CAAM 519: Computational Science I

Module 1: Linux

Linux is an *operating system* (OS): that is, the software on your computer that allows you to interact with the hardware by running other software programs. Other notable examples of operating systems: Windows, Mac OSX, iOS, Android, and so on.

Actually, it's not precisely accurate to say that Linux is "an" operating system. In fact, it is a family of operating systems, each based around the Linux kernel, which has been under development for almost 30 years. Linux distributions are typically free and/or open source, and have become the OS of choice for scientific computing. Therefore, we start this course by introducing you to Linux, and showing you how to perform common tasks.

Interacting with Linux through the terminal

Most modern Linux distributions intended for desktop computers come with a nice graphical user interface that should be familiar to users of Windows or OSX. However, the most powerful way to interact with a Linux OS is through a terminal. A terminal is an application that allows you to interact with the computer through a command line interface; essentially, by issuing it text commands.

A quick note on shells

At the terminal, you will typically talk to through the OS through a shell program. This extends the CLI with various niceties, like colors, configuration, and scripting capabilities. The bash shell is most common, and is typically the default on Linux and OSX. However, other shells exist and are actively used (e.g. csh or zsh). Typically, they don't vary all that much, and zsh is backwards compatible with bash. In any case, don't worry too much about the options at this point; it's easy to switch between them if the need arises.

The basics

How do I figure out how to do X?

In general, Google is your friend. Odds are, someone has had the same question as you in the past, and has asked it on a forum website such as StackOverflow.

Alternatively, you can inspect the manual pages for various programs on your computer via `man`:

```
> man ls
LS(1)                     BSD General Commands Manual
LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls
displays its name as well as
     any requested, associated information.  For each operand that names a
file of type directory, ls dis-
     plays the names of files contained within that directory, as well as
any requested, associated infor-
     mation.

     If no operands are given, the contents of the current directory are
displayed.  If more than one op-
     erand is given, non-directory operands are displayed first; directory
and non-directory operands are
     sorted separately and in lexicographical order.

     The following options are available:

     -@      Display extended attribute keys and sizes in long (-l) output.
...
```

To exit the man pages, press `q`.

Moving around the file system

The file system on a Linux machine is just a big tree of nested directories. A path is read from left to right, with / used a delimiter. So, `/home/joehuchette/Documents` is a folder nested three levels deep.

Opening a terminal, we can see our current directory with

```
> pwd
/Users/jah9
```

This is my “user home” directory, and it is commonly denoted by the special ~ character. Alternatively, the top level directory is denoted by a `/`.

We can move up one directory with

```
> cd ..
```

The double dots indicate one level up in the directory tree than our current location. You can chain them together to go through multiple levels with something like `../../...`.

```
> ls
jah9 other_user
```

We can inspect our current directory with

```
> pwd
/Users
```

which is what you should expect, having moved up one directory. Instead of using the `..` relative directory notation, we can specify an absolute path if we know it

```
> cd /Users/jah9/Documents/caam519
```

Tab completion is quite useful when typing out long paths like this.

We can add a new directory in this location with

```
> mkdir new_directory
```

If we get sick of it, we can remove it with

```
> rmdir new_directory
```

What is a file, anyway?

A file is nothing more than a container of data, represented in binary form (i.e. 0s and 1s). Files typically have an extension in their name (e.g. .pdf, .txt, .csv), which can help programs infer how it should interpret their data in binary form.

You can create an empty file with

```
> touch placeholder.txt
```

You can move it around with

```
> mv placeholder.txt ..
```

(this moves the file up one directory from its current location), and copy it with

```
> cp ../placeholder.txt placeholder.txt
```

This copies the file in the parent directory to the current directory.

To remove the file in the parent directory, do

```
> rm ../placeholder.txt
```

So far the placeholder file isn't too interesting, since there's no data in it:

```
> cat placeholder.txt
```

To add something, we will use redirection. We saw before that `pwd` prints the current absolute path location. We can redirect this output to the file with

```
> pwd > placeholder.txt
> cat placeholder.txt
/Users/jah9/Documents/caam519
```

We can also use the `echo` command to send arbitrary text to the file:

```
> echo "Lorem ipsum" > placeholder.txt
> cat placeholder.txt
Lorem ipsum
```

We observe that the previous content was overwritten. To instead append, we use

```
> pwd >> placeholder.txt
> cat placeholder.txt
Lorem ipsum
/Users/jah9/Documents/caam519
```

It's also worth taking a second to notice when and where line breaks were introduced while we were using piping.

