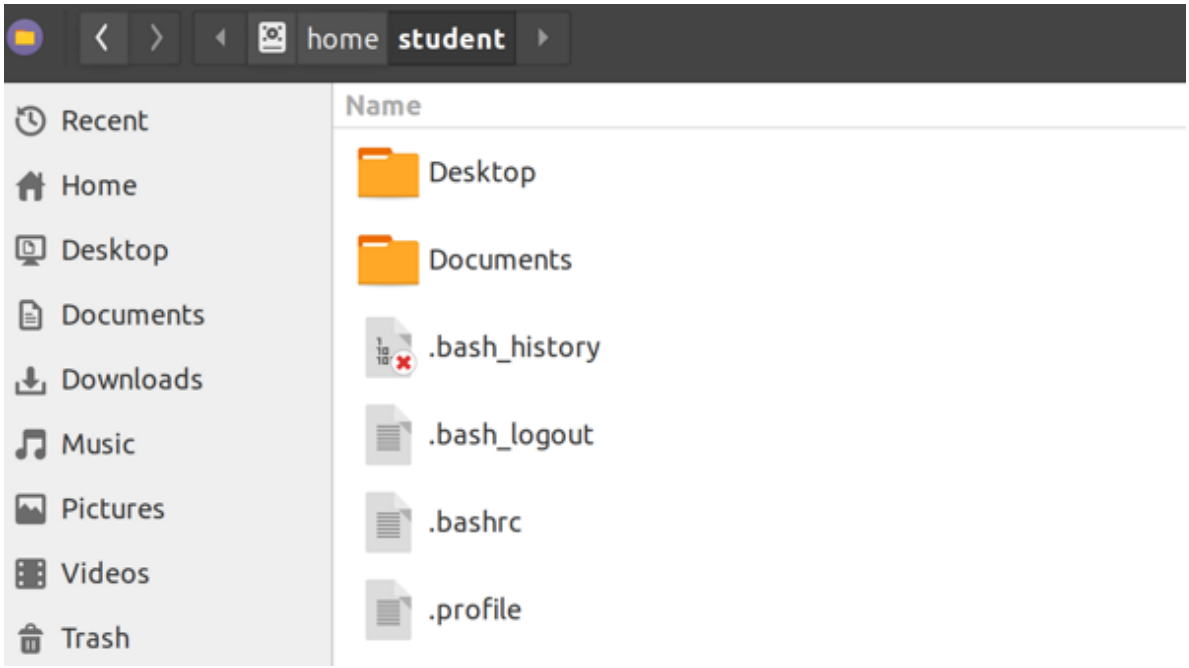
Lecture 2A : Linux File Commands

If you are familiar with how files and directories work in Windows, then you'll be happy to know that it works very similarly in Linux.

When you start up File Explorer in Windows, you see an empty folder for your user, and inside there are folders called 'Documents', 'Music', 'Desktop', and so on.

When you start up the File Explorer in Knoppix or any Linux distribution with a GUI, you will see almost exactly the same thing. Each user has a home directory located here: **/home/.** (Please note the direction of the slashes, they are the opposite of the slashes in Windows!)

Terminology Revisited



nautilus screenshot

- **/home** is the *parent* of the directory **user**.
- There are two *subdirectories*: **Documents** and **Desktop**.
- We talk about moving *into* a subdirectory, end moving *out* into the parent directory.
- As we move up and up, eventually we run out of parent directories. We are now in the *root* directory.
- The *root* directory in Windows is usually C:, which represents our primary file system. As you'll see, Linux is a bit different.

Navigation II

Use `ls -a` to view the contents of your Matrix home. You might see something like this:

```
.   .bash_history  .bashrc  public_html  .viminfo
..  .bash_logout
```

- `cd public_html` is used to enter the public_html directory. You can use `ls` to see that there is only one document in here.
- `cd ..` to go back up. Now we are back where we started, in the directory **/home/student**. Here is our first important Linux shortcut:

`..` = parent directory (one level up).

