Lecture Notes:

These are from the lectures covering UNIX/LINUX. This document goes from chmod to just before covering directories in Linux pt.1 slides.

```
$ echo $0 #shows what shell you're using
-bash

$ which sh
/bin/sh

$ which bash
/bin/bash

$ echo $0
-bash
$ bash
$ echo $0
bash #no longer using bash as the login shell

$ ps jf #gives a process status to see what's going on
PPID PID  PGID SID  TTY    TPGID STAT   UID  TIME COMMAND
7063 7064 7064 7064 pts/11 6252  Ss    7843 0:00 -bash #login shell bash
7064 6049 6049 7064 pts/11 6252  S     7843 0:00 \_bash #created this subshell
6049 6252 6252 7064 pts/11 6252  R+    7843 0:00   \_ps jf
$ exit
exit
$ ps jf
PPID PID  PGID SID  TTY    TPGID STAT UID  TIME COMMAND
7063 7064 7064 7064 pts/11 6252 Ss   7843 0:00 -bash #using login shell
6049 6252 6252 7064 pts/11 6252 R+   7843 0:00   \_ps jf
```

When you see the hyphen in the command name (-bash) you know that's the login shell.

```
$ tcsh
> echo $0
tcsh
> exit
#using tcsh shell and exiting from it

$ zsh
% echo $0
zsh
% exit
#using z shell
```

- “.” and “..” is universal to every file system (hard links)
- “.” refers to the current directory. self-referential.
- “..” refers to the parent directory, useful to move things up without knowing the exact names.

```
fox01:~$ mv file.txt .. #moves file to the parent
fox01:~$ mv file.txt ../.. #moves to the parent of the parent
```

```
$ echo -e “line one\n\n\nline two”
  line one
  #first \n
  #second \n
  #third \n
  line two
```

- “~” has a special meaning (home directory)

ex:

```
fox01: ~$ #indicates you’re in the home directory
$ pwd
/home/ssilvestro #takes you to home directory
```

- “echo” is a built-in command and every shell supports it

```
$ echo Hello, world
 Hello, world
#prints out the input from echo
```

```
fox01:~$ echo ~
/home/ssilvestro
#taking the ~ and replacing it with the $HOME value of the variable

if you want to print the actual ~ you can cancel it via blackslash
fox01:~$ echo ~
/home/ssilvestro
fox01:~$ echo \~
~
#putting it as “~” or ‘~’ will also bypass expansion.

fox01:~$ bash echoargs.sh this is a test
arg 1 = this
arg 2 = is
arg 3 = a
arg 4 = test

fox01:~$ bash echoargs.sh “this is” “a test”
arg 1 = this is
arg 2 a test
```

```
fox01:~$ bash echoargs.sh “this is a test”
arg 1 = this is a test
```

If you use filename completion in the shell, for spaces it’ll just “escape the spaces”. if you escape the spaces, they’re no longer used to tokenize the command line/break it apart.
So for the above example you can get the same thing by:

```
fox01:~$ bash echoargs.sh this\ is\ a\ test
arg 1 = this is a test
```

```
fox01:~$ cp cs34_ #The professor has 3 different files beginning with cs3423. by hitting tab, it auto completes the “cs3423”. after specifying further via: cs3423_su and hitting tab here again, it fully completes the name of the file.
```

```
fox01:~$ cp cs3423_su20_assign2-solvn-v1.0.4.zip .. #copying this to the parent directory
```

using mv will move it to another directory such as:

```
fox01:~$ mv cs3423_su20_assign2-solvn-v1.0.4.zip ../..
```

-e means “escape sequence”

```
fox01:~$ echo “a\n\n\nb”
a\n\n\nb
fox01:~$ echo -e “a\n\n\nb”
a
#first \n
#second \n
b
```

Line Suppression:

```
fox01:~$ echo hello world
hello world
fox01:~$ echo -n hello world
hello worldfox01:~/courses/cs/3423$
```

Shell Processing Steps:

1. Reads an input line

   ```
   fox01: ~$ echo hello world
   hello world
   #it reads the line after pressing enter
   ```

2. Breaks the input line into tokens based on spaces
   It begins to tokenize that line, so any amount of any white space counts as a separator.

Step 2a. Understands quotes and escapes:
ex:
fox01:~$ **echo hello world**
hello world
Despite the spaces it still has the same output because echo is token 1, hello is token 2, and world is token 3
fox01:~$ **echo “hello world”**
hello world

Step 2b. Expands aliases
fox01:~$ **alias hw=”echo ‘Hello, world’”**
fox01:~$ **alias hw**
Hello, world

3. Parses the tokens into simple and compound commands (separated by “|”, “;”, “&&”, “||”)
   fox01:~$ **echo A; sleep 2; echo B; sleep 2; echo C**
   A
   B
   C
4. Performs expansions
5. Performs redirections and removes the redirection tokens from the command argument list.
   If they were left in there, the program would be confused.
   fox01:~$ **cat < test.c**
   #this prints test.c, but the program removes < because cat doesn’t know what that is and would try to open it. This is because < is meaningful to bash, not to cat.
6. Executes the command, passing the expanded parameter list.
   fox01:~$ **slklskdfksd $USER < test.c**
   We could do steps 1 - 5, and still not know whether or not the command will execute. We could do the above and and then fail because there is no command ‘slklskdfksd’
   output would be:
   slklskdfksd: command not found
7. Optionally waits for command completion and exit status
   fox01:~$ **sleep 5.**
   #the command prompt would wait 5 seconds before spawning another
   fox01:~$

   fox01:~$ **sleep 5. &** #gives you your prompt back immediately, & is also a command separator.
   output:
   [1] 1015

#bash is saying you have [1] process running behind the scenes and its process ID is 1015

Whenever you hit enter after 5 seconds, bash will notify you that the process completed successfully

output:

[1]+ Done sleep 5

fox01:~$ jobs #will show any other processes