Fall 2021: Computational Science I

Instructor: Mohammad Sarraf Joshaghani
(m.sarraf.j@rice.edu)

Module 1: Linux OS and shell programming

Introduction

Run your first VM

Very basic commands with the shell

How to get help on Linux

Working with files and directories

Simple data processing

File attributes and searching

Control jobs and processes

Editors; the very powerful vim

Customizing shells and dot files

Talking to other machines and remote access

Shell programming

Outline

Lecture 2 (Aug 27, 2021)

- Linux operating systems, shells and terminals
- Working with Virtual Machines (in case you do not have a Linux OS)
- Basic Linux commands

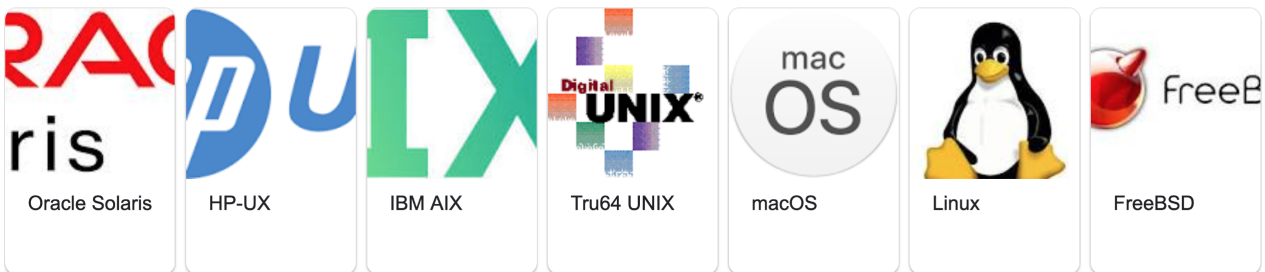
What is your OS?

Linux . macOS . Windows

Unix/Linux OS??

What is a Unix system?

- Unix operating systems were first created at AT&T Bell Labs in the 70s, in part by Ken Thompson and Dennis Ritchie, inventors of the c programming language. (UC Berkeley, Sun, IBM, and others also contributed) $\longrightarrow$ code eventually standardize in series of publications called POSIX.
- There is not just one flavor of Unix: Solaris, HP-UX, IRIX, etc. Apple has macOS (Darwin), and **Linux** is yet another variant.

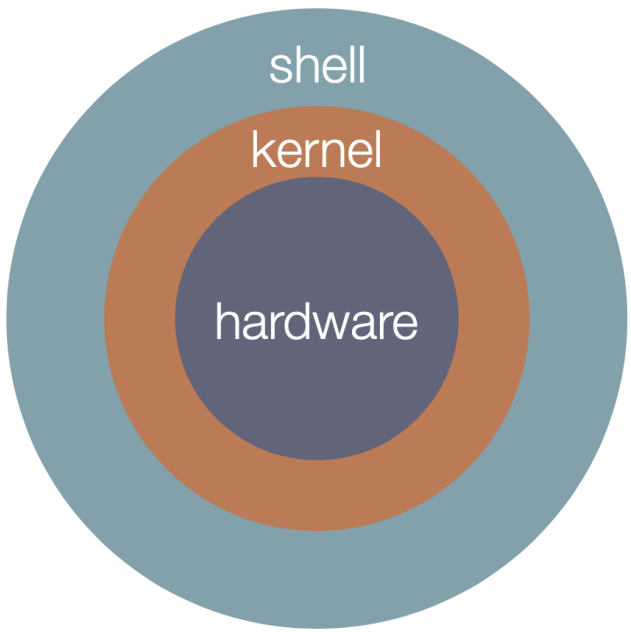


What is a Linux system?

- It is a "Unix-like" operating system written by Linus Torvalds. It was first released September 17, 1991.
- Linux operating systems are *open source*, highly scalable and widely used in scientific computation, on large computer clusters, etc.

Unix/Linux OS

- Unix/Linux has a **kernel** and several different **shells**
- The shell is the *command interpreter* (the program that processes the command you enter in your terminal emulator)
- The kernel sits on top of the hardware (CPU, RAM, disks, network, . . .) and is the core of the OS;
  - ◇ does memory management, task scheduling, handles with file systems, I/O handling
- Kernel receives tasks from the shell and performs them.



Linux distros

Within Linux there are various *Linux distributions* such as Fedora, Debian, Red Hat, and Ubuntu (and over 100 more). See: https://distrowatch.com/

- **Ubuntu** is probably the most well-known Linux distro. Ubuntu is a cleaned-up version of Debian, but it has its own software repositories
- **Linux Mint**: Ubuntu-based distro with elegant apps
- **Debian**: is an operating system composed only of free, open-source software. The Debian project has been operating since 1993 — over 20 years ago! This widely respected project is still releasing new versions of Debian, but it's known for moving much more slowly than distros like Ubuntu or Linux Mint
- **Fedora**: Fedora is a project with a strong focus on free software (de-corporatized Red Hat Linux) — you won't find an easy way to install proprietary graphics drivers here, although third-party repositories are available.
- **CentOS/Red Hat** is a commercial Linux distribution intended for servers and workstations. It's based on the open-source Fedora project, but is designed to be a stable platform with long-term support. CentOS is free analog of red Hat Enterprise

Shell flavors

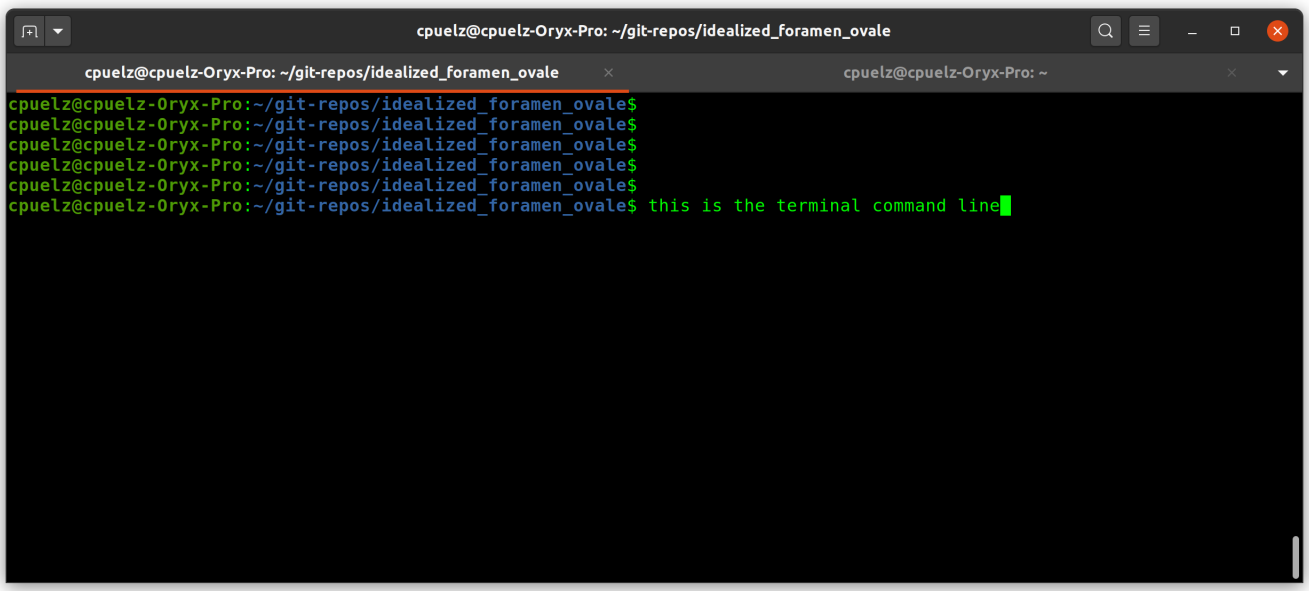
- A program that takes keyboard commands and passes them to the OS to carry out
- Different shell types exist that can extend the command line interface (where you type commands) with various nice features like colors, configuration, auto-completes and scripting capabilities.
- The **bash** shell is most common, and is typically the default on Linux and OSX. However, other shells exist and are actively used (e.g. csh or **zsh**, or Fish). Typically, they don't vary all that much, In any case, don't worry too much about the options at this point; it's easy to switch between them if the need arises.

I prefer **zsh** → (expand wild cards; will give you a list of switches; give tab completion)
see this: What shell I sould use?

Will talk more about that later!

What is the terminal?

- Graphical User Interface (GUI) program that allows users to interact with the shell
- Examples: `iterm2`, `Alacritty`, `mac Terminal`, `xterm`, and `Terminator`



Sort them out in a Breakout room

1

Me typing .....

