Lab 1 - Working in the Terminal and More

Getting Used to the Linux computers in the Lab 1000G.

In CSci 127, you should have learned the basics of getting around on the Department's Lab 1000G Linux systems. This tutorial assumes that you know at the very least how to login and logout when in Lab 1000G, and how to use the Linux desktop environment, including its menus and file browsers. (Our lab uses Fedora Linux, which defaults to the Gnome desktop, but you might have learned another such as KDE.) When you are not in the lab but connect remotely to the Linux hosts, you will have no choice but to work within a ***terminal window***. Even when you are in the lab, you will have to do much of the work within a terminal window. Therefore, the main purpose of this tutorial is to teach you the basics of working within terminal windows.

Why, you may be wondering, do you have to work in a terminal window? There are two reasons.

- UNIX systems, which includes Linux systems, support the ability for someone who is in a remote location to login to their UNIX account. For example, you can connect from your home computer to a computer in the lab and work on your files on that lab computer from home. However, for reasons of efficiency, the only way that you can do this is to work within a terminal window. If you are at home and try to run an application on a CS lab machine that uses the full graphical user interface (GUI), everything will "crawl"[1]. Therefore, to be able to work from home or work or any other remote location, you need to know how to use the command-line interface in a terminal.
- When you are in the lab and you want to work on C++ programs, you will need to use the command-line tools for compiling and running your program. All of your programs will be run by typing their names on the command line because you will not learn in this course or any of its sequels how to design programs that create their own graphical user interfaces. So at the very least you will need to work in a terminal to run your programs. Furthermore, many of the tools for working on software are command line tools. The most basic tool is, of course, the compiler/linker. We will be using the GNU compiler collection for this purpose.

How can you open a terminal window when you are in the lab?

- After you have logged in and are looking at your desktop, you can open a terminal in one of two ways. There may be a Terminal icon on the toolbar. In this case you can click it. If not, then open the *Applications* menu, select *System Tools*, and select the *Terminal* tool that is displayed in its sub-menu. If you do not have a *Terminal* icon in your menu-bar, you can drag it to your toolbar for easy access in the future.

Once you have an open *Terminal* window, you can start entering Linux commands in that window.

You should be aware that, although you are working on a computer that is physically in the lab, all of the files that you will work with are actually stored on the CS Department file server, whose name is *biocs*[2]. Each CS computer, including *eniac*, remotely mounts the *biocs* file system containing your home directory. This makes it possible for you to access your files from any computer in the CS network. Although you may not understand how this works, just trust it for now. Later we will explore commands related to file systems and mounting, and some of it will be demystified.

[1] The reason for this is simply that the amount of data that must be passed back and forth with each mouse movement, mouse click, and so on is generally in the range of tens of thousands of bytes or more, whereas in a terminal it is dozens of characters that get transferred back and forth.

[2] Its full name is *biocs.geo.hunter.cuny.edu*.

What if you cannot login to the computer in the lab?

- To login to any computer in the lab, you will need to have a Computer Science Department account. You should have been given that account at the start of the semester. Use your account's username and password in the dialog boxes on the Linux startup screen. If you have problems logging in, talk to your instructor or email the lab administrator, Tom Walter at twalter@hunter.cuny.edu with the description of your problem.

Some Rules When Working in the Lab

- It is very important that you ***do not turn off the computer*** on which you are working! Other students or instructors may be logged in to these machines remotely. If you turn off your computer they lose their work! If something goes wrong and the machine crashes or becomes unresponsive, do not turn it off, put a sign on it, saying that it is out of order, login to another machine and email Tom Walter describing what happened (twalter@hunter.cuny.edu).
- Never unplug any machines! If you want to use your laptop in the lab, that is fine, but you will have to run it on its battery.
- Never unplug any cables in general, and in particular, do not unplug a machine's Ethernet cable! If you want to connect to the network from your laptop while in the lab, use its wireless card.

Accessing Your Account From Other Locations

From anywhere outside of the CS network, the only way to access any CS machine is by logging in to the CS *gateway* machine, which is named *eniac*. Its full name is *eniac.geo.hunter.cuny.edu*. To login to *eniac* remotely you will need an `ssh` client (for working on the lab machines and/or *eniac* remotely) and / or `sftp` client (for transferring files to and from your CS lab account). There are many free clients available on the Internet; here are a few:

- SSH Client 3.2.9, Windows, SSH and SFTP client (link via my website)
- PUTTY, Windows, SSH client
- FileZilla, cross-platform, SFTP client
- Apple systems should come with SSH client built in, you may need to enable it
- Linux systems come with SSH client built in, and many SFTP clients.

iIf you never used `ssh`, read my tutorial *"SSH, SFTP, and Remote Logins"*.

