The Linux Minimum: A Hopefully Comprehensive yet Non-overwhelming Guide to Linux

Paul Wang

Contents

1 Why Linux? 5

2 So You Want to Code Along With This Guide? 7

3 Life In Command Line: Directories 10

4 Files 14

5 Command Flags 22

6 Absolute and Relative Paths 29

7 Black Magic: Makefiles 31

8 Black Magic 2: Using Python Script 43

9 Closing 50

List of important Linux commands:

- `pwd`: print working directory

- `ls`: list contents of working directory
  - `ls BLAH`: list contents of directory `BLAH`
  - `ls -a`: list all contents of working directory, including these hidden

- `mkdir BLAH`: create a directory called BLAH

- `cd BLAH`: change working directory to `BLAH`
  - `cd ..`: go to the parent directory of the working directory
  - `cd ~`: go to home directory

- `rm filename`: delete the file `filename`
  - `rm -r directory`: delete the directory `directory`

- `mv file1 DIRECT`: move file `file1` to directory `DIRECT`
- `mv file1 file2`: rename file `file1` to `file2`
- `mv DIRECT1 DIRECT2`: move directory `DIRECT1` to directory `DIRECT2`. If there's no directory called `DIRECT2` in the current working directory, then rename directory `DIRECT1` to `DIRECT2`

- `cp file1 file2`: duplicate file `file1` to file `file2`
- `cp -r DIRECT1 DIRECT2`: duplicate directory `DIRECT1` to directory `DIRECT2`

- `python3`: start a python3 shell
  - `quit()` from the python shell will quit it and put you back in the directory where you `python3`'ed.

- `python3 blah.py`: run the python script `blah.py`

- `gcc hw.c -o hw.x`: compile the C file `hw.c` and name the output executable `hw.x`

- without the `-o hw.x` explicit naming, the output executable will be named `a.out`

- `./a.out` runs the executable `a.out`

- (To the savants) other compilers: `clang`, `g++`, `clang++` are used in the same fashion

- `vi filename`: open the file `filename`. If no file with name `filename` exists, create one

- `cat filename`: display contents of file `filename`

- `less filename`: display contents of file `filename` on a clean page

- to quit from `less`: hit `q` on keyboard

- to go to line 176: `176+G`, or `176+shift+g`

-to copy content from Linux shell to local clipboard: select the text you want to copy, then

Windows: left click mouse

Mac: `command+c`

-to paste content from local clipboard to Linux shell:

Windows: right click mouse

Mac: `command+v`

-go up a command: hit the up key on keyboard

-go down a command: hit the down key on keyboard

-write command line output to a file: `COMMAND > FILENAME`

List of important vi commands:

-hitting `i` key on keyboard: enter insert mode
-hitting `esc` on keyboard: enter edit mode
- `:q`: quit vi
- `:w`: save
- `:wq`: save and quit
- `:q!`: force quit vi, even if there are unsaved changes
- `:u`: undo
- `x`: delete the current character on cursor
- `dd`: delete the current line on cursor

- paste from local computer's clipboard to vi:
  Windows: in insert mode, `shift+ins`
  Mac: in insert mode, `command+v`
- copy from vi to vi: go to edit mode, select the text you want to copy, press `y`, put cursor at pasting position, then `P` or `shift+p`
- cut from vi to vi: go to edit mode, select the text you want to cut, press `d`,put cursor at pasting position, then `P` or `shift+p`
- copy from vi to local clipboard: this is actually quite tricky, but there's a workaround method. Use `less` or `cat` to display the file on command line, then use the method for copying from command line (left click for windows, command+c for mac)

-go to line 176: in edit mode, `176+G`, or `176+shift+g`

-search for a string in file: in edit mode, do `/`, then enter the string you want to search for, then hit enter. To go to the next instance of occurrence: `n`; to go to the previous instance of occurrence: `N`

1 Why Linux?

There's no way around Linux for those who wish to do technological or scientific work these days.

The primary reason is that when working in a group, whether it's a company or a research group, most of the work you need to do can't be done on your personal local computers, sometimes for confidentiality reasons, and sometimes simply because the computation is so large that a local computer cannot handle it. For example, a typical set of test run for a compiler can take an entire afternoon, while a typical computation in quantum mechanics can easily take 20 hours. If you choose to do that on your local computer, that means you can't use it for the next 20 hours while the job is running. Big time inconvenience!

Therefore, most, if not all groups have supercomputers that can be remotely logged into. Users can log into these supercomputers from their local laptop, and do work on them. These supercomputers are powerful so they can host multiple users simultaneously. More importantly, they are there to do one job: to do the computations required by users. They don't have to carter to everyday needs (e.g. Youtube, Netflix, Reddit, Steam) like personal computers, so it is ok if they spend 20 hours on some computation. Most of the time they are known as "servers".

Because servers don't have to cater to everyday needs, most of them don't have graphical interfaces. This means they don't have a nice graphical "desktop" where the files and folders are displayed with little icons and you can click on those icons to open things. When you log onto a server like this, your mouse will be next to useless (it can still do a few things, as we will see). Really, you have logged onto a different style of operating system: the Linux command line.

(To the savants: I will make no effort in distinguishing between Unix and Linux.)

It's simply a different operating system than the usual graphical one. The

files and folders are still there, but you access them, edit them, and use them in a different way. In other words, you "operate" on them in a different way.

Logging onto such a server can be overwhelming at first. It has always confused me why there is no course that teaches Linux when Linux skills are basically needed by every group in industry and academy. I write this guide from my experience. Hopefully it can get you up to speed with Linux.

One last thing before we dive in: from now on what layman call "folder" will be called "directory".

2 So You Want to Code Along With This Guide?

If you are a mac user, you can actually operate within your own personal local computer with Linux. **However, until you know what you are doing, do not attempt this!** Files deleted from Linux command line are irreversible and cannot be recovered. For example, the command

```
rm -rf *
```

will irreversibly delete everything in the current directory. If you don't know Linux, the above line might be accidentally typed (by just wiping your keyboard, for example), and you'll be screwed.

If you want to code along with this guide, you can use the remote access for UofT's Engineering Computing Facility (ECF):
`https://undergrad.engineering.utoronto.ca/undergrad-resources/engineering-computing-facility-ecf/remote-access/`

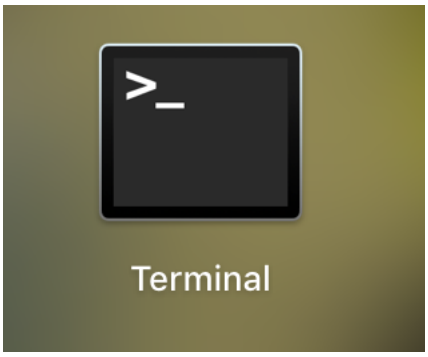
Of course, to log into such a server, you need to be a UofT engineering student, which luckily you all should be. If you are not, you can always try a free online Linux shell simulator, like
`https://bellard.org/jslinux/`

There's a slight technicality: **ECF account is not the same as your regular UTORid account!** All engineering students need to activate their ECF account and set an ECF password before losing their remote access virginity:
`https://undergrad.engineering.utoronto.ca/undergrad-resources/engineering-computing-facility-ecf/ecf-account-activation/`

You don't have to use the same password as your UTORid password for ECF password.

Once you have the account activated, you can now log onto ECF servers! The way to do this is different for mac and windows.