2

This is where I type

3

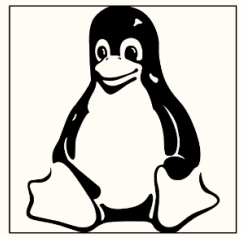
Extend my CLI with nice features

4

I am a low-level program interfacing HARDWARE

5

Operating systerm



Free

proprietary $$$$

GNOME Terminal

iTerm2

csh

Bash

~/.bashrc

disk

RAM

library calls

Ubuntu

Mint

IBM

Alacrity

zsh

I/O handling

Fedora

Terminator

Red Hat

macOS (Darwin)

Fish

~/.zshrc

task scheduling

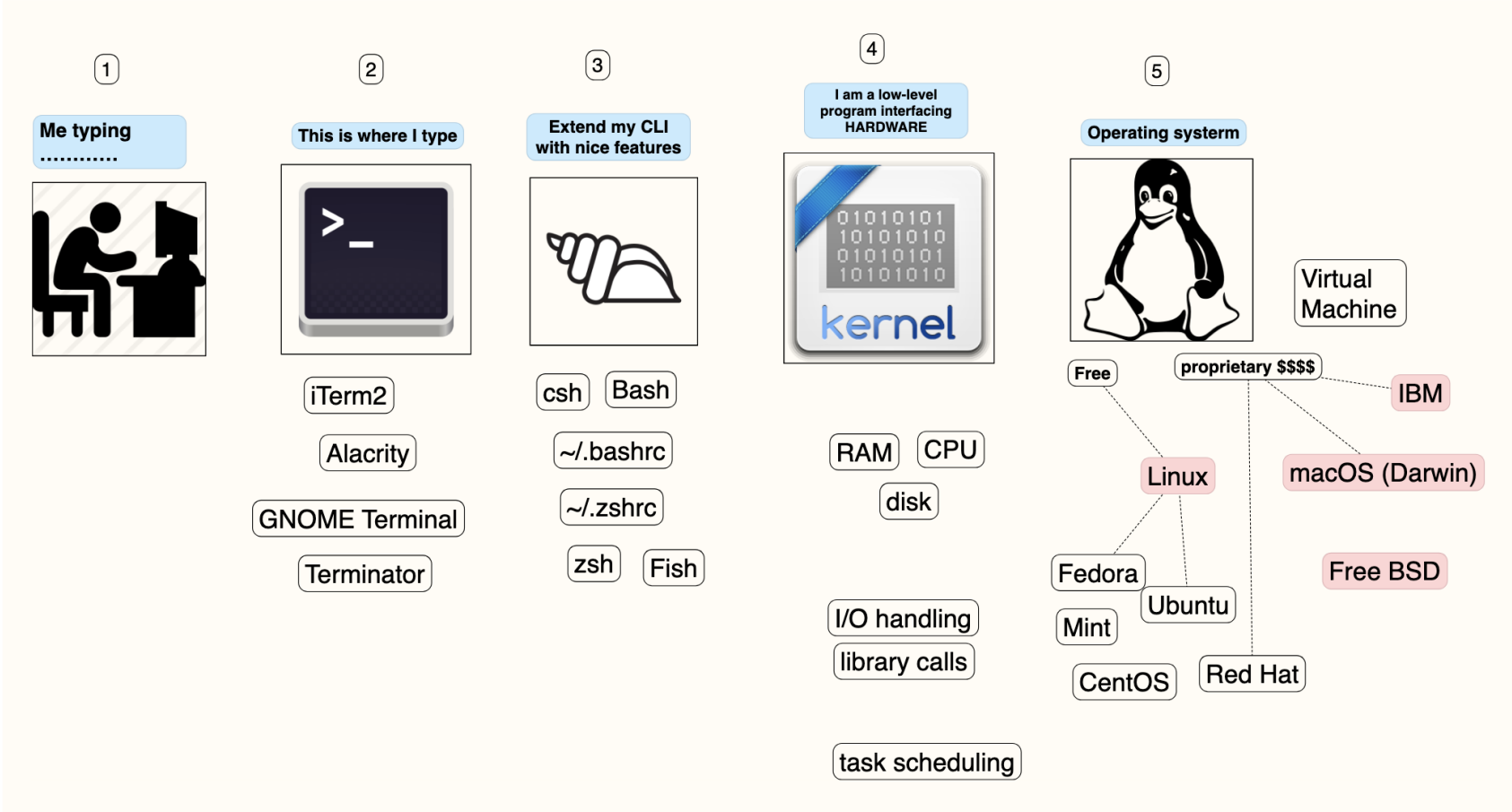
CPU

CentOS

Free BSD

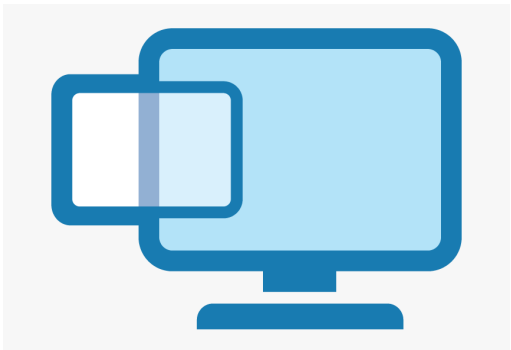
Virtual Machine

Sort them out in a Breakout room



What is a Virtual Machine

- Virtual Machine is a computer inside a computer
- Hypervisors is a software doing the virtualization for us (like VirtualBox, VMware, Hyper-V)
  - Two classes of Hyperviosors: Type I (does not need a host OS) and Type II (Install on a host OS)
  - Create virtual (fake) hardware (borrowed from the actual one)
- You can have more than one VM (macOS, Linux, etc) on your machine



Standardizing our OS for this class

for this class, we will work within an ubuntu os, a linux operating system that we will install within VirtualBox.

https://www.virtualbox.org/

there are other ways to run ubuntu, like dual booting or doing a fresh install on your computer. VirtualBox is the easiest right now...

it will run an ubuntu operating system like an application *within your native operating system*.

Installing VirtualBox

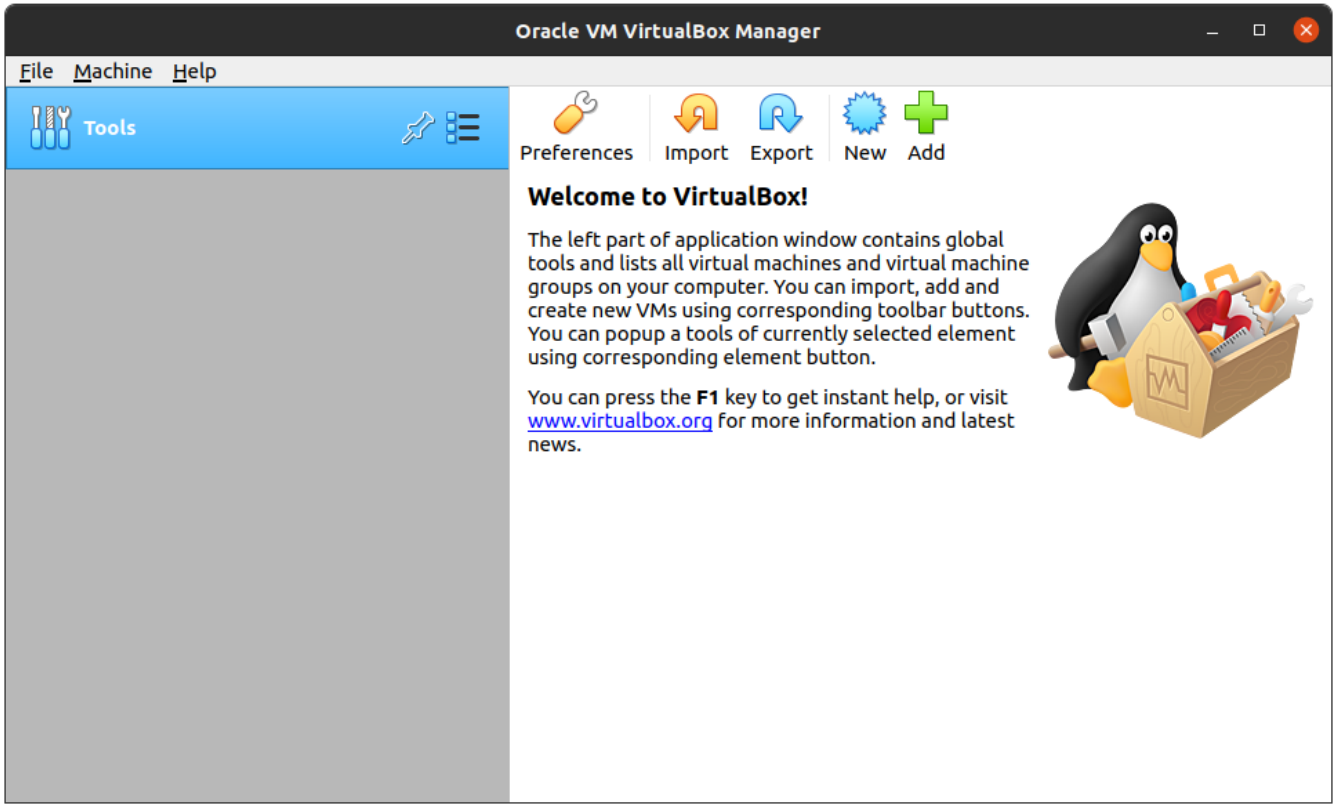
go to https://www.virtualbox.org/wiki/Downloads and select the proper host, i.e. if you are running a Windows OS, you need a Windows host, etc.

