Computer Science 381
Programming Unix in C
The College of Saint Rose
Fall 2013

Lab 1: C and Unix Introduction

Due: 10:25 AM, Wednesday, September 4, 2013

In this first lab, you will be introduced to the basics of C and Unix.

You may ask your instructor and classmates for help as you complete this lab, but the work you submit must ultimately be your own. If you are completely unfamiliar with Unix, don't hesitate to ask questions! On the other hand, if you have some experience, don't hesitate to help a classmate!

Preliminaries

Before you begin work on this lab, you should make sure you can log into the Macs in the lab (these should accept your regular username and password) and open a command terminal (under Applications/Utilities) and can log into our remote-access Linux system `mogul.strose.edu` (a separate account that needs to be set up if you have not used this system for a previous course).

Also, read over the description of the types of items you will encounter in our labs on the course home page.

Motivation

GUIs are nice, but they can be slow to navigate and too restrictive for some purposes. You can often work much more efficiently ny working in a Unix environment and interacting with the system by typing commands at the Unix *shell*, or *command line*. When you log in, you will be presented with a prompt. This is your direct interface to issue commands to the operating system. When you type a command here, the shell will execute the command on your behalf, print out any results, then reissue the prompt.

Of course, the command line is useless if you don't know what commands it understands. You will learn about several important commands in this lab and many more throughout the semester. One of the most important is `man` – the Unix manual. Every Unix command has a manual page, including `man`. To see the manual page about `man`, type the command:

```
man man
```

The Emacs Editor

Emacs (`emacs` from the Unix command line) is a powerful text editor, which is very good for programming in a language like C and for general plain-text editing. You will need to become familiar with it.

To try it out, you will use it to create your `lab1.txt` file that will contain your answers to this week's lab questions. For at least this lab, you are to create this file in your home directory on mogul.

Log into `mogul.strose.edu` using ssh from a Terminal window on the Mac. If your username on `mogul.strose.edu` is `jcool`, you would issue the command

```
ssh mogul.strose.edu -l jcool
```

at the terminal prompt. Log in with your `mogul.strose.edu` password. You should be presented with a prompt that looks something like:

```
[jcool@mogul ~]$
```

and mogul is now ready to accept your commands. More on those later.

Now open a second Terminal window and log into `mogul.strose.edu` on that one as well.

In one of the windows, launch `emacs` on the file `lab1.txt`:

```
emacs lab1.txt
```

Emacs should start up, and present you with a text-based menu across the top (which we will purposely ignore), a large area where you can edit the file, and two lines of status information across the bottom.

Type your name and "Lab 1 Questions" in the Emacs window that is editing the file `lab1.txt`:

In the other window, launch another `emacs` session where you can type some text and then identify the function of and experiment with these Emacs commands:

```
C-x C-s   C-x C-c   C-x C-f   C-x C-w   C-g   C-a   C-e
C-d       C-_       C-v       M-v       C-s   C-r   M-%
C-k       C-y       M-gg      C-x u
```

`C-` before a key means hold down `Ctrl` and hit that key. `M-` indicates the "Meta" key, which on most systems is `Esc`. To issue a Meta command, hit the `Esc` key, release it, then hit the key(s) for the command you wish to issue. Use the keystrokes rather than the menus. It will save you time in the long run! Note: for some of these commands, a very small buffer (that is, the contents of the file you are editing) will not allow you to see what they do. So create a file with several screens full of text before you go too far.

? Lab Question 1:
Complete your Emacs command descriptions in `lab1.txt` (3 points).

Directory Structure

It is always important, but especially so when working with the Unix command line, to know where the files in various directories (often called "folders" on Macintosh and Windows systems because of how they are visually represented in GUIs) you might be using are actually stored, and where and how those are accessible.

On the Macs in our labs, your home directories are (unfortunately) local to each station. Any files you save there are only on that computer and are not guaranteed to remain for a later session. If you want to save files on your college network space, you will need to "mount it". To do so, choose "Connect to Server" from the "Go" menu in the Finder (or Command-K) to connect and then make sure you save your files to the volume that you mount.

On `mogul.strose.edu`, we find a more standard Unix style environment. Each user has a *home directory* where only that user has permission to read and write files. Your home directory is the initial *current directory* or *working directory* when you first log in.

The working directory is where the program will look for files unless instructed to do otherwise. You'll hear Unix users asking a question like "What directory are you in?" and the answer to this is your working directory.

The command `pwd` will instruct the shell to print your working directory.

? Lab Question 2:
What is your home directory on `mogul.strose.edu`? (use `pwd`)

Note: lab questions are worth 1 point each unless otherwise specified.

? Lab Question 3:
What is your home directory when you open a Macintosh Terminal window?

? Lab Question 4:
What is the path to the directory where you mounted your college network volume?

You can also list the contents of your working directory with the command `ls`.

? Lab Question 5:
What output do you see when you issue the `ls` command on `mogul.strose.edu`?

Other important operations to navigate and modify the directory structure are changing your working directory (`cd`), creating a new directory (`mkdir`), and removing a directory (`rmdir`).

Create a directory in your account for your work for this course (`cs381` might be a good name), and a directory within that directory for this assignment (`lab1` might be a good name).

Lab Question 6:
Change your working directory to the one you just created and issue the `pwd` command. What does this show as your working directory?

In your shell window and in your home directory (note: you can always reset your working directory to be your home directory by issuing the command `cd` with no parameters), issue this command:

```
uname -a > linux.txt
```

This will execute the command `uname -a`, which prints a variety of information about the system you are on, and "redirects" the output, which would normally be printed in your terminal window, to the file `linux.txt`.

