Intoduction to UNIX

version 0.1

Last generated: December 10, 2019

Table of Contents

Processes

Creating, Reading, and Editing Files

Printing and Reorganizing Files

Images

Keyboard Shortcuts and Special Characters

Other Helpful Things

Quick Intro to Computer Admin-ing

Running Jobs

Fun Commands

Introduction

This is a list of commands and their contexts for getting started with using Unix. It's probably going to be somewhere between nothing and StackOverflow in terms of explanation. The odds are high that if you have a very specific thing that you want to do, that someone on the internet has answered something similar to it on StackOverflow .

PDF Download

Command Line

First, you're going to need a way to access the command line. If you're on Linux or Mac machine, this is easy! Both have applications that come with your computer named Terminal. If you're on a Windows device, you'll need to download a Terminal emulator, such as PuTTY or MobaXterm . After opening Terminal, you'll probably feel like a hacker. That's cool. The feeling takes a long time to go away.

> ℹ **Note:** When using the command line, there is always who you are and which folder you're in to the left of where you type. This bit ends with a `$`, which will be shown in every command line example in this guide.

UNIX and Unix-Like File Systems

UNIX is a family of computer operating systems that meet a set of criteria (some examples include Apple's macOS and Oracle Solaris). Unix-like systems are computer systems that behave similarly to UNIX, without necessarily meeting the Single UNIX Specification. Linux operating systems are Unix-like systems, and may be known as GNU/Linux, due to being a GNU derivative. Linux distributions (i.e. CentOS, Ubuntu, Debian, Fedora, Red Hat, Mint… there are a lot) are free and open-source, and all follow a similar layout in how they are set-up.

Root Directory

The root directory is the penultimate directory on a Unix system, and can be accessed through `cd /`. The root directory is set up in a way so that only the administrator(s) (known as the "root user(s)" can make changes in this directory. This is a safety issue–just messing around with any old file or directory without knowing the purpose can royally screw up the system, since these files describe the operating system itself.

Bin Directories

The bin directories usually hold programs. The root bin directory can be accessed through `cd /bin`. If you do that in a Terminal, and then use the ls (page 24) command, you will list a bunch of different programs. You'll even find stuff like ls (page 24)! That's because commands are programs that are accessed to do their intended purpose. Additional bin directories are:

```
/bin/
/usr/bin/
/usr/local/bin/
/sbin/
```

Home Directory

The home directory is your safe place. It's essentially your owner folder on the computer, which is where you'll create all of your personal files and folders from. You can access your home directory at any time using the cd (page 21) command. The home directory typically has a path of `/home/username/` which is equal to `~/` to the computer. Hence, anytime you're away from the confines of your own

home directory (like when copying files between computers), to navigate back to it, you'll need to remember the tilde. Home directories are automatically created when a new user is added to the computer.

Scratch Directory

The scratch directory is usually given the largest disk partition on a computer, meaning it has the biggest space allocation. While home directory creation is automatic, scratch directories need to be created by the user. To create your own, use: `$ mkdir /scratch/username`.

It is possible that you need to be a root user to create your own personal scratch directory. If you're a root user, you can do this with sudo (page 98). If you are not, then ask an administrator for help.

Graphical User Interface

Remember how I talked about the command line? Well, if you've ever used a computer like a normal person, then you've had a nice visual component that enabled you to never think about the command line before. That visual component is known as the Graphical User Interface, or GUI. Some programs only function via a GUI, or perform best through a GUI. That's why when you're remotely connecting to computers, you may encounter times that you need to set up stuff like graphical forwarding (known as X11). X11 allows you to see the user interface that you're remotely accessing on your own screen, and interact with it (albeit slowly). Using the GUI is likely more intuitive while sitting at the physical computer (since it's designed that way), so there are likely a large number of commands that will be easier through the GUI than the command line (such as copying files between folders). Those commands become important, however, when you cannot physically access the computer.

Shells (bash and csh)

Shells are command-line interpreters. They are related to terminals (text input/output environment) and consoles (physical terminals). While consoles are terminals are similar; the shell is slightly different. The shell is primarily used to start other programs, so you use commands in a shell environment through the terminal or console. That said, there are multiple types of shells, and computers can generally switch between them. To determine which shell type you are using, type `$ echo $SHELL` into your Terminal window. I almost exclusively work in a **bash** environment, which returns `/bin/bash` to the Terminal. Bash is the default shell for most Linux distributions. However, as I mentioned, there are other shells, like csh (C-shell), zsh (Z shell), fish (friendly interactive shell), tcsh (TENEX C-shell), and ksh (KornShell).

If I wanted to change from a bash shell to a C-shell, I would type `csh`. To switch back, I would type `bash`. Some programs require the use of a different shell type, which may also differ by how it was installed. Gaussian, for instance, likes C-shell, but newer editions have install instructions for bash shells (in case you're wondering, the difference is having a `.login` for C-shell and a `.profile` for bash).

Bash Configuration File (the .bash_profile)

The `.bashrc` (and other `.bash` files) are resource files found in the home directory. Because they are hidden files (i.e. their filename starts with a period so that they do not accidentally get deleted), you need to use ls -a (page 24) to see them. They list different things, like aliases or variables that should be available across your computer upon startup. In general, items from original `.bashrc` file should not be deleted, because they reference other hidden files that may contain similar information. There is usually a commented line that says to add user-specific information after that line.

On my Mac laptop, the `.bashrc` file is called the `.bash_profile`, which is shown below:

```
#Access VMD Executable
alias vmd='/Applications/VMD\ 1.9.2.app/Contents/Resources/VM
D.app/Contents/MacOS/VMD'

#Access Chimera Executable
alias chimera='/Applications/Chimera.app/Contents/MacOS/chimer
a'

#aliases
alias work='ssh -Y username@my-work-computer.org'
alias local='ssh username@my-local-computer.com'
alias dist='ssh username@some-distant-computer.edu'

# Setting PATH for Python 3.6
# The original version is saved in .bash_profile.pysave
PATH="/Library/Frameworks/Python.framework/Versions/3.6/bin:${P
ATH}"
export PATH

# added by Anaconda3 5.1.0 installer
export PATH="/anaconda3/bin:$PATH"
```

As you can see, I use mine mostly for [aliases]](UNIXguide-aliases.html). If you are using a Linux system with a `.bashrc` file, it is generally a good practice to save aliases under a separate `.bash_aliases` file.

Every time something is added to a `.bash` file, the `source` command needs to be used to tell the computer to “reload” that file. This is because every time a Terminal is opened, the `.bash` files are read as-is to set up the environment you’re working in, and changes are not tracked throughout the session. To source a specific file (in the following example, `.bash_aliases`, use

```
$ source ~/.bash_aliases
```

Bash files are different on every computer, so if you have specific things you put into your `.bash` files to make your life easier, you’ll need to copy those lines into the `.bash` files on a new system.

C-Shell Configuration File (the .cshrc)

Like bash shells, the C-shell has a configuration file filled with information that helps set up the environment. C-shell uses the `.cshrc` and the `.login` files. The default of a `.cshrc` includes the following text, which should not be deleted.

```
if (-e /usr/local/etc/csh.cshrc) then
  source /usr/local/etc/csh.cshrc
endif
```

After those lines, or a commented line that specifies you can now add information, you can add in specific information that you would like for the environment (like aliases and environment variables (page 12)). Like with the `.bashrc`, the `.cshrc` file needs to be sourced through a command like `$ source ~/.cshrc`.

Similarly, the configuration files are different on every computer, so if you have specific things you put into your `.cshrc` file to make your life easier, you'll need to copy those lines into the `.cshrc` file on a new system.

Environment Variables

Environment variables are variables that help set the environment. Since that was a horrible definition, I'll try to define by example. The environment is the scene of the play, and the variables are the props that help set the scene (everything from the character's clothing to the backdrop). Basically, the environment variables are strings that the computer sees as something else. One example of this is the `$AMBERHOME` variable, which is used to run the Amber program. Instead of typing `/usr/local/amber18`, users can simply type `$AMBERHOME/` with the rest of the command they want to use. The specified path of an environment variable can be checked through `$ echo $VARIABLE`

A list of all environment variables will be given with the `env` command.

Setting Bash Environment Variables

To set an environment variable in a bash environment, use:

```
export VARIABLE=/path/to/variable
```

The AMBERHOME variable would be set through

```
export VARIABLE=/path/to/variable
```

which can be checked with

```
$ echo $AMBERHOME
```

Setting C-Shell Environment Variables

To set an environment variable in a C-shell environment, use:

```
setenv VARIABLE /path/to/variable
```

$PATH

The `$PATH` variable makes it possible to access programs simply by running their name, which is actually the case for most commands (like ls (page 24) and head (page 87)). PATH makes all of these programs accessible simultaneously, by making them available anywhere. Installed programs are added to the PATH. You can also add directories or files to the PATH by defining them in the configuration file.

For a `.bash_profile` or `.profile` (you shouldn't add user-defined path definitions to the `.bashrc`):

```
export PATH=$PATH:/home/rest/of/path
```

For a `.cshrc` or `.login`:

```
setenv PATH $PATH\:/home/rest/of/path
```

Remember to `source` the file after things have been added.

Since the PATH includes a lot of important things, and almost none of them are user-defined, you should not delete variables from the PATH.

Housekeeping

Using the command line is going to be a lot easier if you keep a few simple tips in mind.

1. For the love of everything, please name files and directories (folders) following a specific set of conventions. What are those conventions? Well, for starters, don't use any of the symbols in the table below (page 14).

Table: Symbols on the naughty list

# pound	< left angle bracket	$ dollar sign
% percent	> right angle bracket	! exclamation point
& ampersand	* asterisk	' single quotes
{ left curly bracket	? question mark	" double quotes
} right curly bracket	/ forward slash	: colon
\ back slash	blank spaces	@ at sign
; semicolon	~ tilde	| pipe

- "But I already used these symbols in my past naming! What do I do now?" Well, my personal suggestion (for everything other than spaces), is that you should rename them through the normal way of accessing folders and right-clicking the name. Trying to rename them through the command line is just going to return errors. Spaces can be addressed through using `\ /` in place of the space and renaming them through mv (page 30), but this can get rather annoying if you have similarly named files, all with spaces.
- "I can't use spaces! What do I do now?!?" Welcome to Computer 101. The easiest way around using spaces is through `_` (the underscore) or `-` (the dash). People that do a lot of web-work are vehemently against underscores because of the implications in search engine indexing and website creation. So if that seems like you, or will eventually be you, you probably just want to use dashes.

1. Do not start or end file names with spaces, periods, hyphens, or underscores.

2. Keep file names relevant, but on the shorter side. Linux has a max filename length of 255 characters (who needs that many?!?) and a maximum path length of 4096 (information on paths can be found in pwd (page 26)).
3. Operating systems are case sensitive, so it's always a good idea to use lowercase for everything. I use uppercase for directories and acronyms; keep yourself consistent. The file `Alpha.txt` is different from `alpha.txt`, and thus both could exist within a given directory.
4. Don't name files the same thing as a Unix command. Why you would even consider this, I don't really know.
5. File extensions are important. Try to use widely-known ones, like `.txt` for a text file. Scripts usually end with `.sh`, python scripts usually end with `.py`.

Regular Expressions

Regular Expressions (aka regexes) are strings that describe specific characters. They may be used in place of something else, or to make searches more versatile, and vary between programming languages. More common regexes include `\t` for tab, `\n` to specify a new line. Others are shown in the table below (page 16).

Table: Common regular expressions.

Character	Usage
^	matches beginning of lines
$	matches end of lines
.	matches single characters
*	matches zero or non-initial character appearances
[chars]	matches the specified characters; ranges denoted with - symbol
[0-9]	matches a single number
[a-zA-Z]	matches a single letter

ssh

Connecting to other computers (or clusters, etc.) involves a command called `ssh`. It stands for “secure shell,” and allows for secure remote access to other devices. This is helpful.

Oftentimes, there are people much smarter than me that want to heavily protect their computers. Universities are one example. They create strong firewalls to protect information, which means that a special way to remote access computers must be used, and that way is through VPN. VPN stands for “virtual private network,” and extends the private network over the public sphere for those allowed access. For those affiliated with the University of North Texas (UNT), the easiest thing is to download Cisco AnyConnect. More information on how to download Cisco AnyConnect for UNT students is available at: UNT ITS

Once installed and connected, accessing computers, clusters, and systems on the UNT network is possible. The VPN address for UNT is `vpn.unt.edu`, and the login is the same as your UNT email login.

After connection, the command

```
$ ssh username@ipaddress.or.title
```

is used to access the device. For instance, if someone regularly logs in as `euid123` and the computer’s network address is `talon3.hpc.unt.edu`, then their ssh line would be:

```
$ ssh euid123@talon3.hpc.unt.edu
```

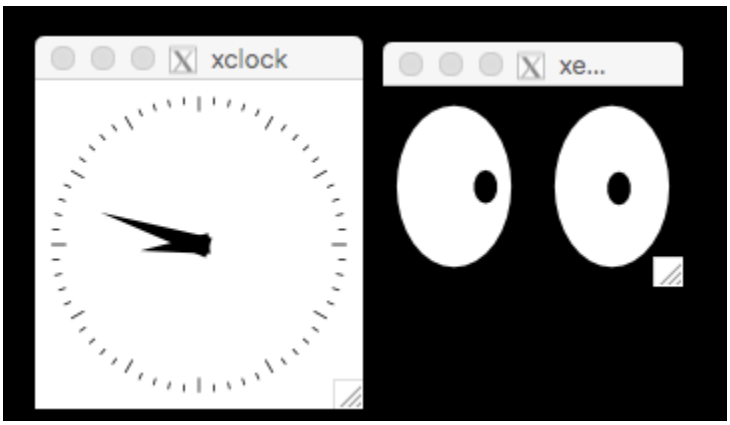
ssh with Graphical Forwarding (X11)

There is a way to set up graphical forwarding (known as X11 forwarding). Basically, this means that visual windows opened through the Terminal are forwarded to your screen. X11 capability is available through PuTTY and MobaXterm on Windows, XQuartz on Mac, and the regular Terminal on Linux. Sending computers (where you've accessed via `ssh`), and receiving Unix-like machines need to have two options located in `/etc/ssh/sshd_config` enabled. These are `X11Forwarding yes` and `X11UseForwarding yes`.