We can redirect the other way as well. For example, if we want to count the words in the file placeholder.txt, we can do

```
> wc < placeholder.txt
       2       3      42
```

To interpret this output, the text file has 2 paragraphs, 3 words, and uses 42 bytes of memory.

STDOUT, STDOUT, and STDERR

Every program run from the shell has three streams of data associated with it: input, output, and error. The standard names, along with the numbers used to identify them in various commands, are `STDIN=0`, `STDOUT=1`, and `STDERR=2`, respectively.

If we try to cat a file that doesn't exist, we get an error:

```
> cat nonexistent_file.txt
cat: nonexistent_file.txt: No such file or directory
```

We can redirect the error message to a log file with:

```
> cat nonexistent_file.txt 2> errors.log
> cat errors.log
cat: nonexistent_file.txt: No such file or directory
```

If we want to log both output and error streams, we can redirect one to the other:

```
> ls -l placeholder.txt nonexistent_file.txt 2> errors.log 2>&1
> cat errors.log
ls: nonexistent_file.txt: No such file or directory
-rw-r--r--  1 jah9  staff  42 Aug 27 18:50 placeholder.txt
```

This functionality can be useful to log output from numerical simulations, so that you can inspect them later.

Hidden files

Now let's create a file with an unusual name:

```
> echo "A hidden file" > .hidden_file.txt
```

Note the leading `.` in the name. If we inspect the contents of the directory now, we don't see the file we just created:

```
> ls
placeholder.txt
```

The leading `.` is convention for a hidden file, and won't be visible (be default) to commands like `ls`. Hidden files are often used for configuration files or to hide directories from unrelated programs, such as git. If you want to see all files, hidden or not, pass an extra flag to `ls`:

```
> ls -a
.                ..               .hidden_file.txt placeholder.txt
```

Very simple globbing

Glob patterns allow you to perform simple pattern matching using wildcard characters. For example, imagine you have a directory with some random files, along with multiple log files, each from a different run of your simulator:

```
> ls
my-simulator sim-results-1.log sim-results-2.log sim-results-3.log
something-else.log
```

If we want to delete the 3 log files from our simulator, but leave everything else, we can use the * wildcard to match only those files:

```
> rm sim-results-*.log
> ls
my-simulator something-else.log
```

Using this glob is equivalent to explicitly passing each simulator log file as an argument:

```
> rm sim-results-1.log sim-results-2.log sim-results-3.log
> ls
my-simulator something-else.log
```

A more powerful version of globs are regular expressions, which are outside the scope of this course.

A warning when using Linux

Linux can be an unforgiving computing environment: it is easy to create major problems on your machine with very minor mistakes to simple and common commands. So: be careful before you hit `RETURN`!

The program where it is easiest to shoot yourself in the foot is `rm`. Recall that `rm *.txt` deletes all files whose filename ends in `.txt`. However, if you accidentally add a stray space to the command, you will find that `rm * .txt` will first delete every file matching `*`--namely, everything in the current directory--before then attempting to delete a file named `.txt` (and throwing an error in the likely event such a file does not exist). Moreover, this deletion is permanent in Linux; there is no guaranteed way to recover your accidentally deleted files.