For mac:
1. Open **Terminal** on your local mac laptop



2. Type in the command

```
ssh <YOUR_UTORID>@remote.ecf.utoronto.ca
```

For example, if your UTORid is "blah123", you would do

```
ssh blah123@remote.ecf.utoronto.ca
```

You might get a warning saying "Warning: Permanently added the RSA host key for IP address '128.100.8.48' to the list of known hosts". You can safely ignore it.
(To the savants: "ssh" stands for "secure shell" and is used for remote logins.)

3. Enter your ECF password. Again, this should be the ECF password you just set up, not your usual UTORid password. Note that the password will not be displayed on your Terminal screen. Not even stars will be displayed. This is normal behavior, don't worry.

4. A bunch of unimportant messages will be displayed. Below that you'll see a line that goes like

```
[<YOUR_UTORID>@remote ~]$
```

You're now in.

For windows:
1. Download Putty
`https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`

2. Follow the instructions under **Via SSH to remote.ecf.utoronto.ca** on
`https://undergrad.engineering.utoronto.ca/undergrad-resources/engineering-computing-facility-ecf/remote-access/`

You can ignore everything that says something about "VNC" on this page.

After you enter the address in Putty, hit enter (or click "open"). A command line window will pop up. Follow the instructions to enter your UTORid and ECF password. Again, this should be the ECF password you just set up, not your usual UTORid password. Note that the password will not be displayed on your putty terminal screen. Not even stars will be displayed. This is normal behavior, don't worry.

3. A bunch of unimportant messages will be displayed. Below that you'll see a line that goes like

```
[<YOUR_UTORID>@remote ~]$
```

You're now in. Note that Putty might use a colon (:) or pound (#) instead of a dollar sign ($) for the previous line, and there might not be square brackets around.

3 Life In Command Line: Directories

The Linux command line revolves around files and directories. There is no analog to the idea of an "application" in the usual everyday graphical operating system.

The filing system on any computer is a tree. There is a root directory at the head. The root directory contains subdirectories and files, each of the subdirectories can contain their own subdirectories and files, and so on.

In Linux, **you are always in some directory**. There's no such thing as "I'm on the desktop" or "I'm in some application". The directory you are currently in is called the **working directory**. The command to see your current working directory is `pwd` (for "print working directory").

When you log onto ECF (or any Linux server), you always log on to your home directory. Let's see what it is!

```
[blah123@remote ~]$ pwd
/u/d/blah123
[blah123@remote ~]$
```

Note: lines starting with `[blah123@remote ~ ]$` are the commands you type, i.e. inputs. You type commands after the dollar sign. Lines that don't start with this are Linux's response to your commands, i.e. outputs. After typing a command, hit the enter key to execute it. After the outputs of a command, Linux will generate a new line that awaits an input command.

You have logged on to your home directory, which is `/u/d/blah123`. This chain of strings means your home directory `blah123` is a subdirectory of `d`, which is in turn a subdirectory of `u`.

There's a shortcut character for your home directory: `~`. Keep this in mind, we'll circle back to it later.

Of course you will want to see the contents of your working directory. To do this the command is `ls` (for "list"):

```
[blah123@remote ~]$ ls
[blah123@remote ~]$
```

If this is your first time logging into your remote ECF server, you won't have anything in your home directory. Since there's nothing in your home directory, there will be nothing to list, so there's no output. Hence, Linux directly goes to the next line where it awaits an input command.

Depending on how ECF purges and stores things you might see a directory `OLDFILES`.

```
[blah123@remote ~]$ ls
OLDFILES
[blah123@remote ~]$
```

That directory is not important for us today.

We can create our own directories. The command is `mkdir`, for "make directory". Straightforward enough.

```
[blah123@remote ~]$ mkdir hw
[blah123@remote ~]$ ls
OLDFILES hw
[blah123@remote ~]$
```

You might have noticed that this time the command has an argument called `hw`, whereas the previous ones `pwd` and `ls` were standalone. The idea of arguments is the same as that for functions: some commands can take in arguments, others don't need arguments. The specifics depend on the command you're using.

Different arguments are separated by spaces. The arguments and the command itself is also separated by spaces. This is why when naming things, you want to avoid spaces. For example, you want to create a directory called "hello world". You would think `mkdir hello world` would do the job, but this command actually makes **two** directories, namely `hello` and `world`. This is because `hello` and `world` are separated by spaces and hence are treated as two individual arguments to `mkdir`. I would suggest you do

`mkdir hello_world` on such occasions, i.e. use underscores when you want spaces in names of things.

The list command `ls` can also take in arguments. `ls BLAH` will list contents of directory `BLAH`. We'll see this in action later on.

We want to enter the directory we've just created. The command is `cd`, for "change directory".

```
[blah123@remote ~]$ mkdir hw
[blah123@remote ~]$ ls
OLDFILES  hw
[blah123@remote ~]$ cd hw
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
[blah123@remote hw]$
```

As you can see, new directories are empty when they're created.

Also you can see that the little tilde $\sim$ before the dollar sign has changed to `hw`. We know that the stuff after the dollar sign is the command we type. The things before the dollar sign actually tells you your login ID and your current working directory. Remember $\sim$ is the shortcut character for your home directory? That's why there's a $\sim$ when you firsts log into ECF, since you log into your home directory.

To go back to the parent directory, do `cd ..`

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
[blah123@remote hw]$ cd ..
[blah123@remote ~]$ pwd
/u/d/blah123
[blah123@remote ~]$
```

`..` is the shortcut character for the parent directory.

There's another shortcut character: `.` is the shortcut character for the current working directory.

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
[blah123@remote hw]$ cd ..
[blah123@remote ~]$ pwd
/u/d/blah123
[blah123@remote ~]$ ls
OLDFILES  hw
[blah123@remote ~]$ ls hw
[blah123@remote ~]$ cd hw
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ cd .
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ cd ~
[blah123@remote ~]$ pwd
/u/d/blah123
```

You might think that a shortcut character for the current directory is kind of pointless, but it will have its uses.

4 Files

The most important difference between Linux operating system and the usual graphical operating system is probably that in Linux files are not necessarily typed. In the usual graphical operating system all files have a suffix (.docx, .pdf, .txt, .jpg, .tex, .py, .c, and so on), and different suffixes represent files that have different formats. Files of different formats needs to be opened and edited with different applications (Word, Excel, Preview, Adobe Viewer, and so on).

There is no such distinction between files in Linux. All files are treated as plain text (i.e. .txt) and edited by the same editor.

Not only that the files don't have types, you also cannot associate types to them even if you want to. Before going there, however, we need to learn how to create files.