also, go to https://ubuntu.com/download/desktop to download the iso file for ubuntu 20.04. we will use this file to install the linux os on our VirtualBox.

we can follow roughly these instructions for the install:
https://brb.nci.nih.gov/seqtools/installUbuntu.html

Step 0: set up a virtual machine

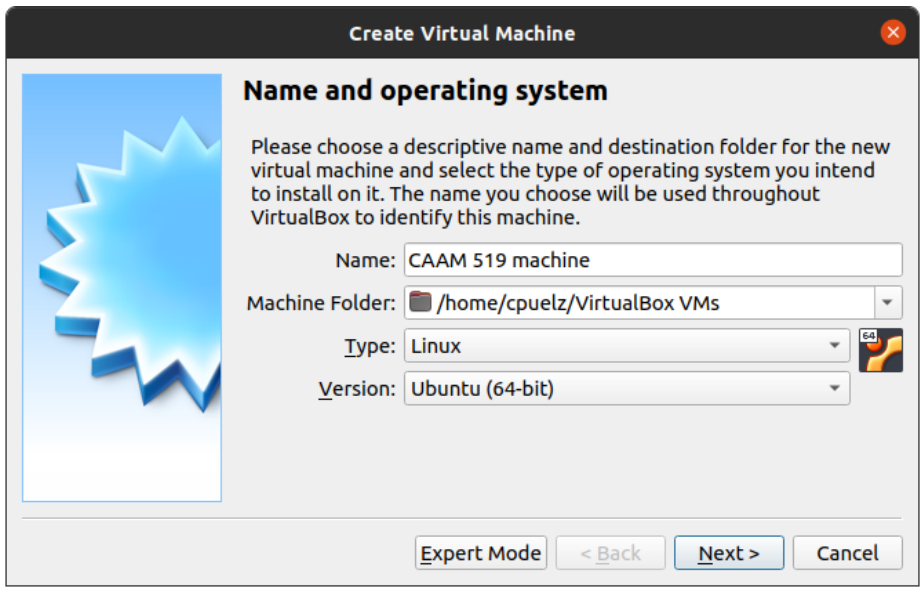
click “NEW” in the menu...



Step 1: name your machine

give your machine a name.

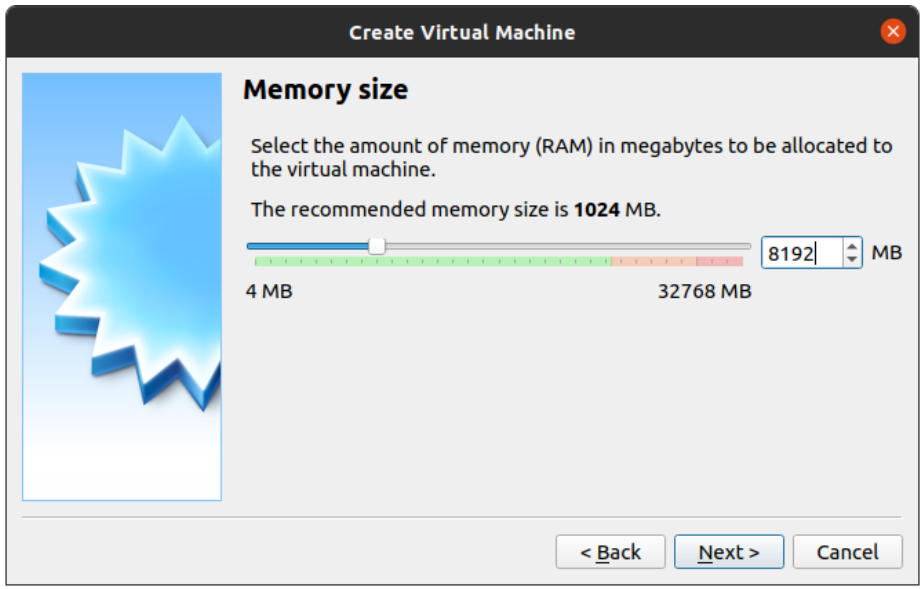
the type should be “linux,” and version will be 64-bit ubuntu.



Step 2: pick amount of RAM

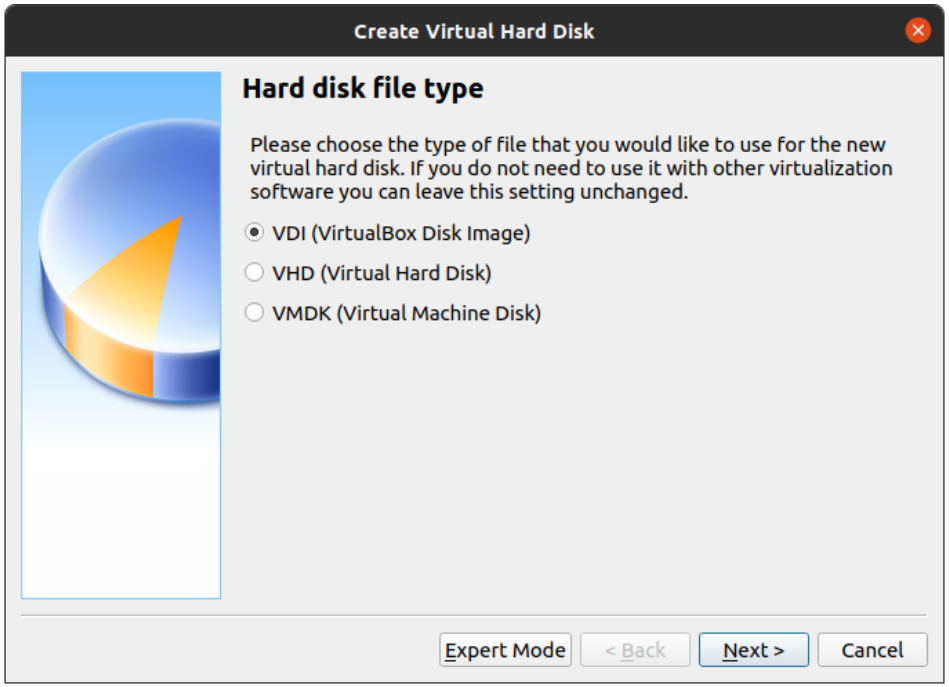
ubuntu recommends 4Gb for the os version we will be installing

see: https://ubuntu.com/download/desktop



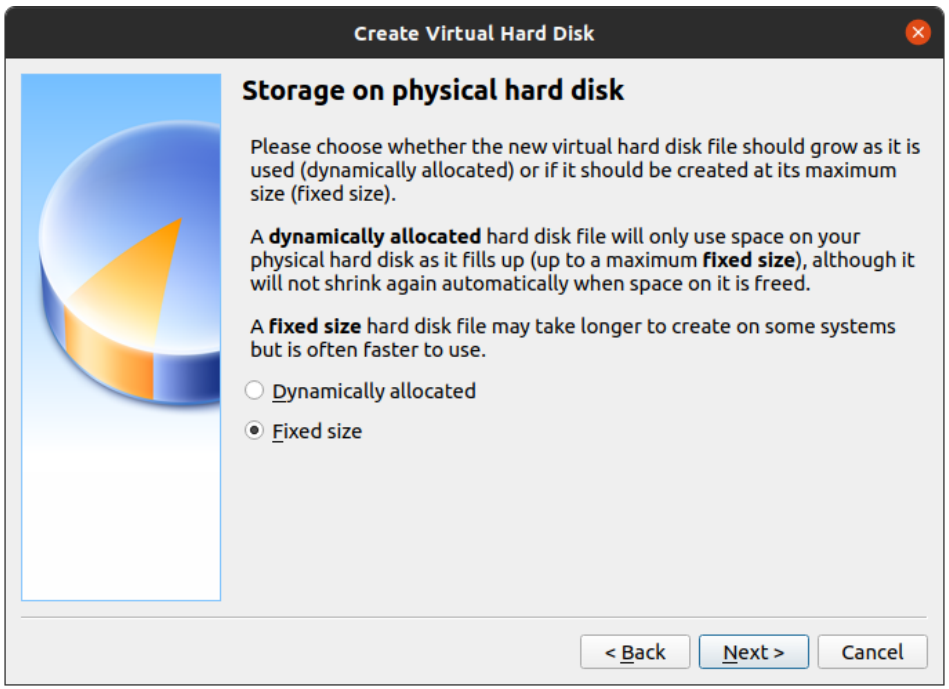
Step 3: setup hard disk

Step 4: virtual disk image



Step 5: let the hard disk be a fixed size

this will be faster.