But it gets worse! The `-r` flag to the `rm` program recursively walks all subdirectories, deleting all matching files. If you had accidentally added this flag, the program would have attempted to delete *all* of your files down to the root directory. That's a lot of damage for one command![1]

Permissions

Let's add an extra flag to the `ls` command, which prints the output in "long" format. The output fills the screen, so we'll pick out three lines in particular:

[1] Your implementation of `rm` will likely try to save you from yourself in this situation, and prompt the user before deleting write-protected files. However, there is *another* flag you could pass to `rm` that would bypass this last countermeasure.

```
> ls -l /etc
...
lrwxr-xr-x   1 root  wheel      15 May 17 05:10 aliases -> postfix/aliases
-rw-r-----   1 root  wheel   16384 Feb 22  2019 aliases.db
drwxr-xr-x  10 root  wheel     320 May 17 05:13 apache2
...
```

The first inscrutable block of characters has the following interpretation. The first character says that this object is either a file (`-`), a directory (`d`), or a symbolic link (`l`) which points to the a location elsewhere in the filesystem to find the file contents of interest. The next three characters are the owner permissions: if there's an `r`, that means the owner can read the file, if there's an `w`, the owner can write to it, and if there's an `x`, they can execute it. Who, exactly, is the owner? If you look right a few blocks of text, you can see that each object is owned by the `root` user. The next three characters in the first block specify the permissions for the group that owns the file (`wheel` for each of the files above), and the last three are permissions for all other users.

Who is the root user? Also called the superuser, root has ultimate permissions over everything on the computer. I am not logged in as root on my machine, so since `aliases.db` only has read permissions for root user and the wheel group, I get

```
> cat /etc/aliases.db
cat: /etc/aliases.db: Permission denied
```

Now I can issue commands as root by asserting myself as the superuser via the sudo program (the !! means rerun the last command):

```
> sudo !!
Password:
```

The command then prints a long mess of nonsense, since the database is a binary format.

If you own a file, you can change the permissions with the `chmod` command.

The package manager

Oftentimes you'll need something at the command line that doesn't come preinstalled. In fact, the two text editors we'll introduce in a moment don't come preinstalled on Ubuntu. Thankfully, many common Linux operating systems come with a built-in package manager to help you easily install command software. Since our class is officially based around Ubuntu, we will

introduce their package manager, `apt`.[2] Briefly, the main functionality you will need is the ability to add new packages:

```
> sudo apt install git
```

And update your entire manifest of packages:

```
> sudo apt upgrade
```

Note that you must run these commands as super user!

Text editors

If you've programmed before, you've used a text editor of some form or another to actually write the code. Popular options include Atom and Notepad++, among many, many others. We'll focus on two stalwarts that you can use directly through the terminal: vim and emacs. Both take some getting used to, but can be extremely powerful if you put the time in. We'll only cover the basics, since you may need to interact with one or both (when writing a git commit message, for example). In particular, you need to know the magic incantations used to save an exit the programs, as otherwise you may get stuck.

Different people hold very strong opinions about which text editor is superior. In reality, there is no "right" choice, and I suggest you select the tool you are most comfortable with. However, I'm most familiar with Vim, so you will see much more of it in the course as a result.

Vim

To get started with Vim in a fresh Ubuntu install, you will need to install it via the package manager. We'll circle back to what this command means later, but for moment just enter

```
> sudo apt install vim
```

Enter your password, and wait a moment.

When interacting with Vim, you are in one of three modes: normal, insert, or visual. Insert mode is what you might expect from a text editor, where you can manipulate the text directly. Normal mode allows you to enter commands; for example, to save or rename a file. Visual mode allows you to select blocks of text for manipulation.

---

[2] If you use a Debian system, `apt` will also be available, while if you're on an Arch Linux install, the `pacman` package manager is provided.

Many a beginning vim user has gotten stuck in one of the modes without knowing how to get out, terribly confusing themselves. The `ESCAPE` key will exit insert mode for normal mode. A sampling of the most important commands in normal mode are:

- `i` -- Enter insert mode at the character preceding the cursor
- `a` -- Enter insert mode at the character following the cursor
- `v` -- Enter visual mode
- `:w` -- Save the file (in normal mode)
- `:q` -- Exit the file (in normal mode)
- `:q!` -- Force exit the file, deleting unsaved changes (in normal mode)
- `dd` -- delete current line
- `yy` -- yank current line
- `p` -- paste clipboard
- `/keyword` -- search for next instance of “keyword” in text
- `k` -- move up a line
- `j` -- move down a line
- `h` -- move left a character
- `l` -- more right a character
- `:some_number` -- jump to line #some_number
- `SHIFT+e` -- jump to end of word
- `SHIFT+b` -- jump to beginning of word
- `$` -- jump to end of line
- `^` -- jump to beginning of line

