# Lecture 2B : Linux Files II



Here is some basic information about naming files and directories in Linux:

- Linux file names (and directory names) are *case sensitive*. Keep this in mind when you name your files.
- Linux file names can contain any number of strange symbols including *, #, and more. But it's *much much* better to avoid these.
- Linux file names can contain spaces, but when you use them in the shell you need to use an escape character (₩).
- There can't be two Linux files or directories with the *same name* in the same location.

You can try this now.

`mkdir "this is a test"` Now type `cd this` (with the space) and press tab. Did it autocomplete? Now try again, but type `cd this\` (with a space). This should work.

---

## Relative-to-Home Paths

So far in these notes we've talked about:

- Relative Paths
  These paths specify a path from your *current location* to some other location. They usually start with either the shortcut to your *parent directory* ( `..` ), or with a *subdirectory* in your current location. You can use this type of file path to travel *up* and *down* the file tree.

- Absolute Paths
  These paths specify a path starting from the *top* of your file system down to some other location. They start with the shortcut for *root* ( `/` ). You can only this type of file path to travel *down* a file tree.

- Relative-to-Home Paths (**New!**)
  These paths start from *your home*. For example, if you are logged in as user `student` then your home is `/home/student` . They always start with a tilde ( `~` ). A *relative-to-home* path might also specify a different user. For example, the filepath `~eric.brauer/example` specifies user `eric.brauer` 's home. You should only use this type of file path to travel *down* a file tree.

To demonstrate this, log into Matrix and enter `pwd` .

```
/home/student
```

Your actual path will be different, since everyone is logged in as different users. But this path that you see can be shortened using a tilde ( ~ ).

Try this now:

```
cd ~eric.brauer/uli101
```

Use `ls` and see if you can find a file called `tux.png` .

Now you could copy this to your home very quickly:

```
cp tux.png ~
```

But you might want to use this image on your webpage instead. Copy this file to your `public_html` directory:

```
cp tux.png ~/public_html
```

```
cd ~/public_html
```

Use `ls` to verify that the image was copied. And then use `pwd` to compare the *relative-to-home path* with an *absolute path*.

**~/public_html == /home/student/public_html**

### Relative-to-Home Paths For Other Users

You can also specify the home directories of other users. So for example, you have been accessing the assignments using this path:

```
~uli101/assign1
```

What does this actually mean? Well, there is a user called `uli101` , and that user has a home directory that contains the assignments. So the absolute path for assignment 1 would be:

```
/home/uli101/assign1
```

Here's another demonstration: I'm going to use a file called `frankenstein.txt` to demonstrate commands such as `more` and `less` . If you don't have a copy of this file, I will share mine. Enter this command to get it:

```
cp ~eric.brauer/uli101/frankenstein.txt ~
```

Again, the first argument is the *target*. That's located in *my* home. The destination is *your* home.

---

## Echo

Let's go over some ways that text can be *put into* and *read from* files. This will also introduce our first *file redirection*.

The `echo` command behaves... like an echo.

```
echo Hello
```

```
Hello
```

We just asked the shell to send what we wrote to *STDOUT*, or standard output. *STDOUT* is what we've been reading so far, as we explore the file system and so on. By default, anything sent to *STDOUT* is displayed on our terminal. This isn't too helpful yet, but it will come in handy when we get into variables!

---

## File Redirection

Instead of using `echo` with *STDOUT*, let's send it into a file that we can call 'friendly'.

`>` = Redirect (Write).

```
echo Hello > friendly
```

Did you see anything printed after you ran the command? Hopefully not, since we asked the shell to *redirect* the output.

Use `ls` to find the 'friendly' file.

---

## Reading the Contents of Files

How do we read from the file? There are several ways. Let's start with the simplest.

```
cat friendly
```

```
Hello
```

`cat` is short for `concatenate`, and it is also the command we would use if we wanted to *combine* two files.

Let's try it again:

```
echo "Hello World" > friendly
```

```
cat friendly
```

```
Hello World
```

---

## Append Redirection

Notice that the previous output was *overwritten* by our most recent command. This can be dangerous, if you are echoing into a configuration file.

`>>` = Redirect (Append). This will *add* to the end of a file, rather than overwrite.

```
echo 'Hello, User!' >> friendly
```

```
Hello World
Hello, User!
```

This will also work if you are creating a new file:

```
echo "Hello World" >> friendly2
```

```
cat friendly2
```

```
Hello World
```

---

## Other Ways of Reading Files

`cat` works well when the file isn't too big. But how does it work for larger files? Try this:

```
cat ~eric.brauer/uli101/frankenstein.txt
```

You'll find that all the text flys past without any way to slow it or to navigate. This is when it's useful to switch to another command. Use the up arrow key to find your last command. Use **Ctrl+a** to move to the start of this command, and change `cat` to:

```
more ~eric.brauer/uli101/frankenstein.txt
```

or...

```
less ~eric.brauer/uli101/frankenstein.txt
```

Of the two, `less` is much easier to use. Use arrow keys, to navigate, q to quit. You can also search for a word by pressing the **/** key and entering a search term. (Use **n** to navigate to the next instance of the search term).

