CS 121 **Lab Assignment #1**
Bruce Bolden **10 Points**
January 15, 2015 **Due:** January 29, 2015

Lab 1: Introduction to Unix

The goal of the exercises in this section is to familiarize the reader with basic Unix operations.

Unix is an operating system originally designed in the 60's and 70's to be multi-user, multi-tasking operating system. It is one of the most, if not the most, influential operating systems ever developed.

Unix must have a way of organizing the users' files and of allowing multiple programs to run simultaneously without interfering with each other. Files are organized hieratically in what is known as a 'tree' structure. The root of the tree is called `/` (slash). Each 'branch' of the tree is a subdirectory. Each subdirectory has its own name (subdirectories are equivalent to folders in a Windows or Macintosh environment).

Every user is assigned their own subdirectory whose name is the same as the user's login. Thus, the user `tsoule`, keeps all of his files in a subdirectory called `tsoule` or in subdirectories within the `tsoule` subdirectory. Each subdirectory is protected so that only certain users are allowed access to it. In addition, there are several levels of access, for example you might be allowed to read the files in a certain subdirectory, but not to modify those files. (This is common in subdirectories containing programs that every user wants to use, such as the C++ program, but that users shouldn't be allowed to change.)

There are a number of basic commands that are used to negotiate Unix's file structure:

- **ls** Short for LiSt, this command lists the files and subdirectories in the current subdirectory. It does not list 'hidden' files—files whose names starts with a '.' (dot).
- **ls -a** This command lists all of the files and subdirectories in the current subdirectory including the hidden files.
- **ls -l** This command lists all of the files and subdirectories in the current directory in the 'long' format – it lists information other than just the names.
- **pwd** Short for Print Working Directory, this command tells the used where they are in the files structure. A typical output would be: `/users/faculty/tsoule/` showing that the user is currently in the subdirectory `tsoule`, which is in the subdirectory `faculty`, which is in the subdirectory `users`, which is in the root directory `/`.

- **cd *directoryname*** Short for Change Directory. The command changes which directory the user is currently located.

- **cp *filename1 filename2*** Short for CoPy, this command copies filename1 into filename2. If filename2 already exists, it is replaced with the new file. The file names can include directories.

- **mv *filename1 filename2*** Short for MoVe, this command copies filename1 into filename2 and removes filenmae1. If filename2 already exists, it is replaced with the new file. The file names can include directories.

- **mkdir *directoryname*** Short for MaKe DIRectory, this command creates a new subdirectory.

- **rm *filename*** Short for ReMove, this command removes (deletes) a file.

- **rmdir *directoryname*** Short for ReMove DIRectory, this command removes (deletes) a directory. A directory must be empty before you can remove it.

- **script *filename*** The script command makes a record of everything printed on the screen by writing it to the file *filename*. (If no filename is given it is written to a file called *typescript*.) To use the script command type script *filename*, run the program, **then type exit**. Typing *exit* stops the script. The script file can be printed and turned in as the sample output of a program.

  There are two errors to avoid when using the script command. First, if you fail to type exit everything you do will continue to be dumped into the script file. In particular, if you attempt to open the script file before exiting the script command you will set up an infinite loop. Second, make sure that the *filename* you use is not the same as the name of a program you wrote; otherwise the script file will overwrite (erase) the program and you will have to rewrite the program.

- **man *commandname*** Short for MANual, this command presents help information on the given command.

All Unix directories include two special subdirectories called `.` and `..` (*dot* and *dot dot* respectively). The directory called `.` is the current directory. Thus, for example, the command `cd .` moves the user to the current directory, which has no effect. The directory called `..` refers to the directory above the current directory.

Exercises/Deliverables

These exercises test some of the commands in Unix and provides some practice with text editors in Unix.

1. Use `ssh` to log into the CS Unix system.
2. In your home directory use a text editor to create a file called `hw1`. (You may want to begin with `nano` (or `pico`), a fairly easy to use text editor found on most Unix systems. However, I suggest that you learn `emacs` or `vi`. Editors are a very personal issue.) Record your name, section number, the assignment number (Lab #1) and current date in the file. Save and exit the file.
3. Create a new directory (`mkdir`) in your home directory called `CS_121`.
4. Change (`cd`) to your CS_121 directory .
5. Create a new directory in your CS_121 directory called 'Lab1'.
6. Move the file 'hw1' into the directory called 'Lab1'.
7. Change your current directory to be the new directory 'Lab1'.
8. Use the `pwd` command to determine where your current directory is in the directory hierarchy. Record the results in the 'hw1' file.
9. Create a new file called `program1.cpp`.
10. Enter the following program into the file:

```
/* Lab #1                                        Your name
   Your section number                           The date
 */

#include <iostream> // include the library with I/O commands

using namespace std;

int main()                      // beginning of the program
{
    cout << "Hello, world." << endl;  // print some output

    return 0;
}
```

11. Use the `g++` command to compile the file 'program1.cpp'. If all goes well, an executable file named `a.out` will be created.

12. Run the compiled program (`./a.out`, recall `.` is the current directory and `/` is the standard directory separation symbol). Describe the results of running the program in the 'hw1' file.

13. Rerun the program using the *script* command to record the output. Don't forget to turn off script before doing anything else.

14. Use the `ls` command to display the files in the current directory. Record the results in the 'hw1' file.

15. Print and turn in the 'hw1', the file program1.cpp, and the output of your program created using the `script` command.