Emacs

To install emacs, run

```
> sudo apt install emacs25
```

GNU Emacs functions slightly differently; you don’t have to worry about different modes, as you issue all commands using more complicated commands (e.g. combos using `C=CTRL` and `M=ALT/OPTION`). The basic commands are:

- `C+x C+s` -- Save current file
- `C+x C+c` -- Quit emacs
- `C+s keyword` -- search for next instance of “keyword” in text
- `C+p` -- move up a line
- `C+n` -- move down a line
- `C+f` -- more left a character
- `C+b` -- more right a character
- `M+g M+g some_number` -- jump to line #some_number
- `M+f` -- jump to end of word

- `M+b` -- jump to beginning of word
- `C+e` -- jump to end of line
- `C+a` -- jump to beginning of line

Note that if you're using Emacs on a Mac, the "meta" key is `OPTION`, and you will need to alter the keymapping in your terminal slightly to get things working properly.

Communicating with running programs

A command with very limited utility is the sleep function:

```
> sleep 60
```

If you try running this on your machine, the terminal will lock up (or fall asleep for 60 seconds). If we get restless, we can talk to the running `sleep` command and tell it to interrupt what it's doing. Pressing `CTRL-C` will send what's called a `SIGINT` interruption signal to the running process, telling it to interrupt what it's doing. We can use this to exit out of the sleep command we foolishly entered. `SIGINT` will be your friend if, for example, you ever start a long-running process (say, a simulation), and realize immediately after hitting `RETURN` that you have a mistake in you configuration that you need to correct.

One slightly silly thing you can do is run the sleep command in a background process, using the & chaining command:

```
> sleep 60 &
[1] 12596
```

The output tells you that process 12596 is running the sleep command in the background. The number is the process ID, or PID for short.

We can see that the process is, indeed, running with

```
> top
```

This will print a live updating table of processes running on your machine; use the PID to track down the sleep process.

You can stop a running process with

```
> kill 12596
```

Shell histories

Your shell will keep a history of all commands that you run, typically stored in a plain text file named something like `~/.bash_history`. If you want to access a complicated old command you previously ran, you can try a number of things. First, you can tab through previous commands in chronological order by using the ↑/↓ arrows. Alternatively, you can use the reverse-i-search functionality to find the old command. This functionality, accessible via `CTRL+r`, will search backwards through your history incrementally for previous commands that match your search string. You can search for the exact text, or use more powerful regular expressions if you would like.

Talking to other machines

Oftentimes you will need to run programs on another machine; for example, a supercomputing cluster, or your advisors desktop. To do this, you can use the terminal to set up a secure network connection. For example, I can log into the CAAM cluster using my normal credentials

```
> ssh joehuchette@harvey.caam.rice.edu
joehuchette@harvey.caam.rice.edus password:
```

My username is `joehuchette`, and I am attempting to connect to a machine whose name is `harvey.caam.rice.edu`.
After typing in the password, I am logged into the remote machine:

```
harvey1.caam.rice.edu% pwd
/home/joehuchette
```

And can run whatever programs I need to. Alternatively, I might want to move files around between computers. Here I can use the `scp` program. If I have an active session on my personal computer, I can copy a file from the CAAM cluster with

```
> scp joehuchette@harvey.caam.rice.edu:/home/joehuchette/my-file.txt
/Users/jah9/Documents
joehuchette@harvey.caam.rice.edus password:
my_file.txt
100%   71     1KB/s   00:00
```

Note the `:` delimiter, which separates the machine and account to communicate with on the left, with the absolute path of the file to be copied on the right (you can also use relative paths with respect to the user home directory as well). Here, `joehuchette` is my username on the remote

machine, and `harvey.caam.rice.edu` is the remote machine name. Note also the second argument, which is the path on my local machine where I want the file copied to.

Shell scripting 101

The shell that you're using is really just an interpreter for a programming language (e.g. bash), and as such you can program the shell by writing shell scripts. I will only cover the very basics.