Assuming that you have a Computer Science Department account and an ssh client, you can login to *eniac*. Once you login to *eniac*, you *must* login to one of the lab machines (e.g., `ssh cslab10`). The names of the lab machines are *cslab1, cslab2, ..., cslab28*. It does not matter which one you use; all of your files are accessible identically from every host, and they are all the same type of machine (same computing power, memory, etc.) Just pick a number you like.

Getting around in Unix/Linux

Passwords

If you just got a new account on the system, the first thing you must do is change your password. *Always. No exceptions.* If someone intercepted the email that contained your password, they can access your account and even change your password so that you cannot get into it!

To change the password, type the command `passwd` at the prompt in the terminal window. Note that it is 'password' without the 'or'. You will be asked to enter the current password and then asked for the new password, twice. If you make any mistakes, the password will not be changed. The prompt will always be denoted by the dollar sign `$` in these notes.

```
$ passwd
Changing password for user sweiss.
(current) UNIX password: you type here
New UNIX password: you type here
Retype new UNIX password: you type here
passwd: password has been changed.
$
```

If you give a poor password, it will not accept it. You must enter a strong password, which is one that contains:

- at least one uppercase letter
- at least one lowercase letter
- at least one digit
- preferably some punctuation mark such as a hyphen, period, underscore, and so on.

What does a command look like in UNIX?

First of all, when you use a terminal, you are really providing input to a special program generally called a ***command line interpreter***, which reads your entered command, interprets what it means, and then executes it. In UNIX these command line interpreters are called ***shells***, and on most Linux systems, the specific shell that people use is called `bash`. `bash` is short for "Bourne-again shell."

In all shells, a simple command is of the form

```
$ commandname command-options argument1 argument2 ... argumentn
```

The command is executed only after you type a newline character. The command has three parts: the command name, command options, which are, as the name implies, optional things you can give to the command, and arguments. Many commands require no arguments. Examples of simple commands:

`date`	display the current date and time
`who`	list who is currently using the computer
`man <command>`	display the online manual page for the given command

This last command is very important – it is how you can learn all about a command whose name you know. Remember it. In fact, try typing the commands

```
$ man who
```

and

```
$ man date
```

Then type

```
$ man man
```

to learn about the `man` command itself!

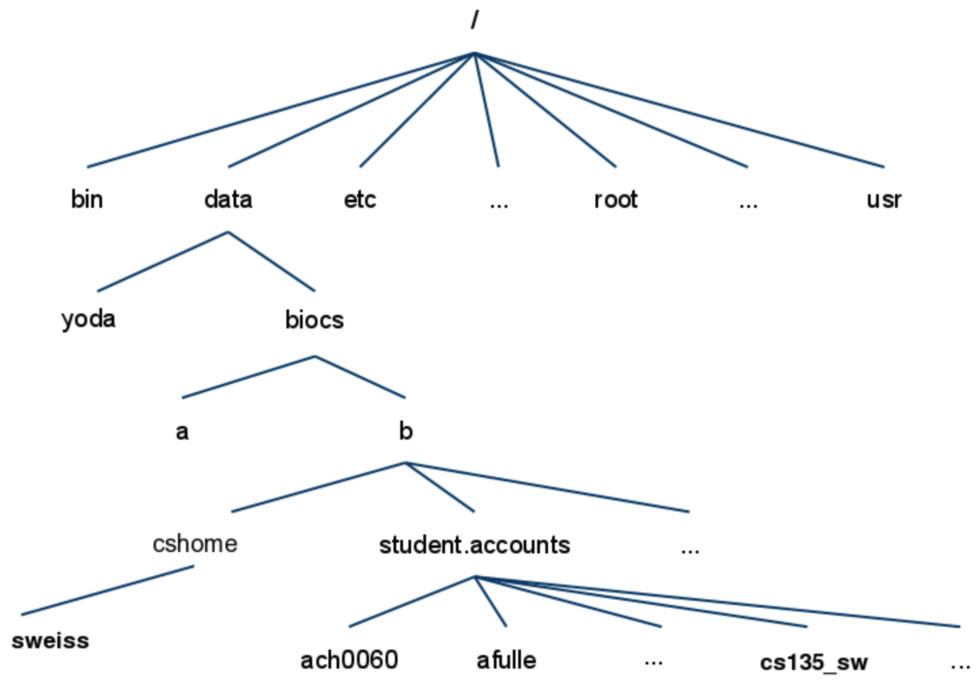

Fig. 1: Partial directory structure on *eniac*.

