COMPASS Introduction to the LINUX Command Line

Vasileios Paschalidis[1], Erik Wessel[2]
[1] *Departments of Astronomy & Physics, University of Arizona, Tucson* [2] *Department of Physics, University of Arizona, Tucson*

This note provides a basic introduction to the terminal interface of the LINUX operating system. Through the terminal we will interact with the bash shell, and gain familiarity with many useful command line tools that can be accessed through it.

PACS numbers:

I. WHY LEARN ABOUT THE LINUX COMMAND LINE?

Why are we insisitng on starting this workshop by forcing those of you on Windows to install LINUX? Even stranger, why are we making you learn to use LINUX through text commands that harken back to the phospherescent tele-type consoles of the Cold War era?

As it turns out, there are fundamental reasons why using a text-based interface like the one we are teaching you today is still a useful computing skill, and will remain one into the future. In addition, familiarty with LINUX is very beneficial, even if you keep using Windows most of the time.

To explain why, I need to establish a few basic concepts.

A. Operating System Terminology

With practically no exceptions, every modern computer has an *operating system* (OS). This is a system of software components that allow users and programs to interact with the computer, obtaining resources and using them in ways allowed by the system. An operating system consists of a core program called the operating system *kernel*, and a supporting cast of *utilities*, *drivers*, and *system libraries*:

- The kernel runs constantly and is responsible for the core functionality of the OS:
  - Allocating and managing resources.
  - Scheduling when other programs can run.
  - Controlling user permissions.
- Utilities are a bunch of additional programs that come bundled with the OS and do specific things not handled by the kernel.
- Drivers are pieces of software that allow the computer to talk to other equipment like displays and keyboards.
- System libraries contain the code for performing many common computations in this specific system. This allows programs to work on different systems than the ones they were originally written on

B. UNIX

It turns out that not all ways of designing an operating system are equally effective: there are some ideas that work well, and others that cause problems. In the early 70's UNIX was released, and it quickly gained popularity as a particularly well-designed operating system. The more people who used UNIX, the more software was written to work on UNIX-like systems, eventually leading many other OSs to implement the basic behavior of UNIX systems.

One of the many UNIX-like operating systems to be developed was NeXTSTEP, which eventually became the basis modern version of macOS. But the first UNIX-like operating system that could be downloaded, used for free, and modified by anyone was LINUX.

C. LINUX

LINUX was developed as a personal project of Linus Torvalds while he was a student at the University of Helsinki, as a result of his frustration with the licensing constraints on other UNIX-like OSs. It was released in 1991, and as the first totally free UNIX-like OS it took off quickly.

Today, there are many different versions of LINUX, called LINUX *distributions* or *distros* for short. These all feature the same LINUX kernel (with only slight modifications), but differ in the collections of utilities, drivers, and system libraries they come packaged with.

Places where you will find LINUX include:

- Many personal computers in the form of easy-to-use distros like Ubuntu.
- Most mobile and wearable devices in the form of the Android OS, which is a variant of LINUX.
- Nearly every high-performance computing cluster.
- Nearly every internet server and cloud-computing cluster.

D. Terminals and Shells

Terminals are a completely text-based method of interacting with a computer. Despite the fact that they are one of the oldest ways to use a computer, terminals remain with us today due in part to their extreme simplicity, and in part to the impressive capability of shell interfaces.

A shell is a program that interprets user input in real time. Shells provide a computer language that they interpret in real time, allowing you to describe what you want the computer to do. Essentially, they let you talk to a computer in a way that is easy to learn but also flexible enough to let you do almost anything. In fact, it is very hard to develop a graphical interface that allows you to do everything that can be done with a shell.

Here's a culinary analogy: if you want to get dinner, you can go get food from a resturaunt, or cook food yourself. If you eat out, you save yourself the trouble of learning how to cook, but you are limited by the menu options of the resturaunts in your area. This is frustrating: even if the resturaunts around you have the ingredients and equipment to make some dish, if it doesn't appear on their menu, it isn't something you can order. However, if you put in the effort to learn to cook for yourself, then as long as you have the ingredients, you can put them together in the way you see fit. Similarly, while graphical user interfaces may have an easier learning curve, they are fundamentally limited: by learning a shell language and talking directly to your computer with it, you can transcend those limitations.

