CS121 - Computer Science II
Lab Assignment #1 - UNIX Review
Fall 2011

The goal of this lab exercise is to (re-)familiarize you with basic Unix operations.

Unix is an operating system originally designed in the 60's and 70's to be a multi-user, multi-tasking operating system. It is one of the most, if not **the** most, influential operating system ever developed. Most of the concepts we now take for granted when we use computers were developed, or at least introduced, with Unix.

Unix was developed at Bell Labs, at the time the research arm of AT&T, and the source code (written mostly in C) is proprietary. In the early 90's an open source project was initiated to mimic the functionality of the original Unix system, called Linux (named after the project's initiator, Linus Torvalds). Linux is freely downloadable, and today its basic functionality is virtually identical to Unix. In the rest of this document we will refer to either system as Unix.

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 in 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, a user named `dargus` keeps all of his files in a subdirectory called `dargus` or in subdirectories within the `dargus` 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++ compiler, but that users shouldn't be allowed to change.)

After you log in, a program called "the shell" is executed; it is the shell that accepts commands from the keyboard and executes them.

There are a number of basic commands that are used to negotiate Unix's file structure (note that Unix is case-sensitive, and all commands are lower case):

- `ls` Short for **li**s**t**, this command lists the files and subdirectories in the current subdirectory. It does not list 'hidden' files; ones 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. In Unix, command options are specified with a '-' (dash).
- `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 **p**rint **w**orking **d**irectory, this command tells the user where they are in the file system. A typical response might be:

`/users/faculty/dargus/` showing that the user is currently in the subdirectory `dargus`, which is in the subdirectory `faculty`, which is in the subdirectory `users`, which is in the root directory `/`.

- `cat` *filename1 filename2* > *outfilename* Short for con**cat**enate, this command will write the contents of *filename1*, followed by the contents of *filename2*, to the file *outfilename*
- `cd` *directoryname* Short for **c**hange **d**irectory. The command changes which directory the user is currently in.
- `cp` *filename1 filename2* Short for **c**o**p**y, this command copies *filename1* into *filename2*. If *filename2* already exists, it is replaced with the new file. The file names can include directories.
- `cscheckin` This is a "local" (not standard Unix) command that you will use to submit (or "checkin") all lab assignments for this course. All lab assignments must be submitted using this command. No alternate form of turning in assignments is permitted. You will also be required to hand in a printed copy (or "hard copy") of your assignment to the lab instructor. Please note that no credit can be given for an assignment unless it is submitted using the `cscheckin` command.
- `g++` This is the C++ compiler. It actually performs the several steps necessary to translate standard C++ code into an executable program. These steps include the preprocessor, the compiler, assembler, and linker (which adds the necessary libraries and other runtime functions).
- `man` *commandname* Short for **man**ual, this command presents help information on the given command.
- `mv` *filename1 filename2* Short for **m**o**v**e (although perhaps this command should have been called "rename"), this command copies *filename1* into *filename2* and removes *filename1*. If *filename2* already exists, it is replaced with the new file. The file names can include directories.
- `mkdir` *directoryname* Short for **m**a**k**e **dir**ectory, this command creates a new subdirectory.
- `rm` *filename* Short for **r**e**m**ove, this command removes (deletes) a file.
- `rmdir` *directoryname* Short for **r**e**m**ove **dir**ectory, 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`*, do the commands whose output you wish to capture, then type `exit`, which stops the script. For example, 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.

All Unix directories include two special subdirectories called `.` and `..` The directory called `.` (dot) is the current directory. The directory called `..` (dot dot) refers to the directory above the current directory.

In Unix, a command is really just an executable program that has the same name as the command (For example, typing `ls` simply runs a program named `ls`.) When you enter a command, the shell starts searching for a program with the command name. The list of subdirectories the shell searches is contained in a "shell variable" named `PATH` (and conveniently called "the path variable!"). To print the value of a shell variable, you can use the `echo` command. For example:

```
echo $PATH
/usr/local/bin:/bin:/usr/bin
```

(The `$` is necessary to indicate that you want the value of the shell variable named `PATH`, rather than the word `PATH` itself.) In this example, the shell would first look in `/usr/local/bin`, then `/bin`, then in `/usr/bin` for the `ls` command.

It is usually the case that your home directory (which, BTW, is held in the shell variable `HOME`) or your current directory is *NOT* included in the search path. Thus, when you try to run a program you just created (in the current directory), it is not found (for example, typing `a.out` won't run your newly compiled program, so you must explicitly specify the current directory, `./a.out`). You can add the current directory to the path if you wish:

```
export PATH=.:$PATH
```

This causes the value of the path variable to change, so that the current directory will be searched before any of the other directories in the path. (Note that to *define* a shell variable, you do not include the `$` in front, but to *refer* to a shell variable, you must include the `$`.

Finally, you can include shell commands that you would like to have executed each time you log in in a file named `.profile` . For example, to always include the current directory in your search path, you could place the above command in `.profile` - the shell executes all the commands in `.profile` before it completes the login process and prints the prompt.

There are several editors available on most Unix systems - one of these should be used to create and modify files (such as C++ source files):

- `nano` - A very simple editor, similar to Notepad on Windows systems.
- `vi` - (**vi**sual editor) or `vim` (**vi-im**proved) - a considerably more powerful, but somewhat cryptic, editor, specifically designed to be used as a program editor.
- `emacs` - ("**e**diting **macros**"), another very powerful editor with over 100 separate commands.

`vi` is available on all Unix systems; `nano` and `emacs` are also widely available. It is important to learn at least one of these editors, although it doesn't really matter which one you actually use for this class.

For this exercise, you are to create a C++ program of your choosing, compile it, and submit the results using cscheckin. Use the script command to:

1. Print the program source code
2. Compile the program using g++
3. Execute the program to display the program results

The program that you create (and all programs you write for this class) should include comments that include your name, section number, date, and lab assignment number, along with a brief description of what the program does. For most lab assignments, you will submit the source code for the program(s) that you write. However, for this lab you should submit the script file that you created, showing the above steps.