CS107 / AC207

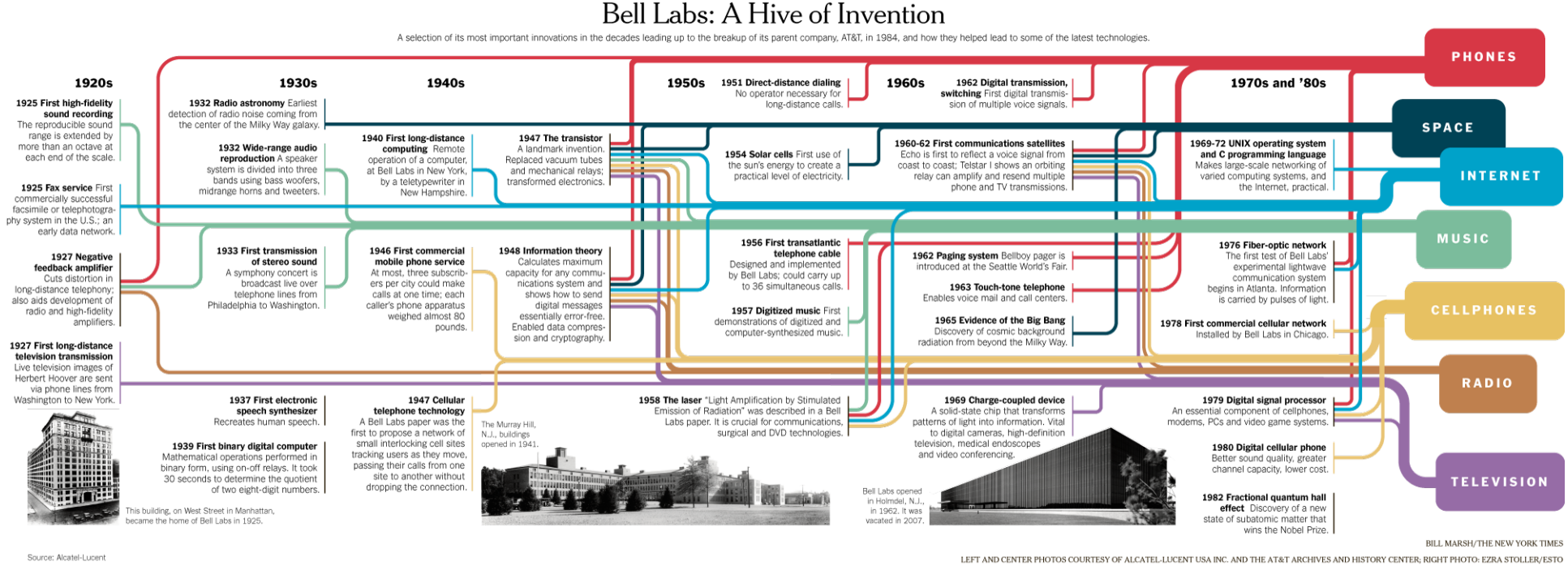
SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 1

Tuesday, September 7th 2021

RECAP OF LAST TIME

- Course introduction and policies
- Bell labs and its impact on the computer as we know it today
- Gentle transition from Unix to Linux
- How to list content with the `ls` command (list)



OUTLINE

- More on Linux commands and the `man`-pages
- Working with the shell
- Regular expressions and `grep`
- File attributes and `find`ing files
- Short journey into text editors

PAIR-PROGRAMMING SECTIONS

- Pair-programming is attendance graded and we check the work you push to your GitHub account.
- You must attend *one* section per week (cycle).
- ***We implement a 5-minute late tolerance.*** After 5 minutes past the section start it will not be possible anymore to join the ongoing section.

The pair-programming cycles ***start on Friday*** morning (new PP-exercises handed out) and ***end on Thursday*** after the last section.
Hand-in deadline of the PP-exercise is the following Thursday.

YOUR GITHUB REPO FOR THE CLASS

- Your GitHub repo for the class should be *private* and follow the naming convention `cs107_firstname_lastname`. It should look similar to this:

```
$ tree cs107_fabian_wermelinger
cs107_fabian_wermelinger/
├── homework
│   └── HW1
│       └── HW1-final
├── lectures
├── pair_programming
│   └── PP1
│       └── README.md
└── README.md
```

- For the HW, you work on a branch called `HWn-dev` where `n` is the homework number. For the current homework the branch is `HW1-dev`.
  You do not need to create pull-requests for pair-programming exercises.
- Put your HW solution file(s) inside the `HWn-final` directory and commit it on the `HWn-dev` branch. Create a pull-request for merging branch `HWn-dev` into your `main` or `master` branch.
- The teaching staff will grade and provide feedback to you via the ***open*** pull-request. ***Do not merge this pull-request until you have received our feedback.***
- See https://harvard-iacs.github.io/2021-CS107/pages/coursework.html