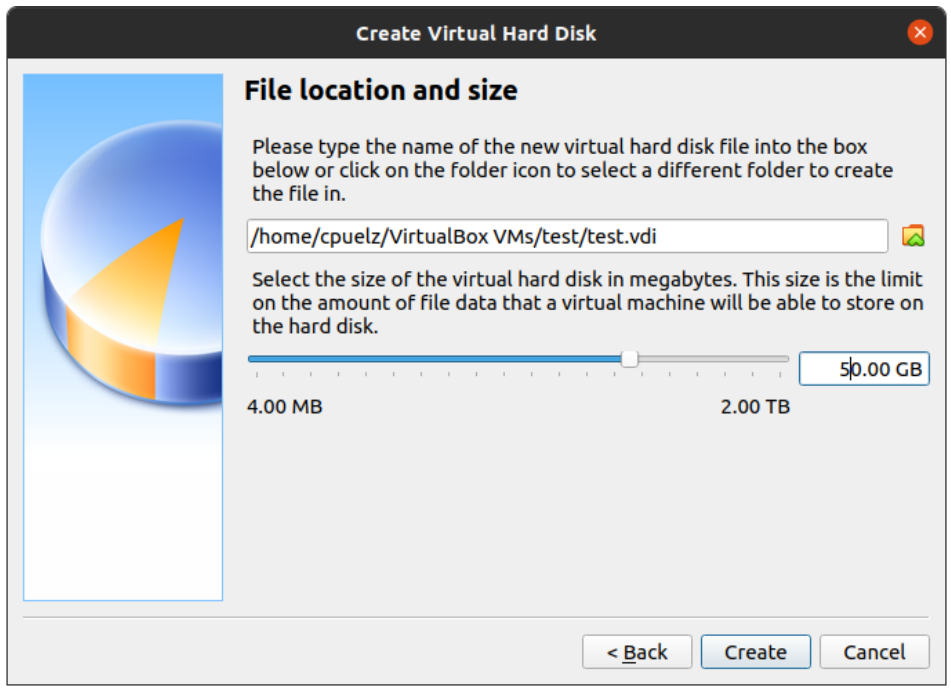


Step 6: pick size of hard drive

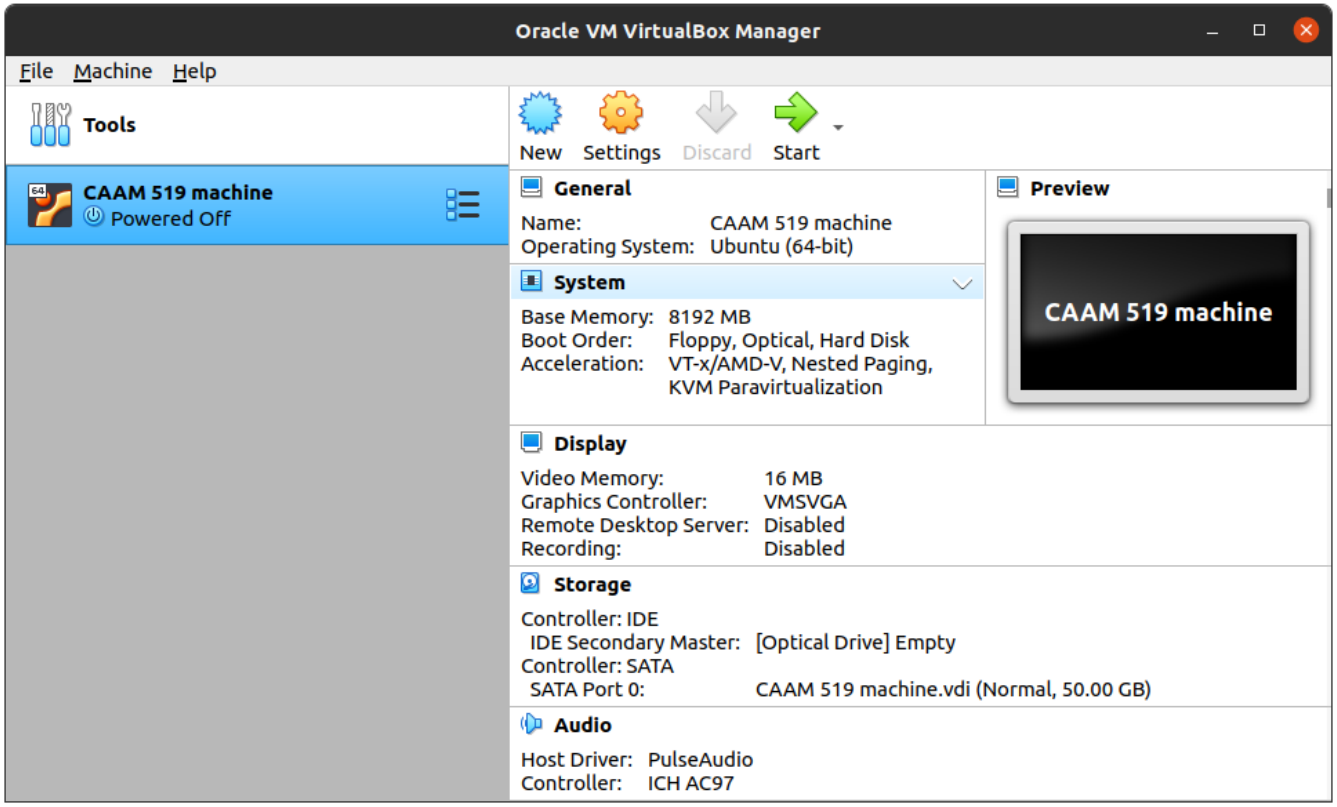
ubuntu recommends 25Gb for the os version we will be installing

i am picking a bit more, for a "buffer"

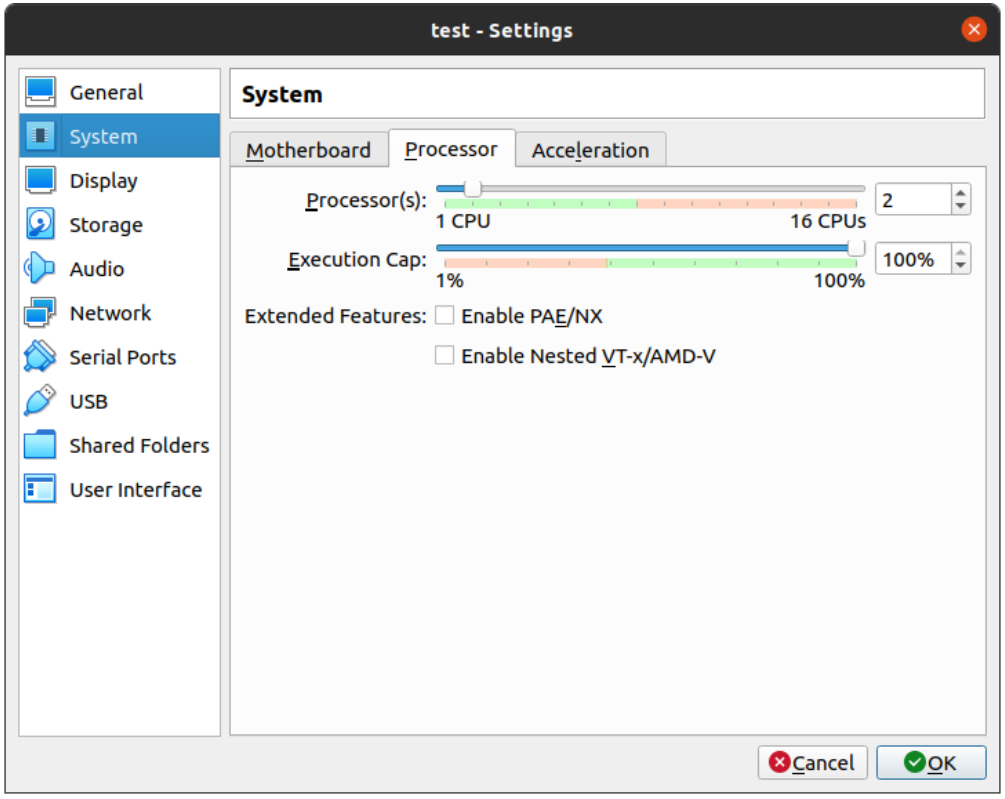
see: https://ubuntu.com/download/desktop