All files are edited by the same editor, called `vi`. (To the savants: yes, I use vi instead of vim. Don't turn me in!) To create a file named, say `some_text`, we do this:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ vi some_text
```

At this point the command line window will jump away from the main Linux shell and enter this new interface that looks empty, and on the bottom line says `"some_text" [New File]`. We're now in the vi editor and have opened up the newly created file `some_text`. Notice that the filename does not require a suffix!

The vi editor has 2 modes: **edit mode** and **insert mode** (to the savants: I'm ignoring visual mode as it is not essential).

Insert mode is analogous to the familiar word/pages/googledocs: you type in the content of the file in this mode. However, when you enter vi you always land in **edit mode**. Basically, insert mode is for typing in file contents only. All other actions like saving, undoing, copy-pasting and cut-pasting are done in edit mode.

The bottom line of the vi interface shows you your "vi commands". All vi commands start with a colon (:). When in edit mode, you cannot insert contents into the file body, and every vi command you type will be displayed on this bottom line. On page 4 there's a list of frequently used vi commands.

By default vi does not display the line numbers. Let's type in our first vi command: `:set nu`. See what it did? Also notice that when you type in `:set nu`, it became displayed on the bottom line. The bottom line in edit mode will not just display everything you typed in. It only starts displaying if you have typed a colon first. Also, if you typed in something that's not a vi command, vi will bark at you. Try typing in `:setnu` and see what happens.

Also type in this command right now: `:set mouse=a`. This will enable you to use your mouse to control the cursor and scroll up and down in vi. Without this vi feels extremely cumbersome.

We want to insert some content into the file! To go to insert mode from edit mode, hit the `i` key on your keyboard. The bottom line will say `--INSERT--`, reminding you that you are in insert mode now. We can now type in whatever content we want, as plain text!

To quit from insert mode to edit mode, hit the `esc` key on your keyboard.

Remember that saving a file is done in edit mode? The command to do this is `:w`, w for "write". Now that the file is saved, we can quit from the vi editor with `:q`, q for "quit". Quitting the vi editor will drop us back to the Linux command line shell. There's also `:wq` and `:q!`. Check page 4 for

what they do.

You can open the file you've just created also with `vi some_text`. The `vi` command creates a file if there's no file with the name indicated by its argument, and opens the file if there is.

You might have noticed that on page 4, there are two commands that do not start with a colon: `x` and `dd`. The thing is they are not commands per se, but they are also used in edit mode, so I listed them there. In fact, all of these little hotkeys are used in edit mode, for if you type these hotkeys in insert mode you simply type them into the content of the file instead of performing their functionality. Try `x` and `dd` out! There are also some copy-pasting tricks for vi on page 4. Play with them if you want. Don't worry about `cat` and `less`, we'll get there later.

Exercise: create a file in your home directory named `.vimrc` (strictly follow this name, even the dot!) with the following content:

```
:set mouse=a
:set nu
```

Solution:

Step 1: `cd ~`
Step 2: `vi .vimrc`
Step 3: hit `i` to go into insert mode
Step 4: Type in the contents
Step 5: hit `esc` to go into edit mode
Step 6: `:wq`

Now, go back to your directory where you stored `some_text` and open the file. What do you notice?

I will not go into how this change happened. But now, every time you open vi on a file the line numbers will be there and you can use your mouse on default. This trick works as long as this `.vimrc` file is there in the home directory. Since logging out of and relogging back into ECF server won't make you lose files, you've just set up your vi for eternity.

If you try `ls` your home directory now, you won't actually see the file `.vimrc`. Don't worry, it's there. I'll talk about what's going on here at a later time.

Let's get back to the idea of typeless files. Create a file named `hw.c` in the `hw` directory with the following contents:

```
def hw():
    print("Hello world!")

if __name__ == "__main__":
    hw()
```

And do this:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
some_text hw.c
[blah123@remote hw]$ python3 hw.c
Hello world!
[blah123@remote hw]$
```

What just happened? Did we just run python on a C file?

First things first: yes, `python3 file` is the command that runs python3 on `file`. You can use this command without arguments to start a python shell, i.e. `python3`. To quit from that shell to Linux, `quit()`.

What happened is that the `.c` file isn't really a C file. As we know, all files are treated as plain text in Linux. The file `hw.c` is simply a text file with the name "h-w-dot-c"!

When you run this file with the `python3` command, it interprets this plain text file as python. If what you've written is legal python, then good; if not, then you'll see some python error messages. For example, let's create another file named `hw.py` with the following contents:

```
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
```

and run `python3 hw.py`. Although the file ends in `.py`, you will still see python barking at you!

However, it is extremely bad practice to name python files `.c` and C files `.py`. Let's take this opportunity to learn how to delete and rename things. Let's delete the python file (ends in `.c` right now) and rename the C file from `.py` to `.c`.

The delete command is `rm`, for "remove". The command to rename is `mv`,

for “move”. They’re really simple to use:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
some_text hw.c hw.py
[blah123@remote hw]$ rm hw.c
[blah123@remote hw]$ ls
some_text hw.py
[blah123@remote hw]$ mv hw.py hw.c
[blah123@remote hw]$ ls
some_text hw.c
[blah123@remote hw]$
```

(A trick: you don’t actually have to type out the entire file/directory name on the command line. You can use tab for auto-complete! For example, if at line 8 you do `mv h` and then hit tab, Linux will auto-complete it to `mv hw.py` for you.)

The `mv` command is actually intended for moving files and directories around. Why it can be used to rename things is a different story and I’ll not get into it, but just know that it can. For how to use `mv` to move things around, check the important Linux command list on page 2. Here’s an example:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
some_text hw.c
[blah123@remote hw]$ mkdir test
[blah123@remote hw]$ ls
some_text hw.c test
[blah123@remote hw]$ mv hw.c test
[blah123@remote hw]$ ls
some_text test
[blah123@remote hw]$ ls test
hw.c
[blah123@remote hw]$ mv test/hw.c .
[blah123@remote hw]$ ls
some_text hw.c test
[blah123@remote hw]$ ls test
[blah123@remote hw]$
```

Here we can see some use of `.` as the character for current directory.

We can inspect the contents of a file from the command line without needing to going into vi. The command is `cat`:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
some_text hw.c test
[blah123@remote hw]$ cat hw.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote hw]$
```

Sometimes the file is very large and outputing its contents to the command line will overflood the screen. In such cases we can use `less hw.c`. It opens up a new vi-like interface and displays the file content there. To go to a certain line number in that interface, enter the line number on your keyboard, and then do `shift+G` (or capital `G`). To quit from that `less` interface to the main Linux command line, hit `q` on your keyboard. To search for a string in the file: do `/`, then enter the string you want to search for, then hit enter. To go to the next instance of occurrence: `n`; to go to the previous instance of occurrence: `N`

Going to some line number and searching for a string in file in vi uses the same trick (do this in edit mode).

It's now time to compile some C. The command is `gcc`. (To the savants: `clang`, `g++`, `clang++` are also installed on ECF servers.)

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
some_text hw.c test
[blah123@remote hw]$ cat hw.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote hw]$ gcc hw.c
[blah123@remote hw]$ ls
a.out some_text hw.c test
[blah123@remote hw]$ ./a.out
Hello world!
[blah123@remote hw]$
```

By default the output executable of `gcc` will be named `a.out`. To run it, do `./a.out`.

We can rename the output using `mv`, but it's a little awkward. Wouldn't it be nice if there's a way to specify the name of the output executable in the compile command? That's what command flags are for.

5 Command Flags

We already know commands can have arguments. That's nice and dandy, but commands can also have something called **flags**. All command flags start with a dash (`-`). Command arguments control the target of the command. Command flags change slightly what the command does.

`gcc` has many, many command flags, but the most used one is `-o` (o for "output"). The flag itself takes in an argument that controls the name of `gcc`'s output.

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
a.out some_text hw.c test
[blah123@remote hw]$ rm a.out
[blah123@remote hw]$ cat hw.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote hw]$ gcc hw.c -o hw.x
[blah123@remote hw]$ ls
some_text hw.c hw.x test
[blah123@remote hw]$ ./hw.x
Hello world!
[blah123@remote hw]$
```

(I usually name gcc executables by the `.x` suffix. It's not a real suffix in terms of the operating system, but a nice reminder to myself.)

You can place flags (and their arguments) anywhere in a command and the effect will be the same. In other words, `gcc hw.c -o hw.x` and `gcc -o hw.x hw.c` will do exactly the same things. Note that `-o hw.x` needs to be moved around as a whole, since that is a flag and an argument to that flag.

Not all flags need arguments. For example, many commands have the `-r` flag, r for "recursion". Deleting a directory needs this flag:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
some_text hw.c test
[blah123@remote hw]$ rm test
rm: cannot remove ‘test’: Is a directory
[blah123@remote hw]$ rm -r test
[blah123@remote hw]$ ls
some_text hw.c
[blah123@remote hw]$
```

(To the savants: why recursion? Files are organized in a tree structure. Well, to delete a directory, you need to delete everything in it, which means you need to visit all of its child nodes, and all the child nodes of these child nodes, and so on. To traverse a tree given its root node as the argument, recursion is the best way. Remember in-order, pre-order and post-order traversals?)