Once these things have been configured, you can start an `ssh` session with X11 forwarding through

```
$ ssh -Y euid123@location
```

The `-Y` uses a secure connection. Otherwise, a `-X` flag can be used if security isn't a major concern. To check the X11 forwarding, the commands `xclock` or `xeyes` can be used.



Use of `xclock` (left) and `xeyes` (right) to test X11 forwarding.

passwd: Changing your Password

Changing your password can be done through the command line with `passwd`.

```
euid123:~$ passwd
Changing password for used euid123.
(current) LDAP Password:
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

New passwords should be slightly complex (meaning digits 0 through 9, punctuation marks, and different cases of letters), otherwise the command will reject it (and if not, it *should* reject it). Passwords should be at least 6 characters long. If the system requires regular password changes, then it will alert you that it is close to expiring.

exit

Anytime you want to exit the Terminal (or sever your `ssh` connection), just type `exit`. If you're every in a territory where `exit` isn't letting you exit, try `quit` or `q`.

cd

Summary: If you've ever decided to organize, ever, then you've probably thought of putting things in folders. Computers are no different, except their folder system is called a directory tree. Each folder is known as a directory.

One of the most important commands created is `cd` which stands for “change directory.” Basically, if I’m in folder `A`, but want to be in `Folder_17` inside of it, then I would use

```
A$ cd Folder_17/
```

If you were going through multiple directories, you would need some slashes. Slashes are important, because they signal the need to sift through a few directories to get to where you want to go. A specific example of this is:

```
$ cd /storage/scratch/share/insert_PI_here_group/
```

A slash after cd but before the location signifies that the computer has to backtrack a little bit (i.e. get away from your home directory), and a slash at the end of the command signifies the end of the folder name. The very final slash isn’t crucial to the success of the command. Unix is very helpful, in that it will tell you when it can’t find what you tried to make it find with an error appearing similar to:

```
-bash: cd: storage/scratch/share/insert_PI_here_group/: No suc
h file or directory
```

which allows you check yourself and retry. Instead of retyping everything you just did (because computer people are lazy, and commands can be overly long), just hit the up arrow key. A log of every command you’ve typed is available in the hidden trenches of your computer, so you can scroll through all of them with the up and down arrow keys. Additionally, if you’re typing a name of something located in that folder (or following that path), you can hit the tab key to auto-fill the word you’re trying for. However, if you have two files, one named

`CrazyPantsA.txt` and the other named `CrazyTownA.txt` then tab will finish filling in `Crazy`, require you to type a `P` or `T`, and then a second tab will finish the title.

`cd` can also be used to go back to the previous directory (like if you went one folder too far). This is achieved through `cd ..`. To backup multiple directories, you add `/..` for each additional jump to make. If I was in folder `D`, I would use the following to return to folder `A`.

```
$ cd ../../../
```

Similarly, using `cd -` will take you back to the previous directory.

```
~ $ cd 82/98/27
27 $ cd ~/ab/ef/gh
gh $ cd -
27 $
```

mkdir

`mkdir` stands for “make directory,” and can be used to create new directories (folders) from the command line. The command to make a folder titled `testingR` would be:

```
$ mkdir testingR/
```

Check out ls (page 24) to find out what’s inside!

ls (and an intro to permissions and flags)

To list items in directories, the command `ls` is employed. The “s” specifies that it is a short list, as opposed to `ll`, which is a long list. If you’re unfamiliar with how the computer lists things the long way, `ls` arranges the information in a more common way.

```
[euid123@computer ~]$ ls
bin         chm5710         lib         chm5660         dropbo
x         mdinputs         test-R
[euid123@computer ~]$ ll
total 28
drwxrwxr-x 12 euid123 eagle12 4096 Apr 22 14:31 bin
drwxrwxr-x   7 euid123 eagle12 4096 Dec 13 13:38 chm5660
drwxrwxr-x   3 euid123 eagle12 4096 Apr 12 11:08 chm5710
drwxrwxr-x   2 euid123 eagle12 4096 Aug 17     2018 dropbox
drwxr-xr-x   4 euid123 eagle12 4096 Apr 22 14:17 lib
drwxrwxr-x   5 euid123 eagle12 4096 Nov    5   2018 mdinputs
drwxrwxr-x   2 euid123 eagle12 4096 May 22 09:55 test-R
[euid123@computer ~]$ ls -lthr
total 28K
drwxrwxr-x   2 euid123 eagle12 4.0K Aug 17     2018 dropbox
drwxrwxr-x   5 euid123 eagle12 4.0K Nov    5   2018 mdinputs
drwxrwxr-x   7 euid123 eagle12 4.0K Dec 13 13:38 chm5660
drwxrwxr-x   3 euid123 eagle12 4.0K Apr 12 11:08 chm5710
drwxr-xr-x   4 euid123 eagle12 4.0K Apr 22 14:17 lib
drwxrwxr-x 12 euid123 eagle12 4.0K Apr 22 14:31 bin
drwxrwxr-x   2 euid123 eagle12 4.0K May 22 09:55 test-R
[euid123@computer ~]$
```

The above example shows several different listing options and what they produce. Some of these have flags, denoted by their `-`, that place conditions or remove restrictions on the command’s use. The flag in `ls -lthr` is actually a combination of 4 different flag types. The `l` shows the long list formatting, the `t` displays newest files first (based on their timestamp), the `h` puts file sizes in “human readable format”, and the `r` reverses their listing (so now the newer files are at the bottom). Including an `a` flag (`ls -a`) for the short list lists everything in the folder, including hidden files (explained further in aliases (page 96)).

In the long format, you can see different information blocks. The `drwxr-xr-x` group shows the permissions associated with the files. The first `d` tells you if the item is a directory (d) or not (-). The next three letters, `rwx` show the user’s

permissions, which in this case are given and include "**R** ead," "**W** rite," and "e **X** ecute." If these were not given, there would be more dashes. The next three are the permissions for the user's group, and the final three are for all users of the computer or cluster. They follow the same rwx/rwx/rwx format. Changing permissions is possible with the chmod (page 120) command, which is discussed later on.

The next group tells you how many items are in the directory (2, 6, and 7, respectively). The next group lists the owner (`euid123`). After that is the group the permissions are assigned to (`eagle12`). The final groups are the file size, the date and time modified, and the name of the file or directory. Note the difference in how file size is given based on the `-h` flag.

pwd

If you ever get lost in your computer, then `pwd` is for you. This command stands for “print working directory” or “pathway directory.” It prints your current location. For example, if you’re located in a shared `insert_PI_here_group` folder and used `pwd`, it’d look like: \begin{lstlisting}[style=P1] $ pwd /storage/scratch/share/insert_PI_here_group/ \end{lstlisting} This location can be copied from the command line and pasted into various locations where it is needed (like code input lines). Paths are important for programs, copying files, creating new files; many commands are path-dependent. Think of the path as the computer’s Google Maps. Without it, you wouldn’t get anywhere. Similarly, you can think of an alias (page 0) as the routes you use so often, you have them memorized.

hostname

Have you ever forgotten which computer you're using? Probably not, since most people aren't working with multiple systems at one time. But it can happen! Which is why `hostname` exists. It's not always helpful, depending on what will be returned, and not always necessary because most terminals include where you're connected, but it is an option you can use.

```
$ hostname
hawkeye
```

whoami

Along the same lines of getting lost in your computer or which computer you're using, you can forget which user you're acting under. The aptly labelled `whoami` command tells you what username you're acting under.

```
$ whoami
username
```

which

When there are multiple installations of a program in different locations on a computer, it can be helpful to know which installation you're trying to access. The `which` command can be used to prompt the path for the program location you would be accessing.

```
$ which python
/anaconda3/bin/python
```

mv

> **Summary:** You can copy files and folders from a number of starting locations. You can also rename them, or permanently delete them, from the command line.

Renaming files is as easy as `mv` (who am I kidding, I'm not funny). To use this command, you need to include the current name and the new name in the command.

```
$ mv current_name.txt new_name.txt
```

That's it! Your file has been renamed!

cp

Copying files locally can be achieved through `cp`. This command can be used to copy a file in the same directory, or copy it from a directory to another directory. The following example demonstrates copying a file titled `current_name.txt` to a file titled `copy.txt`.

```
$ cp current_name.txt copy.txt
```

That wasn't so bad! Now, if I had a file in directory `A`, but I want it to have the same name in directory `C`, I would follow:

```
A$ cp current_name.txt /path/to/dirC/copy.txt
```

where `/path/to/dirC/` would be based on whatever `pwd` says from inside directory `C`. Entire directories can be copied by making the copy recursive with the `-r` flag. This means the folder, and its contents, will all be copied to the location you specify.

```
$ cp -r Folder_A/ Folder_B/
```

scp

Secure copy, or `scp`, is a primary means (along with rsync (page 33)) of copying files between computers/clusters/etc. It works in a similar way to `cp`, but with some of the information of `ssh` thrown in.

```
$ scp current_name.txt euid123@talon3.hpc.unt.edu:~/testingR/co
py.txt
```

In this example, the item `current_name.txt` is being copied from the local computer (or wherever you're currently located) to the user `euid123` on the device `talon3.hpc. unt.edu`. The file is being copied to the the folder `testingR` off the home directory (remember the tilde) of user `euid123`. Since the file name was changed (it doesn't have to be; if it isn't, then the `copy.txt` portion is either left off or also `current_name.txt`), it was included in the path. After this command is run, a password prompt will appear, asking for the password of `euid123@talon3.hpc.unt.edu`. After that is input, then the copying begins. Another use for `scp` is to copy entire directories, and not just files, and it is thus very useful. For copying an entire directory, the `-r` (recursion) flag must be used.

```
$ scp -r directory/ place@to.go.to:~/location/
```

rsync

A way to copy files to protect against newer files being overwritten by those being transferred is using rsync. This is not recommended between personal computers and UNT computer clusters, as it takes a lot of time due to the fact-checking nature of the command. Essentially, if folder A has files 1-23, and folder B has files 20-35, then files 1-19 will be copied just fine. Files 20-23 will only be copied if the files in B have an older “last updated” stamp.

For this example, navigate to the folder that you want things to be copied from. The `*` (asterisk) specifies “from here.”

```
$ rsync -azvp --progress * euid123@talon3.hpc.unt.edu:/home/eui
d123/directory
```

The flags in the order of `azvp` stand for archive, compress, verbosity, and permissions. Essentially, the files will be compressed to cut down on transfer time and the permissions won’t change on the synced files. Additionally, the `--progress` portion will show completion information for each file being transferred.

For this example, navigate to the folder that you want things to be copied into. The last bit (`* .`) basically says “from there to here.”

```
$ rsync -azvp --progress euid123@talon3.hpc.unt.edu:/home/euid1
23/directory/* .
```

This will copy the files from Talon3 to the local computer. The reason both types are specified for rsync is because sometimes the way the computer systems you work with will only allow one or the other (because they hate you). So, it is helpful to be able to work both ways.

sftp

For transferring files between two remote systems (aka through an `ssh` connection), you can use `sftp` (Secure File Transfer Protocol).
First, navigate to the folder that you would like to send or receive information from. Once `sftp` is initiated, the folder for the origin computer cannot be changed. Then, start the protocol with

```
$ sftp username@place
```

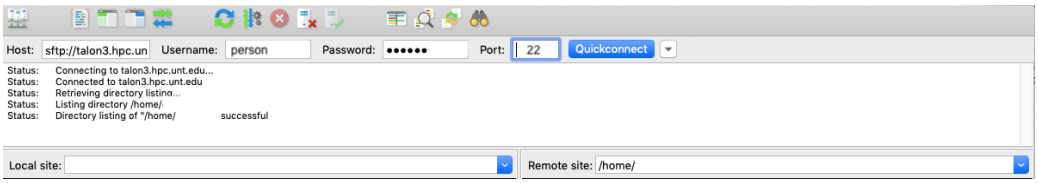
You can specify the folder to navigate to, if you want, but do not use the `~` in place of `/home/username` (it will return an error).

```
$ sftp person@computer.location.org:/home/person/path/to/folde
r/
Password:
Connected to computer.location.org.
Changing to: /home/person/path/to/folder/
sftp>
```

You can create new directories in the receiving end normally with `mkdir`, and use `cd` as normal. To send information, use `put` and to receive use `get`.

```
sftp> put filename
sftp> get filename
```

Filezilla is a helpful program for doing `sftp` through a graphical interface. When using Filezilla, you enter your username and password for where you're trying to connect. Typically, you will use port 22 for connections.



Setting up Filezilla.

rm

Sometimes you need to permanently delete things. For instance, you realized after 2802 attempts that attempt 1 was just horribly incorrect and is taking up valuable memory. The command to use is `rm`, which means to “remove.” To remove an individual file, titled `badfile.txt`, use

```
$ rm badfile.txt
```

Sometimes, that’s not enough though, and your entire directory for `Attempt_1` needs to go. In that case, you’ll need the `-r` flag, which makes the deletion recursive.

```
$ rm -r Attempt_1
```

gzip and gunzip

Summary: Because data management is so essential to computational work, it is important to know about compressing and expanding files. You're probably familiar with .zip folders if you've ever downloaded anything remotely large from the internet. Dealing with zipped (compressed) files is another thing you can do with the command line.

The most common way to compress files in Unix is though `gzip`. Zipping a file turns it into a binary file, which is no longer readable (until it is uncompressed). To zip a file using this command, the syntax (for any number of files, or perhaps a directory) is

```
$ gzip fileA fileB...
```

If you use the `-9` flag, then the files will be compressed as much as possible (`gzip -9 fileA...`). Using `gzip` adds a `.gz` extension to the end of the filename. To unzip a `.gz` file, the command is `gunzip`. Unzipping returns them in their original state.

```
$ gunzip fileA.gz fileB.gz...
```

If you want to use this command to zip or unzip every file in your directory, use a wildcard.

```
$ gzip -9 *
$ gunzip *
```

bzip2 and bunzip2

The less common way of compressing files is using `bzip2`. Like with `gzip`, the syntax is

```
$ bzip2 fileA fileB...
```

Using `bzip2` adds the extension `.bz2`. To uncompress a `.bz2` file, you can use the `bunzip2` command.

```
$ bzip2 fileA.bz2 fileB.bz2...
```

If you want to use this command to zip or unzip every `.bz2` file in your directory, use a wildcard.