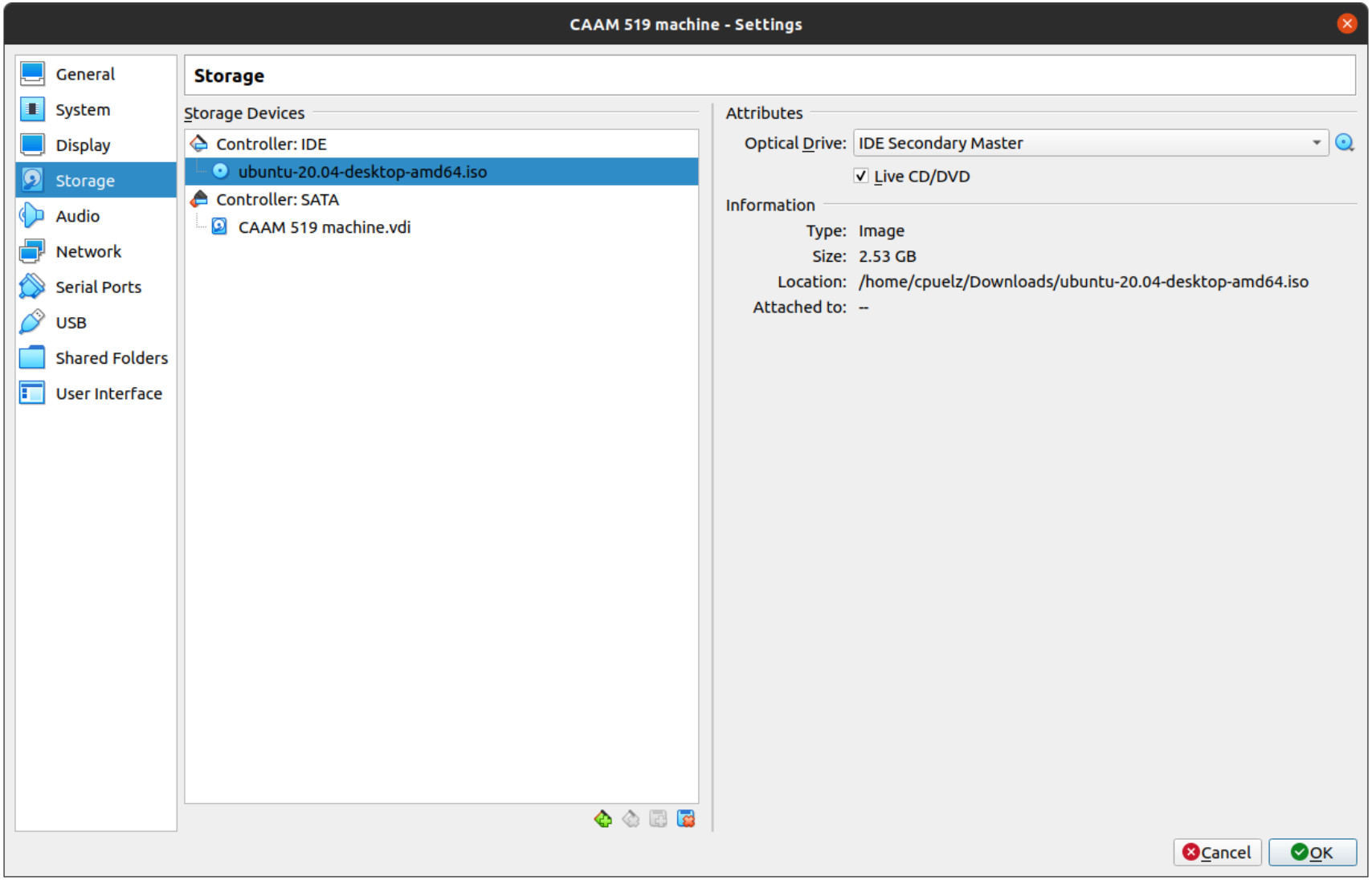
Step 7: done!


Oracle VM VirtualBox Manager
File Machine Help
Tools
CAAM 519 machine
Powered Off
New Settings Discard Start
General
Name: CAAM 519 machine
Operating System: Ubuntu (64-bit)
System
Base Memory: 8192 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization
Preview
CAAM 519 machine
Display
Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled
Storage
Controller: IDE
IDE Secondary Master: [Optical Drive] Empty
Controller: SATA
SATA Port 0: CAAM 519 machine.vdi (Normal, 50.00 GB)
Audio
Host Driver: PulseAudio
Controller: ICH AC97

Step 8: pick number of processors



Step 9: point machine to image of ubuntu os for install



Step 10: reboot the machine!

note: when you reboot, you should "remove the installation medium, and then press ENTER."

this requires you to go to **Settings** > **Storage**, and "eject" the iso file.

Step 11: (optional) install guest additions

this step seems to be helpful for graphics within the virtual machine. by the way, this change might not be possible on certain operating systems.

go to **Devices** $>$ **Insert Guest Additions CD Image**, and following prompts.

Other options

I'm not going to require that you use Linux through a virtual machine if you prefer not to. Some other options at your disposal:

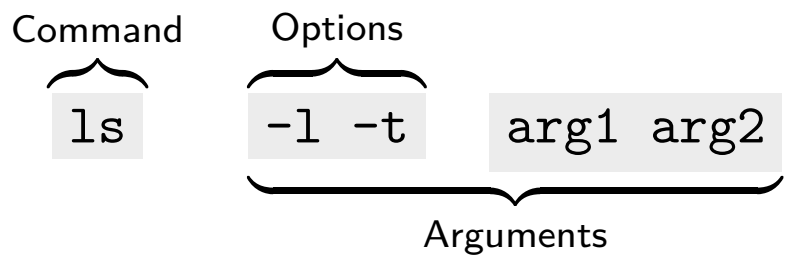
- Use your existing OS, if you have a machine running Linux or MacOS
- Install a partition on your machine running Linux.

$\rightarrow$ I'll do best effort to help troubleshoot computing problems on other platforms, but I cannot guarantee success. The answer will very likely be "use the virtual machine."

Basic *nix Commands-Users & System

When, what, where, who . . . ?

Structure of a UNIX command



When?

- `date` ...display the full date
  - ◇ `date +%m` ...the `%m` is a format specifier (**m**onth)
  - ◇ `date +%h` ...the `%h` is a format specifier (**m**onth string)
- `cal` ...display the calendar
- `history` ...display the entire history of commands

  use `up arrow` to scroll through commands you have used
- `history` ...display the entire history of commands

  Tips:
  - ◇ use `up arrow` to scroll through commands you have used
  - ◇ use `Tab` to autocomplete
  - ◇ use `!<cmd>` to expand most recent invocation of `<cmd>`
  - ◇ use `!!` to expand last commands again

Q What does `Sudo!!` do?

Q How to clean your history?

what?

Lecture 3 (Aug 30, 2021)

List

- `ls` ... list directory contents
  - `ls -a` ... list **a**ll
  - `ls -t` ... list and sort by **t**ime
  - `ls -l -h` ... list **l**ong listing format and **h**uman readable **Q** How to get the size of a directory?
- `file` ... report the type of files
- Multiple options can be combined, for example:
  - ◇ `ls -l -h` ≡ `ls -lh`

Environment variable

- The system uses environment variables to store system and user info
  - ◇ define with `export <MY_NEW_VAR>='NewVar'` (**b**ash shell)
  - ◇ access with `$`, e.g., `echo $MY_New_VAR`

Q What is the difference between `export a=5` and `a=5`?

What?

What is 2+2?

- for simple calculations in terminal you can use: `echo $((2 + 2))`
  - The syntax is `$((...MathExpr...))`
  - spaces between operator and number is obligatory.
- Better way to calculate expression is to use `expr`
  - `expr 2 + 2`
  - ```
    x=20
    y=5
    expr $x / $y
    ```

Q How to calculate 5×5 using `expr` and `echo` commands?
    ```

Where?

Navigation commands

- `pwd` ... **p**rint **w**orking **d**irectory
- `cd` ... **c**hange **d**irectory
  - ◇ `cd $HOME` or better, `cd ~`, even better `cd`
  - ◇ `cd ..` ... go up one level
- Absolute path (directory structures) start with `/`, the root directory
- `tree` ... make tree-shaped list (how to set depth-level?)

Tip: Install **tree** with:

```
sudo apt-get install tree
```

When `apt-get install` is unable to locate a package, the package is not found withing Ubuntu repositories. Follow this instruction to add and update repo

who?

Machine and OS

- `hostname` ... print the machine's hostname
- `uname` ... print other system information
  - `uname` ... Description
  - `uname -a` ... print all system information
  - `uname -m` ... print machine hardware
  - `uname -n` ... same as `hostname`
- `lshw` ... print hardware configurations as root user
- `lscpu` ... print CPU detail from `/proc/cpuinfo`
- dmidecode ... Extract more detailed information
  - `sudo dmidecode -t memory` ... information about memory

Q How to access the content of `/proc/cpuinfo` file? (use cat)

Q How to add a user named "john" and then list all users?

who?

Users

- `passwd` ... change your password
- `who` ... print logged in users(see users)
  - ◇ `who am i` ... print your own username `whoami`
- `finger` ... display information about users

Getting help

- **Google** is your friend. Odds are, someone has had the same question as you in the past, and has asked it on a forum website such as *StackOverflow* or *StackExchange*.
- **Man documentation** `man ls` brings up the **man**ual pages for `ls`. Unlike Google results, manpages are system-specific.
  - ◇ Use `/`+`<keyword>` to perform a search in manpage
  - ◇ Use `n` to jump to successive search results

Man Overview I

`man` (short for manual) is the built-in help system in Linux.

Q How to access the help in a man page?

Read the man page manual: `man man` (type h for help and q to quit)

Man page navigation Man uses the less pager and vim keybindings.

- `j` ... Forward (down) one line
- `k` ... Backward (up) one line
- `Crtl+f` ... Forward one window
- `Ctrl+b` ... Backward one window
- `g` ... Go to first line
- `G` ... Go to the last line

Man Page Conventions

- **bold text** ... type exactly as shown
- italic text ... replace with appropriate argument
- [-abc] ... arguments within [ ] are optional
- -a|-b ... options cannot be used together
- `argument ...` ... argument is repeatable

Man overview II

Combining options: only one hyphen (dash) is needed for multiple single-character options:

```
ls -l -h
ls -lh
```

The same option can come in two forms: `-w, -width=COLS`

```
ls --width=60
ls -w 60
ls -w60
```

Man page Sections: some commands may have man pages in different sections. (Most commands are in Section 1, for system administrations check Section 8, low-level Linux programming check Section 2 and 3.)

Q How to search man pages? use `man -k <text>`

Q How to specify which Section to use for a command? You can specify the section to use: `man <section> <command>` For example: `man 2 mtools`

man Overview III

Shell Builtins: some commands don't have dedicated man page. These commands are shell built-ins (internal commands).

Q Where is the man page for `cd`?

- use `help <command>` to get help files
- use `type <command>` to get type of command (builtins or not?)

```
help while
type cd
```

Getting extra help with option –help:

```
--help
-h
ls --help
```

Outline

Lecture 4, Sep 1, 2021

- Review Linux file system
- Learn how to manipulate and view files
- Pipes and redirects

File system

The File: UNIX treats everything as a file. Directories and devices like the hard disk, DVD-ROM, and printer are files to UNIX

Three types of files

- **Ordinary file:** also known as a regular file, contains only data as a stream of characters
- **Directory file:** a fold containing the names of other files and subdirectories
- **Device file:** represents a device or peripheral (e.g., printer, scanner, etc)

Q How to check Linux disk space for each drive?

Q How to properly mount or safely eject a hard drive? (use udisksctl)

The File System Hierarchy I

- UNIX files are organized in a hierarchical structure:
  - `"/"` ... root of the file system
  - `"/home/username"` ... your home directory (also `~`)
  - `"/bin"` ... contains important binaries or executable essential to OS
  - `"/sbin"` ... contains system binaries (for super user)
  - `"/usr/bin"` ... contains non-essential binaries
  - `"/usr/local/bin"` ... contains locally compiled binaries
- `$PATH` tells Linux where to find binaries
- `which` checks where the binary lives (e.g., `which ls`)

Q What does -> mean when files are listed (e.g., `ls -l /`)?
The path after -> is the source of the symbolic link

The File System Hierarchy II

Other important directories are

- `"/boot"` ... things for booting.
- `"/etc"` ... host specific configurations for applications like text editors and web browsers
- `"/lib"` ... shared libraries
- `"/dev"` ... removable devices like usb drives
- `"/tmp"` ... temporary files lost after reboot
- `"/var"` ... variable files that change during normal operation, like log and cache files
- `"/opt"` ... optional adds-on files (rarely used)
- `"/proc"` ... illusionary file created on fly for monitoring

Relative Pathnames

Relative path shortcuts

- `.` (a single dot) represents the current directory
- `..` (two dots) represents the parent directory

Command Line Expansion

Bash can transform the command-line input using expansions

- **Brace expansion** `{begin..end}` or `{begin..end..increment}`

```
echo {Z..A}
echo Front-{A,B,C}-Back
```

- **Arithmetic expansion** `$((expression))`

```
AA=50
echo $((AA++))
BB=$((AA*2))
echo $BB
```

- **Parameter expansion** (mostly useflul in bash scripting)

```
echo $USER
echo $HOME
```

Making and Removing directories

Making directories

- `mkdir myDir` ...creates `myDir` in the current directory
- `mkdir myDir1 myDir2` ...creates multiple directories in one command
- `mkdir myDir1/myDir2` ...creates `myDir2` inside `myDir1` (must exit)

Q How to create 10 directories named dir1 to dir10 in one command?

Q How to create 10 empty files?

Copy directories

- `cp file1 file2` ...copy `file1` to `file2`
- `cp -R myDir1 myDir2`
  - ◇ `-R` option copies recursively, meaning all subdirectories will be copied as well

Making and Removing directories

Move directories

- `mv file1 file2` ... when used this way it's basically a rename utility
- `mv file1 file2 myDir` ... moves `file1` and `file2` into the directory `myDir`

Removing directories

- `rm file1 file2` ... removes both files
- `rm file*`
  - `*` is a *wildcard*, meaning "anything", the command will remove all patterns that match file with anything following
  - can be dangerous. With the right permissions `rm -Rf /*` would remove most of the files on your hard drive without warning
  - protect yourself `rm -i`

Outline

Lecture 5, Sep 3, 2021

- Simple data processing
- pipes and redirects

Displaying and Concatenating Files

- `cat` displays the contents of one or more files
- `touch` ...creates new/empty files or update existing file
- `cat > file` ...enter text into file (use `Crtl+d` to save and exit)
- Outputs by default to "stdout", but often redirect to a file using `>`

```
cat foo.txt bar.txt > foobar.txt
```

More or Less

- `more` and `less` display files one page at a time
- use `more` for large file
- `less` is more `more`
  - ◇ Allows movement backwards in a file
  - ◇ Faster than most standard text editors
- `man` by default uses `less`

Navigation in `more` and `less`:

`spacebar` or `f` . . . one page forward

`b` . . . one page backward

`j` . . . one line forward

`k` . . . one line backward

`/`+`foo` . . . searches forward expression `foo`

Simple data processing

- `wc` ... counts lines, words, characters
  - ◇ `wc -l file` ... counts line in file
  - ◇ `wc -w file` ... counts word in file
- `head` and `tail` ... prints the first and last few lines of a file
- `sort` ... sorts alphabetically or numerically

Task

Download `us-states.csv` file from class website.
What are the first and last row?
How many lines it has?
show all lines that contains "Texas"

Combining and cutting streams

- `paste` command let you merge two or more input stream side by side

```
seq 5 > serial1.txt
seq 1 5 30 > serial2.txt
paste serial1.txt serial2.txt > newserial.txt
```

- `cut` prints out selected section from each line of an input stream or file (needs to be separate by tab if used without options)

```
cut -f 1,2 <file>
```

Task Print out only 'state' column in `us-states.csv`
(hint: use delimiter option of `cut`)
How can we sort them alphabetically?

Redirect and Append

Redirect command output to a file by using `>`

- Think of redirection characters as arrows. For example

```
ls /dev > text.txt
```

  redirects the output of `ls` into a new file named `text.txt`

Append output to the end of a file by using `»`

```
ls .proc >> test.txt
```

Stndards I/O

There are **3** std I/O file stream descriptors. Most processes initiated by UNIX commands write to standard **output** (i.e., the screen), and take their input from standard **input** (the keyboard). There is also standard **error**.

- **stdin** . . . use `<`
- **stdout** . . . use `>` (the same as `1>`)

```
echo 'print "hello world!" ' > hello.py
python < hello.py
```

Q How to do this in single command?

Stndards I/O

There are **3** std I/O file stream descriptors. Most processes initiated by UNIX commands write to standard **output** (i.e., the screen), and take their input from standard **input** (the keyboard). There is also standard **error**.

- **stdin** . . . use `<`
- **stdout** . . . use `>` (the same as `1>`)

```
echo 'print "hello world!" ' > hello.py
python < hello.py
```

Q How to do this in single command?