On page 2 in the list of common Linux commands I've listed another command called `cp`, for "copy". I listed its base form and its `-r` form. Let's play with it:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
some_text hw.c
[blah123@remote hw]$ mkdir direct1
[blah123@remote hw]$ ls
direct1 some_text hw.c
[blah123@remote hw]$ cp hw.c hww.c
[blah123@remote hw]$ ls
direct1  hw.c  hww.c  some_text
[blah123@remote hw]$ cat hww.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote hw]$ cp hw.c direct1/hw1.c
[blah123@remote hw]$ ls direct1
hw1.c
[blah123@remote hw]$ cat direct1/hw1.c
```

```
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote hw]$ cp direct1 direct2
cp: -r not specified; omitting directory ‘direct1’
[blah123@remote hw]$ cp -r direct1 direct2
[blah123@remote hw]$ ls
direct1  direct2  hw.c  hww.c  some_text
[blah123@remote hw]$ ls direct2
hw1.c
[blah123@remote hw]$ cat direct2/hw1.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote hw]$
```

To see what flags a command can have, you can of course google, and you can also do `man COMMAND`, `man` for "manual". Try doing `man ls`. It opens up a `less` style page for you. Again, use `number+G` to go to line `number`, `/` for searching a string, and `q` for quitting the `less` style page.

In the man page for `ls` you can see a flag `-a`, standing for "all". It says `do not ignore entries starting with .` , which brings us to our next topic: hidden entries.

Go to your home directory and list all, including hidden entries. You should see something like this:

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ cd ~
[blah123@remote ~]$ pwd
/u/d/blah123
[blah123@remote ~]$ ls
OLDFILES hw
[blah123@remote ~]$ ls -a
.        ..         .cshrc      .lesshst
.python_history   .esd_auth  .login    .vim
.bash_history     .hushlogin-all-message
.logout           .viminfo   .config   .vimrc
.hushlogin-engsoc .msgsrc
OLDFILES   hw
[blah123@remote ~]$
```

Oh boy, that's a lot of things! Our `.vimrc` file that configured the vi line number and mouse activation is also hidden here! In Linux, if you name a file or a directory to start with a dot (`.`), it will be hidden under normal `ls`. This is for a good reason: these files are usually system files that shouldn't be changed by users unless they know what they're doing, so Linux hides their existence from the average user altogether. All operating systems do this actually. If you have a mac, start a new terminal shell and do `ls -a ~`. You will be surprised at how many hidden system files there are.

There are two familiars among these hidden files and directories: `.`, the current directory character, and `..`, the parent directory character. They're there in every directory, since, well, every directory has a "current directory" and a "parent directory". This is the reason why these characters have dots starting, so that when you `ls` normally, they won't show up and pollute your screen view.

The last thing I want to mention regarding flags is that you can clump flags together. For example, `rm` has two flags `-r` and `-f`. We already know what `-r` does. `-f` stands for "force". On some servers when you `rm`, the server will prompt you a nice message basically asking you "are you sure?" It can be quite tiring to answer to that query every time. `-f` will silence this message and go straight to the deletion.

Now, you can clump these two flags into one: `-rf`. This clumped flag will perform the functionality of both `-r` and `-f`. Since the order of flags don't matter, you can also clump them as `-fr`

There's a quick way to operate on all files with some common elements in their names: use the `*` symbol. This is another reason why it is good practice to name your files with proper suffixes. The following example shows how this works. There are 4 `.c` files, one `.py` file, and one subdirectory in a directory called `ayayaya`.

```
[blah123@remote ayayaya]$ ls
hw.c hw1.c hw2.c hw3.c barney.py subdirect
[blah123@remote ayayaya]$ rm *.c
[blah123@remote ayayaya]$ ls
barney.py subdirect
[blah123@remote ayayaya]$
```

The `*` basically reads as "anything", so `rm *.c` will delete all files with the form `<anything>.c`, which are the 4 `.c` files.

What about `rm *`? Well, it will delete all files whose names are of the form `<anything>`, which is all of them. It won't delete directories because this is a standalone `rm`, but `rm -rf *` will. Now you see why this command deletes everything in the current directory.

I will now tell you an interesting pitfall. Try to see whether you can make sense of what's happening.

DO NOT CODE ALONG WITH THIS EXAMPLE!
DO NOT CODE ALONG WITH THIS EXAMPLE!!!
DO NOT CODE ALONG WITH THIS EXAMPLE!!!!!!!!

If you are a mac user, then you know that when you take a screenshot, a `.png` image will be stored with the name `Screen Shot 2021-09-25 at 9.26.16 PM`. Of course, the later part of the string is the timestamp.

Say you have stored a bunch of these screenshots on your desktop. In the filing system this means these `.png` files are in the `Desktop` directory. You want to delete these screenshots, but since there're so many of them, deleting them one by one would take forever. Knowing what you know about Linux, you open up mac's Terminal, go to the `Desktop` directory, and entered

```
[my_macbook: Desktop]$ ls
Blizzard
Clubs
Food
Games
movies
Podcasts
Screen Shot 2021-09-25 at 9.26.16 PM
Screen Shot 2021-09-25 at 9.26.17 PM
Screen Shot 2021-09-25 at 9.26.19 PM
Screen Shot 2021-09-25 at 9.26.22 PM
Screen Shot 2021-09-25 at 9.26.59 PM
Screen Shot 2021-09-25 at 9.28.09 PM
Screen Shot 2021-09-25 at 9.32.14 PM
Screen Shot 2021-09-25 at 10.00.46 PM
Studying
xcode
[my_macbook: Desktop]$ rm -rf Screen *
```

You rationale is simple: all of these screenshot files are named in the form `Screen <something>`, so if I use a `*` to represent `<something>`, all the screenshot files should be deleted.

Then, to your horror, you started realizing that everything on your desktop is starting to disappear! The command is deleting every file and every directory on your desktop!!

You press `control+c` to kill the process (just like when Python is stuck, you can use `control+c` to kill a running command), so some of your desktop is kept, but there's still a lot of recovering to do.

The question: what happened? And what should you have done to delete all the screenshots?

Here's the answer: remember that spaces separate arguments to a command? Thus, the command `rm -rf Screen *` is running `rm -rf` on two things: `Screen` and `*`. There's nothing called `Screen` in the working directory (`Desktop`), so `rm` ignores it. Then the `*` tells `rm` to delete everything in the current directory. Since the `-r` flag is turned on, subdirectories will also be deleted.

The correct thing to do was `rm -rf Screen*`. In this form there is only one argument to `rm -rf`, and that is every file and subdirectory whose name is of the form `Screen<something>`. In the case of our screenshot file `Screen Shot 2021-09-25 at 10.00.46 PM`, that `<something>` is `Shot 2021-09-25 at 10.00.46 PM` (with a space as the first character).

So, again, don't use spaces in file and directory names.