```
$ bzip2 *
$ bunzip2 *
```

zip and unzip

Finally, for the times you need to create a folder in the zip folder format, then you can use `zip`, which puts things in the `.zip` format.

```
$ zip zipfoldername.zip file1 file2 file3
```

To zip a directory, use the `-r` flag.

```
$ zip -r zipfoldername.zip dir
```

To unzip the folder, use

```
$ unzip zipfoldername.zip
```

tar

A common way of packaging up files into a single combined file is to use the `tar` command to create a tarball. Oftentimes, the source code for a program is downloadable as a tarball file, since it's easier to distribute and receive. The following command shows the basic syntax, where a folder (or folders) or a bunch of files can be made into a tarball. It is common to just put the files into a single folder and `tar` that folder.

```
$ tar [options] tarball.tar folder/files
```

The `tar` command is used to both package and unpackage tarballs. Thus, the options used are important. Using the `-z` flag will zip the folder (and thus compress its contents), and yield the `.tgz` extension. The `-v` flag stands for verbose and prints as much information as possible while using the command. The `-f` flag specifies that the contents should be put into an archive file. Finally, the `-c` flag specifies that the tarball will be created.

```
$ tar -zcvf tarball.tgz folder/
```

To unpack a tarball, use the `-x` flag (for extract) instead of the `-c` (creation) flag.

```
$ tar -zxvf tarball.tgz
```

If the tarball extension was `.tar` instead of `.tgz`, then you wouldn't need to include the `-z` flag, since the file wasn't compressed through gzip (page 36) during creation. [Note: if you wish to compress the file using bzip2 (page 37), use a `-j` flag instead of the `-z` flag.]

top

To see what processes (aka, what's taking up all of your CPUs) are running on your computer, use `top`.

different process IDs and general information for all running tasks

Use of `top` to show different processes.

The way to exit `top` is by hitting `q`. To the very left is the process ID (PID). That number can be used to kill processes with kill (page 42).

ps

Another way to see what processes are running, but only for what is in your current shell or window, is with `ps`. With `ps`, the `PID` is the process identifier, `TTY` shows the Terminal window running the process, and `CMD` is the command that is running.

```
[euid123@cruntch3 ~]$ ps
   PID TTY          TIME CMD
 60966 pts/12   00:00:00 bash
 61012 pts/12   00:00:00 ps
```

Because the example was run in a `bash` environment, that appears as a line. If it were run in a C shell script, `csh` would appear.

kill

The `kill` command is used to terminate processes. To use it, simply use

```
$ kill -9 PID
```

where PID is the process ID found by using `top`. The `-9` flag ensures that the process is killed by force.

watch

The `watch` command allows you to continuously watch a command. To use it, start the command with the word `watch` and any options that would be applicable to watch. Options for the command that `watch` is being combined with go after that command. One option that can be used with `watch` is the `-n` flag, which allows you to specify a time (in seconds) that the command should be renewed with. As an example, the following command will `watch` the rsync (page 33) or qstat (page 123) command for a specific user, updating it every 5 seconds instead of the default 2 seconds.

```
$ watch -n 5 qstat -u euid123
```

Because the command is continuously updating, the only ways to end it are with rsync (page 33) or cntrl+c (page 44) or closing the terminal.

Cntrl+C: Abort Abort Abort

Occasionally (or not occasionally, I don't know your life), you submit a command like rsync (page 33) or scp (page 32) and you really wish you hadn't. Maybe it's because it is taking too long and you can't keep your laptop open for the length of time it needs to run, or maybe because you're missing a crucial file for the operation. Fear not! There's an abort command! All you need to do is type `Cntrl+C` (like when you copy things in Microsoft Word). [Note: anything that's been completed won't be undone.]

Cntrl+Z: Background Jobs

Sometimes you have long-term jobs running locally in a Terminal, so the command line is unavailable, but you need to do other things in said Terminal. If you know before the job is submitted that it should be run in the background, then submit it with an `&`. However, if it was submitted without the ampersand, then you can simply use `Cntrl+Z` (like undo in Microsoft Word) to temporarily suspend the job. To make it run in the background, type `bg`. If it is alright to run in the foreground again, type `fg`.

Similarly, if you have an ssh connection, where the job will quit when the ssh session has ended, then adding `nohup` before the command, in addition the ampersand, will ensure it runs when the connection is terminated. This is shown below.

```
$ nohup rsync -azvp euid123@talon3.hpc.unt.edu:/home/euid123/di
rectory/* . &
```

nvidia-smi

The way to check GPUs for running jobs is to use the `nvidia-smi` command. Using this command is not only helpful to see which GPUs are available (so you don't double up on jobs and lead to things like overheating…) but also to check that your submitted jobs are actually running. In the following example, AMBER is running on core 3, and nothing else is running on cores 0-2 and 4-7.

```
Mon Mar  5 21:37:17 2018
+-----------------------------------------------------------------------------
--------------+
| NVIDIA-SMI 367.48                 Driver Version: 367.4
8                    |
|-------------------------------+----------------------
+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volati
le Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Ut
il  Compute M. |
|===============================+======================+=======
===============|
|   0  Tesla K80           On   | 0000:06:00.0     Off
|                    0 |
| N/A   21C    P8    26W / 149W |      0MiB / 11439MiB |
0\%      Default |
+-------------------------------+----------------------
+----------------------+
|   1  Tesla K80           On   | 0000:07:00.0     Off
|                    0 |
| N/A   27C    P8    29W / 149W |      0MiB / 11439MiB |
0\%      Default |
+-------------------------------+----------------------
+----------------------+
|   2  Tesla K80           On   | 0000:0A:00.0     Off
|                    0 |
| N/A   22C    P8    28W / 149W |      0MiB / 11439MiB |
0\%      Default |
+-------------------------------+----------------------
+----------------------+
|   3  Tesla K80           On   | 0000:0B:00.0     Off
|                    0 |
| N/A   62C    P0   147W / 149W |    290MiB / 11439MiB |     9
9\%      Default |
+-------------------------------+----------------------
+----------------------+
|   4  Tesla K80           On   | 0000:0E:00.0     Off
|                    0 |
| N/A   21C    P8    25W / 149W |      0MiB / 11439MiB |
0\%      Default |
+-------------------------------+----------------------
+----------------------+
|   5  Tesla K80           On   | 0000:0F:00.0     Off
|                    0 |
```

```
| N/A   27C    P8    31W / 149W |      0MiB / 11439MiB |
0\%      Default |
+-------------------------------+----------------------
+----------------------+
|   6  Tesla K80           On   | 0000:12:00.0     Off
|                    0 |
| N/A   21C    P8    26W / 149W |      0MiB / 11439MiB |
0\%      Default |
+-------------------------------+----------------------
+----------------------+
|   7  Tesla K80           On   | 0000:13:00.0     Off
|                    0 |
| N/A   28C    P8    27W / 149W |      0MiB / 11439MiB |
0\%      Default |
+-------------------------------+----------------------
+----------------------+

+-----------------------------------------------------------------------
---------------+
| Processe
s:                                                                GPU Me
mory |
|  GPU       PID  Type  Process nam
e                             Usage      |
|=======================================================================
===============|
|    3     64236    C   /share/apps/AMBER/amber16/bin/pmemd.cud
a       288MiB |
+-----------------------------------------------------------------------
---------------+
```

du

The `du` command allows you to check how much storage space is available on the computer. There are two flags that make `du` much more managable: `-h` (for human-readable) and `-s` for summary. Human-readable format translates the number of bytes into bigger sizes (like KB, MB, GB, TB, etc.). Summary totals the amount of space for a folder. Running without the summary will provide individual file sizes.

\begin{lstlisting}[style=P1] [euid123@talon3 ~]$ du 4 ./.mozilla/plugins 4 ./.mozilla/extensions 12 ./.mozilla 4 ./.nv/ComputeCache 8 ./.nv 8 ./.ssh 52777464 ./A/A/A 52777468 ./A/A/B 4267968 ./A/B 50819328 ./A/D/A 52509684 ./A/D/B 50339308 ./A/D/C —-MORE—– 379862560 . [euid123@cruntch3 ~]$ du -h 4.0K ./.mozilla/plugins 4.0K ./.mozilla/extensions 12K ./.mozilla 4.0K ./.nv/ComputeCache 8.0K ./.nv 8.0K ./.ssh 51G ./A/A/A 51G ./A/A/B 4.1G ./A/B 49G ./A/D/A 51G ./A/D/B 49G ./A/D/C —-MORE—– 363G . [euid123@talon3 ~]$ du -sh 363G \end{lstlisting}

sleep

If you've ever wanted to pause a computer and make it practically unusable for a set period of time, then the `sleep` command is for you. The command syntax is `sleep X`, where `X` is the length of time in seconds you would like the computer to sleep for. It is likely that this may only be useful in scripts where you need to a make sure a step has been completed.

crontab: a Scheduling Tool

Commands can be set up to run at specific intervals using `crontab`. For instance, automatic backups to an external hard drive can be scheduled to run weekly. The name is derived from Kronos, the Greek god of time, and “table,” since the information is organized in a tabular way.

To edit your `crontab` and set up jobs, use `crontab -e`. This command brings up a file with a lot of comment lines, which should be left in the file to help future you.

```
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'an
y').#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirec
ted).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and c
ron(8)
#
# m h  dom mon dow   command

30 20 * * 4 /home/george/autobackup.sh
```

The last line is the actual specifics for `crontab`. First, the date and time that the command should be executed is specified. It follows minute | hour | day of month | month | day of the week. The example has this set up to run every Thursday (day 4) at 8:30 pm. The week starts with 0 or 7 assigned as Sunday. The asterisks are

essentially an “it doesn’t matter” measure. Finally, the script and location to be executed, which in this case is a script titled `autobackup.sh` in George’s home folder, is specified.

Other flags for `crontab` include `-l`, which lists the information in your crontab, and `-r`, which removes and unschedules crontab jobs. Additionally, those with sudo (page 98) power can set up, list, and remove the `crontab` of other users by specifying their username in the command.

```
$ sudo crontab -u steve -e
```

Autobackup script

The following is the example script used in the `crontab` section:

```
#/bin/bash

NOW=$(date +"%m_%d_%Y")
cp -r ~/Research /media/george/Seagate\ Expansion\ Drive/Backup
s/
tar -cjvf /media/george/Seagate\ Expansion\ Drive/Backups/Resea
rch_$NOW.tar.bz2 /media/george/Seagate\ Expansion\ Drive/Backup
s/Research/
```

What happens is that the variable `NOW` is set as the current date and time. Then, the `Research` folder is copied to the external device. Then, that folder is compressed into a tarball (page 39). After creation, this file needs to be made executable with chmod u+x autoback up.sh (page 120).

Secondary Autobackup Script

This regular autobackup script is helpful if you have a mounted drive with spaces in the name. One of the things it does is it copies the entire folder as normal to the hard drive, and then adds a secondary compressed folder. For a 1TB hard drive with a 1TB external, that is not going to work. Therefore, I personally would suggest renaming your external device. First, plug the device into the computer. Then, use mount (page 109) and locate the device’s name. A lot of information will come on screen, but you’re looking for a specific line (which is shown below).

```
$ mount
/dev/sda1 on /media/georgina/Seagate\ Expansion\ Drive/ type fu
seblk (rw,nosuid,nodev,relatime,user_id=0,group_id=0,default_pe
rmissions,allow_other, blksize=4096,uhelper=udisks2)
```

The mounted device is listed with both it's device location `/dev/sda1` and it's more human-readable location and name `/media/georgina/Seagate\ textbackslash{} Expansion\textbackslash{} Drive/`. Assuming that the drive is in NTFS format (again, please head to the mount (page 109) section), then an administrator can use the `nstflabel` command to rename the device using underscores.

```
$ sudo ntfslabel /dev/sda1 Seagate_Expansion_Drive
```

This removes errors in running the script due to spaces (because computers hate spaces in names (page 14)). Finally, we are ready to create the run script!

```
#/bin/bash

# Backup destination
backdest="/media/georgina/Seagate_Expansion_Drive/Backups"

#Labels for backup name
type="Research"
NOW=$(date +"%m_%d_%Y")
backupfile="$type-$NOW.tar.bz2"

cd $backdest
tar -cvpzf $backupfile /home/georgina/Research
```

Two things to note here. The first is that we change directories to a specific location in this script, so that tar (page 39) can be used from there. Second, we are using a `.bz2` file extension. You can use a `.gz` if you wish (it's faster), but `.bz2` will compress the files more (and we went through this process to ensure there was enough space). Remember, once this script is created, it needs to be made executable through something like chmod u+x autobackup-tar.sh (page 120).

vi editor

The `vi` editor is a “VIsual” text editor. To open something named `fake_document.txt` in this editor, use

```
$ vi fake_document.txt
```

This editor can also be used to create new files. In place of the name of a current file, use a new name you would like the file to be named. To insert text/numbers/commands within the editor, type `i` to bring up “insert” mode. To quit `vi`, there are two options. The first is to quit with saving. For this option, hit escape before typing `:wq`, meaning “write quit.” The second is to quit without saving. For this option, hit escape before typing `:q!`, meaning “quit.” Additionally, to save without quitting, you can use `:w`. There are other modes, such as replace, macros, and visual, which you can research on your own.

The “undo” button is not cntrl+Z (page 45), since that has already been taken for suspense (ha). Instead, undo is `:u`. Practically, this is `Shift+;+u`, since colon is an uppercase symbol.

Another useful feature from within the visual editor is `Shift+G`. [Note: The + stands for “and” here.] This will bring you to the end of the file you’re viewing. Practically, hit escape before typing `:+Shift+G`, which will leave whatever mode you’ve entered before bringing you to the end. Additionally, to move through the file in vi, several letters can be used in place of arrow keys. These are `H` (left), `J` (down), `K` (up), and `L` (right).

In `vi`, you can use `Cntrl+G` to which line of how many lines your cursor is current on. This be used to tell you how long the file that you’re viewing is. To delete lines, you can specify a number and then `dd`. So `22dd` would delete 22 lines starting with the line your cursor is on. A single `dd` will delete the line your cursor is on.

If you need line numbers, you can turn them on with `:set nu` or `:set number`. They can be turned off with `:set nu!` or `:set nonumber`. You can turn on hidden characters with `:set list` and turn them off with `:set nolist`.

If you’re using a Linux system, there is a good chance that your system administrator has installed the gedit text editor, which can be opened from the command line through `gedit filename`. It is a lot easier to navigate for files that need a massive overhaul, but there are definitely circumstances where `vi` has uses (* *cough supercomputers and jobfiles cough* *).

Searching Files

To search a file in the vi (page 54) editor, use `/`. First, use escape to enter any mode that you're in. Then, type the forward slash before the text that you're searching for, and hit enter. This will bring you the the appearance of the string you're looking for.

Using the keyboard shortcut `Cntrl+B` allows you to go back when doing a string search with `/`.

Search and Replace All

Similarly to find and replace all in Microsoft products, you can use `:\%s` to search and replace in the vi (page 54) editor. Practically, to search for `gobbledygook` and replace it with `balderdash` in your file, first hit escape before typing `:\%s/gobbledygood/balderdash/g` and hitting enter. The `g` makes it global (meaning every instance is changed).

Commenting Out

Something that makes scripting and file editing easier when an outsider is reading them are comments. Comments enable more detail about a code for anyone crazy enough to want to understand it. Basically, a comment is something that the computer knows to skip reading, so it can contain anything. Some lines that have been “commented out” are important, like the beginning line of a bash script. Other times, they could be deleted from the code without any affect. The symbol to specify that a comment is forthcoming is `#`. A comment ends when return has been hit. The below script shows things that have been commented out.