```
echo 'print "hello world!"' > hello.py && python < hello.
    py
```

  - ◇ The `&&` means continue with next command

Std err

- **stderr** . . . use `2>`
  - ◇ redirect stderr to a file: `command 2> error.txt`
  - ◇ most often used to redirect to nowhere
    `find / -name ls 2>/dev/null`
  - ◇ looking into very long error messages
    `command 2>&1`

More on std*

Q What does this redirection mean:
`command > out.txt 2>error.txt`

Q Is there any alternative short form? Redirect both stderr and stdout together? `command &> output.txt`

Q Can you explain the following command
`curl http://www.google.com > /dev/null 2>&1`

The first part `>/dev/null` redirects the `stdout`, that is `curl`'s output to `/dev/null` and `2>&1` redirects `stderr` to the `stdout` (which was just redirected to `/dev/null`). The `&` is used on the right side to distinguish `stdout (1)` or `stderr (2)` from files named `1` or `2`. So, `2>1` would have ended up creating a new file.

Outline

Lecture 6, Sep 8, 2021

- Continue with pipes and directs
- Learn file permissions and ownership
- What are wildcards in Linux?
- `grep` and `find` applications

Pipes

`|`

- As the name implies, a pipe (`|`) takes the output of one command to the input of another command
  - ◇ e,g,. `ls /usr/sbin | less`
- We can actually write short "programs" by stringing pipe together

`tee`

- Split a data stream so it follows both into a specified file and continues as `tee` 's `stdout`
- Useful for saving immediate steps in a long sting of pipe
  - ◇ e.g., `ls /usr/sbin | tee processes.txt | less`

Task Use `du` , `head` , and `sort` comamnds to find biggest files and directories in your $HOME

Task Find the ip address of your machine using `ifconfig` command, `cut` , etc . (my ip is 10.0.0.**)

Another way to join two commands together

- The shell supports, in addition to pipes `|`, another way to join two coomands together.
  - Surround the substituted command with single backquotes

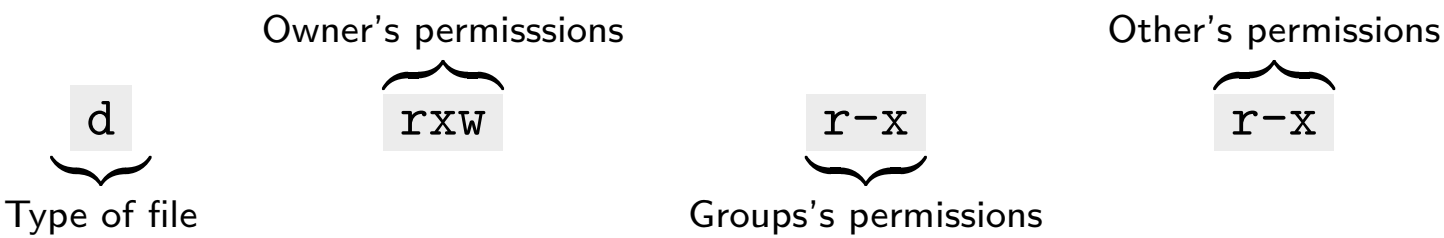
```
echo the date today is `date` and direcoty is `pwd`
```

File Attributes

Listing of the file attributes `ls -la`

```
drwxr-xr-x   1 mohammadsarrafjoshaghani  staff       70 Aug 11 17:42 .gitignore
drwxr-xr-x  17 mohammadsarrafjoshaghani  staff      544 Aug 11 17:42 Figures
-rw-r--r--   1 mohammadsarrafjoshaghani  staff    12703 Aug 24 01:36 Linux.aux
-rw-r--r--   1 mohammadsarrafjoshaghani  staff    42578 Aug 24 01:36 Linux.fdb
-rw-r--r--   1 mohammadsarrafjoshaghani  staff    45202 Aug 24 01:36 Linux.fls
```

Structure of file permissions string



- **Types of file**: plain file `-` , directory `d` , and symbolic link `l`
- **Types of permission**: allowed to read `r` , allowed to write `w` , and allowed to execute `x` .

Change file permissions I

- `chmod [mode] <file>` . . . **ch**anges the permission **mod**e of file
  - `chmod [category + operation + permission] file`

Table: mode representation table

category		operation		permission	
`u`	user	`+`	add permission	`r`	read
`g`	group	`-`	remove permission	`w`	write
`o`	other	`=`	assigns absolute permission	`x`	execute
`a`	all				

change file permission II

- **Octal permissions** uses a single argument string of **3** digits
- Each digit of this number is a code for each of the 3 permission levels (owner, group, other)

octal	permission	Significance
0	`--`	No permissions
1	`-x`	Execute only
2	`-w-`	Writable only
3	`-wx`	Writable and executable
4	`r-`	Read only
5	`r-x`	Readable and executable
6	`rw-`	Readable and writable
7	`rwx`	Readable, writable, and exec

Q What does `chmod 740 file` do?

Changing file ownership

- `chown` . . . **ch**ange **own**er
  - `chown <user> file` . . . change ownership of file to `<user>`
  - `chown <user>:<group> file` . . . change owner and group
  - `chown -R <user> <directory>` . . . change a directory's owner recursively

Q How to to list users and groups on your machine?

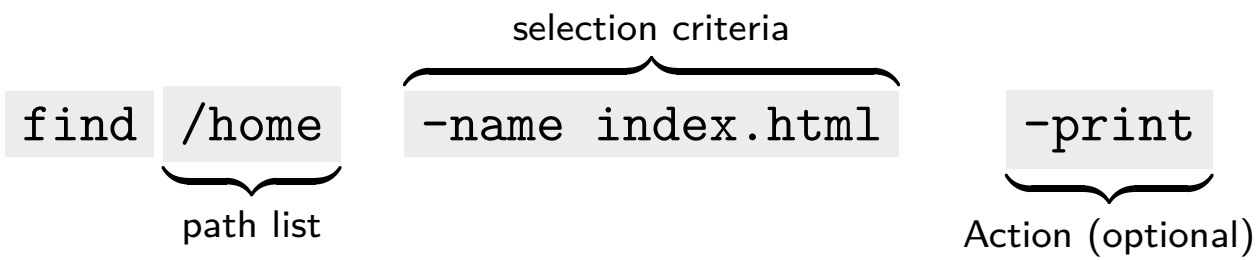
```
cat /etc/passwd
cat /etc/passwd | cut -d: -f1
cat /etc/group | cut -d: -f1
```

How to find a file?

find

- `find` ... recursively examines a directory tree to look for files matching some criteria and then takes action on the selected files

Structure of a `find` command:



How to find a file

Selection criterion:

Selection criterion	Selects file
`-type x`	If of type `x`, where `x` is `f`, `d`, or `l`
`-perm nnn`	If octal permissions match `nnn`
`-links n`	Having `n` links
`-user usrname`	If owned by usrname
`-group grpname`	If owned by grpname
`-name filename`	filename
`-iname filename`	Same as above, but case-insensitive

Action

Action	Significance
`-print`	Prints selected file to stdout
`-ls`	Executes ls -lids command on selected
`-exec cmd`	Executes UNIX command `cmd` followed by `{} \;`

Find examples

- Find directories in `/path/to/search` :

  ```
  find /path/to/search/for -type d
  ```

- Find files by case-insensitive extension, such as '.jpg', '.JPG' in current directory: `find . -iname '*.jpg'`

- Recursively change permissions on files, leave directories alone

  ```
  find ./ -type f -exec chmod 644 {} \;
  ```

Common problem: when searching in `/` directory, you encounter lots of `Permission denied error`. `find / -type f -perm 777`

There are 2 ways to go around it:

```
find / -type f -perm 777 2>/dev/null
sudo find / -type f -perm 777
```

Wild cards! aka metacharacters

Used for sophisticated pattern matching sequences in several applications in the UNIX operating system (e.g., ls, find, grep, sed, awk, vi, emacs).

- `*` ... a string with any length (including zero)
  - ◇ `ls *.txt` ... files end in *.txt
- `?` ... a character
  - ◇ `ls ??` displays two character files
- `[]` ... match any character in a range
  - ◇ `ls [aeiou]*` ... display files start with a,e,l,o,u
  - ◇ `ls [^a-zA-Z]*` ... a non-alphabetic character
  - ◇ `ls[a-f,o-v]*` ... start with a-e and o-v
- `{}` ... a selection of names or wildcards
  - ◇ `cat {1,2}.txt` ... shows `1.txt` of `2.txt`

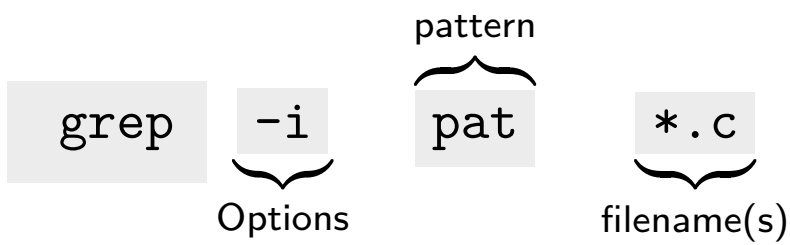
Escaping a metacharacter

- What if you want to search on `.` or remove file named 'chap*'?
  - Use backslash `\` to escape a metacharacter
  - `\.9` will find all occurences of `".9"`
  - `rm chap\*` or `rm 'chap*'`

How to use grep?

- `grep` . . . **g**lobal **r**egular **e**xpression **p**rint
- `grep` scans its input for a pattern, and can display the selected pattern, the line numbers, or the filenames where the pattern occures.

Structure of a `grep` command



`-i`	ignore upper/lower cases match
`-n`	displays line numbers along with lines
`-c`	displays count of number of occurrences

Q How do I find all files containing specific text?

