optional arguments. The ellipsis ("`...`") means that optional additional arguments can follow.

Figure 3 shows an example of a command and its syntax breakdown.

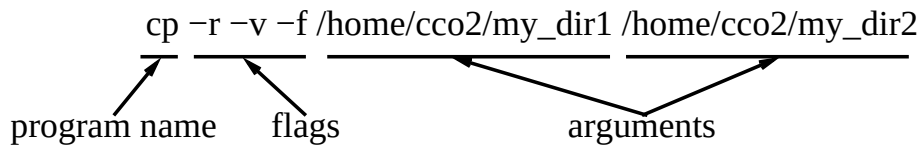

Figure 3: Example command and syntax breakdown.

3 Step by Step Instructions

3.1 Before Discussion Section

1. You should read and try to understand the high-level overview presented in Section 2. Having a good high-level idea of what you are working on will make the later sections clearer and quicker. Be sure to ask a TA if the concepts are unclear.

3.2 Discussion Section

1. Log in to an EWS machine using your netid and Active Directory (AD) password.

2. Launch a new terminal.

   You will be greeted with a GUI upon logging in. You will want to launch a new terminal to access the command line interface. To do so, find "Applications" on the menu bar and click on "System Tools", then "Terminal".

3. Get a copy of the subversion repository by typing: `tar xjf /class/ece199/ece199jl-repo.tar.bz2`

   Subversion is a source control tool that allows you to manage changes to a body of code, even if more than one person is working on the code at the same time. The files for this first lab will initially be a private copy. **Be warned: executing the command in this step again will destroy your stored copy of the lab files!**

   This step is included as a temporary measure until your accounts are fully set up within EWS. Watch the class web board for instructions on the transition once the accounts are ready, hopefully by the end of our first week.

4. Checkout a working copy from your subversion repository (the `repo` directory created in the last step).

   You will need to obtain a copy of whatever you will work on. In this class, we will be using Subversion, a revision control software. Subversion maintains backups of your files as well as merges changes you or your team members make so that multiple members of a group can edit the same file safely.

   There are two parts to Subversion: the repository and the working copy, not to be confused with current working directory. The **repository** is the master location where all the backups are stored. Each backup is associated with a **revision number** that increases each time you or your teammates save a working copy to the repository. The **working copy** is your local copy of the files being saved in the repository. Changes you make to the working copy do not automatically get saved to the repository. You must perform a subversion **commit** to save your working copy to the repository. It is good practice is to commit often so if something happens to your working copy, you have a recent backup, or even lots of backups to restore from. For more on Subversion, read Appendix D.

   To obtain the working copy of whatever you will work on, you will be performing a subversion **checkout**. Run the command: `svn checkout file:///home/my_netid/repo/lab0` This command copies the files from your Subversion repository into your current working directory.

5. Navigate to the file `discussion0.txt` located at `/home/my_netid/lab0/discussion/discussion0.txt`.

   The file `discussion0.txt` is located in the relative path from the directory you checked out your code from: `lab0/discussion/discussion0.txt`. The main commands to navigate the directory structure are `pwd`, `ls`, and `cd`. The `pwd` (print working directory) command prints where you currently are in the directory structure. The `ls` (list) command prints the contents of a given directory. Typing `ls` by itself without any arugments will print the contents of the current working directory. The `cd` (change directory) command changes your current working directory to the path specified by the argument. More information on these commands can be found in Appendix A.

   To find `discussion0.txt`, first you must identify where you currently are. Run the command: `pwd`. Your output should read: `/home/my_netid` where `my_netid` is your netid. Now that you know where

you are and that your destination directory path is `/home/my_netid/lab0/discussion/`, you can run: `cd lab0/discussion` to move your current working directory to your destination directory path.

6. Edit `discussion0.txt` and fill in your name and today's date.

   Now that you've found the file, you should edit it. If you have a favorite editor (`vim`, `gedit`, `emacs`, `nano`, etc), you can use that if you prefer. We recommend `gedit` and `vim`. These instructions will use `gedit` since `gedit` has a very easy learning curve compared to `vim`, but know that `vim` is an extremely common command line editor and we strongly encourage you to sit down and learn `vim`.