```
def is_cool (name) :                #def stands for define
        return (name == "I")

def person(name):
        if is_cool(name):
                print name, "am cool."    #print will print som
ething to the screen
        else:
                print name, "are not cool."

person("I")       #Ah, look at these variables! You can see wha
t I'm going for.
person("You")
```

Something helpful that some systems will do is that they color comments, so you can actually tell that they’ve been commented out. Python does this, which is why that example had gray `#` lines.

less: Safe Viewing

Because with vi (page 54) there is a potential for data loss or overwriting, you may wish to have a “safe” way to view files. Viewing files with `less` does just that. The command syntax is

```
$ less filename
```

To quit `less` mode, you can use `q`, `Q`, `:q`, `:Q`, or `ZZ`. You can also invoke a search for patterns with `/`, typing the pattern, and hitting enter. If you accidentally type a slash, you can just backspace until it is gone. Page navigation can be done through the arrow keys, in addition to page up using `b` (for back) and page down with the space bar. Like with vi (page 54), typing `Shift+G` will bring you to the end of the file.

The man (page 90) page provides a lot more information on `less` mode.

more: Minimalist Safe Viewing

Oddly enough, the phrase “less is more” is true in Unix systems. The `more` command is an older “safe” way to view files. You cannot scroll like by line with arrow keys, like with less (page 58), nor can you search for strings with `/`.

touch

The `touch` program will create empty files. While empty files seem overly useless, it can be critical for installing and compiling programs. Creating files follows the syntax

```
$ touch [options] filename
```

Using the `-m` flag will allow you to update the timestamp on the file. Some clusters or supercomputers will auto-delete files that have been unmodified in a previous timespan, so using `touch -m` to modify the timestamp can be used to prevent those deletions. [Note: some clusters/supercomputers specifically ban doing this, so if that's the case… don't.]

echo

The `echo` command prints text to the screen. It's incredibly useful in scripting, as it can be used to demonstrate how far along a script has gotten during the run through printing statements or the time. `echo` can also print variable locations to the screen, so if you need to access that location without using the variable name, you don't have to search your hidden files.

```
$ echo $AMBERHOME
/usr/local/amber18
```

> and >>: redirecting output

For many commands (or even programs), the standard response when they are run is to print the output to the Terminal. Instead of this, putting the greater than sign (`>`) in your command can force the output to go to a new file. There are two choices for this. The first is a single `>`, which will redirect to a file and overwrite the output. The second is having two (`>>`), which will redirect output to a file, but append the redirected output (i.e. it'll just attach the new to the end of the old).

```
$ echo "ECHO0000000000000000"
ECHO0000000000000000
$ echo "ECHO0000000000000000" > call_and_response.txt
$ cat call_and_response.txt
ECHO0000000000000000
```

And, what's that, a cat (page 63)? What an intro to the next section!

cat (and tac)

The `cat` command can be used to print the contents of a file to the Terminal.

```
$ cat fake_file.txt
cat
dog
fish
elephant
```

The similar command, `tac`, will print the lines in reverse.

```
$ tac fake_file.txt
elephant
fish
dog
cat
```

awk

The `awk` language is useful for whittling down data from a mega-file to a more manageable file. Basically, instead of columns A-Z with different information, you could select out which letters would be of importance to you. The command can even be used to select values greater than or less than different cutoffs, which can make data analysis faster. The information shown below is from a complete file with 24,525 rows of data.

```
 CHR             SNP   A1   A2          MAF  NCHROBS
  21      kgp2850918    C    A      0.03691      298
  21      kgp4753447    A    G      0.03716      296
  21      kgp6829524    A    G      0.06419      296
  21     kgp13210339    A    C      0.05667      300
  21     kgp10927414    A    G      0.06419      296
  21     kgp10658468    A    G      0.06667      300
  21      rs10439884    A    G         0.08      300
```

Using `awk` can select MAF values from a specific cutoff. In the awk line below, the header is printed through `NR == 1` (NR stands for number of records; NF would stand for number of fields and refer to columns), and the remaining data is sorted through column 5 (`$5`) to select out values below 0.05. Then, a new file is created using `>` with the new file name specified.

```
$ awk  'NR == 1; NR > 1 {if ($5<0.05) print}' plink.frq > plink
awk.frq
```

The new opening lines of this now 350 row file are:

```
 CHR             SNP   A1   A2          MAF  NCHROBS
  21      kgp2850918    C    A      0.03691      298
  21      kgp4753447    A    G      0.03716      296
  21      kgp5439554    A    G      0.04667      300
  21      kgp9921880    G    A      0.04333      300
  21     kgp13121553    G    A         0.03      300
  21      kgp1799905    A    G      0.04667      300
  21      kgp4273039    A    C       0.0473      296
```

If, for example, you only wanted to print four specific columns of information to the Terminal (and not a separate file), a command like

```
$ awk  '{print $1, $2, $3, $9}' mega_data_set.dat
```

could be used.

diff

You can check for differences line by line in files using the `diff` command. First, let's examine these two files using cat (page 63).

```
$ cat example-file.txt
cat
dog
fish
elephant
monkey
snake
insect
spider

$ cat other-file.txt
cat
dog
fish
elephant
money
snakey wakey
insect
spider
```

There are two points of difference, which can then be identified with `diff`.

```
$ diff example-file.txt other-file.txt
5,6c5,6
< monkey
< snake
- - -
> money
> snakey wakey
```

Notice, that even though line 6 had `snake`, the difference was flagged because the entire line didn't match.

grep

The command line program `grep` recognizes matching patterns. An example what `grep` can do is shown through the following line. In the line, the `-e` flag tells the program to search for patterns starting with special characters, the `\|` (which is a pipe (page 83)… more on that later) allows multiple things to be found (essentially an "or" feature), and the `>` directs the output to a new file. The whole slew of what you're looking for should be in open quotes.

```
$ grep -e '^ATOM\|^HETATM\|^TER\|^END' 1A31_cleanup_o.pdb > 1A3
1_cleanup_o2.pdb
```

If you used something like

```
$ grep 'word*' *.txt
```

then all of the matching lines starting with "word" found in text files would be printed to the Terminal.

paste

You can combine multiple files together using the `paste` command. If you have a file named `numbers.txt` (that's literally just a list of numbers) and a file you'd like numbered (say `greatcities.txt`), then you can use the below to give you the list printed to the terminal.

```
$ paste numbers.txt greatcities.txt
1       Heidelberg
2       Chicago
3       New Lenox
4       Denton
5       Kirksville
```

Obviously, you can use > (page 62) to direct the output to a new file. It is also good to note that default `paste` use will create the output with a tab delimiter. Changing delimiters can be done through the `-d` flag.

sort

You can sort data with the `sort` command (amazing!). First, use cat (page 63) to print the data.

```
$ cat data.txt
134
127
108
 89
185
...
```

Then we can use `sort`. By default, the data will be sorted in ascending order.

```
$ sort data.txt
  2
  3
 53
 56
 74
 ...
```

The `-r` flag can be used to arrange the values in ascending order, and the `-R` flag can be used to arrange the values randomly. Thus, the flags are case sensitive.

```
$ sort -R data.txt
 94
152
185
105
143
...
$ sort -r data.txt
227
218
213
209
208
...
```

The command can also be used on files with words or letter, where the default is alphabetical order.

```
$ cat words.txt
research
graduate
office
hope
pain
...
$ sort words.txt
chemistry
doctor
education
graduate
hope
...
```

split

Sometimes files are egregiously large and are downright unmanageable to look at. The `split` command can be used to break large files into smaller components, while keeping the original file intact.

Some `split` options include capping at a line number (shown with the `-l5` flag, where there are 5 max lines per file created), specifying the output name prefix (shown as `newname_`), and made verbose, which shows the names of the newly created files.

```
$ split -l5 example_file newname_ --verbose
creating file 'new_aa'
creating file 'new_ab'
creating file 'new_ac'
creating file 'new_ad'
creating file 'new_ae'
```

There are other options, too. `-d` will give the new files a numeric suffix, such as `00`, instead of `aa`. Byte sizes can be specified following the pattern in the table (page 71), where the number is what you want the max file size to be. Alternatively, you can split it into a set number of chunks with something like `-n5` (where it'd be broken into 5 chunks).

Example flags for `split` byte sizes

| Flag | Max File Size | |———-|———————-| |-b2000000 | 2000000 bytes | |-b 50K | 50 kilobytes (KB) | |-b 50M | 50 megabytes (MB) | |-b 1G | 1 gigabyte (GB) |

sed

In the 1970s, `sed`, a Stream EDitor, was created. `sed` reads files line by line, is mainly used for search and replace, and doesn't edit the input file by default (instead printing the information to the screen). The options for `sed` change with each operating system, so I'll stick the the options for GNU ("GNU's Not UNIX", aka the precursor to Linux and what Linux systems use).

Generic `sed` commands follow the syntax of:

```
$ sed 'script' input_file
```

So, if you wanted to search for every instance of the word "hello" and change it to "world" in a file named `input.txt`, creating a new file with those changes called `output.txt`, then the command would look like

```
$ sed 's/hello/world/g' input.txt > output.txt
```

If you wanted these changes to be reflected in the original file by overwriting it, then you would instead use the `-i` flag, which edits in-place.

```
$ sed -i 's/hello/world/g' input.txt
```

In these examples, the `s/` stands for search and the `g` stands for global; together they are the equivalent of "find and replace all." Not including the `g` will make turn an `example.txt` file like

```
__hello__ my dearest friend __hello__
my deepest __hello__ unto you
can you reply to __hello__ with __hello__
I only say __hello__ to myself
```

into this

```
__world__ my dearest friend __hello__
my deepest __world__ unto you
can you reply to __world__ with __hello__
I only say __world__ to myself
```

because, once again, `sed` makes changes line by line. Similarly, `sed` is not recursive, so you can globally replace a word with a phrase containing the world multiple times without causing an infinite loop.

Now, single quotes are not always critical to the use of `sed`, but they won't hurt anything either. Basically, they're necessary for meta-characters (aka anything you shouldn't include in a file name (page 14)). Thus, since nobody understands `sed`, and the people writing `sed` documentation suggest using single quotes every time, you should just learn `sed` using single quotes every time.

You can also delete lines with `sed`. To edit a file by removing the first line, then you would use

```
$ sed -i '1d' filename
```

In the example, `1d` essentially stands for first line deletion.

Using `a\` ("append") can add lines to a file.

```
$ sed '/hello/ a\ Add this line after every line with hello' ex
ample.txt
hello my dearest friend hello
 Add this line after every line with hello
my deepest hello unto you
 Add this line after every line with hello
can you reply to hello with hello
 Add this line after every line with hello
I only say hello to myself
 Add this line after every line with hello
```

You can "insert" lines with `i\`

```
$ sed '/hello/ i\ Look at me go' example.txt
 Look at me go
hello my dearest friend hello
 Look at me go
my deepest hello unto you
 Look at me go
can you reply to hello with hello
 Look at me go
I only say hello to myself
```

or “change” lines with `c\`

```
$ sed '/my/ c\ well this was dumb' example.txt
 well this was dumb
 well this was dumb
can you reply to hello with hello
 well this was dumb
```

These commands can easily be created into a script for editing files. Say you wanted to delete the CRYST line from all the PDB files in a folder. Hmmm, it’s almost like VMD hates this line from Avogadro-generated files. To do this, you could write a script like:

```
#!/bin/bash
sed -i '/CRYST/ c\' *.pdb
```

If you have symbolic links (see ln (page 95), you can have `sed` follow the symbolic links. To do that, use something like:

```
$ sed -i --follow-symlinks 's/^/\t/' *.txt
```

In that example, a tab (`\t`) is inserted at the beginning of each line (`^`) of every `.txt` file, including following through to the referenced files of symbolic links.

Finally, old versions of `sed` only allowed the first line of a script to be a comment, but now comments can be used anywhere in the script, though to be safe they should be entered on their own line of the script.

More information on `sed` is available here and here.

convert

> **Note:** This section will really only be helpful on Linux systems, but considering these things are covered through other means on other operating systems, it's not the end of the world.

Converting between image formats can be a pain or require a photo editor. Luckily enough, `convert` can be used to do it! The command syntax is

```
$ convert [input options] old_image [output options] new_image
```

For instance, you can resize images through something like this (obviously 800x600 is not the only option)

```
$ convert imagename.jpg -resize 800x600 newimagename.jpg
```

Some other options include `-rotate degree_number` and `-crop x{+-}{+-}{%}`.

If you're on a system without ImageMagick, consider downloading it to get this command.

eog

> ℹ **Note:** This section will really only be helpful on Linux systems, but considering these things are covered through other means on other operating systems, it's not the end of the world.

To open image files from your Terminal, you can use the `eog` command. Using `eog image_file_name` will open the image in a new window. Once the image is closed, you can resume using that same Terminal.

evince

> ℹ **Note:** This section will really only be helpful on Linux systems, but considering these things are covered through other means on other operating systems, it's not the end of the world.

While eog (page 76) can be used to open images, the `evince` command can be used to open PDF files. Using `evince file.pdf` will open the PDF in a new window. Once the file is closed, you can resume using your Terminal window for running commands.

Cntrl+A: Begin Again

This command allows you to move the cursor to the beginning of the command prompt that you were typing. This is helpful because you cannot simply click to where in the command you would like to fix; arrow keys must be used to maneuver around prompts.

Cntrl+L: CLEAR!

The Terminal screen can become cluttered with old prompts and printed content. To clean it up (if only for a moment), use Cntrl+L.

Cntrl+R: History's Pal

While history (page 94) will bring up a log of past commands, using `Cntrl+R` will allow you to search through old commands. When I search for “qsub” on insert_computer_cluster, my most recent submission involving “qsub” appears. I can continue going through previous options using `Cntrl+R` again.