Navigating the File System

Whenever you are logged into a Unix/Linux system, you have a unique, special directory called your ***current*** or ***present working directory*** (***PWD*** for short). The present working directory is where you "are" in the file system at the moment, i.e., the directory that you feel like you are currently "in". This is more intuitive when you are working with the command line, but it carries over to the GUI as well.

Many commands operate on the *PWD* if you do not give them an explicit argument. We say that the *PWD* is their *default argument*. (Defaults are fall-backs – what happens when you don't do something.) For example, when you enter the "`ls`" command (list files) without an argument, it displays the contents of your *PWD*. The dot "." is the name of the *PWD*:

```
$ ls .
```

and

```
$ ls
```

both display the files in the *PWD*; the first command because the name of the directory ("`.`") is provided, the second because it is the default directory for `ls`.

When you first login, your present working directory is set to your ***home directory***. Your home directory is a directory created for you by the system administrator. It is the top level of the part of the file system that belongs to you. You can create files and directories within your home directory, and usually almost nowhere else. Usually your home directory's name is the same as your username.

In Unix/Linux, files have ***filenames*** and ***pathnames***. The filename is the name of the file, i.e., the name listed when you type the `ls` command. You might name one of your files, `hwk1.cpp` for example, and its filename would be `hwk1.cpp`. Someone else might also create a file with that exact same name. How can they be told apart? The answer is by their pathnames. A pathname is a unique name for the file. It specifies the location of the file in the file system by its position in the hierarchy. The POSIX.1-2008 standard specifies a system-dependent limit on the number of characters in a pathname (including the terminating null byte).[3].

There are two types of pathnames: ***absolute*** and ***relative***. An absolute pathname is a pathname that starts at the root. It begins with a "/" and is followed by zero or more filenames separated by "/" characters. All filenames except the last must be directory names. For example, `/data/biocs/b/student.accounts/cs135_sw` is the absolute pathname to the directory that we will use in class, as is `/data/biocs/b///student.accounts////cs135_sw`. The extra slashes are ignored. Observe that UNIX (actually POSIX) uses slashes, not backslashes (as in Windows), in pathnames: `/usr/local`, not `\usr\local`.

The image in Figure 1 shows a partial directory structure on biocs. Your home directory is located in `student.accounts` which is in a directory called `b` which in turn is in a directory called `biocs`, which in turn is in `data` which is at the root level of the file system. The absolute pathname to the home directory of `jbgoode` would be `/data/biocs/b/student.accounts/jbgoode`.

A relative pathname is a pathname to a file relative to the PWD. It never starts with a slash! If the current working directory of `jbgoode` is his home directory, and this contains a directory named `cs136`, which contains another directory called `lab01` which contains the file called `lab01_solution.pdf`, then the relative pathname of that file from his PWD is

```
cs136/lab01/lab01_solution.pdf
```

and its absolute pathname is

```
/data/biocs/b/student.accounts/jbgoode/cs136/lab01/lab01_solution.pdf
```

Dot-dot or **..** is a shorthand name for the parent directory. The parent directory is the one in which the directory is contained. It is the one that is one level up in the file hierarchy. If my PWD is currently `/data/biocs/b/student.accounts/cs135_sw/`, then

`..` is the directory `/data/biocs/b/student.accounts`

and

`../../` is the directory `/data/biocs/b/`.

Tilde, or ~, is a shorthand for your home directory.

```
$ ls ~
```

displays the contents of your home directory. If your username is `jbgoode`, then `~jbgoode` is also a shorthand for your home directory.

Simple Versions of Unix/Linux commands

Listed below are the most basic UNIX commands related to navigating around the file system. Only the simple forms of each are listed. Many have more complex forms. You can learn more about them using the `man` command.

