Week 2, Ex. 1 - Moving around

Getting Setup

Let's start up a terminal on linux. We recommend you use the Docker images we've provided. Specifically, the `pementorship/w1_ex1` image (the one for this exercise). You can access it by running this docker command on your system (you should have it installed by now):

```
$ docker run -it pementorship/w2_ex1
```

You should see a prompt like this:

```
[me@88dede95125b ~]$
```

Navigation

We're gonna learn a few basic commands to help us navigate. Type `pwd` and then enter. You should see this:

```
[me@88dede95125b ~]$ pwd
/home/me
```

`pwd` is a command that shows you the present working directory (hence the name). Basically, every process (aka running program) has a directory (aka a folder) from which it's running. This can (and often is) different from where the executable for said program lives.

Now for a few more commands. Run `ls`:

```
[me@828abf2f1f04 ~]$ ls
a_file.md  a_folder
```

This shows you all the files in the current directory. Now run `ls -l` to see the long-form list of files:

```
[me@9f2c81e1b90f ~]$ ls -l
total 8
-rw-r--r-- 1 me me   14 Aug  9 12:43 a_file.md
drwxr-xr-x 2 me me 4096 Aug  9 18:28 a_folder
```

Let's examine one of these lines:

```
drwxr-xr-x 2 me me 4096 Aug  9 18:28 a_folder
```

Here, you can see the name of the file (or in this case, folder) to the very right: `a_folder`. Now let's look at every item left to right:

- `drwxr-xr-x`: This is the permissions of the file, and they specify who can act on the file in what ways.
  - The `d` tells us this is a folder (`d` for directory). If you see a `-` it means it's a regular file. There's other leters for other file types.
  - After that, there's three groups of three characters. Each group defines permissions for a different set of users: user, group, and global, or other, permissions (abreviated `u`, `g` and `o`). Each one of these contain a Read, Write and eXecute flag, which respecively tell you whether the given user can read, write or execute the file. This last one is unique since execute can both refer to running a file as an executable, or, if this is a directory, enter it.
- `2`: the number of hard links to this file (we'll talk about this when we get to file systems)

- `me me`: These two are, respectively, the user that owns the file, and the group that also owns the file. These are used by the file permissions we mentioned earlier. The group is usually something different from the user, and signifies a set of users with some important set of permissions.
- `4096`: The size of the file in bytes
- `Aug  9 18:28`: When the file was last modified

Let's take a look at one of the files. We'll be using the `cat` program:

```
[me@c6677a8ee8f9 ~]$ cat a_file.md
Hello, world!
```

Basically, this prints out the file to standard output, which shows up on your terminal.

Now, let's access the directory `a_folder`:

```
[me@7d5a40e91479 ~]$ cd a_folder/
[me@7d5a40e91479 a_folder]$ pwd
/home/me/a_folder
[me@7d5a40e91479 a_folder]$ ls
big_file.txt  more_data.txt
[me@7d5a40e91479 a_folder]$ cat more_data.txt
Hello, this is another file
With more text!
```

We used the `cd` (short for change directory) command to go into `a_folder`. Notice this is using a *relative path*. In other words, the path is stated relative to the current working directory. You could have also specified it as an *absolute path*, which you can identify because it starts with a `/`, refering to the root directory `/`. As such, running `/home/me/a_folder` (notice this is the same output as the `pwd` command, which always outputs absolute paths).

We ran `ls` again, and notice we now have a file called `more_data.txt`, which we can `cat` to find a more complex piece of text.

One last thing, we didn't take a look at the `big_file.txt` file. However, if you `cat` it you'll notice it's annoyingly long. Run `less big_file.txt` to look at the contents in a scrollable window. You can use the up and down arrows to move through the file, and the `q` key to quit.

Now if you want to go back to `/home/me`, you have three options. You can either `cd` to the absolute path, using `cd /home/me`. You can also use a special directory in every folder that points to the parent directory called `..`. As such, you can do:

```
[me@97b6a023a882 a_folder]$ pwd
/home/me/a_folder
[me@97b6a023a882 a_folder]$ cd ..
[me@97b6a023a882 ~]$ pwd
/home/me
```

Additionally, you can simply run `cd`, without any arguments, to take you back to your user's home directory (`/home/me` in this case):

```
[me@97b6a023a882 log]$ pwd
/var/log
[me@97b6a023a882 log]$ cd
[me@97b6a023a882 ~]$ pwd
/home/me
```

File Editing

Often throughout this course, you'll find yourself needing to write and edit code inside the container. We'll talk about how you might be able to move files between your computer and the docker container, but you'll also need to be able to edit these files directly through your terminal. We've installed three editors in the container, `nano`, `emacs` and `vim`. If you've never used a terminal editor, stick to `nano`. I recommend you take the time to learn one of the other two once you become comfortable with nano, just because they're widely used and standard in the linux world, and they're quite rich in what they can do. From now on, I'll be refering to your editor of choice as `$EDITOR`.