   To launch `gedit` on `discussion0.txt`, run the command: `gedit discussion0.txt`. A graphical window should pop up that contains some text.

7. Commit your `discussion0.txt` changes to your repository with the message "My first commit".

   You've now made changes in this file but they are all local (confined to your working copy). The repository does not have these changes. You must perform an Subversion commit to push your local changes to the repository.

   To save the changes to the repository, run the command: `svn commit -m "My first commit"` The `-m "<message>"` flag is required when you commit changes to the repository. The message is stored in a log file that you can read using `svn log`. Each entry in the log has a revision number and commit message so if you use detailed commit messages, you know what changes were made for that specific revision.

8. Delete `discussion0.txt` from your working copy.

   Now let's say you're looking to delete your `discussion0.txt` file. You can use the `rm` command to delete the file.

   Run the command: `rm discussion0.txt`. If you type `ls`, you should notice that `discussion0.txt` is no longer there. Remember, once you delete the file, you cannot undo it unless you have some sort of backup or revision control software (SVN is a revision control software).

9. Restore `discussion0.txt` from your repository.

   Your working copy no longer has `discussion0.txt` but your Subversion repository has a copy of the file still. You can perform a Subversion **update** to update your working copy with the copy stored in the repository.

   Run the command: `svn update`. If you type `ls`, you should see `discussion0.txt` in your working copy once again.

10. Demo steps 7 and 8 to a TA.

3.3 Survey

1. Inside your working copy, you will have a file called `survey.txt`. Fill out this survey and e-mail it to Prof. Lumetta (lumetta@illnois.edu). Be sure to include "ECE199JL" in your subject line, as he receives a lot of e-mail.

2. Schedule time to meet with the professors. This is an informal meeting where you can get to know your professors and the professors can get to know more about you.

3.4 Compile and Execute Code

1. Inside your working copy, there is a directory called `mp0`. Move into that directory. You should see `hello-world.c`. The `.c` extension indicates that this file is a C programming language source file. A **source file** contains the instructions a specific program should execute. The source file is written in a particular coding language - in this case, the language is C.

2. We will need to **compile** the source files into an executable that the computer can then run/execute. To do this, run the command:

   ```
   gcc -o hello-world hello-world.c
   ```

   This command invokes a program called a **compiler**. The `-o <executable name>` specifies the name of the executable output from the compiler. If this flag is not specified, the default name for the executable is `a.out`. After all the flags, the source files are named. In this case, there is only one source file. For more information, read Appendix C.

3. Run the executable using the command: `./hello-world`. The program should output "Hello World!".

4. If you want to change the program, you will need to edit the source file. Open the source file in a text editor and add "My name is <your name here>!" and your name in place of <your name here> after "Hello World!". So if your name is Chris, the line should read: `printf("Hello World!  My name is Chris!\n");`

5. Anytime you make changes to the program, you will need to recompile the files to generate an updated executable. Do so and run the program again. You should see the output "Hello World! My name is <your name here>!".

6. Commit your changes to your svn repository. Congratulations, you've finished the lab!

Appendix A: Directory Navigation Commands

Fig. 4 is the sample directory structure used to illustrate the commands in Appendix A and B.

Command	Synopsys	Full Name	Function
pwd	`pwd`	print working directory	Print the absolute path of the current working directory

Below shows the output of `pwd`.

```
[netid@linux3 ~]$ pwd
/home/my_netid
```

Command	Synopsys	Full Name	Function
ls	`ls [flags] [directory_path]`	list contents	Prints the contents of the directories specified by `directory_path` or the current directory if `directory_path` is not specified

Below shows the output of `ls` when run with no arguments.

```
[netid@linux3 ~]$ ls
my_dir1 my_dir2 my_file1 my_file2
```

Depending on your terminal, directory and file names could have different colors. By default, `ls` does not show hidden files. To do so, use the `-a` flag.

```
[netid@linux3 ~]$ ls -a
. .. my_dir1 my_dir2 my_file1 my_file2
```

The `.` and `..` are hidden in every directory. `.` refers to the directory itself. This provides a convenient way for a path to reference itself without needing to type out the full absolute path. `..` refers to that directory's parent directory. Other common hidden files are configuration files. All hidden files begin with a `.` as in `.bashrc` or `.vimrc` which are `bash` and `vim` configuration files.