LINUX COMMANDS AND GETTING HELP

UNIX / LINUX CHEAT SHEET

FILE SYSTEM

- `ls` — list items in current directory
- `ls -l` — list items in current directory and show in long format to see perimissions, size, and modification date
- `ls -a` — list all items in current directory, including hidden files
- `ls -F` — list all items in current directory and show directories with a slash and executables with a star
- `ls dir` — list all items in directory dir
- `cd dir` — change directory to dir
- `cd ..` — go up one directory
- `cd /` — go to the root directory
- `cd ~` — go to to your home directory
- `cd -` — go to the last directory you were just in
- `pwd` — show present working directory
- `mkdir dir` — make directory dir
- `rm file` — remove file
- `rm -r dir` — remove directory dir recursively
- `cp file1 file2` — copy file1 to file2
- `cp -r dir1 dir2` — copy directory dir1 to dir2 recursively
- `mv file1 file2` — move (rename) file1 to file2
- `ln -s file link` — create symbolic link to file
- `touch file` — create or update file
- `cat file` — output the contents of file
- `less file` — view file with page navigation
- `head file` — output the first 10 lines of file
- `tail file` — output the last 10 lines of file
- `tail -f file` — output the contents of file as it grows, starting with the last 10 lines
- `vim file` — edit file
- `alias name 'command'` — create an alias for a command

SYSTEM

- `shutdown` — shut down machine
- `reboot` — restart machine
- `date` — show the current date and time
- `whoami` — who you are logged in as
- `finger user` — display information about user
- `man command` — show the manual for command
- `df` — show disk usage
- `du` — show directory space usage
- `free` — show memory and swap usage
- `whereis app` — show possible locations of app
- `which app` — show which app will be run by default

COMPRESSION

- `tar cf file.tar files` — create a tar named file.tar containing files
- `tar xf file.tar` — extract the files from file.tar
- `tar czf file.tar.gz files` — create a tar with Gzip compression
- `tar xzf file.tar.gz` — extract a tar using Gzip
- `gzip file` — compresses file and renames it to file.gz
- `gzip -d file.gz` — decompresses file.gz back to file

PROCESS MANAGEMENT

- `ps` — display your currently active processes
- `top` — display all running processes
- `kill pid` — kill process id pid
- `kill -9 pid` — force kill process id pid

SEARCHING

- `grep pattern files` — search for pattern in files
- `grep -r pattern dir` — search recursively for pattern in dir
- `grep -rn pattern dir` — search recursively for pattern in dir and show the line number found
- `grep -r pattern dir --include='*.ext` — search recursively for pattern in dir and only search in files with .ext extension
- `command | grep pattern` — search for pattern in the output of command
- `find file` — find all instances of file in real system
- `locate file` — find all instances of file using indexed database built from the updatedb command. Much faster than find
- `sed -i 's/day/night/g' file` — find all occurrences of day in a file and replace them with night - s means substitude and g means global - sed also supports regular expressions

PERMISSIONS

- `ls -l` — list items in current directory and show permissions
- `chmod ugo file` — change permissions of file to ugo - u is the user's permissions, g is the group's permissions, and o is everyone else's permissions. The values of u, g, and o can be any number between 0 and 7.
- `7` — full permissions
- `6` — read and write only
- `5` — read and execute only
- `4` — read only
- `3` — write and execute only
- `2` — write only
- `1` — execute only
- `0` — no permissions
- `chmod 600 file` — you can read and write - good for files
- `chmod 700 file` — you can read, write, and execute - good for scripts
- `chmod 644 file` — you can read and write, and everyone else can only read - good for web pages
- `chmod 755 file` — you can read, write, and execute, and everyone else can read and execute - good for programs that you want to share

NETWORKING

- `wget file` — download a file
- `curl file` — download a file
- `scp user@host:file dir` — secure copy a file from remote server to the dir directory on your machine
- `scp file user@host:dir` — secure copy a file from your machine to the dir directory on a remote server
- `scp -r user@host:dir dir` — secure copy the directory dir from remote server to the directory dir on your machine
- `ssh user@host` — connect to host as user
- `ssh -p port user@host` — connect to host on port as user
- `ssh-copy-id user@host` — add your key to host for user to enable a keyed or passwordless login
- `ping host` — ping host and output results
- `whois domain` — get information for domain
- `dig domain` — get DNS information for domain
- `dig -x host` — reverse lookup host
- `lsof -i tcp:1337` — list all processes running on port 1337

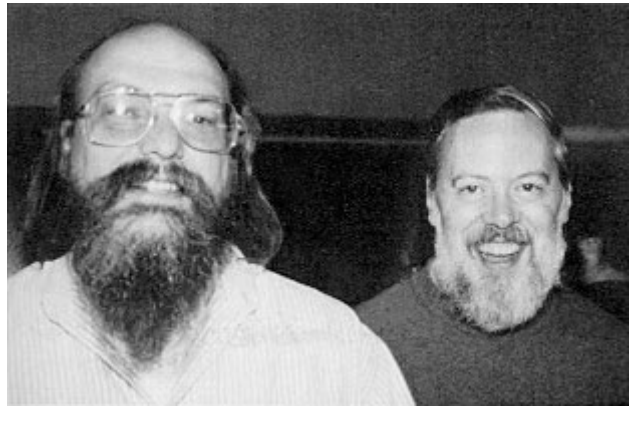
SHORTCUTS

- `ctrl+a` — move cursor to beginning of line
- `ctrl+f` — move cursor to end of line
- `alt+f` — move cursor forward 1 word
- `alt+b` — move cursor backward 1 word

LINUX COMMANDS AND GETTING HELP

There are *numerous* commands available in Linux. They are so numerous because of the core Unix philosophy:

Every command in Unix/Linux does *exactly one* job. In other words, this implies **modularity** and **reusability**. Once you have digested this principle, you will *love it*!