---

## Using the man command

Everything you just learned about `less` is applicable to `man`. `man` should be the first place you look for command help. Every command in Linux has a man page.

```
man cal
```

This will print a manual page where you can read about the various options for using the `cal` command.

`/-w` (followed by Enter)

The next time you run `cal`, add `-w` to see how this works.

Another useful feature of `man` is that you can search to find a tool you might need. Use `-k`.

```
man -k calendar
```

```
 cal (1)              - displays a calendar and the date of Easter
calendar (1)         - reminder service
gcal (1)             - a program for calculating and printing calendars.
ncal (1)             - displays a calendar and the date of Easter
zshcalsys (1)        - zsh calendar system
```

---

## diff Command

Let's use a couple files we created already:

```
 cat friendly
```

```
 Hello World
Hello, User!
```

```
 cat friendly2
```

```
 Hello World
```

To quickly check the differences in files:

```
 diff friendly friendly2
```

```
1,2c1
< Hello World
< Hello, User!
---
> Hello World
```

Another, more useful version is with `-y`.

```
 Hello World                             | Hello World
Hello, User!                           <
```

There are more ways of making `diff` output useful. Refer to the man pages.

---

## find Command

The `find` command does exactly what you expect: it *finds* the files that you have misplaced. This is a crucial problem-solving tool and from this day forward, it should be a tool that you use before anything else!

The `find` tool will require at least two arguments: the place to search and what to search for. Let's use this directory structure for an example:

```
 sample_dir <-- You are here
├── admin
├── cambridge
│   ├── cafeteria
│   └── library
│       └── dir_practice
├── faculty
├── markham
```

```
│   ├── annex
│   ├── annex2
│   ├── building1
│   └── parking
├── outline.doc
├── oxford
│   ├── outline.doc
│   ├── programming
│   │   └── report.pdf
│   └── security
└── stenton
    ├── gen_ed
    │   ├── cars
    │   ├── cars2
    │   └── link-cars
    ├── lib_arts
    │   ├── english.txt
    │   └── match.doc
    └── parking2

12 directories, 14 files
```

Take a moment to understand the file structure. We know that 'cambridge', 'markham', 'oxford' and 'stenton' are all directories, and 'gen_ed', 'lib_arts' are subdirectories of 'stenton'. Additionally, 'programming' and 'library' are subdirectories of 'oxford' and 'cambridge' respectively.

Let's construct our first `find` command. The first argument is where we want to start searching. This search will always be *recursive*, meaning we will be searching all subdirectories. Let's start the search from our *current location*, for which we can use our shortcut **.**

Our second argument will be what we are searching for. We know what to look for, we are looking for 'report.pdf'. So let's plug that in.

```
find . -name report.pdf
```

```
./oxford/programming/report.pdf
```

The answer appeared right away! Let's try a broader search:

```
find ~ -name report.pdf
```

This time, we starting the search in '/home/eric', and the search takes quite a bit longer.

```
find: ‘/home/eric/.cache/dconf’: Permission denied
find: ‘/home/eric/.dbus’: Permission denied
/home/eric/ULI101/sample_dir1/sample_dir/oxford/programming/report.pdf
```

We find our file again, and we also get a few error messages because we don't have permission to be reading from '.dbus' or 'dconf'. This is normal. One more search, but this time it will be *very* broad. It would also return so many error messages that it would be hard to find the actual result, so I'm going to add a bit to the end.

```
find / -name report.pdf 2>errors.log
```

```
/home/eric/ULI101/sample_dir1/sample_dir/oxford/programming/report.pdf
/home/student/Documents/sample_dir1/sample_dir/oxford/programming/report.pdf
```

This time, I searched in root, so the entire file system was included. This search took the longest of all, but I found another result in another user's home folder. I also added another *redirect*, but this time what I redirected was *STDERR*, which stands for 'standard error'.

To see what that looks like, use `cat errors.log` to read our error log.

```
find: ‘/snap/core/5328/var/lib/waagent’: Permission denied
find: ‘/snap/core/5328/var/spool/cron/crontabs’: Permission denied
find: ‘/snap/core/5328/var/spool/rsyslog’: Permission denied
find: ‘/snap/core/5145/etc/ssl/private’: Permission denied
find: ‘/snap/core/5145/root’: Permission denied
find: ‘/snap/core/5145/var/cache/ldconfig’: Permission denied
find: ‘/snap/core/5145/var/lib/machines’: Permission denied
find: ‘/snap/core/5145/var/lib/waagent’: Permission denied
find: ‘/snap/core/5145/var/spool/cron/crontabs’: Permission denied
find: ‘/snap/core/5145/var/spool/rsyslog’: Permission denied
find: ‘/snap/core/4917/etc/ssl/private’: Permission denied
find: ‘/snap/core/4917/root’: Permission denied
find: ‘/snap/core/4917/var/cache/ldconfig’: Permission denied
find: ‘/snap/core/4917/var/lib/machines’: Permission denied
find: ‘/snap/core/4917/var/lib/waagent’: Permission denied
find: ‘/snap/core/4917/var/spool/cron/crontabs’: Permission denied
find: ‘/snap/core/4917/var/spool/rsyslog’: Permission denied
find: ‘/home/apt’: Permission denied
find: ‘/home/lost+found’: Permission denied
find: ‘/home/eric/.cache/dconf’: Permission denied
find: ‘/home/eric/.dbus’: Permission denied
... and so on and so on.
```

How long is this file?