EXERCISE: Modify the `a_file.md` file and add a line that simply says "Hello from the other side".

If you're not sure how to edit your file, simply run: `$EDITOR a_file.md` from your home directory (remember to make sure you're already there using `pwd` and `cd`). If you're using nano, you can just type away and move around like any other file editor. Once you're done press `^X` (Ctrl+X) to exit and save (it'll ask for confirmation, just follow the instructions).

You can now `cat` the file to see if the changes are there:

```
[me@7d5a40e91479 ~]$ cat a_file.md
Hello, world!
Hello from the other side
```

Man pages and help

This is a quick aside about how to get more information on programs. The internet can be a great source of information, but sometimes you want to get information about commands quicker from the terminal, or just need a quick references. For this, we have two mechanisms:

- Most terminal (aka CLI) programs, have a `--help` flag. Append that to a program to get a short, summarised version of help for the program with basic usage information and a list of flags/arguments. For example, run `ls --help` to see what comes out
- Many of the programs, specially those that come pre-installed on your system, have a man page. *man* here stands for manual. These are more long-form versions of the help command, including detailed instructions and usecases, often with plenty of examples. You can run `man man` to read the man page on how to use `man` (there's some useful things there to check). Run `man ls` to get the more detailed information on how `ls` works (you'll be surprised how many ways you can use that).

Piping

The last thing we'll discuss in this week, is piping. This is the operation of passing around the output of one program to the next, or to a file. This is basically how a lot of the work in Linux happens. The basic design philosophy is, create small programs that do minimal things, and chain input/output between programs to perform more complex operations. This doesn't work out in practice for a lot of things, but for your terminal programs, this is the basic philosophy of how they were designed.

Let's first talk about a little program called `grep`. `grep` lets you search through files. If you run `grep --help`, you can see:

```
[me@7d5a40e91479 ~]$ grep --help
Usage: grep [OPTION]... PATTERN [FILE]...
Search for PATTERN in each FILE.
[...]
```

The usage line tells us how to use it. The items in square brackets are optional and the ones without are required. Three dots (`...`) signify they can take more than one of the preceding argument. By itself, `grep` is used to search for a regex pattern (which for now we'll pretend is just like doing a text search) in a file or set of files. However, notice that the `FILE` argument is optional. Here, that means that if no argument is provided, it'll search over standard input. We can make use of this to search over the output of another program. Let's try this:

```
[me@f5d3dd0599d8 ~]$ ls
a_file.md  a_folder  data
[me@f5d3dd0599d8 ~]$ cd data/
[me@f5d3dd0599d8 data]$ ls -l
total 0
-rw-r--r-- 1 me me 0 Aug  9 20:32 afile.md
-rw-r--r-- 1 me me 0 Aug  9 20:32 afile.txt
-rw-r--r-- 1 me me 0 Aug  9 20:32 bfile.md
-rw-r--r-- 1 me me 0 Aug  9 20:32 bfile.txt
-rw-r--r-- 1 me me 0 Aug  9 20:32 cfile.md
-rw-r--r-- 1 me me 0 Aug  9 20:32 cfile.txt
[me@f5d3dd0599d8 data]$ pwd
/home/me/data
```

Notice we have a bunch of files in this `data` folder. Now, `ls -l` output all the files, but what if we wanted to just look at the files with a `.txt` extension? Well, then we can pipe the output of `ls -l` to `grep`, and have that search for lines with a `.txt`:

```
[me@f5d3dd0599d8 data]$ ls -l | grep .txt
-rw-r--r-- 1 me me 0 Aug  9 20:32 afile.txt
-rw-r--r-- 1 me me 0 Aug  9 20:32 bfile.txt
-rw-r--r-- 1 me me 0 Aug  9 20:32 cfile.txt
```

So to sumarise here, the `|` basically means, send the output of `ls -l` as the input of `grep .txt`, and so all we see is the output of the final `grep` command. Notice, both these programs are basically running at the same time, but we're only chaining the output of one to the input of another, so you could have the command on the left constantly creating data and the one on the right constantly consuming that data.

What's next?

Continue to `problems.pdf` where I'll list a bunch of open-ended problems for you to explore and discuss with your mentor!