Ken and Dennis, 1973 (wiki)

LINUX COMMANDS

The commands you will likely need most often:

`ls`	List directory contents
`cd`	Change directories
`mkdir`	Create directories
`rm`	Remove files and directories. *Be very mindful with this command! Unlike other OSs, there is **no trash bin** in Linux.*
`cp`	Copy files and directories
`rsync`	Remote (and local) file sync tool. This tool will be your friend.
`ln`	Create links to files and directories
`grep`	Search file contents for a pattern. This tool is very important and you will use it often. A faster alternative might be `ripgrep`.
`find`	Find files in the file system
`cat`	Concatenate files and print to `stdout`

LINUX COMMANDS

These are already 10 commands. Looking at all of them in detail is not efficient. You will learn these commands most efficiently by **practice**. Once you use them daily, they will become second nature to you.

Command names in Unix/Linux are a mnemonic of *what they do* (**recall:** they have only one job to do). The ancient ones are 2-3 letters short because typing on the Teletype Model 33 was a finger gym.

Finally, one very important command is missing: `man` gives you the manual pages (documentation) of every Linux command.

GETTING HELP

Manual pages are obtained using: `man <command name>`

- The manual page of `man` is:

```
$ man man  # get the manual page for man itself. MAN(1) refers to section 1
MAN(1)                          Manual pager utils                          MAN(1)

NAME
       man - an interface to the system reference manuals
...
```

- `man` pages are split into 9 numbered sections (see `man man`):
  1. Executable programs or shell commands
  2. System calls (functions provided by the kernel)
  3. Library calls (functions within program libraries)
  4. Special files (usually found in `/dev`)
  5. File formats and conventions, e.g. `/etc/passwd`
  6. Games
  7. Miscellaneous (including macro packages and conventions)
  8. System administration commands (usually only for `root`)
  9. Kernel routines (Non standard)

GETTING HELP

If you do not specify a section, `man` will default to section 1:

```
$ man printf
PRINTF(1)                          User Commands                          PRINTF(1)

NAME
       printf - format and print data

SYNOPSIS
       printf FORMAT [ARGUMENT]...
       printf OPTION
...
```

GETTING HELP

Or you can specify the section number explicitly:

```
$ man printf
PRINTF(1)                          User Commands                          PRINTF(1)

NAME
       printf - format and print data

SYNOPSIS
       printf FORMAT [ARGUMENT]...
       printf OPTION
...

$ man 3 printf # explicitly specify the section number with the first argument
PRINTF(3)                      Linux Programmer's Manual                      PRINTF(3)

NAME
       printf,  fprintf,  dprintf, sprintf, snprintf, vprintf, vfprintf, vdprintf, vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
```

GETTING HELP

You can use the `whatis` command to find out more about particular `man`-page entries for a command:

```
$ whatis whatis
whatis (1)           - display one-line manual page descriptions
$ whatis printf
printf (3)           - formatted output conversion
printf (1)           - format and print data
printf (1p)          - write formatted output
printf (3p)          - print formatted output
```

GETTING HELP

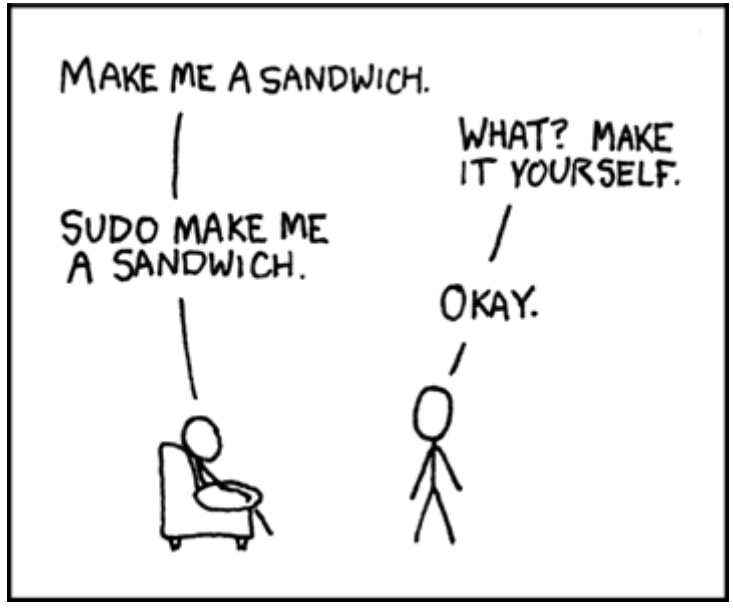
Try out the `man` command with your neighbors:

- Pick a Linux command that you would like to investigate. A few examples are given below.
- Read about it using the `man` command
- Make sure you can all provide a short summary of what it does
- What is one interesting option that this command provides?