6 Absolute and Relative Paths

This chapter is more of a clarification and there's nothing really new.

You can specify the location of a file or a directory using its absolute path or its relative path. Both are ok. Absolute paths are with respect to the home directory `~`. Relative paths are with respect to the current working directory `.`. If you do not specify the root of your path, Linux assumes you are using a relative path.

```
[blah123@remote ayayaya]$ pwd
/u/d/blah123/ayayaya
[blah123@remote ayayaya]$ echo ~
/u/d/blah123
[blah123@remote ayayaya]$ ls
hw.c hw1.c hw2.c hw3.c barney.py subdirect
[blah123@remote ayayaya]$ cat hw.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote ayayaya]$ cat ./hw.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote ayayaya]$ cat ~/ayayaya/hw.c
#include <stdio.h>

int main(){
   printf("Hello world!\n");
}
[blah123@remote ayayaya]$ mv hw1.c ~/ayayaya/hww1.c
[blah123@remote ayayaya]$ ls
hw.c hww1.c hw2.c hw3.c barney.py subdirect
[blah123@remote ayayaya]$
```

The `echo ~` command will print your home directory to the screen. The `echo` command is way more useful than that, but it's not essential for navigating in Linux so I'm not including it. For more on what it can do, check

google or its man page!

When you do `cat hw.c`, since you are not explicitly specifying whether you are using `~` or `.` as the root of your path, Linux assumes you are using relative path, i.e. your root is `.`. Hence Linux looks for a file named `hw.c` in the current directory and `cat` it.

There's an exception to this rule: C executables. To run a gcc-generated C executable, we do `./a.out`. You might think that by the previous rule we can just do `a.out`, but that doesn't work. This is a historical issue and has something to do with security and viruses and what not, but from a practical vantage point just remember it as syntax. In fact it's not that hard to remember: `./a.out` isn't really a command, but just a file. If you want, you can sort of think of `./` as the "run C executable" command in this case only.

The use of absolute paths can come in handy when the files you want to operate on are far away. Say you have a folder for all of your C headers and another folder for all of your C bodies, but the two folders are quite far away. For example `~/install/bin/blah/include` has all of your headers and `~/install/bin/blahblah/body` has all of your bodies. If your working directory is `body` and you want to compile something, it can be cumbersome to type out `gcc project.c ../../blah/include/project.h -o project.x`, and trying to count how many `..`s you need is no fun. Instead you can use absolute path:
`gcc project.c ~/install/bin/blah/include/project.h -o project.x`

7 Black Magic: Makefiles

We have already covered all the basic material. By now you should be able to navigate yourself in a Linux command line and run some small-scale Python and C. The remaining two chapters are not essential to using Linux, but it will show you two of the many ways in which Linux can do some magical and powerful things, and hopefully give you a glimpse of why every development group in industry and academy uses it.

I'll structure these two sections around two example (and quite real) scenarios.

Go back to our `hw` directory and do a `rm -rf *` to clean everything in it. Or, of course, you can just make a new directory altogether. In any case, preparing a clean directory for this section would be helpful. I will assume that you have prepared a clean directory called `hw`.

Matrix multiplication is one of the most important operations for modern algorithms. It is widely used as the essential modules in many applications:

- Numerical solution of systems of differential equations;
- Numerical solution of important partial differential equations, such as the heat equation, the wave equation and the Schrodinger equation;
- Computer vision and image processing;
- Training of machine learning models;
- Graph algorithms;
- And much more...

As such, it is easily imaginable that the speed of matrix multiplication is the bottleneck of the speed of many applications. In fact, MATLAB actually has different functions for usual matrix multiplication and sparse matrix multiplication (matrices whose entries are mostly zero; these matrices can be multiplied without doing a $O(n^3)$ loop), because usual matrix multiplication is very slow.

Research into how to optimize matrix multiplication has been going on for decades. Your boss has asked you to join this army.

We won't go into the fancy methods since that's not what this Linux guide is about. We'll consider one very simple technique: changing the loop order. We assume all matrices are square with size n by n.

In your `hw` directory, create a couple of files

```
[blah123@remote hw]$ pwd
/u/d/blah123/hw
[blah123@remote hw]$ ls
[blah123@remote hw]$ vi genM.c
[blah123@remote hw]$ vi mm.c
[blah123@remote hw]$ vi test_mm.c
[blah123@remote hw]$ ls
genM.c  mm.c  test_mm.c
[blah123@remote hw]$
```

with the following content:

`genM.c`:

```
#define SEED 3.1415926535

void generate_Matrix(double *a, int n){
   int nn = n*n;

   for (int i=0; i<nn; i++){
      a[i] = SEED * (i+1);
   }
}
```

`mm.c`:

```
#define A(i, j) a[n*j+i]
#define B(i, j) b[n*j+i]
#define C(i, j) out[n*j+i]

#define ijk
//#define ikj
//#define jik
//#define jki
//#define kij
//#define kji

```

```
#if defined(ijk)
void MM(double *a, double *b, double *out, int n){
   int i, j, k;

   for (i=0; i<n; i++){
      for (j=0; j<n; j++){
         for (k=0; k<n; k++){
            C(i, j) += A(i, k)*B(k, j);
         }
      }
   }
}

#elif defined(ikj)
void MM(double *a, double *b, double *out, int n){
   int i, j, k;

   for (i=0; i<n; i++){
      for (k=0; k<n; k++){
         for (j=0; j<n; j++){
            C(i, j) += A(i, k)*B(k, j);
         }
      }
   }
}

#elif defined(jik)
void MM(double *a, double *b, double *out, int n){
   int i, j, k;

   for (j=0; j<n; j++){
      for (i=0; i<n; i++){
         for (k=0; k<n; k++){
            C(i, j) += A(i, k)*B(k, j);
         }
      }
   }
}

#elif defined(jki)
void MM(double *a, double *b, double *out, int n){
   int i, j, k;

   for (j=0; j<n; j++){
      for (k=0; k<n; k++){
```

```
            for (i=0; i<n; i++){
                C(i, j) += A(i, k)*B(k, j);
            }
        }
    }
}

#elif defined(kij)
void MM(double *a, double *b, double *out, int n){
    int i, j, k;

    for (k=0; k<n; k++){
        for (i=0; i<n; i++){
            for (j=0; j<n; j++){
                C(i, j) += A(i, k)*B(k, j);
            }
        }
    }
}

#elif defined(kji)
void MM(double *a, double *b, double *out, int n){
    int i, j, k;

    for (k=0; k<n; k++){
        for (j=0; j<n; j++){
            for (i=0; i<n; i++){
                C(i, j) += A(i, k)*B(k, j);
            }
        }
    }
}
#endif
```

`test_mm.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE_FIRST 128
#define SIZE_LAST 1024
#define SIZE_INC 128

```

```
9  void MM(double *a, double *b, double *out, int n);
10 void generate_Matrix(double *a, int n);
11
12 int main(){
13    double *a, *b, *out;
14
15    for (int p=SIZE_FIRST; p<SIZE_LAST; p+=SIZE_INC){
16       a = (double *)malloc(sizeof(double)*p*p);
17       b = (double *)malloc(sizeof(double)*p*p);
18       out = (double *)malloc(sizeof(double)*p*p);
19
20       generate_Matrix(a, p);
21       generate_Matrix(b, p);
22
23       clock_t begin = clock();
24       MM(a, b, out, p);
25       clock_t end = clock();
26       double time_spent = (double)(end - begin) /
   CLOCKS_PER_SEC;
27       printf("Size = %d, Time spent = %f seconds\n", p,
   time_spent);
28    }
29    return 0;
30 }
```

The `genM.c` fills the test matrices, the `test_mm.c` contains the driver `main()` function, and the `mm.c` defines the implementation of the matrix multiplication.

Sidebar: the macro