II. SHELL CONFIGURATION

Today, we will learn to use a popular shell called `bash`. First, open the terminal application on your machine.

- To test whether your account is configured to the bash shell, in the prompt type

  ```
  echo $SHELL
  ```

  If everything is set up properly you should see

  ```
  /bin/bash
  ```

- By default, the shell will start out in your home directory.
- To list the content of your current directory, type:

  ```
  ls
  ```

  You will see a list of all the files in your directory.

The terminal may appear daunting at first. However, Linux is full of shortcuts and failsafes (you will learn the basics). For example, the bash shell creates a hidden file (we will talk about hidden files later) called `.bash_history` which stores your previous commands so you don't have to retype them. You can access these by hitting the up arrow a few times. Try it now.

III. CREATING AND DELETING DIRECTORIES, AND NAVIGATING

One of the most important aspects in keeping your files organized is to learn how to create directories. This can be done with the `mkdir` command.

First, to keep things organized, let's create a new directory for all the COMPASS workshop exercises we are about to do. In the terminal, type the command

```
mkdir COMPASS2024
```

Then, hit the "Enter" key.

Now, we'd like to move into this directory. To do this, we will need another command: `cd` for "change directory". To change direcotry to be inside the `COMPASS2024` directory we just made, type

```
cd COMPASS2024
```

and hit enter.

Now, lets make a few test directories.

```
mkdir dir1
```

You can create several at the same time by typing a sequence of directory names after `mkdir`

```
mkdir dir2 dir3
```

In addition, we can create a whole sequence of subdirectories all at once by using the -p flag

```
mkdir -p dir4/subdir/subsubdir
```

The -p flag will create not only the directory subsubdir, but also the parent directories subdir and dir3.

Go ahead and use `ls` to see what is in your current directory, you should see all 4 subdirectories.

To delete an empty directory you can type

```
rmdir dir1
```

"rm" for remove, or for multiple directories at once

```
rmdir dir2 dir3
```

Now try to remove `dir4` the same ways

```
rmdir dir4
```

Uh-oh! What happened? If you followed the earlier steps when creating `dir4` you should see an error message

```
rmdir: failed to remove 'dir4/': Directory not empty
```

This is because the `rmdir` command has a very specific function: it is only for removing empty directories. If you attempt to use it to remove anything that isn't an empty directory, it will print out an error message and refuse to do anything.

Let's use `dir4` to practice naviating. To navigate into the directory, type

```
cd dir4/subdir
```

Now, type

```
ls
```

You will see only the `subsubdir` is visible from within this folder. Navigate into that folder with

```
cd subsubdir
```

- One of the most important commands is "pwd", which allows you to find your current location. Type

```
pwd
```

You should see something like

```
/home/USERNAME/COMPASS2024/dir4/subdir/subsubdir
```

where "USERNAME" will be your username, while the part before that could be different on different systems. "pwd" stands for "print working directory".

Now we can see that we've fallen into a hole: how do we get out of this directory? We need to go backwards. There are several ways to do this. One, we could type the full path to wherever we want to go. This is done by starting the path with a `/` to indicate that this path starts at the root of the filesystem, instead of starting from the current location (which is what filepaths begining with nothing imply, along with ones that begin with `./`). For example, if we wanted to get back to COMPASS2024, we could type

```
cd /home/USERNAME/COMPASS2024
```

Instead, we could use a short hand for returning to the directory one level up

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

Yet another way to do this is to first call `cd` with no filepath. This takes you to the user directory, and then you can go back into `COMPASS2024` as before

```
cd
cd COMPASS2024
```

Use `pwd` to make sure you're back in the `COMPASS2024` folder now, as expected.

- One of the most convenient shortcuts of linux is the "tab-completion". Which helps in writing all these long paths. Try again changing into the subsubdir using the absolute path

  ```
  cd /home/USERNAME/COMPASS2024/dir4/subdir/subsubdir
  ```

  but each time you hit a letter hit the "tab" button a couple of times, if the path is not auto-completed enter another letter in the path.

