CS 211 Lab 1

Navigating the Unix Shell

Winter 2021

Today we are going over the basics of how to log into a remote computer, use shell commands to create and edit files, and compile and run C code.

The shell works as a textual conversation. It presents a prompt, like `[wsc147@robin]~/cs211%`. (The default EECS prompt shows the username, the hostname, and the current working directory.) You type a command and press enter. The shell executes the command and then prints another prompt, waiting for further commands. For example, to list the files in the current directory, you will run the *ls* command by typing it at the prompt:

There are different shells with slightly different syntax. In this class, we will use a shell called *fish* (for "Friendly Interactive SHell"). Other shells include *tcsh* (the default on the EECS servers), *bash* (the default on most Linux systems), and *zsh* (the default on recent versions of Mac OS).

```
% ls
```

Don't type the `%`. That represents the prompt that the shell prints for you to tell you it's ready.

Before you can do that, though, we have to get you logged in.

Getting around the firewall

The workstations we want to connect to are not directly accessible from off-campus, so we will need to do a little work to reach them. There are essentially two alternatives:

- Use Northwestern's virtual private network (VPN). When you enable this, all your network traffic goes through an encrypted tunnel to Northwestern's campus, which means you are effectively there. This solution is easy, but it can slow down your network, so you'll want to turn it off when you aren't using it.

  To setup the NU VPN, follow these instructions.

- Configure SSH to use a "proxy" through a different host than the one we ultimately want. SSH can do this automatically for you if you set it up just right, but it's a bit harder than using the VPN. Instructions for setting up a proxy appear in the per-system sections below.

Logging in

For the majority of you who are unfamiliar with the Unix shell, it probably seems like a scary foreign concept reserved for computer hackers on TV shows and movies. However, in reality, with a little bit of time and a few basic commands, you will realize that the Unix shell is not something to be scared of, and in fact a very useful tool to embrace as you continue your computer science education. Don't get frustrated if it seems hard at first! Every great computer scientist was at one point also unfamiliar with the shell, just like you, but with a little bit of exposure, it will start to make sense.

SSH (secure shell) is a protocol that allows you to login remotely onto an external system. We will be using it in order to create a connection onto a Northwestern remote server, where we will be learning our first Unix skills. For the first step of establishing the connection, it will be different for Windows and Mac/Linux, but for the rest it should not matter which OS you are on, since you'll be using the remote Unix machine.

You will need your netid and EECS password to log into the computers. If you do not remember it, you can create a new password at https://selfserv.eecs.northwestern.edu/temp_password/. You will need to use your netid (lowercase) and Northwestern password to log into that website. The link will only work once, so pick a memorable password. If you need to reset your password again, you'll have to contact EECS IT: help@eecs.northwestern.edu

Windows

Download the SSH client PuTTY; we recommend the MSI installer, since it's usually easier. The link on the right will take you directly to the Windows installer. After you install PuTTY, open it up. You'll need to enter a hostname to login to. The link on the right will take you to a list of student lab hostnames (such as *batgirl.eecs.northwestern.edu* or *hush.eecs.northwestern.edu*). Ensure SSH is selected, then press Open. You should get some sort of message asking whether or not you trust the host. Press yes. From here, login as your EECS username (probably the same as NetID), and your EECS password (not necessarily your NetID password). You should now be logged into one of the Northwestern EECS boxes!

https://the.earth.li/~sgtatham/putty/latest/w64/putty-64bit-0.74-installer.msi

You can find a list here: it.eecs.northwestern.edu/info/2015/11/03/info-labs.html#workstations

Note that you can and should configure PuTTY so that you don't have to do all of this every time by saving a session.

If you would like to try to configure PuTTY to proxy so that you don't have to use the VPN, try these instructions.

Mac/Linux

For those of you on Mac or Linux, everything you need is already installed. Open up your terminal and at the prompt type a single command of the form

Mac users: search for "terminal" in Spotlight

```
% ssh ⟨eecs-id⟩@⟨eecs-host⟩.eecs.northwestern.edu
```

Don’t type the %. That stands for your shell prompt.

where ⟨*eecs-id*⟩ is your EECS username (probably your NetID) and ⟨*eecs-host*⟩ is replaced by one of the EECS hostnames from the list of student lab hostnames (such as *alfred.eecs.northwestern.edu* or *robin.eecs.northwestern.edu*).

You can find a list here: it.eecs.northwestern.edu/info/2015/11/03/info-labs.html#workstations

You should get a message saying that the authenticity of the host can’t be established, and you should be asked if you want to continue connecting. Type “yes” as prompted and press Enter. Now type in your EECS account password (not necessarily your NetID password), press Enter again, and you should be logged in remotely!

You can configure `ssh` to proxy automatically, and to use a cryptographic key for authentication so that you don’t have to type your password. If you want to try this, run the following command in your terminal and follow the instructions:

```
% bash -c "$(curl -fsLS https://bit.ly/3nBxDH3)"
```

If you’d like to see the script, run just the *curl* command that’s inside the parentheses (not including the parentheses), or click the link.

