1 Intro

- We use computers to do two mechanisms:
  - Display information
  - Perform operations
- We use the shell perform various tasks:
  - Read commands and interpret them
  - OS interface that gets the OS to do things.
- There are 2 main approaches to shells: graphical shell (GUI) versus a command shell (CLI)
- Unix uses two types, both similar/equivalent:
  - sh (ksh, bash) (what we use)
  - csh (tcsh)

2 Linux File System Management

- / is the root.
- Typical subdirectories of the root are:
  - bin (bash, ls), etc (shell)
  - home (username; files that we create on our account are typically stored here).
  - usr (bin, include, share).
- Due to Marmoset being fickle, make sure EVERY file ends with a newline!
- If using Vi/Vim, it should do this for you automatically, but double check to make sure it fits the project's requests.
- Absolute path: starts from root, NOT relative from somewhere else - thus, it will start from /, and isn't based on where it is called.
- We note that the shell will mention your absolute path to your current directory.
- Relative path: starts from where it is called.
  - . represents the current directory
  - .. represents the parent directory OF the current directory
  - ~/... represents the user home directory
  - ~userid/... represents userid's home directory

- More useful bash commands:
  - ls lists all files in current directory We use the -a parameter to show hidden files.
  - mkdir name creates a child directory at your cd
  - rmdir name deletes a directory, BUT it must be empty.
  - rm deletes a file or directory. Use the -r parameter (recursively deletes) to delete all child files, and the directory you specified.
  - "*" represents a wildcard, we can use this to take EVERYTHING that matches before/after the wildcard character (ie: ls -al *.txt) (aka globbing)
  - A slightly more difficult example is something like "ls -al *f*.*". This would list ALL files, hidden or not, that have an f and are files.
  - "echo TEXT" will output to the console whatever you put after it. You could combine this with wildcard — for example, echo *.txt" would print out the file names of all files that are txt files. Note *.txt would print out *.txt, NOT the files!
  - "?" represents wildcard matching for EACH character. For example, let's say we had three folders - CS111, CS 232, CS 246. Putting "echo CS???" in bash would output these three folders. Basically, it must have the same number of ? as the number of characters you are searching for.
  - On a side note, if echo with ? fails to find anything, then it'll just print out the string you were searching for. For example, "echo CS?" will output "CS?" if it finds nothing.
  - [xy]is a pattern for anything that starts with x or y.
  - CS246,1 will look for any file that starts with CS and follows 246 OR 1. The difference is that [] doesn't use commas, BUT this would be an issue with anything longer than one character. requires commas, but works with multicharacter values.
  - cat [filename] displays the content of that file if it exists. Use the -n argument to number the lines.
  - Every process we preform is connected to three streams:
  - stdin (standard input, defaults to the keyboard's input)
  - stdout (standard output, defaults to the screen)
  - stderr (default: screen)
  - Note that cat ¡ file and cat file HAVE A DIFFERENCE. The former is the shell opening it, the latter is cat opening it. We can see this by opening two files at once.
  - We can simulate copying files by doing something like cat f1 ¿ f2 - this is equal, in this case, to cp f1 f2.

- Let's say we wanted to redirect the error message of catting a non-existant file. We would do NOT cat fake.txt ¿ res.txt, but rather cat fake.txt 2¿ res.txt. Note the latter will still create res.txt, just not copy the error message to it.
- Note that this type of redirection overwrites the file.
- We can use head # file and tail # file to specify how many lines from the top or bottom (respectively) to print.
- Piping: It's literally the — thing. Basically it allows us to use a command on the result of another command, say, head -4 test.txt — grep hello
- "uniq file.txt" represents omitting adjacent repeating lines.
- "sort file.txt" will output a sorted file.txt.

3 Pattern Matching

- (e)grep -¿ extended global regular expression print. Identical to grep -e.
- Format is egrep [pattern] [filename].
- A pattern with no special characters (ie; test) will just print out all lines with that string. No need for quotations.
- We can use a pattern like word1—word2 to search for all lines with word1 and/or word2.
- We could extend this - word(1—2) would do the same as above.
- Something we could do is [Ww]ord[12] as well, to, say, detect both capital and lowercase, and track 1 or 2.
- [ˆ..]means not.
- ? means 0 or 1 of any character that is preceding it.
- * means 0 or more of any character preceding it.
- + means 1 or more of any character preceding it.
- . means any non-line-break character (ie: .+ is any non-empty string).
- ...1—10 would mean 1 to 10 occurances of the character preceding it.
- ˆ... (no brackets) means anything that starts with the following string.
- ...$ matches the end of the line.
- ie: chmod a=rx filename
- ie: chmod a+x filename

4 File Management

- ls -l will give you a more detailed list of the files within the directory.
- In the first column, we get a 10 bit line showing what each file/directory can do.
- An example would be something like -rw-r–r–, or something of that sort.
  - - or d means file or directory
  - : rwx for user (read write execute)
  - : rwx (group)
  - : rwx (other)
- In the second column, we get the owner of the file.
- Third column is the group.
- Fourth column is the size in bytes.
- Fifth is the last modified date.
- We use chmod to modify permissions:
- u, g, o, a for group.
- +, =, - to add, equate, or remove permissions.
- r, w, x, for read, write, execute.
- ie: chmod u-w filename
- We can define variables within the shell. For example, x=1.
- echo $x will now print 1.
- In the shell, all these values are of type string.
- We could save directories: dir=~/cs246, for example.
- “PATH” is a predefined variable that will tells Linux where to search for executables.
- A good practice is to use $ to fetch the value of a variable.
- When we create a script, by default, it will NOT have executable permissions. We can easily fix this with chmod.
- Remember to always include “#!/bin/bash” to indicate that this is a bash script.
- We pass arguments to our scripts using $1, $2...
- We can use redirect outputs that we do not want to show to the user to /dev/null. It’s like tmp.

- By default, if an operation is completed successfully, the command line returns 0. If it fails, it returns /!/= 0.
- We can use this to our adavantage, and create if/else statements.
- "$?" holds the status of the most recently executed command line.
- "-eq" checks for equality with left and right sides.
- $# tells us the number of arguments.
- -ne is not equal.
- $0 represents the command line itself.
- <&2 represents stderr.
- exit x ends the program with code x - 0 represents it exited fine, a non-zero value means it failed to run properly.
- Bash if chain goes like this:

```
if [cond]; then CMD
elif [cond]; then CMD
else CMD
fi
```

- Bash while loops go like this:

```
while [condition]; do
CMD
For an accumulator, say, x:  x=\$((x+1))
done
```

- Bash for loops:

```
for varname in listname do
CMD
done
```

- Observe the following shell script:

```
#!\bin\bash
for name in *.cpp; do
    mv ${name} ${name%cpp}cc
done
```

- ${name%cpp}cc produces the value name without the trailing "cpp", and then we replace it with "cc".
- We can create and use functions in scripts.