Open a file named first_shell_script.sh in your favorite text editor, and copy the following into the file:

```
#!/bin/bash
echo "Hello, world!"
```

The first line is known as the "shebang", and instructs the shell which program it should use to run the script. This is important, since different shells can have different syntax and support different functionality.

If we try to run the script with `./first_shell_script.sh`, we will get a permission denied error. To remedy this, we need to add executable permission to the script. We can do that with the (completely inscrutable) command

```
> chmod 711 first_shell_script.sh
```

This gives the owner read/write/executable permissions, and everyone else executable permission. If we had instead done `cmod 755 ...`, this gives the owner read/write/executable permissions, and everyone else read/executable permissions. Look over the `man` pages for `chmod` for why this works; the API is fairly insane. Anyone want to guess what the numbers mean?

Also, if you just want to change one permission, you can do it with something like

```
> chmod u+x first_shell_script.sh
```

A more elaborate example, using interpolation and inputs to the shell script, might be:

```
#!/bin/bash
echo "My name is $0 and you gave me $# arguments, the first of which is $1"
```

Observe what happens if no arguments are passed.

Here's an exercise: write a shell script that takes two arguments:

1. A filename `fname`
2. A string `ext` giving a file extension type

It then redirects the contents of each file in the current directory ending in the extension `ext` to the file `fname`. Then, it prints the number of characters written to the file: "Wrote all files ending in X to a file named Y."

Run it with

```
> ./script.sh out.log .txt
```

Environment variables and start-up files

Where, exactly, do all the programs we have been running live on your machine? Some of them are built into the OS:

```
> which cd
cd: shell built-in command
```

Some are merely files themselves, typically living in the `/bin` directory:

```
> which cat
/bin/cat
> ls /bin
[         cp        df        hostname  link      mv        rm        stty
unlink
bash      csh       echo      kill      ln        pax       rmdir     sync
wait4path
cat       date      ed        ksh       ls        ps        sh        tcsh
zsh
chmod     dd        expr      launchctl mkdir     pwd       sleep     test
```

Even though these programs do not live in our current directory, the shell still knows where to find them because the directory they reside in sit on our path:

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

Here, PATH is an environment variable; you can think of it as usual variable for the bash (or csh, or zsh…) programming language. Note that the $ in the command above performs variable interpolation; without it, we would not get the response we expect:

```
> echo PATH
PATH
```

The shell will walk the PATH, searching the directories from left-to-right, until it finds the program you would like to run. This means that if there are multiple files with the same name on your path, it will choose the first one encountered. Sometimes this will be an issue, particularly if you would like to disambiguate a file in your current directory from one on your path. For example, imagine that, for some reason, we decide to rename our shell script to conflict with another, quite common, program:

```
> mv first_shell_script.sh pwd
```

If you do

```
> pwd
```

the shell will first search for a program named pwd on your path, where it will likely encounter one in your `/bin` or `/usr/bin`. To specifically call the local version, you can use the single dot to denote the local directory location:

```
> ./pwd
```

We can alter our path by adding new directories to it:

```
> export PATH=$PATH:/Users/jah9/new_directory_on_path
```

Now, any files residing in `new_directory_on_path` can be invoked from any directory.

However, this change to our path will go away if we start a new shell session. If we want to make our changes permanent, we can add them to our shell startup files. The startup file name will vary based on OS and shell, but for the bash shell, it will be either ~/.bashrc on Linux or ~/.bash_profile on MacOS.[3] If we add the previous export line to our startup file, we can start a new shell session and observe that our path is appropriately altered.

[3] The difference is actually more subtle; both files have roles on both OSes, but the Terminal applications work slightly different on Linux and MacOS.

```
> echo "export PATH=$PATH:/Users/jah9/new_directory_on_path" >>
~/.bash_profile
> bash
> echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin::/Users/jah9/new_directory_on_
path
```

The startup files will prove useful for a number of things, including setting the PATH, adding aliases for complicated commands via `alias`, or directing applications to find various things such as libraries (`LD_LIBRARY_PATH`).