```
#define A(i, j) a[n*j+i]
```

simply means to store matrices in column major order. For example if $n = 3$, the mapping between array `double *a` and the matrix A is like such:

$$\begin{pmatrix} A(0,0) & A(0,1) & A(0,2) \\ A(1,0) & A(1,1) & A(1,2) \\ A(2,0) & A(2,1) & A(2,2) \end{pmatrix} = \begin{pmatrix} \text{a[0]} & \text{a[3]} & \text{a[6]} \\ \text{a[1]} & \text{a[4]} & \text{a[7]} \\ \text{a[2]} & \text{a[5]} & \text{a[8]} \end{pmatrix}$$

We can compile our project easily using what we already know:

```
[blah123@remote hw]$ ls
genM.c  mm.c  test_mm.c
[blah123@remote hw]$ gcc genM.c mm.c test_mm.c -o mm.x
[blah123@remote hw]$ ./mm.x
Size = 128, Time spent = 0.019914 seconds
Size = 256, Time spent = 0.064901 seconds
Size = 384, Time spent = 0.217672 seconds
Size = 512, Time spent = 0.542181 seconds
Size = 640, Time spent = 1.011029 seconds
Size = 768, Time spent = 3.361336 seconds
Size = 896, Time spent = 2.739842 seconds
[blah123@remote hw]$
```

The current version of `void MM()` is the version under the `ijk` macro in `mm.c`, since the macros for all the other versions are commented out. To switch to a different implementation, we can go back to the C file and comment out different macros, and then recompile. However, doing this many times is quite cumbersome. Try it! You will see what I'm talking about. It gets annoying pretty quick.

Note: in Linux shell you can use you up key to duplicate your previous command. Hitting the up key again will bring you up yet another command. Hitting the down key will bring you down one command. But even with these recompiling 6 times with editing C files in between is still annoying.

There's a better way to do this task. I mentioned that `gcc` has many many flags, but I only told you the output naming `-o` flag. I'll now tell you one of its other flags.

The `-D` flag for `gcc` (and `clang`) is the flag for defining macros at the command line. To see what it does, first comment out all the macros in all three files (except the column-major defining ones), i.e.

`genM.c`:

```
//#define SEED 3.1415926535

...
```

`mm.c`:

```
#define A(i, j) a[n*j+i]
#define B(i, j) b[n*j+i]
#define C(i, j) out[n*j+i]

//#define ijk
//#define ikj
//#define jik
//#define jki
//#define kij
//#define kji

...
```

`test_mm.c`:

```
...

//#define SIZE_FIRST 128
//#define SIZE_LAST 1024
//#define SIZE_INC 128

...
```

Then, when compiling, you can define the macros using the `-D` flag like this:

```
gcc -D ijk -D SIZE_FIRST=128 -D SIZE_LAST=1024 -D SIZE_INC=128
   -D SEED=3.1415926535 genM.c mm.c test_mm.c -o mm_ijk.x
```

For macros that don't have values assigned to them, a single `-D ijk` in the command line is equivalent to `#define ijk` in the C file. For macros that have assigned values, the syntax is `-D SEED=3.1415826535`, and it's equivalent to, you guessed it, `#define SEED 3.1415926535`. If you want to move flags around, remember to move `-D BLAH` as a group. You can now easily switch to different implementations with `-D jik`, or `-D kji`, and so on.

It is not a requirement that the original macros in the C files to be commented out; they can be flat out deleted. It's just good practice to comment instead of delete because it can help remind yourself of what they are.

The beauty of this approach is that you can now run trails of experiments with different implementations and different parameters from the command line directly, instead of changing the C source and recompiling every time. Also, reducing the number of changes to your C file will reduce the chance of making accidental errors.

Play with this! If you want, you can investigate matrix sizes up to 2000s or 3000s (look at the C sources to see which macro to change; I believe you can figure this out), and you can also play with different implementations. You might encounter a couple of surprising findings, like how quickly the performance degrades with matrix sizes, and how much simply switching the order of loops can impact performance. If you get tired of waiting for some matrices to multiply, you can always `control+c` to kill a process.

So now we can interact with our C interface purely from the command line. But the `gcc` command itself is quite long, and typing it out every time is still a little bit of a pain. In fact, for larger projects a `gcc` command can easily take 4 or 5 lines! Is there a way to compile the files without typing out `gcc`?

Introducing the makefile.

A makefile is basically a script to run a series of Linux commands automatically, instead of manually typing the commands out one at a time. (To the savants: yes, I know makefiles can do much more than simply serving as a bash script, but I don't think it's appropriate to introduce them at this level.)

Let's create a very simple makefile that has nothing to do with our matrix C interface. A makefile, just like any other file, can be created and opened as plain text in vi. However, **its name has to be `makefile` exactly**.

```
[blah123@remote hw]$ ls
genM.c  mm.c  mm.x  test_mm.c
[blah123@remote hw]$ vi makefile
[blah123@remote hw]$ ls
genM.c  makefile  mm.c  mm.x  test_mm.c
[blah123@remote hw]$
```

In `makefile` type in the following contents:

```
# makefile comments start with #

DEFAULT:
  pwd
  ls
  echo ~
  mkdir ayayaya
  ls
  rm -r ayayaya
  ls
```

Note: **the indentation in makefiles has to be a tab. No, 3 or 4 spaces will not work.**

Reminder: `echo ~` prints out your home directory.

Now that you have a `makefile`, you can execute it with the command `make`

```
[blah123@remote hw]$ ls
genM.c  makefile  mm.c  mm.x  test_mm.c
[blah123@remote hw]$ make
pwd
/u/d/blah123/hw
ls
genM.c  makefile  mm.c  mm.x  test_mm.c
echo ~
/u/d/blah123
mkdir ayayaya
ls
ayayaya genM.c  makefile  mm.c  mm.x  test_mm.c
rm -r ayayaya
ls
genM.c  makefile  mm.c  mm.x  test_mm.c
[blah123@remote hw]$
```

See what it did? The `make` command is sequentially executing the commands in your `makefile` automatically! For each command executed, it prints out the command, and then prints out the output of that command.

Now, makefiles can have multiple blocks. Let's try another makefile with a slightly modified content:

```
DEFAULT:
  pwd
  ls
  echo ~

CREATE:
  mkdir ayayaya
  ls
```

... and run it:

```
[blah123@remote hw]$ ls
genM.c  makefile  mm.c  mm.x  test_mm.c
[blah123@remote hw]$ make
pwd
/u/d/blah123/hw
ls
genM.c  makefile  mm.c  mm.x  test_mm.c
echo ~
/u/d/blah123
[blah123@remote hw]$ make DEFAULT
pwd
/u/d/blah123/hw
ls
genM.c  makefile  mm.c  mm.x  test_mm.c
echo ~
/u/d/blah123
[blah123@remote hw]$ make CREATE
mkdir ayayaya
ls
ayayaya  genM.c  makefile  mm.c  mm.x  test_mm.c
[blah123@remote hw]$
```

The `make` command can take in an argument `BLOCK`. It will then execute the block named `BLOCK` in the `makefile` in your current working directory. Note that this argument can also be tab-autocompleted in Linux shell.

If you don't give `make` an argument, the first block in `makefile` will be executed. It doesn't need to be named `DEFAULT`. It's just my habit.

Since makefiles automatically execute commands, and `make` is also a command, makefiles can reference themselves.