IV. COMMANDS, ARGUMENTS, AND STRINGS

The tools we've used so far: `ls`, `cd`, `mkdir`, and `rmdir`, are all *commands*. Commands are basically little programs that can be run to do very simple things. Each command has zero or more required *arguments* that are pieces of input that tell the command what to do.

For example, `cd`, `mkdir`, and `rmdir` all take a filepath as an argument. In fact, it turns out, `ls` does to: instead of just listing the current directory, you can list the contents of any directory, for example try

```
ls dir4
```

We also saw a special type of argument for `mkdir` with a dash in front of it. This optional argument is called a *flag*, and these typically go before the main arguments and specify settings the command should obey. In our case, the `-p` flag tells mkdir to create intermediate directories wherever it needs to.

An example of a command that doesn't take a filepath as its argument is `echo`. `echo` simply repeates whatever text argument it is given. However, this argument must be given as a *string*. A string is any set of characters you can type, including spaces. These are entered by surrounding them with `""` so that the shell knows where they start and stop. For example,

```
echo "Hello World"
```

prints exactly that message to the Terminal.

V. FILES

Linux is an extensionless system. It doesn't depend on the extension of a file to tell what type of file it is. For example, a pure text file with the extension .txt or .dat or .png or .jpg is treated as a text file in linux.

- Let create our first file using the "touch" command. Type

  ```
  cd /home/USERNAME/COMPASS2024/dir4/subdir/subsubdir
  ```

  and

  ```
  touch foo.txt foo1.txt foo2.txt
  ```

  This will create 3 empty files in the subsubdir directory. Strictly speaking, touch just updates the timestamp of last access of the files. But, since these files never existed they are created by the system.

  Lets also write something in these files. Type

  ```
  echo "First entry into foo file" >> foo.txt
  ```

  ```
  echo "First entry into foo1 file" >> foo1.txt
  ```

  ```
  echo "First entry into foo2 file" >> foo2.txt
  ```

  Hint: Remember to use the up arrow to access your previous commands, so that you don't have to retype everything.

  The "echo" command simply prints the text in the quotes, while the append operator ">>" just appends to the files the output from echo, i.e., the text in the quotes.

- We can now use the "file" command to figure out the file type, i.e., enter

  ```
  file foo.txt
  ```

  and the output should be.

  ```
  foo.txt: ASCII text
  ```

- When choosing file names and directory names, it is more convenient to not use spaces as they require special treatment. A better practice is to use underscores in the place of spaces, i.e.,

  ```
  echo "First entry into foo file" >> my_fist_foo_file.txt
  ```

- Lets now list the contents of the subsubdir, i.e., type

  ```
  ls
  ```

  Much like `mkdir`, it turns out `ls` also has useful options to obtain more information about the files, i.e.,

  ```
  ls -l
  ```

  which will list the files in alphabetical order and show the permissions of the files (no need to get into this for our purposes), the owner and group of the files, the size of the files in bytes (if you use the flags "-lh" it will print the file size in human readable form), the timestamps for the files and the filenames. We can also use a combination of options. A useful one is

  ```
  ls -ltrh
  ```

which lists the files in reverse order time, i.e., the item that was accessed last appears at the bottom of the list.

- Earlier we mentioned that Linux has hidden files. We can list these by using the "-a" or "–all" flag. Go to your home directory using `cd`, and type

  ```
  ls -a
  ```

  You should now see a number of files that start with a ".". All these are hidden files. You can also see your ".bash_history" file (which contains your previous commands), as well as one of the most important files in Linux the ".bashrc". The ".bashrc" file is loaded every time you log into your linux account and can be used to define useful macros, including aliases, set shell variables (like the $SHELL variable we used) and other useful shortcuts.

For most Linux commands there are manual pages which you can access by typing "man" and the command name, e.g.,

```
man ls
```

In the manual pages you can read more about the command and the different options (flags) it has. Of course the world wide web is always your friend.