```
(reverse-i-search)\`qsub\': qsub basher.sh
```

Pressing an arrow key will bring you out of the search and back to the command prompt, so you can edit the command before executing. As always, Cntrl+C (page 44) will end the search.

Cntrl+U: Clear the Line!

Sometimes the command you've typed into the Terminal's command prompt is just wrong and you want to undo all the typing without deleting it. To clear the prompt, use Cntrl+U.

Tab: the Autofill Key

If you've ever written a paper, then you know that the Tab key exists on a keyboard. This key is a lazy Terminal user's best friend, because it acts as an autofiller for unique names. Say you want to open `theWorldsLongestDatasetwiththeWorldsLongestName.dat`. First, that name is really long. Second, there's a lot of weird capitalization. Say that there are a few file names in the same directory that start with "the," but none of them start with "theW." Thus, you can type `vi theW` and hit the tab key. Suddenly, what you've typed is now `vi theWorldsLongestDatasetwiththeWorldsLongestName.dat`, and you can hit enter to view the file. The tab key can also be used to bring you to the next point of difference, in addition to finishing commands or program names. If I type just `whi`, then hit tab twice, the Terminal will list the two things that start with `whi`, which are the `which` and `while` commands. Thus, by then typing a `c` or `l`, and hitting tab once more, I will have the completed command. This is a great help if you know how a file name starts, but don't remember its unique breaking point.

The Pipe

There's something on the keyboard that looks like a straight line or a weird colon that shares the key with `\`, and that thing is called a pipe (`|`). Use of a pipe allows you to combine multiple commands into a single line, and "piping" allows you to use the output of a command as the input of the following command. To use it in the command, hit `Shift+\`. One of my most commonly used pipe commands is:

```
cat rmsd_all.dat | awk '{print 0.1*$1, $2}' > new_rmsd_protei
n.dat
```

That command prints out the information from the data file, then prints out only the two columns I want and places it in a new file.

Gaze upon the pipe ('|').

*: Wildcards

No, we're not playing UNO. Wildcards are what asterisks are called, because they have many functions, including making lives easier. Say you want to copy every file that has the same extension (like `.txt`) from a folder. To do that, you can do something like

```
$ cp *.txt /path/to/other/directory/
```

Each file will have the same name as it did in the original directory. Similarly, all the `.txt` files from a folder can be permanently deleted with

```
$ rm *.txt
```

If you wanted to list everything that had the `.txt` extension, then you would use

```
$ ls *.txt
```

I'm sure you can see how wildcards are helpful. The only other useful thing that I'll mention here is that if you had things that were the same at the beginning and end, you can use a wildcard for their point of difference. So something like `All_These_<sup>\*</sup>Files.txt` would pertain to anything under those conditions, such as `All_These_Bloody_Files.txt` or `All_These_Silly_Files.txt`.

date, cal, and time

date

No, you're not asking the Terminal out with the `date` command. You're simply asking what time and day it is.

```
$ date
Thu Mar  8 21:11:04 CST 2018
```

cal

Like date, you can print a calendar to the terminal with `cal`. The current date is highlighted.

```
$ cal
     March 2018
Su Mo Tu We Th Fr Sa
              [1]  2  3
 4   5  6  7   8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

time

The `time` command can be used to give information about the time a command or process takes to run. It has a man page, which is kind of elusive on bash systems, as `time` is a builtin (within the shell) there.

If you want to figure out how long a command takes to execute (like scp (page 32), for example), you can use the following (where `command` is the command you're testing). Bash shell on Ubuntu:

```
$ /usr/bin/time -p command
0.00user 0.00system 0:00.00elapsed 0%CPU (0avgtext+0avgdata 243
2maxresident)k
0inputs+0outputs (0major+103minor)pagefaults 0swaps
```

Bash shell on Mac OSX:

```
$ /usr/bin/time -p command
---
        0.00 real         0.00 user         0.00 sys
$ time command
real        0m0.017s
user        0m0.003s
sys        0m0.008s
```

head

The command `head` can be used to print the first few lines of a file. The default use of head with print the first 10 lines of the file, but a flag can be used to print more or fewer lines.

```
$ head -n 25 fake_data_file.txt
```

Use of `-n 25` will print the first 25 lines, as opposed to the original 10. Otherwise, default use would be:

```
$ head fake_data_file.txt
```

tail

Like head (page 87), `tail` can be used to print the final lines of a file. It also will use 10 lines as the default, which can be changed with the `-n` flag.

file

The `file` command can be used to print the format of the data within a file. In this example, the data is a binary NetCDF file. Using cat (page 63) to print the file would make it look like a monster trying to escape the terminal.

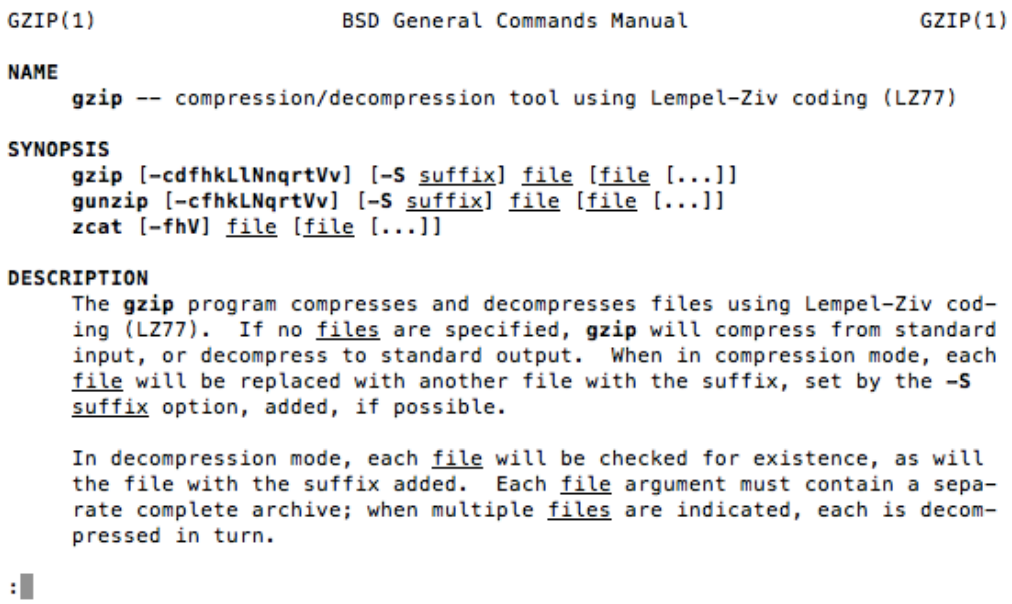
```
$ file trajectory_info.nc
trajectory_info.nc: data
```

For this one, the file is a text file that is easily printed using cat (page 63).

```
$ file trajectory_info.mdcrd
trajectory_info.mdcrd: ASCII text
```

man Pages

Despite their name, `man` pages are for people of all genders. Commands have manual entries that can be accessed through their `man` pages (see top (page 40)).



```
GZIP(1)                   BSD General Commands Manual                  GZIP(1)

NAME
     gzip -- compression/decompression tool using Lempel-Ziv coding (LZ77)

SYNOPSIS
     gzip [-cdfhkLlNnqrtVv] [-S suffix] file [file [...]]
     gunzip [-cfhkLNqrtVv] [-S suffix] file [file [...]]
     zcat [-fhV] file [file [...]]

DESCRIPTION
     The gzip program compresses and decompresses files using Lempel-Ziv cod-
     ing (LZ77).  If no files are specified, gzip will compress from standard
     input, or decompress to standard output.  When in compression mode, each
     file will be replaced with another file with the suffix, set by the -S
     suffix option, added, if possible.

     In decompression mode, each file will be checked for existence, as will
     the file with the suffix added.  Each file argument must contain a sepa-
     rate complete archive; when multiple files are indicated, each is decom-
     pressed in turn.

:
```

Example use of `man gzip`, which describes the gzip command. This command unzips zip folders.

To exit a man page, type `q` for quit or `ZZ`.

find

If you're like most people, using Cntrl+F is the way you live your life. While Cntrl+F isn't something you can use through the command line, you can use `find`. In general, the command syntax is `find [path] expression`. So, to find something with "README" anywhere in the filename from the folder you're using the command in, you would use something like:

```
$ find . -name *README*
```

There are many other options, all of which are readily Google-able or found through the man (page 90) page.

locate

Like find (page 91), `locate` can be used to search for files by name. With `locate`, however, you need not have to include the entire file name; it will search for parts of names. Using `locate` will search your entire computer system, so it can take a significant amount of time depending on specificity and number of files. The general syntax is `locate [options] name`, where options like the `-i` flag will make the search case insensitive.

wc

The `wc` command will find the word count of a file. It has several different flags that make it helpful for determining how big files are. These include

- `wc -w` : gives the word count
- `wc -l` : gives the line count (the last line won't be counted if `\n` isn't included)
- `wc -m` : gives the character count
- `wc -c` : gives number of bytes

history

Unix shells keep a log of your previously run commands (that's how the up arrow function works). To print your history, use `history number`, where the number specifies how many lines to print (the default is to print your entire log, since the last system restart).

```
[euid123@t3-login1 ~]$ history 10
  896  exit
  897  history 10
  898  exit
  899  ls
  900  pwd
  901  top
  902  cd testingR/
  903  ls
  904  cd ..
  905  history 10
```

ln: Symbolic Links

Symbolic links are redirects to a specific file location. Instead of copying a file or folder, you can just provide a link to that in your current working directory, which saves disk space. They can also be used to make scripting easier. The command to create a symbolic link is `ln`. The general syntax is:

```
$ ln -s actual_file symbolic_link
```

In the following example, a symbolic link is created for the `example.txt` file in the home directory to appear on the `Desktop`. The entire file path for both the origin file and the file's symbolic link location must be specified. Then, from the Desktop, `ls -lthr` is used to demonstrate the link is indeed a link, which is specified by the `->`}

```
$ ln -s ~/example.txt ~/Desktop
$ ls -lthr
lrwxrwxrwx 1 simon simon   24 Mar 21 11:03 example.txt -> /hom
e/simon/example.txt
```

Symbolic links can be removed by using rm (page 35) on the link. Specifying that links are links in the link's name may be helpful if you regularly get lost inside your Terminal.

Aliases: for Efficiency and Laziness

Aliases are a powerful tool for the forgetful, the lazy, and the efficient. They take commands that you commonly use and shorten them to a specified command. They are created in hidden files, which are titled with a `.` due to the difficulty to delete them. In your home directory (`cd`), list everything (`ls -a`) to show the hidden files. There is likely something titled `.bash_profile` or `.bash_aliases` . If so, open it with vi (page 54). If not, use `vi .bash_aliases` to create one. [Note: if you're not using a bash shell, look for a similar file for the shell you are using.]

The easiest way to explain the alias, is by giving an example alias. Say you want to make it easy to ssh to a computer. You want to just type `comp` . In this case, make a line in the "." file (make sure to do it after the `#alias` line if there is one!) that looks like:

```
alias comp='ssh euid123@talon3.hpc.unt.edu'
```

and save by using `:.wq` .

Every time you update a `.bash` file, you need to tell the computer that you changed something. You can do this in a few ways. The annoyingly long way would be to restart your computer. Surely you can imagine why that would not be ideal. The easy way is to `source` the file. Basically your computer is like "OH, THAT'S NEW!" The following example will demonstrate that.

```
$ source ~/.bash_profile
```

You might be thinking to yourself, "Why is there a tilde?" In which case, I direct you to the section on home directories (page 7).

Opening Additional Terminals from Terminal

You can open additional Terminals (to the same directory) using system-specific commands. On Ubuntu, use the `gnome-terminal` command. On a Mac OS X running Sierra, use

```
open -a Terminal .
```

You can also use `Cntrl+N` on a Mac to open a new Terminal in the home directory.

sudo: Administrator Rights and Installations

If you have administrator privileges, but are not the complete admin of everything (i.e. the root user), then you are likely able to use `sudo`. If you run into issues where you cannot use a command because permission has been denied, like in this example,

```
[euid123@talon3 local]$ mkdir folder_of_doom
mkdir: cannot create directory \`folder_of_doom\': Permission d
enied
```

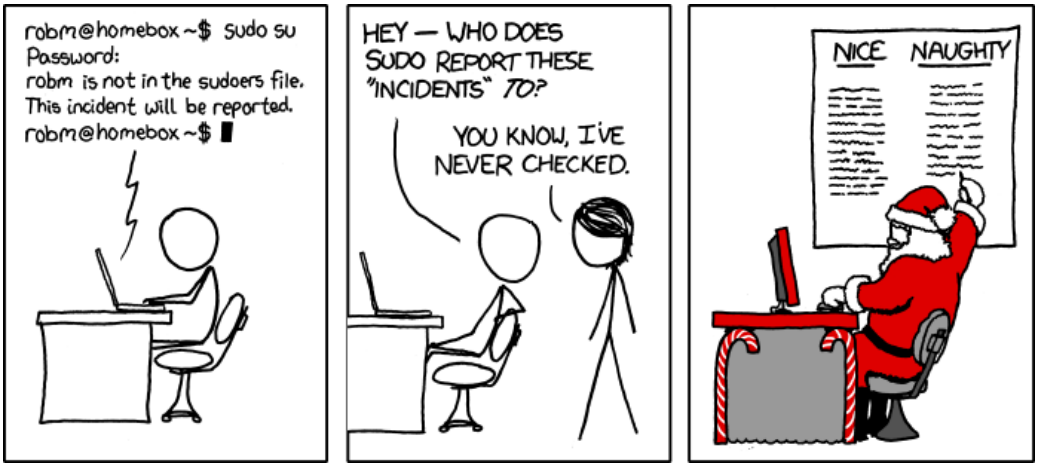
then you can try the command with `sudo` in front of it.

```
[euid123@talon3 local]$ sudo mkdir folder_of_doom
```

You will then be prompted for the `sudo` password. You get 3 tries to get it right (like any use of a password except su (page 114)), and if not, then the instance "is reported to the root user." This is the same as when you try to use `sudo` without the correct privileges.