`ls`, `cp`, `mv`, `ln`, `rm`, `du`, `df`, `wc`, `ps`, `id`, `w`, `vi`, `bc`, `pwd`, `sh`, `chsh`, `bash`, `csh`, `ksh`, `env`, `ssh`, `ssh-keygen`, `man`, `whatis`, `whereis`, `which`, `stat`, `info`, `make`, `sudo`, `echo`, `sort`, `cut`, `uniq`, `sed`, `awk`, `cat`, `tac`, `tar`, `zip`, `unzip`, `head`, `tail`, `gcc`, `top`, `dstat`, `ulimit`, `history`, `passwd`, `useradd`, `usermod`, `userdel`, `mkdir`, `rmdir`, `touch`, `rsync`, `grep`, `find`, `diff`, `jobs`, `kill`, `chmod`, `chown`, `time`, `date`, `sleep`, `mount`, `ping`, `ex`, `pico`, `nano`, `vim`, `reboot`, `shutdown`, `halt`

WORKING WITH THE SHELL

There is this long lasting joke...



Which translates to this in the shell:

```
[wife@husband]$ make sandwich         # husband@wife would also be valid user and host names ;)
make: cannot make target 'sandwich': Permission denied
[wife@husband]$ sudo make sandwich    # see also `man sudo` and `man make`
```

https://en.wikipedia.org/wiki/Sudo

RUNNING A PROGRAM

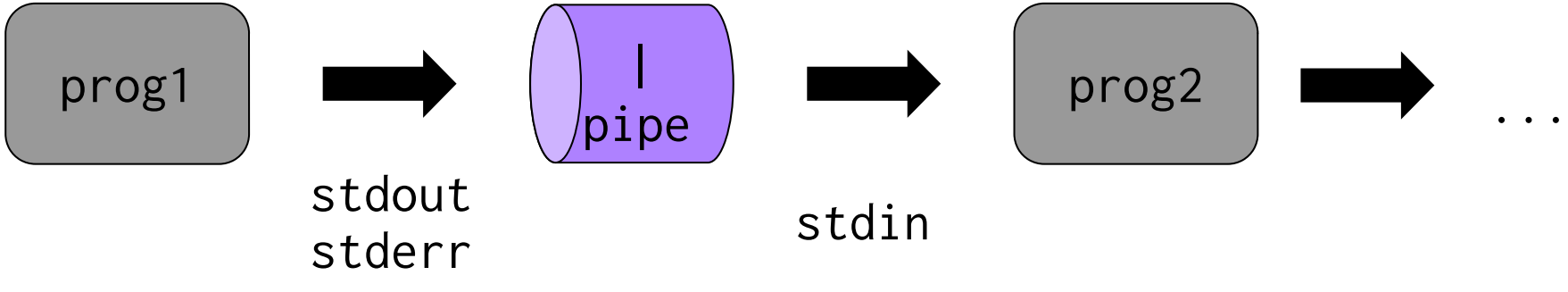
Recall: the shell offers you a prompt to input a character sequence which will be interpreted after you press `enter`.

- The shell reads the character sequence, locates the program(s) and executes it by passing the argument(s) you have specified
- There are **three** standard I/O streams:
  1. Standard *input*: `stdin` (associated to file descriptor 0)
  2. Standard *output*: `stdout` (associated to file descriptor 1)
  3. Standard *erro*: `stderr` (associated to file descriptor 2)

  Also see: `man stdin` (covers all three)
- *File descriptor*: is a reference in the kernel for *open* files. There is a limit to how many files you can have open at the same time. See `ulimit -a` for how many. (The currently open file descriptors are listed in the `/dev/fd` directory.)

UNIX PIPES

- **Recall:** Unix philosophy is one program for a particular task
- Traditional Unix programs therefore act like *filters*
- Most of the time you need *multiple* filters to achieve the desired transformation of your data.
- **How would you achieve that?**
- You need a notion to *connect* the `stdout/stderr` (either one or both) to the `stdin` of the following program
- The notion for this is the "`|`" character (Unix *pipe*)
- `prog1 [args] | prog2 [args]`



UNIX PIPES

Pipes are extremely powerful and comprise a core component in the Unix philosophy.

This is a wonderful trip down memory lane: https://www.youtube.com/watch?v=tc4ROCJYbm0&t

Assume you have the following list of students:

```
$ cat student_list.txt
FirstName  LastName  Seniority  Major
Jane       Smith     Grad       CompSci
Joe        Bloggs    Undergrad  Bio
Ruth       Schmoe    Undergrad  Math
John       Doe       Grad       MechEng
```

You want to create a new list with grad students only and you would like them sorted by last name.

What filter steps are required to achieve this goal?