```
DEFAULT:
  pwd
  ls
  echo ~

CREATE:
  mkdir ayayaya
  ls

# I don't like capital letters
create:
  make CREATE
```

Run it:

```
[blah123@remote hw]$ ls
ayayaya  genM.c  makefile  mm.c  mm.x  test_mm.c
[blah123@remote hw]$ make create
make CREATE
make[1]: Entering directory `/u/d/blah123/hw'
mkdir ayayaya
mkdir: cannot create directory `ayayaya': File exists
make[1]: *** [makefile:9: CREATE] Error 1
make[1]: Leaving directory `/u/d/blah123/hw'
make: *** [makefile:14: create] Error 2
[blah123@remote hw]$
```

When `make` encounters errors it barks at us, which is normal. In this case we are trying to `mkdir` a directory `ayayaya` that already exists, causing the `mkdir` command to fail and hence causing `make create` to fail.

We're ready to use makefiles on our C matrix infrastructure. Change the contents of `makefile` to the following:

```
method = ijk
SIZE_FIRST = 128
SIZE_LAST = 1024
SIZE_INC = 128
SEED = 1.2345678

DEFAULT:
  make COMPILE
  make RUN

COMPILE:
  gcc -D $(method) -D SIZE_FIRST=$(SIZE_FIRST) -D SIZE_LAST=$(
   SIZE_LAST) -D SIZE_INC=$(SIZE_INC) -D SEED=$(SEED) genM.c mm
   .c test_mm.c -o mm_$(method).x

RUN:
  ./mm_$(method).x
```

You can define and use variables in makefiles like this. It's helpful to think of all makefile variables as strings. Similar to the workings of f-strings in python, a dollar sign followed by the variable name enclosed in brakets will be replaced with the string value of that variable. In the above example, the `gcc` command is essentially

```
gcc -D ijk -D SIZE_FIRST=128 -D SIZE_LAST=1024 -D SIZE_INC=128
   -D SEED=1.2345678 genM.c mm.c test_mm.c -o mm_ijk.x
```

Play with the makefile. Appreciate how easy it is. To make life really easy and enjoyable, I usually like to open up 2 `ssh` windows to the same server. One window will be in the vi for `makefile`, and the other will be on the command line. In this fashion I can change the macro definitions in the makefile in one window and `make` in the other window (remember to `:w` in the makefile vi before `make`-ing in the other window), and life has just got really easy.

8 Black Magic 2: Using Python Script

For this chapter, your boss has tasked you with measuring the speed of `gcc`'s factorial computation under an iterative method and a recursive method.

Clean or create a new directory. I'll assume that clean directory is called `hw`. Let's create two C files in our clean directory with the following content:

`fact_compute.c`:

```
long fact_iter(int x){
  long res = 1;
  for (int a=x; a>0; a--){
     res *= a;
  }
  return res;
}

long fact_rec(int x){
   if (x==1){return 1;}
   else {return x*fact_rec(x-1);}
}
```

`fact_driver.c`:

```
#include <stdio.h>
#include <time.h>

//#define SIZE 10

long fact_iter(int x);
long fact_rec(int x);

int main(){
   long a, b;
   int i;
   clock_t begin_iter = clock();
   for(i=0; i<2000000; i++){a = fact_iter(SIZE);}
   clock_t end_iter = clock();
   double time_spent_iter = (double)(end_iter - begin_iter) /
   CLOCKS_PER_SEC;

   clock_t begin_rec = clock();
```

```
18    for(i=0; i<2000000; i++){b = fact_rec(SIZE);}
19    clock_t end_rec = clock();
20    double time_spent_rec = (double)(end_rec - begin_rec) /
      CLOCKS_PER_SEC;
21
22    printf("Results: iter = %d, rec = %d\n", a, b);
23    printf("Time spent: iter = %f, rec = %f\n", time_spent_iter,
       time_spent_rec);
24
25    return 0;
26 }
```

Note: in the driver the core factorial functions are repeated 2000000 times to emphasize the time difference between iteration and recursion.

These files can be compiled and run like this:

```
1 [blah123@remote hw]$ gcc -D SIZE=10 fact_driver.c fact_compute.
    c -o fact.x
2 [blah123@remote hw]$ ./fact.x
3 Results: iter = 3628800, rec = 3628800
4 Time spent: iter = 0.045798, rec = 0.040961
5 [blah123@remote hw]$
```

Seeing that recursion is faster than iteration, you go back to your boss and reported the results.

Looking at your results, your boss, with disappointment written on his face, tells you that it is not enough to do the test just for one number. He wants you to repeat the test for all factorials between 1! and 40!.

Ok, you think to yourself, you know how to do that, just use a for loop in `main()`. Easy enough.

But then your boss goes like this:
"Oh and by the way, keep these 40 compiled executables just in case we need them down the road."

Upon first glance it doesn't seem like a different job, but it is: now you can't simply add a for loop to `main()`. How matter how many times you loop in

`main()`, it will only get compiled to one executable. To compile 40 factorials to 40 executables, you would need to call `gcc` 40 times at the command line! That's absurd on the face of it. And what if he wanted 500? 5000? Do I just sit at my screen and type `gcc` 24-7?

What we want is a way to execute a for loop not at the C level, but at the command line level. In other words, we want a mechanism that loops the action of "typing `gcc` to the command line" for us.

It's the all-mighty Python to the rescue. (It is possible to do a loop in makefiles, but aren't python loops more familiar to us?)

Create a file `runner.py` (this time there's no strict naming requirement like `makefile`) with the following content:

```
1 import subprocess
2
3 if __name__ == "__main__":
4     subprocess.call("pwd", shell=True)
```

The module `subprocess` enables the magic. Try to run it with `python3 runner.py` and see what happens. I think at this point you already know what's going to happen, right?

Exercise: modify `runner.py` so that it compiles the 40 factorial C executables. Also, I didn't include the link to fstrings on page 42 just for aesthetics!

Solution:

```
import subprocess

if __name__ == "__main__":
    for i in range(10, 40, 1):
        subprocess.call(f"gcc -D SIZE={i} fact_driver.c
    fact_compute.c -o fact_{i}.x", shell=True)
        subprocess.call(f"./fact_{i}.x", shell=True)
    # let's do some clean up too. Think: why does the clean up
    need to be outside the for loop?
    subprocess.call("mkdir execs", shell=True)
    subprocess.call("mv *.x execs", shell=True)
```

If you try to run this, you will see a lot of output on the command line. That is normal: they are the printfs from `fact_driver.c`. However, for such a large number of outputs, printing them to the screen doesn't seem like the best way to handle and use them. Since python can be used to deal with files, we would ideally want to store these outputs to some kind of output file.

This is also easily doable in Linux. Try out these commands and then you will see.