A. Moving files around

Copying files or moving them is always useful. Create a new directory in your home directory, typing

```
cd ~/COMPASS2024; mkdir dir1
```

Notice that we used "∼/", which is a shortcut to the home directory path. We also used the semicolor ";" operator to separate the cd command from the mkdir command. You can use multiple semicolons on one line to separate multiple commands that you can execute in the order they appear and hit enter only once. Now copy the "my_fist_foo_file.txt" from subsubdir to the dir1 directory. Using the "cp" (copy) command and the paths to the files

```
cp ./dir4/subdir/subsubdir/my_fist_foo_file.txt dir1/
```

The first dot "." is meant to indicate that `dir4` is in your working directory. The command would be equivalent to

```
cp dir4/subdir/subsubdir/my_fist_foo_file.txt dir1/
```

Remember to always use the tab completion to write the paths, so that you don't have to type the full path.

The syntax of the copy command is "cp [source] [destination]". If you are sure you don't want to copy the file, but move it, i.e., no longer have the file in your source you can use the "mv" command, e.g.,

```
mv ./dir4/subdir/subsubdir/foo.txt ./dir4/subdir/
```

Now list the contents of the subsubdir directory

```
ls ./dir4/subdir/subsubdir/
```

and you should see the file you moved. Also, list the contents of subdir

```
ls ./dir4/subdir/
```

and you will see the file was indeed moved. The copy command can be used to copy directories too.

B. Deleting files

It is also possible to destroy files. When doing this, one should be very cautious, since this action is generally not un-doable. Instead of `rmdir`, which only destroys empty directories, we can use the `rm` command to destroy files and directories.

For example, if we create

```
touch test.txt
```

We can destroy it by

```
rm test.txt
```

By using the `-r` flag, we may instead recursively destroy a whole directory tree. Try,

```
rm -r dir1
```

The directory and its file should now be gone.

VI. GREPPING

When debugging we will often need to find certain expressions in a file. Using the linux command grep is one of the best ways to find an expression in a text, e.g.,

```
grep "First" ./dir4/subdir/foo.txt
```

The syntax is "grep expression file". The above will search and print all the occurences of "First" in the file `foo.txt`. (Of course, in this case there is only one line in the file, so it's not much of a test.)

`grep` also has a useful `-r` option that allows it to recursively search through directories and run on each file. This way, not only can you find content in files, you can find files based on what content they contain.

```
grep -r "First" ./
```

VII. TAR ARCHIVES

One particularly useful utility is `tar`. This program allows you to create *arcives*: collections of files and folders that can be saved as a single file, and extracted later.

The `tar` command has many options, I won't go through all of them here. But, just to get started, here's how you would turn `dir4` into a `tar` archive.

```
tar -cf dir4_archive.tar dir4
```

The `-c` flag tells `tar` that it will be creating a new archive, and the `-f` flag tells it that it will accept a filename for the new archive next. Then the filename is provided, followed by the file(s) or folder(s) to be archived. Whenever a folder is lised, it is assumed that all its contents will also be included in the archive.

To test that this worked, we need to move the archive somewhere else, so that it will not just overwrite the original files when unpacked. Do the following

```
mkdir test
mv dir4_archive.tar test/
cd test
ls
```

You should see that this folder only contains the archive file. Now, extract the archive by executing

```
tar -xf dir4_archive.tar
```

Where the `-x` option tells `tar` that it is time to extract the archive, and the target folder argument is no longer needed.

After this is done, running `ls` will reveal that a copy of `dir4` has appeard. By inspecting it, you can confirm that all the contents of `dir4` have appeared as well.

VIII. MORE USEFUL COMMANDS

So far, we have mostly made folders and a few files, and navigated around, and did some searching. But there are more things we can do. Below are several more useful commands.

To see how much disk space a folder or file uses, use

```
du ./
```

To make the output shorter, use the `-s` option to only show the summary of total disk usage for everything. And, the `-h` option can be used to write the disk usage in human-readable units like Kb, Mb, and Gb, rather than in bytes.

```
du -sh ./
```

To tell of two files differ, the `diff` utility can be used:

```
diff dir4/subdir/subsubdir/foo1.txt dir4/subdir/subsubdir/foo2.txt
```

will show a difference, whereas

```
diff dir4/subdir/foo.txt dir4/subdir/subsubdir/my_fist_foo_file.txt
```

will not.