UNIX PIPES

Assume you have the following list of students:

```
$ cat student_list.txt
FirstName  LastName  Seniority  Major
Jane       Smith     Grad       CompSci
Joe        Bloggs    Undergrad  Bio
Ruth       Schmoe    Undergrad  Math
John       Doe       Grad       MechEng
```

Solution:

1. Print only lines which have seniority `Grad`
2. Sort the second column of input alphabetically
3. Redirect result to a file

```
$ cat student_list.txt | grep Grad | sort -k2 >grad_student_list.txt
$ cat grad_student_list.txt
John         Doe       Grad       MechEng
Jane         Smith     Grad       CompSci
```

UNIX PIPES

Solution:

1. Print only lines which have seniority `Grad`
2. Sort the second column of input alphabetically
3. Redirect result to a file

```
$ cat student_list.txt | grep Grad | sort -k2 >grad_student_list.txt
$ cat grad_student_list.txt
John         Doe       Grad       MechEng
Jane         Smith     Grad       CompSci
```

Question: Would it be a good idea to sort first and then filter `Grad`?

Answer: Sorting can be an expensive task. If your input data is Megabytes or even larger, reducing the input size for `sort` can be a more efficient approach.

MORE USEFUL COMMANDS

COUNTING WORDS, LINES OR CHARACTERS

If you need to count words, lines or characters in a document, you can use the `wc` utility:

```
$ wc -l grad_student_list.txt  # lines
2 grad_student_list.txt
$ wc -w grad_student_list.txt  # words
8 grad_student_list.txt
$ wc -c grad_student_list.txt  # characters (bytes; 1 ASCII char = 1 byte)
84 grad_student_list.txt
```

- When counting words, be careful with markup languages like LaTeX (see detex)
- Note that `wc -c` counts bytes. (Also works with binary files.)

```
$ ls -l grad_student_list.txt  # check the file size
-rw-r--r-- 1 fabs fabs 84 Aug 27 13:42 grad_student_list.txt
```

MORE USEFUL COMMANDS

FINDING FILES

The `find` command is a powerful tool to search for files in your system. You will need it often, especially in scripts.

- Search for files or directories using the `-type f` or `-type d` options, respectively
- Use a search pattern to only match specific file names
- The "`*`" is called ***wildcard***, your shell expands it to match anything
  Example: to match any python script use `find . -name "*.py"`
- You can execute commands on matches that `find` reports using the `-exec` option

MORE USEFUL COMMANDS

FINDING FILES

- Find directories in current directory:

```
$ find . -type d              # recursively
$ find . -maxdepth 2 -type d  # only current directory and 1 level down
```

- Same for files only:

```
$ find . -type f                  # recursively, all files
$ find . -type f -name "*.py"     # recursively, only files ending with .py
$ find . -type f -name "test*.py" # recursively, files starting with "test" followed by
                                  # any char (zero or more times) and ending with ".py"
$ find . -maxdepth 1 -type f      # only current directory
```

- Execute a command on the returned match

```
$ find . -type f -name "*.py" -exec wc -l {} \;
```

What does the above command do?

MORE USEFUL COMMANDS

FINDING FILES

- Execute a command on the returned match

```
$ find . -type f -name "*.py" -exec wc -l {} \;
```

What does the above command do?

1. Find files (`-type f`) using pattern (`-name "*.py"`), i.e. all python scripts
2. On a match execute (`-exec`) the command `wc -l` (count lines)
   - The "`{}`" is a placeholder for the current match
   - The "`;`" terminates the inline command passed to `-exec`
   - It must be *escaped* because it belongs to the *inline* command, not to the `find` command itself
   - It is usually not needed for single commands or if you use the pipe `|`. You could have written this however:

```
$ find . -type f -name "*.py" -exec wc -l {} \;; # the second ";" terminates find
```

GREP

- `grep` is a historical tool for searching content in files
- It was written by Ken Thompson, where it was originally part of the `ed` text editor
- `ed` uses a text processing language to operate on single lines or globally. The command `g/re/p` searches *globally* for a *regular expression* pattern `re` and then prints (`p`) every line containing the pattern
- The command was so *useful* that the corresponding `ed` code was refactored into a standalone tool called `grep`
- `grep` is *absolutely* essential for searching code bases efficiently
- When your code base is really large a faster alternative could be ripgrep (I use it every day)

GREP

Note that `grep` is case-sensitive by default:

```
$ grep Grad student_list.txt
Jane         Smith     Grad       CompSci
John         Doe       Grad       MechEng
```

```
$ grep grad student_list.txt
Joe          Bloggs    Undergrad  Bio
Ruth         Schmoe    Undergrad  Math
```

```
$ grep -i grad student_list.txt  # use the -i option to ignore case
Jane         Smith     Grad       CompSci
Joe          Bloggs    Undergrad  Bio
Ruth         Schmoe    Undergrad  Math
John         Doe       Grad       MechEng
```

REGULAR EXPRESSIONS

- A regular expression (regex) is a notation for specifying a pattern of text
- Many commands make use of this powerful (but confusing) syntax. E.g. `grep`, `awk`, `sed`, `perl`, `vim`, ...
- Any character is a match, but there are certain special characters that are interpreted differently if they are not *escaped*:

`.`	Matches any one character except a newline
`*`	Matches *zero* or more occurrences of the *preceding* character
`+`	Matches *one* or more occurrences of the *preceding* character
`?`	Matches exactly *zero* or *one* occurrences of the *preceding* character

- Potential confusion 1: your shell has a set of special characters too. Recall the shell wildcard `*`, it behaves **not** the same as the `*` in a regex!
  What is the regex equivalent of the shell wildcard?
  Answer: `.*` (more info on shell wildcards)

REGULAR EXPRESSIONS

- Any character is a match, but there are certain special characters that are interpreted differently if they are not *escaped*:

`.`	Matches any one character except a newline
`*`	Matches *zero* or more occurrences of the *preceding* character
`+`	Matches *one* or more occurrences of the *preceding* character
`?`	Matches exactly *zero* or *one* occurrences of the *preceding* character

- To match a special character, you must escape it with the backslash `\`
  - `a.c` matches aac, abc, acc, ...
  - `a\.c` matches `a.c` literally

REGULAR EXPRESSIONS

More special characters:

`()`	Capture group: `(abc)` matches "`abc`" where you can back-reference the match with `\1` (does not work in all regex dialects)
`|`	Logical "OR": `ab|cd` matches `ab` or `cd`
`{}`	Numeral range of occurrences: `a{5}` match exactly five times, `a{2,}` match two or more times, `a{1,3}` between one and three times
`[]`	Character group: `[abc]` match any of `a`, `b` or `c` *once*, `[abc]*` same as before but *many* different combinations possible, `[^abc]` match anything except `a`, `b` or `c`, `[a-g]` match any character between `a` and `g`. The caret "`^`" after the opening `[` means *negation*. The hyphen "`-`" specifies a *range*, e.g., `[0-9]` any number between `0` and `9` once

REGULAR EXPRESSIONS

Convenience classes:

`\d`	Matches a digit `[0-9]`
`\D`	Matches a non-digit `[^0-9]`
`\w`	Matches a word including letters and digits
`\W`	Matches a non-word
`\s`	Matches whitespace including space, tab, carriage return, newline, vertical tab, form feed (Windows)
`\S`	Matches non-whitespace
`^`	Matches the beginning of a line
`$`	Matches the end of a line
`\b`	Matches a word boundary
`\B`	Matches a non-word boundary

Character classes | Boundary classes

REGULAR EXPRESSIONS

Going back to our earlier example:

```
$ grep grad student_list.txt
Joe          Bloggs    Undergrad  Bio
Ruth         Schmoe    Undergrad  Math
```

...does match sub-words.

Adding word-boundaries:

```
$ grep '\bgrad\b' student_list.txt
```

...does match nothing. (Because `grep` is case-sensitive by default, "Grad" is not a match.)

REGULAR EXPRESSIONS

- Potential confusion 2: you must be mindful with escape sequences. The backslash `\` in the shell acts as an escape sequence as well!
- **This will not work:**

```
$ grep \bgrad\b student_list.txt
```

  Why: `\b` will be escaped **before** it is passed as an argument to `grep`. `grep` will see this pattern: `bgradb` where your regex escape sequence has been eaten up by the shell.
- ***Solution 1:*** escape the escape (horror)

```
$ grep \\bgrad\\b student_list.txt
```

- ***Solution 2:*** pass the pattern as a hard-quoted string (prefer this)

```
$ grep '\bgrad\b' student_list.txt
```

REGULAR EXPRESSIONS

- Regular expressions can be exhausting...
- But they will do the job for you when you are confronted with complex search and replace tasks
- It will require *iterations* to get your pattern right, especially for complex stuff (at least I do)
- Watch out for different dialects, they behave slightly different regarding special characters, e.g. compare the `REGULAR EXPRESSIONS` section in `man grep` and `vim -c ':h regexp | only'`

Helpful References

- Pattern composition and testing iterations: https://regexr.com/
- Interactive exercises: https://regexone.com/
- Nice blog post: Basic and Extended Regular Expressions (BRE, ERE)
- Mastering Regular Expressions, 3rd Edition

FILE ATTRIBUTES

FILE ATTRIBUTES

Files in Linux have useful attributes:

- There are *three* timestamps:
  - `atime`: last access time
  - `mtime`: last modification time (content changed)
  - `ctime`: last time file metadata changed (not content)
  - You can use them with `find` too!
- File size obviously
- Ownership and group access (because of time-sharing)
- File permissions (consequence of time-sharing again)

FILE ATTRIBUTES

You get complete information for a file with `stat` (see `man stat`):

```
$ stat my_file
  File: my_file
  Size: 13              Blocks: 8          IO Block: 4096   regular file
Device: 10303h/66307d   Inode: 28969249    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/    fabs)   Gid: ( 1000/    fabs)
Access: 2021-08-27 20:03:32.760407309 -0400
Modify: 2021-08-27 20:01:40.397072908 -0400
Change: 2021-08-27 20:01:40.403739575 -0400
 Birth: 2021-08-27 20:01:40.397072908 -0400
```