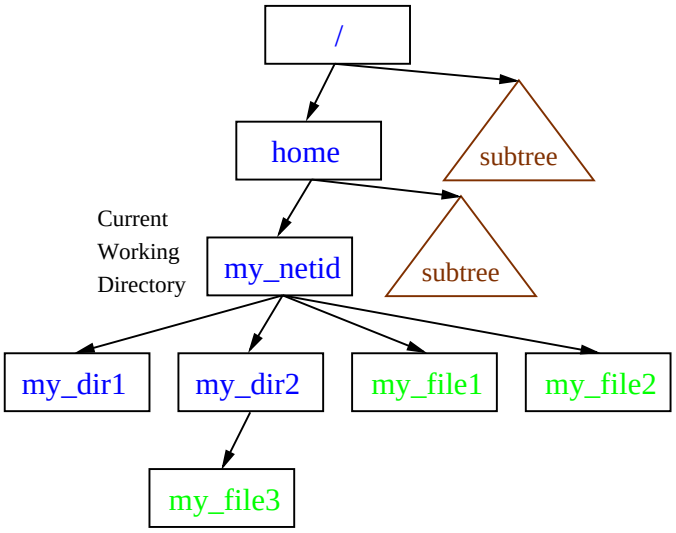

Figure 4: Sample directory structure.

Command	Synopsys	Full Name	Function
cd	`cd directory_path`	change directory	Changes the current working directory to the one specified by `directory_path`

The `directory_path` argument can be an absolute or relative path. To change the directory to `/home`, one could use the absolute path:

```
[netid@linux3 ~]$ cd /home
```

or use the relative path:

```
[netid@linux3 ~]$ cd ..
```

because `/home` is the parent directory of the current working directory `/home/my_netid`. Note that if the `cd` command is successful, there will be no output.

Appendix B: Moving and Removing Directories and Files

Command	Synopsys	Full Name	Function
mkdir	`mkdir directory_path`	make empty directory	Creates an empty directory at `directory_path`

The `directory_path` specifies the name and place the new directory should be created. To create a new subdirectory named `my_dir3` in the current working directory, one would run:

```
[netid@linux3 ~]$ mkdir my_dir3
```

If the `mkdir` command is successful, there will be no output. Use `ls` to verify the new directory was created successfully. Note that one cannot create multiple hierarchical subdirectories at once. Each preceding subdirectory must be created before the next subdirectory can be made.

Command	Synopsys	Full Name	Function
rmdir	`rmdir directory_path`	remove empty directory	Removes an empty directory at `directory_path`

The directory at `directory_path` must be empty for rmdir to succeed. To delete a new subdirectory named `my_dir3` in the current working directory, one would run:

```
[netid@linux3 ~]$ rmdir my_dir3
```

There is a shortcut to removing directories that are non-empty using the `rm` program which is discussed next.

Command	Synopsys	Full Name	Function
rm	`rm [flags] path ...`	Remove file	Removes file at `path`

Suppose one wants to remove `my_file3` located in `my_dir2`, use the following command:

```
[netid@linux3 ~]$ rm my_dir2/my_file3
```

As with the other commands, no output means no errors but use `ls` to verify the command succeeded.

More than one file can be deleted by simply adding more paths at the end of the command. One shorthand way of specifying all contents in the current directory is to use `*`. `*` is a wildcard that refers to all file and directory names in the current directory. To delete all files in the current working directory, run the following:

```
[netid@linux3 ~]$ rm *
```

The `*` can also be used to fill partial names. For example, `my_file*` autofills into `my_file1` and `my_file2` when used in `/home/my_netid`.