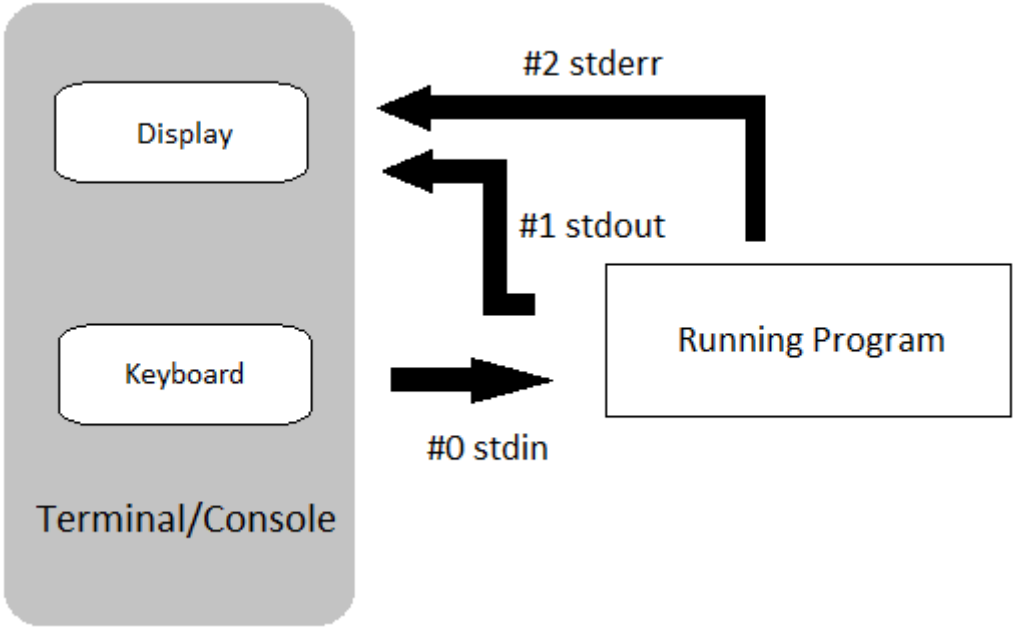
```
wc -l errors.log
```

```
1869 errors.log
```

1869 error messages! **Note**: Use the `man` tool to discover what `wc` is and what `-l` will do.

## Introduction to Standard Streams

Let's a closer look at streams:




From Linux Unit

By default, when we run programs we are getting a mix of *STDOUT* and *STDERR* sent to the display. But as we've seen, it's possible to redirect these streams, for example when we redirected output from `echo` into a file called 'friendly'.

```
output > file
```

Note that there is an implicit '1' in front of that redirect symbol.

Similarly, just now we redirected error output into a log file.

```
output 2> error.log
```

Or maybe we don't care at all about those error messages, in which case we can throw them into a black hole and forget about them:

```
output 2> /dev/null
```

What about *STDIN*? We can demonstrate that with our `cat` program. When you call `cat` without a file, it will listen on *STDIN*. You can try this now, and press Ctrl+d when you want to stop.

```
 Hello
Hello
Is
Is
There
There
An
An
Echo?
Echo?
```

Streaming is at the heart of what makes Linux so powerful, once we get into pipes. For now, be aware of the idea of streams!

Another Useful Link for Understanding Streams

# Summary

## File Paths

- Relative Paths
  These paths specify a path from your *current location* to some other location. They usually start with either the shortcut to your *parent directory* ( `..` ), or with a *subdirectory* in your current location.

- Absolute Paths
  These paths specify a path starting from the *top* of your file system down to some other location. They start with the shortcut for *root* ( `/` ).

- Relative-to-Home Paths
  These paths start from *your home*. For example, if you are logged in as user `student` then your home is `/home/student` . They always start with a tilde ( `~` ). A *relative-to-home* path might also specify a different user. For example, the filepath `~eric.brauer/example` specifies user `eric.brauer` 's home.

> - **STDIN**: Standard Input *into* a program.
> - **STDOUT**: Standard Output, by default it is sent to the display.
> - **STDERR**: Standard Error, by default is *also* is sent to the display.

> ## Commands
>
> - echo: display a line of text
> - cat: dump the contents of a file to STDOUT
> - more: a scrollable way to read files
> - less: another scrollable, searchable way to read files
> - diff: spot the differences between two files
> - man: manual for programs
> - find: find a missing file
> - wc: word count

> ## Shortcuts
>
> - . : current directory
> - .. : parent directory
> - ~ : home directory for current user
> - / : root directory

> ## Redirects
>
> - > Redirect STDOUT and overwrite
> - >> Redirect STDOUT and append
> - 2> Redirect STDERR