You can also sort by time with the `ls` command:

```
$ ls -lt   # mtime by default [long format -l and sort by time -t (newest first)]
-rw-r--r-- 1 fabs fabs  13 Aug 27 20:01 my_file
$ ls -ltu  # -u: atime
-rw-r--r-- 1 fabs fabs  13 Aug 27 20:03 my_file
$ ls -ltc  # -c: ctime
-rw-r--r-- 1 fabs fabs  13 Aug 27 20:01 my_file
```

FILE ATTRIBUTES

Time to look at `ls -l` in more detail:

```
$ ls -l
-rw-r--r-- 1 fabs fabs  13 Aug 27 20:01 my_file
```

From left to right:

- File permissions
- Hard link count (see `man ln`)
- Ownership
- Group access
- File size
- Timestamp
- Filename

FILE PERMISSIONS

- Files (and directories) have a set of permissions that control who can access the data
- There are ***three* permission categories**:
  - `r`: read permission
  - `w`: write permission
  - `x`: execute permission
- There are ***three* types of people** you can trust (or not):
  - `owner`: this is you
  - `group`: this is a group name of other users that you set up
  - `other`: everybody else

FILE PERMISSIONS

- The first entry specifies the *type of file*:

`-`	is a plain file
`d`	is a directory
`c`	is a character device. (The driver communicates with this device by characters, i.e. bytes. E.g. serial ports (Arduino), parallel ports, sound cards.)
`b`	is a block device. (The driver communicates with entire blocks of data. E.g. hard disks, several USB devices.)
`l`	is a symbolic link (see `man ln`)

- The following are permission categories for the three types of people (we distinguish between *files* and *directories*):

Permission category	Set for files	Set for directories
`r`	allowed to read	allowed to see the filenames
`w`	allowed to write	allowed to add and remove files
`x`	allowed to execute	allowed to enter the directory

CHANGING FILE PERMISSIONS

- The `chmod` command is used to change file permissions (see `man chmod`):

```
CHMOD(1)                          User Commands                          CHMOD(1)

NAME
       chmod - change file mode bits

SYNOPSIS
       chmod [OPTION]... MODE[,MODE]... FILE...
       chmod [OPTION]... OCTAL-MODE FILE...
```

- The `mode` can be specified in two ways:
  1. Symbolic representation
  2. Octal number (base-8 number system: 0 to 7)
- Sometimes one method is better suited than the other. You should know both of them.
- Multiple *symbolic modes* can be specified, separated by commas (`MODE[,MODE]...`)

SYMBOLIC MODE

- General form: `[ugoa] [+-=] [rwxX]`
- `u`: user, `g`: group, `o`: other, `a`: all
- `+`: add permission, `-`: remove permission, `=`: set permission
- `r`: read, `w`: write, `x`: execute
- `X`: set to execute only if the file is a directory or already has execute permission. This flag is useful with the `-R` option for recursion.
- There are a few more permissions not discussed here, see `man chmod` for all details.
- See also `man umask` for default file mode creation mask.

SYMBOLIC MODE EXAMPLE

Directory permissions:

```
$ ls -l
total 4.0K
d--------- 2 fabs fabs 4.0K Aug 28 11:23 directory
$ ls directory/
ls: cannot open directory 'directory/': Permission denied
$ chmod a+x directory/ && ls -l
d--x--x--x 2 fabs fabs 4096 Aug 28 11:23 directory/
$ cd directory/ && ls  # && means execute second command only if first succeeded
ls: cannot open directory '.': Permission denied
$ chmod a+r,u+w ../directory/ && ls -ld .  # the -d option only lists directories
drwxr-xr-x 2 fabs fabs 4096 Aug 28 11:33 .
$ ls -l                    # works because we set the a+r permission
---------- 1 fabs fabs 0 Aug 28 11:23 file
$ touch new_file && ls -l  # works because we set the u+w permission
---------- 1 fabs fabs 0 Aug 28 11:23 file
-rw-r--r-- 1 fabs fabs 0 Aug 28 11:33 new_file  # new file default perm defined by umask
```

SYMBOLIC MODE EXAMPLE

File permissions:

```
$ ls -l  # works because we set the a+r permission for the directory before
---------- 1 fabs fabs 0 Aug 28 11:23 file
$ cat file
cat: file: Permission denied
$ chmod a+r file && cat file
Hello
$ echo 'World!' >> file
bash: file: Permission denied
$ chmod u+w file && echo 'World!' >> file && cat file
Hello
World!
```

OCTAL MODE

- Octal mode uses a single octal number for each of the three types of people (3 octal numbers, each can take values 0-7)
- While symbolic mode allows *relative* permission settings (+ and – operators), octal mode is *absolute*
- Setting permissions relative can be convenient in some cases
- Base permissions are assigned the following octal values:
  - 4: read
  - 2: write
  - 1: execute
- Combinations of base permissions are obtained by *summing* their octal values

OCTAL MODE

- Base permissions are assigned the following octal values:
  - 4: read
  - 2: write
  - 1: execute
- Combinations of base permissions are obtained by *summing* their octal values