More Navigation

Try to use `cd ..` to enter `/home`. Use `ls` to view all of the home directories of students and professors (Be prepared to wait!). Ask a friend for their myseneca username, and use `cd <username>` to enter their home directory. Try to use `ls`. Does it let you?

Creating Files

We can use a command called `touch` to make a file called `test1`. `touch test1`

This file is empty, it has nothing in it and its size is zero. You can confirm this by typing in `ls -l`.

```
total 8
drwxrwxr-x 2 student student 4096 Sep  7 14:33 Desktop
drwxrwxr-x 2 student student 4096 Sep  7 15:51 Documents
-rw-rw-r-- 1 student student    0 Sep  9 21:24 test1
```

See the 0 in the fifth column? That is file size.

Copying Files

Now, let's take a look at copy and move.

`cp` = copy

`cp test1 test2` what did that do?

```
total 8
drwxrwxr-x 2 student student 4096 Sep  7 14:33 Desktop
drwxrwxr-x 2 student student 4096 Sep  7 15:51 Documents
-rw-rw-r-- 1 student student    0 Sep  9 21:24 test1
-rw-rw-r-- 1 student student    0 Sep  9 21:24 test2
```

`cp` takes two *arguments*. The first is the *target*. This is a file or directory that already exists. The second argument is what you want to *create*. (Tell the shell what you have, and then tell it what you want).

Renaming Files

`mv` = move/rename.

`mv test2 test3` what did that do?

```
total 8
drwxrwxr-x 2 student student 4096 Sep  7 14:33 Desktop
drwxrwxr-x 2 student student 4096 Sep  7 15:51 Documents
-rw-rw-r-- 1 student student    0 Sep  9 21:24 test1
-rw-rw-r-- 1 student student    0 Sep  9 21:24 test3
```

`mv` is used to *move* files and directories around, but we also use it to *rename* files and directories. This is an example of renaming. Again, tell the shell what you *have* (I have a file called `test2`), and then tell the shell what you *want* (I want this to become a file called `test3`).

Creating Directories

Working with directories requires some different commands in Linux.

In **Documents**, type this command: `mkdir level1` . Then use an `ls` command to check your work.

Creating Directories II

Now type in this command: `mkdir level2/level3` . You should see an *error message*. Reading and understanding error messages is an important skill to learn!

```
mkdir: cannot create directory ‘level2/level3’: No such file or directory
```

The problem here is that we want to create `level3` , but the shell is expecting `level2` to exist, and it doesn't. So it assumes that we've probably made a mistake, gives us an error message, and stops without creating anything. This is the default behavior.

Changing Options

Let's change the default behavior by adding an *option*, or *flag*.

```
mkdir -p level2/level3
```

The `-p` is option tells the shell to create a *parent* directory if it doesn't exist already. In this case, we asked the shell to create **level2** in addition to creating **level3**.

We can use another command `tree` to see a nice output of directories and subdirectories.

Moving Files

Now let's demonstrate the other other use of `mv` : this time our second argument will be a *destination*. Our destination will be the name of a directory.

`mv test3 level2/level3` What do you expect that to do?

We could use `cd level2/level3` then `ls -l` to check what happened, but we can also peek into a directory without entering it.

```
ls -l level2/level3
```

```
total 0
-rw-rw-r-- 1 student student 0 Sep  9 21:24 test3
```

Please note that our *current location hasn't changed*.

Copying Directories

```
.
├── level1
├── level2
│   └── level3
│       └── test3
└── test1

3 directories, 2 files
```

We can see that this wasn't exactly what we wanted. **level2** should probably a subdirectory of **level1**. So we probably want to perform a command to *copy* level2 into level1. We can try using the `cp` command to do this. The first *argument* will be the *target*, the second argument will be the *destination*.

Try this:

```
cp level2 level1
```

What happened?

Try this instead:

```
cp -r level2 level1
```

```
.
├── level1
│   └── level2
│       └── level3
│           └── test3
├── level2
│   └── level3
│       └── test3
└── test1

5 directories, 3 files
```

The `-r` means *recursive*. It means to repeat an action for all things contained in the target (**level2**). necessary for directories and things.