> **Note:** Instances are reported in `/var/log/auth.log` for Ubuntu systems.

Why mention this at all, then, if you probably can't use it? Because on your own personal computer, or other systems where you are an administrator, you can use `sudo` to install certain programs from the Terminal on machines with the pip package management system of Python installed.



**The Incident* from XKCD.*

free

You aren't going to get a sample or anything with this command. However, `free` will give you information on free, total and swap memory. Using the `-t` flag shows the total memory used. The default for `free` is in bytes, but the `-h` flag can put it in human-readable format.

```
$ free -th
              total        used        free      shared  buff/c
ache   available
Mem:            15G        310M        4.3G         25
M         11G         14G
Swap:           15G          0B         15G
Total:          31G        310M         20G
```

ifconfig

Let's start with definitions: IP and MAC. IP stands for Internet Protocol, and is the numeric number assigned to a computer network access the internet. MAC stands for Media Access Control, and the address is the device's unique identifier for the network adapter. Determining IP and MAC addresses can be accomplished through visiting `/sbin/ifconfig`. You don't even need cd (page 21) to do it!

```
euid123@comp:~$ /sbin/ifconfig
enp0s31f6 Link encap:Ethernet  HWaddr 4c:cc:6a:30:f9:c6
          inet addr:10.144.120.1  Bcast:10.144.120.255  Mask:25
5.255.255.0
          inet6 addr: fe80::9d22:8f52:cb14:fb7f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:40194 errors:0 dropped:0 overruns:0 fram
e:0
          TX packets:31924 errors:0 dropped:0 overruns:0 carrie
r:0
          collisions:0 txqueuelen:1000
          RX bytes:21634036 (21.6 MB)  TX bytes:22810381 (22.8
MB)
          Interrupt:16 Memory:df200000-df220000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:774 errors:0 dropped:0 overruns:0 frame:0
          TX packets:774 errors:0 dropped:0 overruns:0 carrie
r:0
          collisions:0 txqueuelen:1
          RX bytes:69303 (69.3 KB)  TX bytes:69303 (69.3 KB)
```

The initial text block `enp0s31f6` contains information referring to the computer's Ethernet port. The MAC address is listed under `HWaddr`, and the IP address is listed under `inet addr`.

Installing Packages with a Package Manager

Most Unix-like systems have a package manager that can be used to install programs. This is typically done through:

```
$ sudo [command] install program_name
```

The command changes between operating systems, and obviously the program name would change depending on what you are trying to install (a short list can be found in the [table below][UNIXguide-package-manager-install.html#table]. The operating systems using `apt-get` use different related commands in different contexts. If you wanted to search available packages, these systems would use the `apt-cache` base command instead. An initial search would be accomplished through:

```
$ [command] search program
```

Table: Package managers for different systems

Operating System	Command
Ubuntu	apt
Mint	apt-get
Debian	apt-get
CentOS	yum
Red Had	yum
Fedora	dnf
macOS X	macports (or brew)

So, to search for and install a specific python on an Ubuntu system, you would use

```
$ apt-cache search python
-{}-{}-hundreds of results returned-{}-{}-
$ sudo apt-get install python3.6
```

However, it is strongly recommended you update the available lists before you install something, through the following (recognizing `command` is the system-specific command).

```
$ sudo command update
```

For MacOS, you'll probably want to install macports , brew , or fink . Depending on what things you'll be installing, you may need more than one of these.

Fully Removing Packages (and Kernels) with a Package Manager

During program installations (page 102), there are dependent system components that are also installed, that may be spread in various places throughout the computer. Thus, to safely uninstall them, you would use

```
$ sudo [command] autoremove program_name
```

One area where this is critical is removing the old kernel (after testing the system!), because the old kernel keeps a complete previous back-up, using valuable system memory in `/boot`.

Groups (a supplement to chmod)

To add people to a group, you need to know their username (imagine that). The command to add `mark` to the `grads` group would be:

```
$ sudo usermod -a -G grads mark
```

The `-a` flag stands for append (or add) and the `-G` flag specifies that they should be added to the group name following.

Checking Groups

To check who is included in a certain group (we're going to stick with the `grads` example), then you would use:

```
$ grep grads /etc/group
```

Which will give you back the usernames of everyone in the group.

Removing Users from Groups

Well, if you can add people, it's only logical you can remove them too. To remove `mark` from the `grads` group, you would use:

```
$ sudo desluser mark grads
```

> ❗ **Warning:** If you forget to include the group category in this command, you will just… delete the user. That's not good! Thus, a safer way to do this would be through the "Users and Groups" window of the GUI (page 8), which allows you to manage groups.

last

To check user activity, and receive information such as reboot, boot, and kernel version, you can use the `last` command. The kernel version is the central component to the operating system, and is essential for memory, process, task, and disk management.

```
$ last
charlie        pts/8                  10.144.120.7
4                    Wed Mar 21 09:20        still logged in
reboot        system boot          4.4.0-116-generi
Thu Mar 15 15:06        still running
```

`lsof` stands for “list of open files.” Using this command, thus, gives a list of all the open files. Often crazy, ridiculously-named things are being used by programs and show up in top (page 40), so using `lsof` can show where those files originate.

```
$ lsof
lsof   5982    root  mem     REG      8,18    2981280   5085636
7 /usr/lib/locale/locale-archive
lsof   5982    root  mem     REG      8,18     138696    498141
7 /lib/x86_64-linux-gnu/libpthread-2.23.so
lsof   5982    root  mem     REG      8,18      14608    498142
0 /lib/x86_64-linux-gnu/libdl-2.23.so
```

You can also specify usernames with `lsof -u username`.

Mail

Mail in UNIX is both a command for sending emails, and a user file that acts as an inbox for system messages.

Checking Mail

The mail spool is located at `/var/spool/mail/username` (where username is your username), and is a file that can be opened with vi (page 54) (or other file viewing commands).

Sending Mail

You can send mail to users (or email addresses, if you've gone through the work of configuring it….) through

```
$ mail euid123@their.system.email.com
```

Typically, the system email involves their username and hostname (so on Talon3 it would resemble `euid123@talon3.hpc.unt.edu`). Some options include the `-r` flag that can be used to specify the "from" address, `-b` for blind carbon copies, `-c` for carbon copies, and `-s "Mail Subject"` to specify the subject (in quotes).

mount

The `mount` command is used to attach the file directory of a device to the Linux operating system's directory tree. Using the generic `mount` will list all of the mounted devices, which is helpful if you need to rename a device (we'll get to that in a little bit).

Actually mounting devices (outside of "plug it in") requires an administrator, and can be done through something like

```
$ sudo mount /dev/sda1 /mnt
```

This will mount it to the `/mnt` folder. Unmounting devices is done through something either of

```
$ sudo unmount /mnt
$ sudo unmount /dev/sda1
```

Mount also has a `--move` flag, which displays information in a different place.

```
$ sudo mount --move /mnt/Files/Research /home/user/Research
```

Just plugging in devices assigns them a `/dev` name. The device's visible name (ex: Mike_USB_2012) would be found in the file system from the `/media/username` folder.

> ℹ **Note:** Mac likes to put devices in `/Volumes`.

Each user's media folder is automatically created when they plug in a device.

Now, mounted devices are formatted different ways. To change their assigned name, you'll need to know their formatting type. To change their formatting upon use (which will wipe the device), the easiest way is to do it through the GUI (page 8), by right-clicking the name and choosing `Format`. A list of formatting types is shown in the table below (page 110).

Table: List of device formatting types

Type	Description
NTFS	“New Technology File System” – the default for modern Windows-compatible external hard drives. The only limit is the size of the drive, so you could have a single file taking up the entire memory of the external device. Unfortunately, NTFS is read-only for Mac and some Linux distributions.
FAT32	“File Allocation Table” – the oldest file system in computing. Remember floppy discs? They’re worth a Google if you don’t. FAT32 allows files up to 4GB in size, so you can see why you wouldn’t want this for an external.
exFAT	A FAT32 derivative. Ever used a digital camera? This is the common default for memory cards. The issue with it is that since it’s proprietary, Microsoft limits its usage by license obligations.
ext2, ext3, and ext4	“Second/Third/Fourth Extended Filesystem” – built for compatibility with the Linux kernel. It’s the default for several Linux distributions. The system lacks a journal and is ideal for SD cards and USB flash drives on Linux systems.
HFS+	“Hierarchical File System +” – this is the Mac OS Extended format. As you can probably guess, it’s what Mac’s use. When you use an external device on a Mac, 9/10 times it needs to be wiped and reformatted to this format.

NTFS is likely the default formatting type. Each formatting type has a different command to rename a mounted device. For NTFS, that command would follow:

```
$ sudo ntfslabel /dev/sda1 new_name_of_device
```

where `/dev/sda1` is the device tag associated with the device, found through the standalone `mount` command. The device will need to be unmounted to change its name though, which can be done either using `unmount` or through clicking the up-arrow-esque eject button from the directory window.

For FAT32, the command would be (including the two colons):

```
$ sudo mlabel -i /dev/sda1 ::new_name_of_device
```

For ext2/ext3/ext4, the command would be:

```
$ sudo e2label /dev/sda1 new_name_of_device
```

ping

One way to check that computers are online (or responsive) is through `ping`. This command demonstrates how much information (packets) is transmitted and how many are received. If both transmitted and received have the same number, then the computer is online and accessible.

```
euid123@computer1:~$ ping computer2
PING computer2 (10.144.120.1) 56(84) bytes of data.
64 bytes from computer2 (10.144.120.1): icmp_seq=1 ttl=64 tim
e=0.457 ms
64 bytes from computer2 (10.144.120.1): icmp_seq=2 ttl=64 tim
e=0.486 ms
64 bytes from computer2 (10.144.120.1): icmp_seq=3 ttl=64 tim
e=0.380 ms
^C
--- computer2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.380/0.441/0.486/0.044 ms
```

To get out of `ping`, Cntrl+C (page 44) must be used. If the computer isn't on the same network, then a hostname issue will be returned. If the computer is offline, an "unreachable" message will be returned.

reboot

Occasionally, you'll need to reboot (restart) a computer through the command line. Believe it or not, there are multiple death screens before the blue screen of death (*cough* the black screen with white underscore *cough*), which make it difficult to safely reboot a computer. Thus, through a Terminal via ssh (page 17), you can use

```
$ sudo reboot
```

and enter the sudo password. The connection will automatically disconnect, which makes sense because it is restarting. It is always a good idea to use top (page 40) and nvidia-smi (page 46) (if there's a graphics card) to ensure that other users are not in the middle of important jobs or processes. The polite thing in that situation would be to ask if you can reboot, and then proceed with rebooting.

su: Switch User

Say you're helping someone on their computer, where you are also a user, and want to access something like your `.bash_profile` in order to properly assist them. Logging them out to log you in would be ridiculously frustrating, and unfortunately no other computers are around for you to access the system through an `ssh` connection. Never fear! To access your files through the command line, without going through `sudo` 12 times, you can continue using the Terminal as yourself after switching users. This is done through the `su` command.

```
alice@computer $ su bob
Password:
```

This time, you only get one attempt, before you get hit with "su: Sorry."

It is worth noting that the default use of `su` without a specified user is to switch to the `root` user. (You can read more about that in UNIX systems (page 7))

uptime

Computers (like humans) get finicky when they've been awake for a long time. To check the amount of time that a computer has been online, use the `uptime` \ command.

```
$ uptime
 8:37  up 8 days, 17:19, 2 users, load averages: 1.75 1.98 2.10
```

The response displays how long the system has been running, how many users are currently logged in, and the load average for 1, 5, and 15 minute intervals. Load average should ideally stay around 0.70*(number of CPU cores) for Linux systems. A load average of 1.0 per core would essentially mean the system has a full lane of traffic, and over 1.0 means there are cars stopped on the highway on-ramp waiting for a chance into traffic. It is important to know that macOS's load averages act differently, so a completely idle system would show a load average of 1.0. If things are amiss, check the Activity Monitor application.

useradd

Root users can create new user accounts with

```
$ useradd -m newusername
```

The `-m` flag creates a home directory for the user (in this case `newusername`). Once the user is created, you must set a password through

```
$ passwd newusername
```

users

The `users` command shows which users are currently logged in.

w

The `w` command combines the [uptime]{UNIXguide-uptime.html} and who (page 144) commands into a single letter. It also shows what users are currently doing, and where they are logged in from.

```
$ w
 09:05:19 up 5 days, 17:59,  1 user,  load average: 0.00, 0.0
0, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU
WHAT
euid123  pts/8    10.144.120.74    09:05    1.00s  0.04s  0.00
s w
```

The load averages in this example look a lot better, since this command was run on a Linux system. ?

Running Jobs Locally

Jobs are how you produce data and run analyses. There are several different resources attributed to different jobs.

Some jobs are better run on local computers, as opposed to on a cluster, using CPUs. If you're in a computational lab, it is likely that some of your local computers will have GPUs to run jobs on as well.