0: no permissions	4: read only
1: execute only	5: read and execute (4+1)
2: write only	6: read and write (4+2)
3: write and execute (2+1)	7: read, write and execute (4+2+1)

OCTAL MODE EXAMPLE

```
$ ls -l
d--------- 2 fabs fabs 4.0K Aug 28 11:23 directory
$ ls -l directory/; touch directory/new_file
ls: cannot open directory 'directory/': Permission denied
touch: cannot touch 'directory/new_file': Permission denied
$ chmod 755 directory/ && ls -l
drwxr-wr-w 2 fabs fabs 4.0K Aug 28 11:23 directory
$ touch directory/new_file
$ ls -l directory/
-rw-r--r-- 1 fabs fabs 0 Aug 28 12:23 new_file
```

FILE PERMISSIONS

- Assume you start with the following file

```
---------- 1 fabs fabs 0 Aug 28 12:22 file
```

  What is the octal mode equivalent of `chmod a+r,u+w file`?
- **What does `chmod 777` do?** Discuss some of the repercussions.

TEXT EDITORS

TEXT EDITORS

- You can not get around the task of editing text files
- Because you spend the majority of time editing files, you need an editor you feel most comfortable with. The choice is personal.
- There are many text editor in Linux and you will meet them in the pair-programming sections:
  - `pico` and `nano`, easy to get started and minimal.
  - `vim`, powerful but steep learning curve.
  - `emacs`, powerful but also much more than just an editor.
  - `ne`, offers three user interfaces, one via menus.

HISTORICAL EVOLUTION OF VI(M)

- We met `ed` before when talking about `grep`. Very first line based Unix editor written and used by Ken Thompson.
- `ex` is an *extended* version of `ed`.
- `vi` is a full screen version of `ex` (before that there were teleprinters not screens!)
- `vim` is an *improved* version of `vi`.
- `vi` or `vim` are tools that you will have at your disposal on any *nix type operating system.
- Because `vi`/`vim` are ancestors of `ed`/`ex`, they inherit similar syntax that is found in other tools such as `sed` or `awk` (learn one use by many).

VIM

- `vim` is a *modal* editor. It has 7 basic modes and 7 variations of the basic modes. The 3 most important ones are:
  1. Normal mode
  2. Insert mode
  3. Command-line mode
- Normal mode is the default and used for navigation and operations on text(-objects).
- Insert mode allows you to enter text with the keyboard (press `i` to enter insert mode and `ESC` to return to normal mode).
- Command-line mode allows to enter `ex` commands that operate on the file contents (e.g. pattern substitutions, writing the file or quitting the editor). Enter command-line by pressing `:` in normal mode.

VIM

- Starting the editor: `vim my_file`

```
VIM(1)                          General Commands Manual                          VIM(1)

NAME
       vim - Vi IMproved, a programmer's text editor

SYNOPSIS
       vim [options] [file ..]
...
```

- Press `i` and type "`Hello vim!`" (you can move around with the arrow keys)
- Press `ESC`, followed by `:w<CR>` (`<CR>` means carriage return or enter, this will `write` your changes), followed by `:q<CR>` which will exit `vim`
- Check the contents of `my_file`:

```
$ cat my_file
Hello vim!
```

USEFUL VIM COMMANDS

All of these commands are typed in normal mode:

`:q!`	Exit without saving the document. Your changes will be lost.
`:wq`	Save and quit
`:wqa`	Save all open files and quit
`/pattern`	Search for `pattern`. This can be a regex too. Type `n` for the next forward match and `N` for the next backward match.
`dd`	Delete the line where the cursor is on
`yy`	Copy (yank) the line where the cursor is on
`I, i, a, A`	Insert text: at beginning of line (`I`), before the cursor (`i`), after the cursor (`a`), at end of the line (`A`)
`p`	Paste the last yank/cut/deleted text
`gg`	Go to first line
`G`	Go to last line

VIM RESOURCES

- `vim` tutor: type `vimtutor` in your shell
- Practical Vim: Edit Text at the Speed of Thought 2nd Edition
- Cheat sheet
- Vimcasts.org
- `git` plugin for `vim`: `vim-fugitive` and screencasts

A NOTE ON IDE

- IDEs are *Integrated Development Environments*. They are graphical tools that combine many development tasks in the same graphical environment. (All of these tools exist in the shell as well.)
- They can be convenient and powerful but often require Gigabytes after installation and can take a while to start up. Examples are:
  - Spyder
  - Eclipse
  - Visual Studio
  - PyCharm
  - Jupyter (somewhat)

Assume you are a performance engineer at Netflix and an expert Eclipse user. Saturday 2AM the phone rings due to an emergency situation on an important Netflix server. You must fix the problem ASAP on the remote machine without Eclipse. Stay calm.

RECAP

- Linux `man`-pages
- The Unix philosophy and pipes
- Regular expressions (practice!)
- Linux file attributes and permissions
- Find an editor you are comfortable with and make it your own
- When you own it, get matching key caps...