One can remove full directories using the `-r` flag. The `-r` flag will remove a directory and all of its contents (including subdirectories) specified by the path.

```
[netid@linux3 ~]$ rm -r my_dir2
```

removes `my_dir2` and its all of its contents from the directory structure. Be very careful when using the `rm` program. Once a file or directory is removed, it's gone. There is no recycle bin to find your removed files.

Command	Synopsys	Full Name	Function
mv	`mv source_path destination_path`	Move file/directory	Moves file or directory from `source_path` to `destination_path`

To move `my_file3` from `my_dir2` to `my_dir1`, use the following command:

```
[netid@linux3 ~]$ mv my_dir2/my_file3 my_dir1
```

A few scenarios exist based on what the `destination_path` points to. If the `destination_path` points to a directory, then the source file or directory will be moved inside that directory. If the `destination_path` points to a new path that does not exist yet, the source file/directory will be moved and renamed to the destination. If the `destination_path` is a file that already exists and the source is a file, the file pointed to by the `destination_path` will be overwritten without any notification.

Command	Synopsys	Full Name	Function
cp	`cp [flags] source_path destination_path`	copy file/directory	Copies file or directory from `source_path` to `destination_path`

The `cp` program behaves the same as `mv`, except that the `cp` duplicates files and directories rather than just moving them. The `-r` flag is use to copy whole directories and subdirectories.

Appendix C: Compiling Code

Once you've edited your source code, you'll need to **compile** the code into an **executable**. This executable is also known as a program or application. In this class, we will use the GNU C compiler (gcc). The syntax is:

```
gcc [flags] file1 ...
```

Flags are denoted with a leading dash. Some flags have arguments associated with them. Below is an example that compiles the source code `foo.c` into an executable.

```
[netid@linux3 ~]$ gcc -g -Wall -o foo foo.c
```

The `-g` flag specifies to compile with debug symbols. These debug symbols allow you to attach an application known as a debugger to help you find bugs in your code. The `-Wall` flag enables all warnings generated during compilation to be displayed as the output from the `gcc` command. The `-o arg1` flag specifies the name of the resulting executable as `arg1`. If `-o arg1` flag is not given, the executable will be named `a.out`.

To run the executable created from the commands above, type:

```
[netid@linux3 ~]$ ./foo
```

Appendix D: Revision Control

Revision control (or version control) is a tool to manage changes to files. Have you ever made a change to a document that you later regretted? A revision control system allows you to keep track of the different versions of a document, so that you can revert back to before you made an unwanted change. Revision control is especially useful in team projects where multiple people are making changes to a set of documents. Revision control software automatically handles separate changes to the same file and merges them so you do not have to.

In this class, we will be using a revision control system called Subversion. Subversion maintains a central repository which has one or more revisions of a directory and all of its contents. The repository is placed on a separate computer/server. You can get a copy of that directory in a process called a **checkout**. You can make local changes to your copy of the directory, and then add the new version to the central repository in a process called a **commit**.

We've already set up a central repository for you. You can view the contents of your repository using with a web browser; just go to:

`https://subversion.ews.illinois.edu/svn/fa12-ece199/[netid]`

where [netid] is a placeholder for your netid. If you can't access your repository, let a TA know.

svn checkout - The program for doing all work in subversion is called `svn`. The checkout syntax is as follows:

```
svn checkout url directory
```

From your home directory, you'll want to do something like this:

```
[netid@linux3 ~]$ svn checkout https://subversion.ews.illinois.edu/svn/fa12-ece199/[netid] ece199
```

This will create a **working copy** of your repository, which is a local copy for you to work on. This working copy is separate from the repository. This command will put this working copy in a directory called `ece199`.

svn status - This command allows you to see what has changed in the your local working copy of the repository Here's a sample output from the command:

```
[netid@linux3 ~]$ svn status
M       foo.txt
```

The "M" means that the file has been modified from the original working copy you checked out. A "D" indicates a deleted file and an "A" indicates a file to add to revision control.