Local CPU Run Script

Running AMBER locally on CPUs eliminates the need for the queue scheduler lines. When running these scripts, keep in mind that you will need to edit the `while [ \$f -lt 11]` part to run the loop for the correct number of mdin files. This specific script assumes that there are 10 mdin files running minimization, heating, and equilibration, and an 11th specifying production. Thus, the first 10 are run on CPUs through this script. This script also will throw up a flag if the counters for `e` and `f` are set with brackets and not parentheses, so make sure to change that from a different one too. Additionally, before running on CPUs, you will need to modify the permissions (page 120) on the script to make it executable. As a reminder, this can be done through `chmod u+x example_script.sh`. After, you can run the script with nohup (page 45), through something like `nohup ./example_script.sh &`.

```
#!/bin/bash

e=0
f=1

while [ $f -lt 11 ]; do

$AMBERHOME/bin/pmemd -O -i mdin.$f \
-o WT_protein_system_wat_init$f.out \
-p WT_protein_system_wat.prmtop \
-c WT_protein_system_wat_init$e.rst \
-r WT_protein_system_wat_init$f.rst \
-x WT_protein_system_wat_init$f.mdcrd \
-ref WT_protein_system_wat_init$e.rst

e=$((e+1))
f=$((f+1))
done
```

Local GPU Run Script

As you'll see from a later "GPU Run Script" (page 127) section, there are some differences between running locally and running through a queue scheduler. For one, essentially all of the queue scheduler lines just… disappear. Additionally, the "export" line may or may not need to be commented out. You only need it if there are multiple GPUs you could run on–to specify the core you want.

```
#!/bin/bash
export CUDA_VISIBLE_DEVICES=0

e=0
f=1

while [ $f -lt 201 ]; do

$AMBERHOME/bin/pmemd.cuda -O -i mdin.11 \
-o WT_protein_system_wat_md$f.out \
-p WT_protein_system_wat.prmtop \
-c WT_protein_system_wat_md$e.rst \
-r WT_protein_system_wat_md$f.rst \
-x WT_protein_system_wat_md$f.mdcrd \
-ref WT_protein_system_wat_md$e.rst

e=$[$e+1]
f=$[$f+1]
done
```

Executing Scripts (and Changing Permissions)

Computers, while magical beasts, can't always read your mind. When you want to use most scripts (i.e. not one submitted to a queue scheduler), you'll need to use `chmod`. This command changes the permissions of the generated file. So, to make the script runnable (known as executable in computer speak), you would use

```
$ chmod u+x
```

The letters all stand for something, and follow a syntax of `user + privilege`. To add the permission for all users, the user group would be `a`. Similarly, to add only yourself, the option is `u`, and to add group permissions, the option is `g`. All other users fall under the `o` specifier. The different privileges available are read (`r`), write (`w`), and execute (`x`). If you wanted to remove permissions, then you would instead of `+` (for add, see what they did?!?), you would use `-`. After using `chmod`, your job can be run (probably using nohup (page 45), but I'm not you).

A few other things with `chmod`: first, if you're trying to make a directory tree accessible, you'll need the `-r` flag to make it recursive. Second, you can also achieve `rwx` status through using a command with numbers. The table below (page 121) shows what some of these numbers are.

Based on that table, you could use something like the following command to make `folder_a` and all its contents have `rwx` access for the user, `rw-` access for the group, and `r-x` access for other users. Sometimes you'll need to have sudo (page 98) privileges to do this. Also, ensure that you are using the uppercase `-R` flag for recursion, because otherwise you can lose read access.

```
$ chmod -R 765 folder_a/
```

Table: Numeric permissions with `chmod`

Number	Function	Listed	Binary Reason
0	No permissions given	-{}-{}-	000
1	Execute only	-{}-x	01
2	Write only	-w-	010
3	Write and execute	-wx	011
4	Read only	r-{}-	100
5	Read and execute	r-x	101
6	Read and write	rw-	110

Number	Function	Listed	Binary Reason
7	Read, write, and execute	rwx	111

Using PBS Schedulers

The most common queue manager is Portable Batch System (PBS). It was created by NASA, and many clusters use it for scheduling. A complete guide to using the PBS queue manager can be found in the PBS Professional® User's Guide .

> ⚠ **Important:** Each PBS system has is set up slightly differently in terms of the information that you need to pass to it. Ask an administrator for an example *specific* to your system. The examples provided here are based on the system I often work on.

CPU Run Scripts

Run scripts, or jobfiles, contain all the necessary information to run a job. An example is the `basher.sh` script (thanks, Alice!).

```
#/bin/bash
#PBS -q my_cpu_alloc              ## queue allocation
#PBS -l nodes=1:ppn=20,mem=20GB   ## 20 processors
#PBS -j oe                        ## same output and error file
#PBS -r n                         ## Job not re-runnable
#PBS -o err.error                 ## name of error file
#PBS -N WT_protein                ## name of job for queue

cd $PBS_O_WORKDIR

cat $PBS_NODEFILE  > $PWD/PBS_NODEFILE
module load amber/19-mvapich2

e=0
f=1
while [ $f -lt 11 ]; do

mpirun -np 20 -hostfile $PWD/PBS_NODEFILE $AMBERHOME/bin/pmem
d.MPI -O -i mdin.$f \
-o WT_protein_system_wat_init$f.out \
-p WT_protein_system_wat.prmtop \
-c WT_protein_system_wat_init$e.rst \
-r WT_protein_system_wat_init$f.rst \
-x WT_protein_system_wat_init$f.mdcrd \
-ref WT_protein_system_wat_init$e.rst

e=$[$e+1]
f=$[$f+1]
done
```

The very first line specifies that it is a bash script (feel free to read more about that on WikiBooks). The next line specifies where the job should be run on the cluster (using CPUs). The `#PBS -N` line is the job tagline to appear in the queue. The `module load` line tells the cluster to locate the shared location for the cluster's program. In this case, instead of a local installation of AMBER18 in an individual's home directory, the entire cluster can use the installed AMBER18. The list of available modules can be checked through the `module avail` command. Finally, the final lines specify that you want the script to run 10 times, with structures 1-10, based on the appropriate `.mdin` file. If you wanted to change that to structures 3-10, then the `e=0` line would become `e=2`, and the `f=1` line would become `f=3`. To actually run this script, you would need an appropriately named `.prmtop` and an `init0.rst` file. This file is just the original `.inpcrd` copied with a new name, since both are AMBER7 restart files.

The next example is of a Gaussian 16 run script (thanks, Mark!).

Note: To use Gaussian 09, mess with the commenting.

```
#!/bin/bash
#PBS -N My_Gauss_Job                ## name of job for queue
#PBS -j oe                          ## same output and error fi
le
#PBS -o error.err                   ## name of error file
#PBS -q my_cpu_alloc                ## queue allocation
#PBS -l nodes=1:ppn=20,mem=20GB     ## 20 processors
#PBS -r n                           ## Job not re-runnable
#PBS -m abe                         ## Mail on abort, begin, er
ror

## Go to the place you submit the job from
cd $PBS_O_WORKDIR

## input is the name of your job input without the file extensi
on
## and ext is the file extension
## so this job is for test.gjf
## Old versions would resemble test.com
input=test
ext=gjf

#--------------------#
# SET JOB ENVIRONMENT #
#--------------------#
BaseScrDir=/scratch
PbsId=`echo ${PBS_JOBID} | cut -d "." -f1`
ScrDir=${BaseScrDir}/${USER}.${PbsId}

## Change following line to where you are at
cp $PBS_O_WORKDIR/$input.$ext $ScrDir
cd $ScrDir

## Gaussian 09 (uncomment if wanted)
#export g09root=/share/apps/GAUSSIAN/g09d01
#source $g09root/g09/bsd/g09.profile

## Gaussian 16 (comment if unwanted)
export g16root=/share/apps/GAUSSIAN/g16a03
source $g16root/g16/bsd/g16.profile

## Run the job
#time g09 < $input.$ext > $input.log
time g16 < $input.$ext > $input.log
```

```
## Bring the log file, checkpoint, and wavefunction files
## back to the place you submitted the job from
cp -r $ScrDir/$input.log $PBS_O_WORKDIR
cp -r $ScrDir/$input.chk $PBS_O_WORKDIR
#cp -r $ScrDir/$input.wfn $PBS_O_WORKDIR

exit 0
```

In this example, your input file is copied to the scratch directory to run the job, and when the job is completed, the output files are returned to your home directory. This is done so that the calculations are not run on the head (login) node, thereby saving you from being yelled at and receiving a temporary cluster ban. To modify this script, change the `input` and `export` lines to match the file name of the Gaussian input (either `.com` or `.gjf`), which can be generated in Gaussview or Avogadro .

> ℹ **Note:** This Gaussian script is *very* PBS system-specific. Sometimes it's a module. Sometimes you have a bunch of lines like these. Do *not* run Gaussian without a script unless your administrator has told you to do so.

GPU Run Script

There are not many differences between a script to run on GPUs versus CPUs, other than specifying the actual location to run. The following is an example of the `bashercuda.sh` AMBER script to run on GPUs (thanks again, Alice!).

```
#!/bin/bash
#PBS -q my_gpu_alloc       ## queue allocation
#PBS -l nodes=n11-12-13    ## use this GPU node
#PBS -j oe                 ## same output and error file
#PBS -r n                  ## Job not re-runnable
#PBS -o err.error          ## name of error file
#PBS -N WT_protein         ## name of job for queue

export CUDA_VISIBLE_DEVICES=3

cd $PBS_O_WORKDIR

#module load amber/19-cuda_mvapich2
module load amber/19-cuda_serial
#export MV2_ENABLE_AFFINITY=0

e=0
f=1

while [ $f -lt 201 ]; do

#nohup mpirun --bind-to none -np 4 \
#-hostfile $PWD/PBS_NODEFILE
$AMBERHOME/bin/pmemd.cuda -O -i mdin.11 \
-o WT_protein_system_wat_md$f.out \
-p WT_protein_system_wat.prmtop \
-c WT_protein_system_wat_md$e.rst \
-r WT_protein_system_wat_md$f.rst \
-x WT_protein_system_wat_md$f.mdcrd \
-ref WT_protein_system_wat_md$e.rst

e=$[$e+1]
f=$[$f+1]
done
```

Like before, this script specifies a loop, to create MD files 1-200. If you wanted to add a maximum job time of 12 hours (a requirement for some supercomputers), then add the line

```
#PBS -l walltime=11:59:59
```

to your input file, after the other `#PBS -l` line. The reason why you shouldn't specify 12:00:00 for your run time is that the queue sorts by the time limits, and this will place priority on your jobs. An important thing to note in this script is that you specify the node (`#PBS -l nodes=`) and the core (`export CUDA_VISIBLE_DEVICES=`) that you want to run on, which can be verified through nvidia-smi (page 46). This example uses core 3 on node n11-12-13.

R Script PBS Submission

R is a programming language often used for data processing and statistics. Because our simulation analysis generate text files for single runs, it can be helpful to average them across replicates, and R is a great choice for this purpose.

On many clusters, several packages have been installed for all users, meaning that they do not need to be locally installed. To run R and access these packages, you need to first create a `~/.Renviron` file with something like the following line:

```
R_LIBS=/share/apps/R/3.6.0/pkgs
```

As this is a hidden file that's sourced by the module, you only need to make it once.

While most scripts with R are pretty simplistic, scripts that average huge trajectories or perform a lot of math take up valuable computer memory. Some packages also have a lot of overhead can also slow down a cluster for other users. Thus, these types of things (e.g., things that run with the tidyverse package or EDA averaging scripts) should be submitted through the queue using a script like:

```
#/bin/bash
#PBS -q my_cpu_alloc
#PBS -l nodes=1:ppn=1,mem=20GB
#PBS -j oe
#PBS -r n
#PBS -o R.error
#PBS -N R-test

cd $PBS_O_WORKDIR

module load R/3.6.0

Rscript name_of_actual_R_script.R
```

qsub

Submitting jobs is done with the `qsub` command. To submit a jobfile named `jobfile`, the command would simply be:

```
$ qsub jobfile
```

The jobfile has a string of information for running the job, and must include the `\#!/bin/bash` line at the start.

Dependent submissions, i.e. job B needs the output from job A but job A isn't finished and you want to submit job B right now going to sleep, can also be accomplished. To create a dependent job, first submit the first job with a normal `qsub jobfile`. That job will return a job ID (which can be checked through qstat (page 131)), which you'll need to submit the dependent job. The dependency submission would follow:

```
$ qsub -W depend=afterok:12345 jobfile
```

where `12345` is the job ID of `job A` and `jobfile` is the jobfile of `job B`.

> **Note:** On some clusters, the job ID will show something like `12345.my.address.org`; you only need the `12345` part.

If you need to run a job interactively (i.e. there are some command prompts you need to respond to within the job), then use

```
$ qsub -I -q my_cpu_alloc
```

which will allow you access a specific node. To leave interactive mode when the job is finished, use `exit`.

qstat

If you want to see what jobs you have running or waiting to run (queued jobs), then use `qstat`. Using `qstat` alone will show the jobs status for every single user within the PBS manager. To check the queue for a specific job, then you would need to do something like

```
$ qstat 1323523
```

where 1323523 is the job number that was given when the job was submitted. Alternatively, to show just what you, a specific user, euid123, are running, use `qstat -u euid123`. The `-n` flag will give information about the specific computer node that your simulation is running on. Thus, your command could become `qstat -u euid123 -n`. This is a very good command to create an alias (page 96) for.

qdel

Sometimes you scream out in horror when you realize that you shouldn't have submitted a job yet, or it's taking too long and you'd rather just kill it. On PBS systems, this can be done with `qdel`. Again, the job number will need to be added, so that it's practically look like:

```
$ qdel 1323523
```

where `1323523` is the job number that was given when the job was submitted.

Using SLURM Schedulers

A SLURM queue manager is a slightly more unpopular queue scheduler. Unlike a PBS or SGE scheduler, the commands are slightly less straightforward. If this quick guide doesn't provide enough detail, there is more information available on UNT's HPC website or the Slurm website .

CPU Run Scripts

Run scripts, or jobfiles, contain all the necessary information to run a job. An example is the `run-R.job` script.

```
#!/bin/bash
#SBATCH -p public                  # partition
#SBATCH --qos general              # quality of service (priorit
y)

module load R/R-devel

R CMD BATCH clogit.R OUT1.R
```

The very first line specifies that it is a bash script (feel free to read more about that on WikiBooks). The next two lines specify where the job should be run on the cluster. The `module load` line tells the cluster to locate the shared location for the cluster's program. In this case, instead of a local installation of R in an individual's home directory, the entire cluster can use the installed R. Finally, the last line is the command used to run a specific function. In this case, it's to use R on the pre-created R script, `clogit.R` , and give the output file as `OUT1.R` . The list of available modules can be checked through the `module avail` command.