```
[blah123@remote hw]$ ls > i_am_a_file_name.txt
[blah123@remote hw]$ ls -a > i_am_another_file_name.txt
[blah123@remote hw]$ pwd > i_am_yet_another_file_name.txt
```

I am letting the examples do the explanations for me now, since now that we're at the last bit of the last chapter you should already be quite fluent with Linux in general.

So we have

```
import subprocess

if __name__ == "__main__":
    for i in range(10, 40, 1):
        subprocess.call(f"gcc -D SIZE={i} fact_driver.c
    fact_compute.c -o fact_{i}.x", shell=True)
        subprocess.call(f"./fact_{i}.x > fact_{i}.txt", shell=
    True)

    subprocess.call("mkdir execs", shell=True)
    subprocess.call("mv *.x *.txt execs", shell=True)
```

You might want to download the resultant `execs` directory that stores all of these beautiful results to your local computer, don't you? This is easily done on mac with the `scp` command. For windows it's trickier but not impossible. See here.

On mac, open a new Terminal window (not the one you're currently in that's logged into ECF). Do not sign into ECF, as `scp` needs to be called from the local computer in order to start a download from the server. The command is

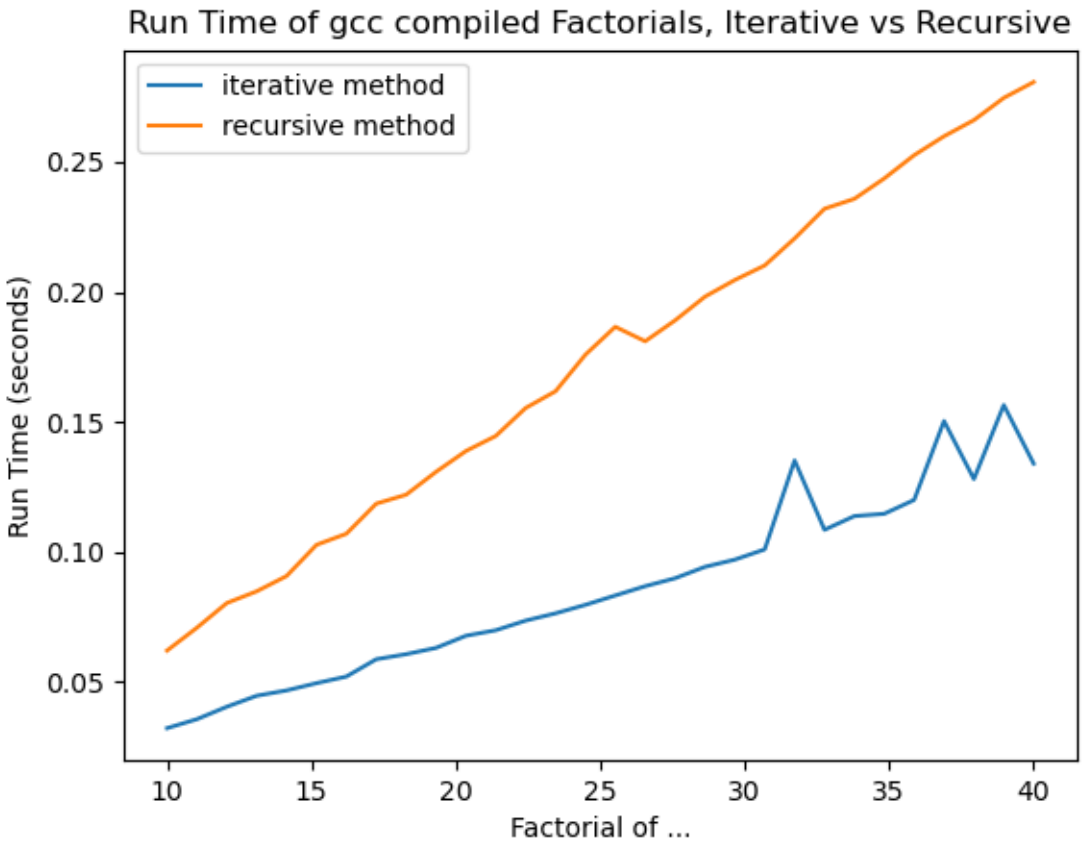
```
scp -r blah123@remote.ecf.utoronto.ca:~/hw/execs a_local_name
```

This will create a folder `a_local_name` on your local computer with the same contents as `execs` on ECF server. A good name for the local name would be `Downloads/execs`. You will be prompted to enter your ECF password. Of course, change the path on ECF to a different one if your directory tree is slightly different than the one laid out in this guide. Note that there is no space between the colon(:) and the tilde (∼). Do you know why? You should have enough to figure out why. Think about what spaces mean.

Bonus: we can produce nice little graphs that show runtime versus factorial number. I'll include the script on the next page without explanations. You should have enough python brain to figure out what it is. Ask google when you see unfamiliar functions or string methods!

```python
import subprocess
import numpy as np
import matplotlib.pyplot as plt

def extract_time(filename):
    with open(filename, "r") as file:
        time_spent_line = file.readlines()[1]

        comma_loc = time_spent_line.find(",")
        iter_equal_loc = time_spent_line.find("=")
        rec_equal_loc = time_spent_line.rfind("=")

        iter = float(time_spent_line[iter_equal_loc:comma_loc].strip("= "))
        rec = float(time_spent_line[rec_equal_loc:].strip("= ").strip("\n"))

        return (iter, rec)


if __name__ == "__main__":
    factorial_numbers = np.linspace(10, 40, 30)  # 30 numbers from 10 to 39 inclusive
    iter_times = np.zeros(30)
    rec_times = np.zeros(30)

    for i in range(10, 40, 1):
        subprocess.call(f"gcc -D SIZE={i} fact_driver.c fact_compute.c -o fact_{i}.x", shell=True)
        subprocess.call(f"./fact_{i}.x > fact_{i}.txt", shell=True)

    subprocess.call("mkdir execs", shell=True)
    subprocess.call("mv *.x *.txt execs", shell=True)

    for i in range(10, 40, 1):
        (iter, rec) = extract_time(f"execs/fact_{i}.txt")
        iter_times[i-10] = iter
        rec_times[i-10] = rec

    plt.plot(factorial_numbers, iter_times, label = "iterative method")
    plt.plot(factorial_numbers, rec_times, label = "recursive method")
    plt.legend()
```

```
39    plt.xlabel("Factorial of ...")
40    plt.ylabel("Run Time (seconds)")
41    plt.title("Run Time of gcc compiled Factorials, Iterative vs
       Recursive")
42    plt.savefig("
      Run_Time_of_gcc_compiled_Factorials_Iterative_vs_Recursive.
      png")
```



Note: ECF doesn't have matplotlib preinstalled, so I produced this graph on my local computer. Everything in this guide works with mac, since the mac Terminal is simply a Linux command line for your local computer. Most professional supercomputers will have matplotlib preinstalled.

9 Closing

Linux is powerful. Very powerful. I hope that the final example with the plotting has illustrated this.

I have hidden a lot of dirty details from you in this guide. For example, I told you that all Linux files are plain text, but they're actually not. C executables are a clear counterexample. The correct statement is "almost all files are plain text, except those that are not" (lol). And what about that `.vimrc`? What is it exactly? Those are two of the many questions that I avoided touching on in this guide.

However, I think this approach is worth it. Mentioning the dirty details to starters will only raise more questions then it answers. I tried to make it as practical as possible, in terms of getting new Linux users started. In my travels I have found that the best way to learn is through examples, so that's what I did in this guide. I am not planning to tell you everything there is to know about Linux, nor am I able to. However, I do believe I am telling you enough for you to get started and not be scared with Linux.

As is always the case in university, industry and academia, you never stop learning. Every resource can only provide a limited amount of information. You have to build your own knowledge base through exposure and researching on your own.

I sincerely hope that this guide has done a good job in getting you started with Linux. If that's the case, then I would be very happy.

The name of this guide is a reference to Professor Leonard Susskind's series *The Theoretical Minimum*. It carried the same philosophy and got me started with the basics of modern theoretical physics.

Live long and prosper, stay calm, and use Linux.

Best,
Paul