Moving Directories

Let's use `mv level2 level2a` to rename that second `level2` directory in our current location. You should now have a tree that looks like this:

```
.
├── level1
│   └── level2
│       └── level3
│           └── test3
├── level2a
│   └── level3
│       └── test3
└── test1
```

```
5 directories, 3 files
```

Use `cd level1` to set `level1` as your current location.

Now let's try *pulling* instead of *pushing*. The idea here is that your *target* is somewhere other than your current location, and your *destination* is the current destination. This works fine.

Try this command: `mv ../level2a .` (Again, use TAB to make less work!)

```
.
├── level1
│   ├── level2
│   │   └── level3
│   │       └── test3
│   └── level2a
│       └── level3
│           └── test3
└── test1

5 directories, 3 files
```

Notice that the `mv` command *doesn't need the `-r` option!* This is just a weird inconsistency of these core utilities that you'll have to remember.

`.` = current directory. We are *pointing at* a target that's far away, and pointing at the current location as destination.

`mv level2a/level3/test3 .` What did this do?

Remove Files/Directories

Finally, a lesson in removing files and directories. The command here is `rm`. `rm test3` This should work!

`cd level2a`

Now try `rmdir level3` This will remove an *empty* directory.

- `cd ..`
- `rmdir level2` (doesn't work!)
- Now try `rm -r level2`
- Use `ls` to make sure that level2 is gone. The `-r` is required to delete a directory that *isn't* empty. Be careful, because as you see, this command will remove *all subdirectories and files*!
- Use `touch` to create a new file, and then try using `rm -i` to delete it. What does the `-i` option do?

Navigation II

Type: `tree ~` `~` is a shortcut that means **home** for the current user. Our current user is `student`. So `~ = /home/student`.

- `cd ~/public_html` where are we?

Now try this: `cd` without any argument will take us home.

Summary for Filepaths

Terminology

absolute
: print the entire file path. This is very specific, but a lot to type. It will work regardless of where you are in the filesystem. (example: `/home/student/level1/level2/level3`)

relative
: print the path relative to something else. Allows us to use shortcuts like: `..`, `.`. It's easier for humans to use, but can be unpredictable since the result will change based on where you are. (example: `../level2a`)

commands
: We always interacting with the shell by entering a command. Example: `ls`

options/flags
: changes a command's default behaviour. These will always be listed in a command's man page. Example: the "-l" in `ls -l`.

arguments
: specifies (usually) a location, destination, a file, or something else. Example: the "test" in `mkdir test`.

> ### Commands
>
> - `pwd` : Print Working Directory (where am I?)
> - `cd` : Change Directory
> - `ls` : LiSt files
> - `touch` : create an empty files
> - `cp`, `cp -r` : CoPy, CoPy Recursively
> - `mv` : MoVe (also how we rename files)
> - `mkdir`, `mkdir -p` : MaKe DIRectory, MaKe DIRectory and Parent
> - `rm`, `rm -r` : ReMove, ReMove Recursively
> - `rmdir` : ReMove DIRectory

> ### Shortcuts Used in *Relative Filepaths*
>
> - `.` : Current directory (we use this to *pull* files)
> - `..` : Parent directory (one directory closer to root)

The Linux Filesystem

We started by talking about the Windows filesystem, and comparing it to the Linux filesystem. We talked about C:₩ being *root* in Windows, but not really how it works in Linux. Let's try that now. Enter `cd ..` until your command prompt no longer changes

when you enter the command. You can go no further. Take a look at your command prompt:

```
student@eric-eriga:/$
```

Now use `pwd` to see where you are.

```
/
```

/ is root. You are at the top of the filesystem.