Output Capture:
`linux.txt` for 1 point(s)

Look at the contents of the file `linux.txt` with the command:

```
cat linux.txt
```

Do the same in a Mac terminal window, saving the output of `uname -a` in a file called `mac.txt`.

Output Capture:
`mac.txt` for 1 point(s)

Lab Question 7:
What do you think the information in `linux.txt` and `mac.txt` means?

Unix Commands

Identify the function of and experiment with these Unix commands (a few of which you have already used):

```
ls     cd      cp      mv      rm      mkdir   pwd
man    chmod   cat     more    grep    head    tail
ln     find    rmdir   wc      diff    scp     touch
```

Lab Question 8:
Give a one sentence description of each command. (4 points)

Using appropriate commands from the above list, move the `linux.txt` and `mac.txt` files you created in your home directory into the directory you created on mogul for your work for this assignment.

Show that this has worked by issuing the following command from inside of your course directory (but not inside the directory for this assignment):

```
ls -laR > ls.out
```

Then move the file `ls.out` into the directory for this assignment.

Output Capture:
`ls.out` for 3 point(s)

Using the Unix manual, your favorite search engine, or in discussion with your classmates, determine the answers to these questions:

Lab Question 9:
How do you change your working directory to be "one level up" from the current working directory? (Give the command.)

Lab Question 10:
Give two or three different ways to change your working directory to be your home directory. All likely involve the `cd` command, but will take different parameters.

The C Programming Language

C is a widely-used, general purpose language, well-suited to low-level systems programming and scientific computation.

We will initially study it assuming you have Java experience, focusing on the features that make C significantly different from Java. Fortunately, Java borrowed much of its syntax from C, so it is not difficult for a Java programmer to read most C programs.

C++ is a superset of C (that is, any valid C program is also a valid C++ program, just one that doesn't take advantage of the additional features of C++). C++ adds object-oriented feautures. In this course, we will look only at C, not C++.

A Very Simple C Program

We will begin by seeing how to compile and run a very simple C program (`hello.c`) in a Unix environment.

See Example:
`/home/cs381/examples/hello`

For you to run this, you will want to copy the example to your own directory. Create a directory called `hello` under your directory for this lab and copy the C file into that directory.

Change to that directory and compile and run it:

```
gcc hello.c
./a.out
```

Things to note from this simple example:

- We run a program named `gcc`, which is a free C compiler.
- `gcc`, in its simplest form, can be used to compile a C program in a single file:

  ```
  gcc hello.c
  ```

  In this case, we're asking `gcc` to compile a C program found in the file `hello.c`.

  Since we didn't specify what to call the executable program produced, gcc produces a file `a.out`. The name is `a.out` for historical reasons.

- When we want to run a program located in our current directory in a Unix shell, we type its name.
  - For example, when we wanted to run `gcc`, we typed its name, and the Unix shell found a program on the system in a file named `gcc`.
  - How does it know where to find it? The shell searches for programs in a sequence of directories known as the *search path*. Try: `env`.
  - So if we want to run `a.out`, we should be able to type its name. But our current directory, always referred to in a Unix shell by "`.`", is not in the search path. We need to specify the "`.`" as part of the command to run:

    ```
    ./a.out
    ```

- Of course, we probably don't want to compile up a bunch of programs all named `a.out`, so we usually ask `gcc` to put its output in a file named as one of the parameters to `gcc`:

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

  Here, the executable file produced is called `hello`.

- And in the program itself, let's make sure we understand everything:
  - At the top of the file, we have a big comment describing what the program does, who wrote it, and when. Your programs should have something similar in each C file.

  - We are going to use a C library function called `printf` to print a message to the screen. Before we can use this function, we need to tell the C compiler about it. For C library functions, the needed information is provided in *header files*, which usually end in `.h`. In this case, we need to include `stdio.h`. Why? See `man 3 printf`. (More on the Unix manual later.)
  - A C program starts its execution by calling the function `main`. Any command-line parameters are provided to `main` through the first two arguments to main, traditionally declared as `argc`, the number of command-line parameters (including the name of the program itself), and `argv`, an array of pointers to character strings, each of which represents one of the command-line parameters. In this case, we don't use them, but there they are.
  - Our call to `printf` results in the string passed as a parameter to be printed to the screen. The `\n` results in a new line.
  - Our `main` function returns an `int` value. A value of 0 returned from `main` generally indicates a successful execution, while a non-zero return indicates an error condition. So we return a 0.

- Notes for Java programmers:

  - Good news: much of the syntax of Java was borrowed from C, so a lot of things will look familiar.
  - There are no classes and methods, just *functions*, which can be called at any time. Any information a function needs to do its job must be provided by its parameters or exist in *global variables* – variable declared outside of every function and which are accessible from all functions.

Practice Program

Write your own C program named `helloloop.c`, much like the "Hello, World" example, but which prints some other message and prints it 10 times inside of a `for` loop. The C `for` loop is much like Java's `for` loop, except that the loop index variable needs to be declared before the loop. That is, a Java loop that looks like this:

```
for (int i=0; i<10; i++) {
  ...
}
```

would need to have the declaration of `i` outside of the loop:

```
int i;

// any other code that happens before the loop
```

```
for (i=0; i<10; i++) {
  ...
}
```

Make sure your program compiles and runs on either the Mac or mogul using `gcc`.

This program is worth 10 points.

Note: there are no formal "Programming Assignments" this week.

Submission

Please submit all required files as email attachments to *terescoj@strose.edu* by 10:25 AM, Wednesday, September 4, 2013. Be sure to check that you have used the correct file names and that your submission matches all of the submission guidelines listed on the course home page.

Grading

Grading Breakdown	
Lab questions and output captures	20 points
Practice program	10 points