`ls`	list content of the current working directory
`ls dir_name`	list content of the directory named `dir_name`
`cd dir_name`	`cd` stands for change directory, changes the current working directory to `dir_name`
`cd ..`	move one directory up in the directory tree
`cd`	change the current working directory to your home directory

[3] The variable `PATH_MAX` contains the maximum length of a string representing a pathname. On many Linux systems it is 4096 bytes.

`pwd`	print the name of the present working directory
`cp file1 file2`	copy `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names
`mv file1 file2`	move `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names
`rm file`	remove a file (there is no undoing it, so be very careful!)
`mkdir path`	make a new directory at the specified `path`
`rmdir path`	remove the *empty* directory specified by the `path` (there is no undoing it, so be very careful!)

Class Directory

I will put code, tutorials and labs that you need to access in the directory

```
/data/biocs/b/student.accounts/cs135_sw
```

Do not be bothered about why it is `cs135_sw` and not `cs136_sw`. There are so many students who are in both classes that it is easier just to use a single directory for both. The 135 and 136 classes will share materials.

If you don't want to worry about remembering the path to this directory you can make a link to it in your home directory. To do that, first navigate to your home directory and then create the link:

```
$ cd
$ ln -s /data/yoda/b/student.accounts/cs135_sw cs135link
```

Then, you can always navigate to it by typing

```
$ cd ~/cs135link
```

This directory has several subdirectories, including one named labs and one named tutorials. Files placed in these directories can be copied into your own directories so that you have your own private copies of them. It is a good idea to copy assignments and tutorials when they become available. There will be code examples as well, and these should also be copied. You have only read permission on the files inside this directory, so if you ever need to edit something, you should copy the files first to your own directory (you can do so from the command line, or simply drag files from one directory to another in the gui - of course the second option is not available when you are logged in remotely).

Compiling Your C++ Programs

One of the things that you going to do over and over again in this class is compiling and running your code. We will use the GNU C++ compiler, named g++, for this purpose. I have written a more detailed tutorial on the GNU C++ compiler, which is on my website; see *A Tutorial and Brief Summary of GCC*.

The exercise portion of Lab01 goes through several examples of how to use g++ to compile and build your programs.

Documentation and Comments

You should make a habit of writing documentation and comments even for the simplest programs. I will talk about documentation in separate files later on in the semester, for now, let's start with comments. Comments should serve as a way of documenting your code. You do not need to explain the programming language in the comments, since we assume that whoever reads your code knows C++. You do have to explain the purpose of variables (unless you give them descriptive names) and the methods of doing things. You might look at your own code several weeks or months (or even years) after you have written it and not know why you chose to do something; the comments will help you to remember if you write them well.

Every distinct source code must contain a preamble with a title, author, brief purpose and description, date of creation, and a revision history. The description should be a few sentences long at the minimum. A revision history is a list of brief sentences describing revisions to the file, with the date and author (you in most of the cases) of the revision. This is an example of a suitable preamble:

```
/******************************************************************************
  Title          : simple_C++_io.cpp
  Author         : Stewart Weiss
  Created on     : January 26, 2006
  Description    : Copies standard input to output a char at a time
  Purpose        : To demonstrate typical text I/O in C++
  Usage          : simple_C++_io
  Build with     : g++ -o simple_C++_io simple_C++_io.cpp
  Modifications  :

******************************************************************************/
```

For the simple programs at the beginning of the course you may not really need all the parts of the preamble - just leave them blank if that is the case. The required parts, those you should always specify, are: Title, Author, Description, Usage, and Build with.

Sometimes there is information you may want to include at the very top of the program, to help the reader understand more about it. I usually include notes in the preamble for this reason. Following is an example of a preamble that you will find in one of the demo programs in our class directory. In fact, the preamble contains the name of the file itself (nestedfor_01.cc), so you can look for it.

```
/******************************************************************************
  Title          : nestedfor_01.cc
  Author         : Stewart Weiss
  Created on     : Feb. 28, 2007
  Description    :
  Purpose        : To introduce the concept of a nest for-loop, this
                   program shows how the inner and outer loop index variables
                   change.
  Usage          : nestedfor_01
  Build with     : g++ -o nestedfor_01 nestedfor_01.cc
  Modifications  :
  Notes          : The program introduces a convenient function found in
                   UNIX operating systems called usleep(). usleep() is given
                   integer argument that is the number of microseconds to
                   delay the program. usleep(1000000) delays 1 second for
                   example, and usleep(500000) delays 1/2 second. It adds
                   effect to the program.

******************************************************************************/
```