The next script is a Gaussian run script for Talon3. It follows the same idea as the one for PBS, but with some extra SLURM commands.

```
#!/bin/bash
#SBATCH -J My_Gauss_Job        # name in queue
#SBATCH -o Gauss_job.o%j       # output
#SBATCH -e Gauss_job.e%j       # error
#SBATCH -C c6320               # contraint -- specific nodes
#SBATCH -p public              # partition
#SBATCH -N 1                   # Nodes
#SBATCH -n 16                  # Tasks per node
#SBATCH --mem-per-cpu=150MB    # memory allocation
#SBATCH -t 12:00:00            # Wallclock time (hh:mm:ss)

## Loading Gaussian module
module load gaussian/g16-RevA.03-ax2

## input is the name of your job input without the file extensi
on
## and ext is the file extension
## so this job is for test.gjf
## Old versions would resemble test.com
input=test
ext=gjf

## Define scratch directory
export GAUSS_SCRDIR=/storage/scratch2/$USER/$SLURM_JOB_ID
mkdir -p $GAUSS_SCRDIR

## Copy your current folder to the scratch directory
cp $SLURM_SUBMIT_DIR/$input.$ext $GAUSS_SCRDIR

## Go to the scratch directory to run the calculation
cd $GAUSS_SCRDIR

## Run the program
time g16 < $input.$ext > $input.log

## Bring log file, checkpoint, wavefunction and info files
## back to the place you submitted the job from
cp -r $GAUSS_SCRDIR/$input.log $SLURM_SUBMIT_DIR
cp -r $GAUSS_SCRDIR/$input.chk $SLURM_SUBMIT_DIR
#cp -r $GAUSS_SCRDIR/$input.wfn $SLURM_SUBMIT_DIR

#echo "Job finished at"
#date

exit 0
```

GPU Run Scripts

The following is an example script to run GPU AMBER jobs on Talon3.

```
#!/bin/bash

#SBATCH -J WT_protein           # name of job in queue
#SBATCH -o WT_protein.o%j       # output file (%j appends job n
ame)
#SBATCH -p public               # partition
#SBATCH --qos general           # quality of service
#SBATCH --ntasks=1              # Number of nodes
#SBATCH --gres=gpu:2            # 2 GPUs
#SBATCH -t 12:00:00             # Wallclock time (hh:mm:ss)

### Loading modules
module load amber/18-cuda-mpi

e=0
f=1

while [ $f -lt 101 ]; do

$AMBERHOME/bin/pmemd.cuda -O -i mdin.4 \
-o WT_protein_system_wat_md$f.out \
-p WT_protein_system_wat.prmtop \
-c WT_protein_system_wat_md$e.rst \
-r WT_protein_system_wat_md$f.rst \
-x WT_protein_system_wat_md$f.mdcrd \
-ref WT_protein_system_wat_md$e.rst

e=$[$e+1]
f=$[$f+1]
done
```

sbatch

Submitting jobs is done with the `sbatch` command. To submit a jobfile named `jobfile`, the command would simply be:

```
$ sbatch jobfile
```

The jobfile has a string of information for running the job, and must include the `#!/bin/bash` line at the start.

Dependent submissions i.e. job B needs the output from job A but job A isn't finished and you want to submit job B right now going to sleep, can be accomplished through something like:

```
$ sbatch --dependency=afterok:12345 jobfile
```

where `12345` is the job ID of `job A` and `jobfile` is the jobfile of `job B`. The job ID is given when `job A` is submitted, but it can also be checked in the queue with squeue (page 135).

squeue

If you want to see what jobs you have running or waiting to run (queued jobs), then use `squeue`. Using `squeue` alone will show the jobs status for every single user within the SLURM manager. To check the queue for a specific job, then you would need to do something like

```
$ squeue 1323523
```

where `1323523` is the job number that was given when the job was submitted. Alternatively, to show just what you, a specific user, are running, use `squeue -u`.

scancel

Sometimes you scream out in horror when you realize that you shouldn't have submitted a job yet, or it's taking too long and you'd rather just kill it. On SLURM systems, this can be done with `scancel`. Again, the job number will need to be added, so that it's practically look like:

```
$ scancel 1323523
```

where `1323523` is the job number that was given when the job was submitted.

Using XSEDE's Comet

Comet's IP address is `comet.sdsc.edu`. Comet also uses a SLURM (page 132) queue manager. More information available on Comet's website or the Slurm website. To use Comet, you need to have an XSEDE allocation and be added to a project. To apply for an XSEDE account, visit the XSEDE User Portal and follow their steps for account creation. Then, email your XSEDE username to the person with the XSEDE allocation to be added to a project. It may take a few days between being added on the XSEDE portal to actually being allowed to use the resources.

> ☑ **Tip:** One of the "pro-tips" for using Comet is to shave a second off of your wallclock time. Think of how many people aren't doing this but submitting jobs of the same wallclock time–that thought is how many people you're jumping in the queue. Think like a computer!

CPU Run Scripts

An example CPU runscript for Comet is the `comet-cpu-jobfile.sh` script. One of the new parts of this script is the `#SBATCH -A abc123` part. What goes in place of `abc123` comes from using the `show_accounts` command on Comet. You're using this line to pick which allocation is charged system units (XSEDE's computer currency) for running the job. This was modified from Alice's Comet scripts to incorporate SLURM's environment variables.

```
#!/bin/bash

#SBATCH -A abc123            # PI allocation
#SBATCH --nodes=2            # request 2 nodes
#SBATCH --tasks-per-node=24  # number CPU
#SBATCH -J WT-protein      # job name
#SBATCH -o amber.out         # output and error file name (%j e
xpands to jobID)
#SBATCH -t 11:59:59          # run time (hh:mm:ss)
#SBATCH --export=ALL

## mkdir /oasis/scratch/comet/$USER/temp_project/
## NOTE: You should from this a directory off of the above path

#Set up the amber environment
module load amber/18

## The name of the directory that these files are in
## (used to copy mdinfo to your comet home directory)
prefix=WT_protein_system

## Copy the necessary files from the submission location
## to the place the job will run
cp $SLURM_SUBMIT_DIR/*.prmtop /scratch/$USER/$SLURM_JOBID
cp $SLURM_SUBMIT_DIR/*init0.rst /scratch/$USER/$SLURM_JOBID
cp $SLURM_SUBMIT_DIR/mdin* /scratch/$USER/$SLURM_JOBID

## Access the place to run the job
cd /scratch/$USER/$SLURM_JOBID

## Loop variables to restart calculation
## e=input, f=output
e=0
f=1
while [ $f -lt 4 ]; do

ibrun $AMBERHOME/bin/pmemd.MPI -O -i mdin.$f \
-o WT_protein_system_wat_init$f.out \
-p WT_protein_system_wat.prmtop \
-c WT_protein_system_wat_init$e.rst \
-r WT_protein_system_wat_init$f.rst \
-x WT_protein_system_wat_init$f.mdcrd \
-ref WT_protein_system_wat_init$e.rst

## if calculation will not finish within 48 hours, make sure to
```

```
## copy calculation so far to permanent scratch dir INSIDE loop
#cp -R /scratch/$USER/$SLURM_JOBID/* $SLURM_SUBMIT_DIR

## Puts time info in home directory
cp mdinfo $HOME/mdinfo.$prefix

e=$[$e+1]
f=$[$f+1]
done

## these lines copy the files into the submission directory
## after the calculation has finished--make sure to be within
## the wallclock time!
cp -R /scratch/$USER/$SLURM_JOBID/*md$f.out $SLURM_SUBMIT_DIR
cp -R /scratch/$USER/$SLURM_JOBID/*md$f.rst $SLURM_SUBMIT_DIR
cp -R /scratch/$USER/$SLURM_JOBID/*md$f.mdcrd $SLURM_SUBMIT_DIR
```

GPU Run Scripts

An example GPU runscript for Comet is the `comet-gpu-jobfile.sh` script. This was modified from Alice's Comet scripts to incorporate SLURM's environment variables.

```
#!/bin/bash

#SBATCH --nodes=1          # request 1 node
#SBATCH -p gpu-shared      # queue (partition) -- normal, devel
opment, etc.
#SBATCH --gres=gpu:2       # resources you want to use (2 GPUs)
#SBATCH --tasks-per-node=2 # number GPUs
#SBATCH --export=ALL       # Keep the current environment stuff
#SBATCH -J 1DNA-5mrC       # job name
#SBATCH -A abc123          # PI allocation
#SBATCH -o amber.out       # output and error file name (%j exp
ands to jobID)
#SBATCH -t 23:59:59        # run time (hh:mm:ss)

## if you want to submit to gpu rather than gpu-shared
## have number of gpus, tasks per node, and
## OMP_NUM_THREADS equal to 4

## Set up the job environment
module unload intel
module load amber/18
module load cuda

## Set number of threads, should equal number of GPUs
#export OMP_NUM_THREADS=2

## The name of the directory that these files are in
## (used to copy mdinfo to your comet home directory)
prefix=WT_protein_system

## Loop variables to restart calculation
## e=input, f=output
e=0
f=1

## All files should be located in the Lustre filesystem
## So, place them in:
## /oasis/scratch/comet/$USER/temp_project/$prefix

## Copy the necessary files from the submission location
## to the place the job will run
cp $SLURM_SUBMIT_DIR/*wat*.prmtop /scratch/$USER/$SLURM_JOBID
cp $SLURM_SUBMIT_DIR/*md$e.rst /scratch/$USER/$SLURM_JOBID
cp $SLURM_SUBMIT_DIR/mdin* /scratch/$USER/$SLURM_JOBID
```

```
## Access the place to run the job
cd /scratch/$USER/$SLURM_JOBID

while [ $f -lt 101 ]; do

ibrun $AMBERHOME/bin/pmemd.cuda.MPI -O -i mdin.4 \
-o WT_protein_system_wat_md$f.out \
-p WT_protein_system_wat.prmtop \
-c WT_protein_system_wat_md$e.rst \
-r WT_protein_system_wat_md$f.rst \
-x WT_protein_system_wat_md$f.mdcrd \
-ref WT_protein_system_wat_md$e.rst

## Puts time info in home directory
cp mdinfo $HOME/mdinfo.$prefix

## Puts output files into directory accessible outside of job
## Environment--MUST BE IN LOOP
cp /scratch/$USER/$SLURM_JOBID/*md$f.out $SLURM_SUBMIT_DIR
cp -R /scratch/$USER/$SLURM_JOBID/*md$f.rst $SLURM_SUBMIT_DIR
cp -R /scratch/$USER/$SLURM_JOBID/*md$f.mdcrd $SLURM_SUBMIT_DIR

e=$[$e+1]
f=$[$f+1]
done
```

finger (and chfn)

Did I include this program under fun because body parts are funny? Indeed. The `finger` program is used to find out information about a specific user given their username. The command syntax is `finger username`. I'm sure you can think of the word that Unix uses to describe this incredibly appropriate action.

```
[euid123@talon3 ~]$ finger euid123
Login: euid123                          Name: Scrappy Stu
dent
Directory: /home/euid123                Shell: /bin/bash
On since Thu Mar 29 08:03 (CDT) on pts/1 from some-computer.un
t.edu
   13 minutes 37 seconds idle
On since Thu Mar 29 08:28 (CDT) on pts/2 from some-computer.un
t.edu
New mail received Tue Mar 27 15:07 2018 (CDT)
     Unread since Thu Mar 22 14:38 2018 (CDT)
No Plan.
```

As you can see, you can use `finger` on your own username as a way to check to see if you have mail (page 108).

Additionally, you can update your personal information that appears with `finger` using `chfn`. Some system administrators block you from changing this information, though, so knowing this probably isn't very helpful.

mesg

To set whether users can send you messages with the write (page 144) command, use `mesg`. Just typing `mesg` will register what your choice is currently set as (y allows messages to appear; n prevents messages from appearing). To change the settings, use `mesg y` or `mesg n`, depending on the setting you want.

wall

The `wall` command can be used to write a message to every Terminal window currently open on a computer, including your own. This type of thing would be useful if you wanted to warn people about an impending shut down, but it can also be a fun way to mess with your coworkers (though, I may have a warped sense of fun…). The ability to send a message is brought up with `wall`, and the message is sent using Cntrl+D (note: on a Mac it is indeed Cntrl and NOT command). This allows you to use enter to break up information into paragraphs if you so wish.

```
kevin@raphael$ wall

I thought it\'d be fun to send myself a message via Terminal, b
ecause I\'m ridiculous. So here I go.

WHY AM I LIKE THIS?

Lyk this iff u cri evertim

Broadcast Message from kevin@raphae
l
        (pts/7) at 20:34 CS
T...


I thought it\'d be fun to send myself a message via Terminal, b
ecause I\'m ridiculous. So here I g
o.

WHY AM I LIKE THI
S?

Lyk this iff u cri everti
m

my_mac_computer:~ Owner$
```

The first iteration is what I sent, and the `Broadcast Message` is the message that everyone received (because I received a copy when I sent it). It includes who sent it (`kevin@raphael`) and which Terminal it was sent from (`pts/7`).

who and write

Similar to `wall`, messages can be sent to an individual user using `write`. First, you need to know `who` that individual user is (another straightforward command–amazing).

```
emmett@splinter:~$ who
emmett   tty7         2018-03-08 12:18 (:0)
hatice   pts/2        2018-03-07 09:40 (10.144.120.1)
hatice   pts/18       2018-03-07 09:52 (10.144.120.1)
hatice   pts/9        2018-03-08 10:49 (10.144.120.1)
hatice   pts/1        2018-03-08 14:42 (10.144.120.1)
emmett   pts/20       2018-03-09 09:53 (10.144.120.22)
emmett   pts/21       2018-03-09 09:53 (10.144.120.22)
```

In the example, 6 Terminal windows are open on `splinter`. The user `hatice` has 4 windows open, while `emmett` has 3 open, one of which is through an `ssh` session. The dates and times specify when the window was first opened. Now that we know who has windows open, and what they are (shown in the second column), we can send a message to any of those windows. I chose to write a message to myself. This is what the writing terminal looks like.

```
emmett@splinter:~$ write emmett pts/20
Hi pal! Nice you see that I can talk to you via ssh!
I\'m glad I can take over your Terminal. Now get back to work!
```

Like before, I can use enter for line breaks, and I send by using Cntrl+D (and not command if on a Mac). This is what the receiving Terminal looks like (it literally takes over so you can't do work).

```
emmett@splinter:~$
Message from emmett@splinter on pts/21 at 09:54 ...
Hi pal! Nice you see that I can talk to you via ssh!
I\'m glad I can take over your Terminal. Now get back to work!
EOF
```

After the `EOF` (End-of-File), you can hit enter to get the command prompt back.