Use `ls -l to take a look at the contents. They will look *very* different from what you are used to.

```
 total 139
drwxr-xr-x   2 root root 12288 Sep  5 17:06 bin
drwxr-xr-x   5 root root  3072 Sep  6 08:24 boot
drwxrwxr-x   2 root root  4096 Mar  5  2016 cdrom
drwxr-xr-x  22 root root  5440 Sep  9 21:15 dev
drwxr-xr-x 181 root root 12288 Sep  7 14:32 etc
drwxr-xr-x   4 root root  4096 Sep  7 14:23 home
drwxr-xr-x  26 root root  4096 May  1 23:28 lib
drwxr-xr-x   2 root root 12288 Jun  4 11:55 lib32
drwxr-xr-x   2 root root  4096 May  1 23:21 lib64
drwxr-xr-x   3 root root  4096 Mar  7  2016 media
drwxr-xr-x   2 root root  4096 Apr 22  2016 mnt
drwxr-xr-x   8 root root  4096 Jun 18 09:48 opt
dr-xr-xr-x 294 root root     0 Sep  4 10:25 proc
drwxr-xr-x  34 root root  1240 Sep  9 21:15 run
drwxr-xr-x   2 root root 12288 Aug 20 11:00 sbin
dr-xr-xr-x  13 root root     0 Sep  7 15:51 sys
drwxrwxrwt  25 root root 24576 Sep  9 21:51 tmp
drwxr-xr-x  12 root root  4096 Nov 10  2016 usr
drwxr-xr-x  14 root root  4096 Apr 22  2016 var
```

- What do all these directories mean? Let's look at *some*:
- `/bin` : Contains binary files (executables). You know all those commands we've been using, such as `ls` , `pwd` , etc.? They all live here.
- `/etc` : Contains a lot of directories and some files that end in .conf. This is where you find a look on configuration files. If you need to set a static IP address, for example, this is where you might do that.
- `/mnt` : One of the locations where secondary drives are mounted to. For example, a network drive might be found here once you've mounted it.
- `/tmp` : Temp. The contents of this directory are cleared when the computer is rebooted. Some scripts and programs might make use of it to store– you guessed it– temporary files.
- `/home` : This is where you will spend most of your time, since you don't really have permission to change any of the important stuff. :) `/home/` is where we store anything that is specific to one user. For example, on my personal computer, all of my Steam games are stored in `/home` . My roommate could create a user on my machine, install Steam and then install games from their account, and I wouldn't have access to any of them since they would be installed in *their* home directory.
- `/dev` : Devices. This one is interesting. Here you can find files that correspond to hardware. There is a directory here that is called **cpu** for example. For the most part, you shouldn't try to manipulate anything in here: "What is /dev/mem?" for example..

But there is one location that we will use a lot: **/dev/null**. Redirecting things into **/dev/null** is like throwing them into a black hole. You'll see this used later on.

In Linux, *Everything* is a File!

Understanding the Information in ls -l

Take a look at our previous example:

```
drwxr-xr-x   2 root root 12288 Sep  5 17:06 bin
```

Permissions	Links	Owner	Group	Size	Time Last Modified	Name
drwxr-xr-x	2	root	root	12288	Sep 5 17:06	bin

The first letter in 'Permissions' tells us that this is a *directory*. There are several options:

- – : normal file
- d : directory
- s : socket file
- l : link file

We'll talk more about permissions and links soon. Note that the owner and group are 'root'. Isn't **/** root?

- *User*: Much how **/** is the root of the filesystem, "root" is a term that we use for the *administrator* of the computer. As normal users, we don't have enough permission to read other people's directories or to change crucial system files. To do that, we would need *root privileges*. Since `/bin` is a directory containing incredibly important executables, we want to restrict a normal user's ability to wreck things inside. Thus, the owner is 'root', and only a root user would be able to edit files inside.

- *Group*: has a similar purpose. Every user is part of a group, but we can use groups to further control what users can do. For example, we may want to give students and teachers different levels of access to the system. A teacher might have access to edit the assignments, while the students only have access to read or execute the assignments.

- *Size*: By default, this displays the size of the file in *bytes*. There is a handy option that we can use with `ls` that gives us a *human-readable* file size.

```
ls -lh
```

```
drwxr-xr-x   2 root root  12K Sep  5 17:06 bin/
```

Now it's easier for us to see that `bin` has a size of 12 *kilobytes*.