To see how many words a file contains, use `wc`

```
wc -w dir4/subdir/foo.txt
```

Or use the `-l` option to count lines, or the `-c` option to count characters (a.k.a bytes).

To launch an interactive calculator program, call

```
bc -l
```

This will allow you to type expressions into a calculator and get numerical answers. To quit, type "quit".

To print a list of the currently running processes in your shell, use

```
ps
```

To download things from the internet, you can use the `curl` utility

```
curl --output comic.png https://imgs.xkcd.com/comics/tar_2x.png
```

IX. COMBINING COMMANDS

As fun as this is, the real power of using these command line utilities comes from stringing them together.

For example, lets say we wanted to know how many lines the word "First" appears in for all the files in our directory. We can take the output of `grep`, which lists all the occurances, and use `wc -l` to count how many lines there are.

To do this, we use something called *piping* to send the output from one command into the first input of another command, as if that output were a file.

This is done with the | symbol, like this

```
grep -r "First" dir4 | wc -l
```

Another thing we can do is use commands that are designed to take other commands in their arguments.

For example, lets say we are waiting for a process to stop running, and we'd like to monitor it. We can use a command called `watch` (Note: unfortunately, watch is not installed by default on macOS) to repeatedly run `ps` every few seconds so that we can keep an eye on it. Since `watch` is set up to take a command in its input, we just do

```
watch -n1 ps
```

Combining commands unlocks much of the power of the shell interface, and leads eventually to shell scripting, which is a topic we will cover tomorrow.

X. WILDCARDS

So far, we have written out filepaths explicitly. However, there are many times when you may want to write a long list of files that follow some pattern, without having to do so manually. For situations like this, we can use *wildcards*. Wildcards are characters in a filepath that essentially mean "anything goes here"

For example, go to the following directory

```
cd /home/USERNAME/COMPASS2024/dir4/subdir/subsubdir
```

In this directory, we could type `ls` to list all the files. But, we could also just list the files that obey the pattern "foo¡number¿.txt". To do this, we use the `?` wildcard, which means any character can go there

```
ls foo?.txt
```

There is also the `*` wildcard, which stands for "any string goes here". This can substitute a string of any length.

For example, we could list all the .txt files

```
ls *.txt
```

Which, of course, are all the files in the directory in this case.

It is also possible to use brackets to specify specific character ranges, for example

```
ls */out_[2-5].pdb
```

Give this a try to list out only certain types of files on your home directory!

XI. READING FILES

Linux has a number of tools to read files

- Cat <file>

  cat stands for concatenate. It will print the entire file in one go on your screen. Good for files that are only a few lines long.

- less <file>

  It will print the file in pages that you can browse through using the up/down arrows or the up/down page keys. Type "Q" to exit.

- head -nA <file>

  Here A is a integer number. The command print on screen the first A lines of the file

- tail -nA <file>

  Here A is a integer number. The command print the last A lines of the file.

XII. MORE CHALLENGES!!

Have you gotten this far! Great! In that case, it's a good idea to keep practicing. At the following link you will find a GitHub repository with extra fun challenge exercises on it. Download the files and then follow the instructions in the README.txt" `https://github.com/mssalvatore/command-line-challenges`

Have fun!

XIII. EDITORS

One of the most important tools in building our programs are text editors. Almost all Linux distributions these days have vi, vim and the emacs editors. Here we have written a quick guide to vi and vim, and emacs, so that you can get started using either.

A. Vi/vim

First of all, you need to check what is installed on your system. You might have vim, or just vi. Fortunately, the usage of these two is almost the same.

To check what is set up, we can use the `which` command.

```
which vim
```

If you see a filepath to the `vim` executable, then you can use `vim`. If not, you will probably have to use `vi` instead, in place of most of the commands below. There also may be slight differences, which we will handle if they arise.