Notes: 1) Run this *locally* on your own computer, not while logged into a remote workstation. 2) Type or paste that command *exactly* as written.

Basic shell navigation

There are a few basic commands we will be using frequently throughout this exercise in our shell: *cd*, *ls*, and *pwd*, and *man*.

cd stands for “change directory,” and is used to change the current directory we are looking at in our shell (our working directory). You can think of a directory as a folder from your regular interactions with your computer. For example the command `% cd Documents` will look for a directory inside our current directory called Documents, and if it exists, our working directory will become that Documents directory. If you ever want to go back to your home directory, the command `% cd` with no argument will switch your working directory back to your home directory. The command `% cd ..` will switch your working directory up one level from where you currently are.

As usual, don’t type the %.

pwd stands for “print working directory,” and is used to print out the current working directory of your shell. For example, if you have been navigating around for a while and you are lost you can type in the command `% pwd` and you will see your directory printed out into the shell.

ls is short for the word “list,” and is used to list the contents and subdirectories within your current working directory. You can type the command `% ls` into your shell, and you will see all files and directories within your current working directory.

Play around with these three commands for a few minutes in your shell, and see what directories and files already exist on your EECS box!

man is short for "manual," and is used to access the system manuals. For example, you can read the manual pages for *pwd* and *ls* by running the commands `% man pwd` and `% man ls`. To learn about man, you can of course run `% man man`.

Hit q to quit.

Once you are done playing around, type `% cd` in to navigate back to your home directory. We will be making a new directory for this lab using the *mkdir* command.

Creating new files

mkdir stands for make directory, and is used to create a new directory within our current working directory. For example, `% mkdir fun-project` will create a new directory inside our current one called fun-project that we can *cd* into if we so desire. We can create hierarchies of directories to keep our files well organized.

Create a new directory inside your home directory called lab01-dir. Change your current working directory to lab01-dir, and we will now practice editing and compiling some C source files!

The `% emacs` command in the shell will open up the Emacs text editor. (On Mac/Linux, you will probably want to use `% emacs` to avoid starting the X Window System..) Pass in a file that you want to edit (even if it hasn't been created yet), and you can start editing that file! For example type `% emacs my_code.c` and you can start editing a file called my_code.c within your current working directory.

Text editor preferences can be a fairly contentious issue among software engineers, and if you already have experience with one of Vim or Emacs, feel free to use whichever you already have experience with instead of Emacs. However, for the purpose of this class, we will be teaching using Emacs. Emacs can also seem scary at first, but after you learn a few simple commands, it will quickly start making sense.

Inside your lab01-dir directory create and open a file using Emacs called animals.txt. Note that the .c file extension is what we will be using to indicate C files. You will see a text editor pop up that does not look dissimilar to a *Notepad.exe* or *TextEdit.app* editor from your Windows or Mac. However, you will notice that clicking a location using your cursor will not move your cursor to where you click :(

Inside this text editor, type in a list of your 3 favorite animals. Once you have typed in your list, you are going to want to save your file so you can use it later. On Emacs, saving is slightly different than other programs. Instead of using Command- or Ctrl-s, you are going to use Ctrl-x followed by Ctrl-s. (In Emacs, this is spelled "C-x C-s.") This will save your file to your current working directory. Now, we want to close our Emacs window and get back to our Unix shell. In order to close our Emacs window, we will type C-x C-c (that is, Ctrl-x followed by Ctrl-c).

If you are curious about more Emacs commands, there is a nice basic list here: http://www.cs.cornell.edu/courses/cs312/2003sp/handouts/emacs.htm. You can also run an Emacs tutorial inside Emacs. Press C-h t – that is, Ctrl-h followed by t (no Ctrl).

We can ensure that our file was properly created by using the *cat* command in the shell. *cat* is short for "catenate," and prints out contents of a given file. `% cat` ⟨*filename*⟩ will print the contents of the file to the shell. If you run `% cat animals.txt` you should see the file you just created on your shell.

Setting up the CS 211 development environment

As you saw in class, we will be using the *fish* shell in CS 211, because it's friendlier and generally easier than other shells. However, *fish* isn't available by default on the EECS workstations, so getting access to it requires a bit of setup.

Additionally, editing programs will be much nicer if you have at least a basic Emacs configuration.

Both of these things are configured in *hidden* files in your home directory. The directories where *tcsh* (the default shell on these workstations) looks for programs (called the "path") is configured in .tcshrc, and Emacs looks for its configuration in a file called .emacs.

On Unix-line operating systems, include Linux and Mac OS, files whose name begin with a period (`.`) are omitted from directory listings by default. Passing the `-a` flag to *ls* causes it to show these hidden files as well.

We've provided a script to create these files for you. If you already have these files, it will first back them up and tell you the names of the backup files. Here is the command to run:

```
% ~cs211/setup211
```

Remember not to type the prompt.