svn commit - Remember that all modifications you make are all on your local working copy. Nothing has changed in the repository. To modify the repository with your changes, you must commit your working copy. An example of this is below:

```
svn commit -m ‘‘Removed last line of foo.txt"
```

In this case, the `-m` (for message) is an option flag with an argument. It specifies a message to be associated with the new version you are sending to the repository. These messages will be useful if you ever realize you made a mistake and need to go back and find a particular revision, so make them good summaries of the changes you've made.

svn add - This command tells the repository what files it should retain. By default, the repository is empty and you must manually add the files you want the repository to track. An example is below:

```
svn add foo.txt
```

`foo.txt` must be a valid file. This command tells the repository to start tracking changes to the `foo.txt` file. When we run `svn status`, we see an "A" next to `foo.txt`. Note that you must run `svn commit` to tell the repository you've added this file.

svn mv, svn rm, svn cp - When we're working with a local copy of a subversion repository, the best way to perform file moving operations is to use the subversion wrapped version of these programs, so that subversion knows what is going on and doesn't get confused. A wrapped version of a program is one that does what the original does with some additional functionality. These commands will work just like the original versions, except that subversion does some additional bookkeeping to track of what has changed:

```
svn mv source destination
svn cp source destination
svn rm file1 ...
```

svn revert - If you make a mistake that you haven't committed, you can revert your changes back to the last commit with the revert command:

```
svn revert [-R] file_or_directory ...
```

You can revert individual files, or entire directories. You will need to use the `-R` flag to revert directories and their contents. Notice that this is a capital R instead of a lowercase r. Unfortunately, different programs don't necessarily agree on the meaning of flags; you'll just have to remember that svn uses a capital R.

svn update - When someone else commits changes to your repository, your working copy will become out of date. To update your working copy of your repository, run the following command from inside your working copy:

```
svn update
```

You should always run this command before you commit to make sure your working copy is up to date.

Appendix E: Other Tips/Commands to Make Your Life Easier

man: To get very detailed syntax for a particular program, use the man utility:

```
man [section] program
```

For example, to get help on using `rm`, run:

```
[netid@linux3 ~]$ man rm
```

To scroll down, hit the return key To exit, hit "q".

Tab Completion: Another useful command line trick is tab completion. Whenever you're typing a path into the terminal, you can just enter the first few characters, hit the tab key, and the shell will attempt to complete the rest for you. You can use this trick as often as you like in a command. On other shells, hitting tab twice will also show you a list of possible completions.

Command History: Similar to tab completion, command history allows you to cycle through previously executed commands in the shell. You can use the up and down arrows to navigate the command history.

Vim: Vim is another popular text editor and our personal recommendation. Vim is a command line based text editor and is a universally recognized text editor. Because it is command line based, it can still be used on operating systems that don't have GUIs (typically found on server/cluster operating systems). Vim has a steeper learning curve but once you get the basics, simple editing becomes quick. There are also lots of special advanced commands that can also make your life easier should you choose to learn/use them. Vim has a graphical counterpart called gvim if you have a GUI available to you. You'll want to read the separate vim reference guide if you've never used vim before and want to try it out. To launch vim, type:

```
vim [file]
```

Makefile: As code projects get bigger and bigger, the line(s) you write to compile your code get larger too. Wouldn't it be easier to just type one command to compile all of your code for you? **Makefiles** provide this. Makefiles are special files that, when executed, auto-generate the compilation commands and flags.

We have provided a sample Makefile for you. In order to execute the compile line from before, all you need to type is:

```
[netid@linux3 ~]$ make
```

Similarly, if you want to remove all of the compiled files and start from scratch (which you will have to do very often), you run the command:

```
[netid@linux3 ~]$ make clean
```

This is just the beginning of the potential uses for Makefiles. They are very powerful, especially when dealing with many files. Keep in mind that they must be written and tailored to auto-generate the commands required for the specific codes you are compiling. They have a special syntax that you must follow. We will give you more information on Makefiles later in our class.