To launch `vim` call it with the path to a file you want to edit or create

```
vim my_test_file.txt
```

You will see your terminal be taken over by the `vim` text editor. This editor is entirely text-based: it runs completely within the terminal interface and doesn't require any graphical elements or interface to be set up. Because of this, `vim` and `vi` are installed on almost every system, and are a frequent go-to when trying to read and edit files over terminal and ssh interaces, when you don't have a chance to set up a more complicated and full-featured editor.

It would be nice to populate this blank file with some text. But you may notice: your keys aren't doing what you expect! The reason is that, in the default mode, keys in vim are reserved for doing commands, such as to navigate the file, select text, and change modes.

In order to edit the file, we need to activate the *insert* text mode, which we can do by pressing the `i` key.

Once in insert mode, you will be able to type normally. Go ahead and type a long sentence, such as,

```
Speak the speech, I pray you, as I pronounc'd it to you, trippingly on the tongue.
But if you mouth it, as many of our players do, I had as live the town crier spoke my lines.
Nor do not saw the air too much with your hand, thus, but use all gently; for in the very torrent, tempest,
O, it offends me to the soul to hear a robustious periwig-pated fellow tear a passion to tatters, to very r
I would have such a fellow whipp'd for o'erdoing Termagant.
It out-herods Herod. Pray you avoid it.
```

The downside of insert mode is that you can't run any commands! To exit insert mode, hit the `ESC` key.

Now we have a good amount of text on our file, we can practice navigating around. To move, use the arrow keys as you normally would. You'll see the cursor moving around the text.

There are also navigational shortcuts: press `$` to jump to the end of a line, and `^` to jump to the beginning.

You can also use `w` to jump to the next word, and `b` to jump to the previous.

To go to the very end of the file, you can press capital G `shift+G`.

You can also jump to a specific line with `:<line number>`

To select text, you can press `v` to enter visual mode: here, moving your cursor will select text just like a normal text edtior.

To change the text that is selected, you can press `c`, which will take you into insert mode again.

Finally, you can save by pressing `:w` to "write" the file.

To quit `vim` you will use `:q`. If you try to quit without saving, you'll get an error. To quit anyway, use `:q!`.

I have to admit, this is only the tip of the iceburg for what vim can do. To access documentation, you can use the `:help`. The real power of vim is in the ability of stringing commands together. We may cover more on this on friday.

B. emacs

```
emacs my_fist_foo_file.txt
```

You can now start to edit this file. Add a few sentences to it.

- To move fast between the text you can use the "pg up" and pg dn" keys, the "home" and "end" keys, you can keep the "ctrl" key pressed down while using the left, right, up and down arrows to jump between words and text. To move a page down you can also type

  ```
  ctrl+v
  ```

  and to move up

```
Meta+v
```

where the “Meta” key is typically the “alt” key (the key left of the space bar).

To move to the beginning of a line you can also type

```
ctrl+a
```

and to the end of a line you can also type

```
ctrl+e
```

These shortcuts will expedite greatly how quickly you move the cursor through the text. If you want to go directly to the end of the text (buffer) type

```
Meta+<
```

and the beginning of the text (buffer)

```
Meta+>
```

- To save what you wrote type

  ```
  ctrl+x+s
  ```

- To close emacs type

  ```
  ctrl+x+c
  ```

- Copying and pasting inside a file is always useful. Open the file with emacs again

  ```
  emacs my_fist_foo_file.txt
  ```

- Move the cursor to the beginning of the text you would like to copy/cut and hit

  ```
  ctrl+space
  ```

- This activates the marking area tool. Use the arrows to highlight the area you want to cut/copy. To cut the text type

  ```
  ctrl+w
  ```

- To copy the text type

  ```
  Meta+w
  ```

- To paste the text, move the cursor where you want to paste the text and type

  ```
  ctrl+y
  ```

  where “y” stands for yank.

- To remove (kill) an entire line in your text you can type

  ```
  ctrl+k
  ```

- To undo a move type

  ```
  ctrl+_
  ```

  In other words

  ```
  ctrl+shift+-
  ```

  To redo a move type

  ```
  ctrl+shift+space bar+-
  ```

  While these may sound like many commands, keep in mind that practice makes perfect!