The changes will take effect the next time *tcsh* restarts. You can log out and back in, or run the command `% exec tcsh` to reload *tcsh*.

Once you've done so, you should be able to start *fish*:

```
% fish
```

You only need to do the above setup once, but you'll want to run *fish* each time you log in.

Getting the files

We provide archives of starter code for both this lab and the homework assignments (as well as code from the lectures) on the EECS login boxes in a place where you can access them to make a copy. You'll want to use these rather than starting from scratch because they include the build system we'll be using (more on this below) as well as configuration options that you need.

So, you will find lab code in found in `~cs211/lab`, and the code for this lab in particular may be found in `~cs211/lab/lab01.tgz`. But what is a `.tgz` file and how can you use it?

On Unix, in front of a user's name is the path to their home directory, so this means the lab/ subdirectory of belonging to a user named `cs211`.

The `.tgz` file extension is used for "gzipped tarball," which is like the Linux equivalent of a .ZIP file. The name "tar" stands for "tape archive," because it was invented when computer systems still stored backups on magentic tape.

In order to extract the contents of an archive into your current working directory, the command is:

```
% tar -xkvf ⟨archive⟩
```

This week, ⟨*archive*⟩ should be `~cs211/lab/lab01.tgz`.

The letters after the hyphen are *flags*, which specify various options to the program:

k tells `tar` to **k**eep any existing files rather than overwrite them, which is the default. You might want to make a habit of this so that you never accidentally replace your completed homework with starter code, but you can leave the `k` out if you feel like living dangerously.

x tells `tar` you want to e**x**tract files from the archive, as opposed to **c**reating an archive or just **t**elling you what's in it.

v tells `tar` to be verbose, meaning it will print out the names of the files that it extracts; so if you don't like seeing that part, you can leave it out.

f tells `tar` to extract from the file whose name follows (as opposed to from some other place, like its *standard input*).

Now, you have our new directory with the files you need, so change your directory to lab01/ using the *cd* command. Now list its contents using *ls*, and notice that there is a Makefile file, and a src/ directory. The Makefile file is a *make* configuration file which you won't have to worry about too much right now; the src/ directory contains the file *hello.c*, which we have provided you.

Using our build system

As briefly mentioned in class, *make* is our build system we will be using for the first few weeks of the course at least. We will usually be giving some sort of starting structure for the projects you will work on, and right now is no exception.

If you are the directory with the Makefile then can build your program using the command *make*.

This means you should see it when you run *ls*.

The basic purpose of *make* is to build your project into an executable file. In your build directory, each time you update your code, you can run

```
% make ⟨target-name⟩
```

to create your executable called ⟨*target-name*⟩. In this case, run `% make hello` to build a program called hello, which will be put in the current directory. You can run the program like this:

```
% ./hello
```

This should spit out a nice greeting.

Updating our code

So, we gave you a basic function and you were able to run it, but how do you change the code?

Open up the src/hello.c file using Emacs, and edit it so it now says "Aloha 211 student!" instead of "Hello 211 student!." Make sure to save and exit Emacs.

Remember C-x C-s to save and C-x C-c to exit.

Then try running `% ./hello` again. Did anything change?

The reason why you still see "Hello 211 student!" on your screen is because while you changed your C++ code, your computer doesn't understand the C++ code, but only the machine code you create by using *make*. So now, run `% make hello` once again, and try `% ./hello`. Notice how you now have the correct output! Each time we want to change our code, we are going to need to remember to rebuild our executable. Don't worry if you have error messages your first few times trying to write new code, this is completely normal. Even the best developers in the world usually need a few tries before they can properly build their files, so just take a deep breath, and try and figure out what went wrong.

Conclusion

Knowing how to use the shell is an extremely important tool in computer science. Don't worry if it is still hard for you to use, like much of life, it is one of those things you'll just need to practice with until it seems much more familiar! On your own time, it would be a good idea to continue learning more about the shell and playing around with some more commands. Of course, come to office hours or post on Campuswire with any questions or if you want any more challenges!

A good resource for some basic commands is here: http://www.computerworld.com/article/2598082/linux/linux-linux-command-line-cheat-sheet.html.

Useful links

EECS login server hostnames it.eecs.northwestern.edu/info/2015/11/03/info-labs.html#workstations

Simple command line cheat sheet http://www.computerworld.com/article/2598082/linux/linux-linux-command-line-cheat-sheet.html

PuTTY MSI installer https://the.earth.li/~sgtatham/putty/latest/w64/putty-64bit-0.74-installer.msi

PuTTY "proxy" instructions https://nu-cs211.github.io/cs211-files/putty_setup_guide.pdf

Nice Emacs guide http://www.cs.cornell.edu/courses/cs312/2003sp/handouts/emacs.htm

Nice Vim guide http://www.openvim.com/

VPN instructions https://kb.northwestern.edu/page.php?id=94726