```
grep -rnw /path/to/somewhere/ -e 'pattern'
```

Comments on grep

- You can use regular expressions in patterns and filenames
  `grep "wo[od,de]house" *.c`
  Will print all matches to either `woodhouse` or `wodehouse` in all the C program files located in the current directory.
- Works well with pipe (`|`)
  `ls /usr/bin | grep gcc`
  Prints all executables in `/usr/local` with gcc in the command name.
- Works well with `-exec` command with `find`
  Use `find` to find a file matching some regular expression then use `grep` to look inside the file for an additional pattern.
- see also: the `Perl` script `ack` and also `fzf`
  - ◇ a better `grep`

Powerful `awk` and `sed`

Job control

- A "process" is a running program
- Find out the current running processes
  - `ps` (process status) . . . non-interactive
  - `top` or `htop`, or `bashtop` . . . interactive
  - ▶ check this linkfor more details on htop
- Interup a running processes
  - `Crtl+d` . . . terminate input or exit the shell
  - `Crtl+z` . . . suspend foreground processes
  - `Crtl+c` . . . kill foreground processes

Other job control operations

- `&` ...run at backgorund
- `jobs` ...show the active processes
- `fg` ...bring to foreground
- `bg` ...put to backgournd
- `kill` ...terminate a process
- `kill -15 <pid>` ...to kill a process gracefully
- `kill -9 <pid>` ...to kill a process forcefully

Editing a file in terminal

In order to edit files within the terminal, you need to use a text editor.

The choice of text editor is a personal one.

The two most popular text editors are **vim** and **emacs**

Figure out which one you like, if either I use vim. Both editors have their own keystrokes for doing basic things like file navigation, inserting, deleting, saving, etc.

Check here for a discussion on vi and emacs learning curves

The vi/vim editor

vi

- The standard UNIX file editor
  - Originally written by Bill Joy (of Sun Micro systems) when he was a student at Berkeley working on BSD UNIX
- Found by default on every UNIX system
- `vim` is `vi` improve
  - Written by Bram Moolenaar
  - Standard editor on most Linux systems
  - Today `vi` and `vim` are usually spoken interchangeably to mean `vim`
- To edit a file type: `vi <file>`

vim modes

- Normal mode
  - Move around file
  - Copy and paste text
  - Search and replace
- Insert mode
  - Insert text

Insert mode

Enter insert mode from normal mode

Command	Function
`i`	**i**nsert text left of cursor
`a`	**a**ppend text right of cursor
`I`	**I**nsert text at beginning of line
`A`	**A**ppend text to the end of line
`o`	**o**pen new line below current line
`O`	**O**pen new line above current line
`r`	**r**eplace a single character under cursor
`R`	**R**eplace all text to the right of cursor (overwriting)
`s`	replace a **s**ingle character under cursor and stay in Insert Mode
`S`	replace a **S**ingle character under cursor and all other characters in the line

- use `esc` to get back to normal mode

Motions in vim

Motions are the keys associated with moving around in Vim

- `h`, `j`, `k`, `l` ... left, down, up, right
- `w`, `W` ... to start of next word or WORD
- `b`, `B` ... to start of previous word or WORD
- `e`, `E` ... to end of word or WORD
- `$` ... to end of the line
- `^` ... to first word in line
- `0` or `|` ... to move beginning of the line
- `<numb>G` or `<num>|` ... to line `num` or to column `num`

Scrolling normal mode

Command	Function
`Crtl+f`	scrolls full page forward
`Crtl+b`	scrolls full page back
`Crtl+d`	scrolls half page
`Crtl+u`	scrolls half page back

Simple text editing normal mode

Deleting text

Command	Function
`x`	deletes single character under cursor
`X`	delete text left of the cursor
`dd`	delete entire line

Moving text

Command	Function
`p`	puts text to right of cursor
`P`	Puts text to left of cursor
`J`	Join the current line with the one below
`»`	indent line
`«`	de-indent line

Learn to speak vim!

Using vim is like talking to your editor

Rule 1: `operator motion` sentences

- Learn some verbs (or operators): **v** (visual), **c** (change), **d** (delete), **y** (yank/copy)
- learn some text subjects (or motions): **w** (word), **s** (sentence), **p** (paragraph), **b** (block/parentheses), **t** (tag, works for html)

Speak to the editor in sentences

- `d$` ... delete to end of the line
- `yW` ... copy till end of the WORD
- `cE` ... delete till end of the word and go to insert mode

Rule 2: use COM which means `[count] operator motion`

- `3dw` ... delete a word 3 times

Learn to speak vim!

Rule 3: speak in more details `operator modifier motion`

- learn some modifiers: **i** (inside), **a** (around), **t** (till..finds a char), **f** (find...like till except including the char), **/** (search..find a string)

- `diw` (delete inside word) . . . delete the current word
- `cis` (change inside sentence) . . . change the current sentence
- `ci"` (change inside quote) . . . change a string inside quotes
- `c/foo` (change search foo) . . . change until next occurrence of 'foo'
- `ctX` . . . change everything from here to the letter
- `vap` (visual round paragraph) . . . visually select this paragraph

Rule 4: when an operator is called twice it produce its effects on the entire line

- `dd` . . . delete whole line
- `yy` . . . yank whole line
- `cc` . . . delete whole line and go to insert mode

Recording a macro!

To enter a macro use: `q <letter> <commands> q`

The complete process looks like:

1. `qd` start recording to register `d`
   - Each register is identified by a letter `"a"` to `"z"`
2. $\cdots$ your complex series of commands
3. press `q` to stop recording
4. `@d` execute your macro
5. `@@` execute your macro again

Search and replace

Command	Function
`/<pat>`	searches forward for `<pat>`
`?<pat>`	searches backward for `<pat>`
`n`	repeats search in same direction
`N`	repeats search in opposite direction
`:n1,n2s/<s1>/<s2>`	replaces first occurrence of string or regular expression s1 with string s2 in lines n1 to n2
`:%s/find/look/g`	replaces all occurrences of find

Visual selection

Command	Function
`v`	activates character-by-character highlighting
`V`	activates line-by-line highlighting

Other useful editing commands

Command	Function
`u`	undo last command
`U`	undo all changes to line
`.`	repeat last command
`:ab`	create shortcut
`:ab fb foobar`	shortcuts key sequence fb to foo

Exit vim

Enter **ex** mode with `:`

Command	Function
`:w`	saves file and remains in insert mode
`:x` or `:wq`	saves file and quits insert mode
`:w`	newfile "save as", creates newfile
`:w!`	as above, but overwrite existing file
`:q`	quits vi/vim
`:q!`	quits without making changes
`:!cmd`	runs shell cmd and returns

Tips

- The earlier you get used to navigating with `j` , `k` , `l` , `m` , etc. the better. Don't use the arrow keys!
- Customize your settings (again .vimrc) Learn about vim plugins

Customizing vim

- `vim` can be tailored by redefining keys or abbreviating frequently used expressions.
- Many preset customizations are available via the `:set <command>`
  - Available from **ex** mode
  - `:set all` lists all available preset settings
  - Setting can be undone by `:set no<command>`, for example `set nonumber`

- Some popular `:set` commands are:
  - `autoindent` . . . next line starts at previous indent level
  - `showmatch` . . . shows momentarily a match to a (or {
  - `ignorecase` . . . ignores case when searching patterns

Mapping keyboard keys

- The `:map` command allows you to map keyboard shortcuts
- Examples:
  - Lock line marker at the top: `map j jzt`
  - To map the `g` key in Normal mode to save the buffer (i.e., `:w`)

    `:map g :w^M`
  - ▶ vim interprets the `^M` commands as `Enter`
- To make key sequences in Insert mode use `:map!`
- Use `:unmap` (`unmap!`) to undo key sequence mappings

The .vimrc file

- Place all commonly used `:set` commands, keyboard mappings, and abbreviations in a file named `.vimrc` in `~` directory (create if necessary)
  - ◇ To start you can download a minimal `.vimrc` from course website

```
wget -O ~/.vimrc https://msarrafj.github.io/CAAM519-FA21/Files/dotfiles/vimrc_minimal
```

  - ◇ View or download my everyday `.vimrc` file

```
wget -O ~/.vimrc https://msarrafj.github.io/CAAM519-FA21/Files/dotfiles/vimrc
```

  - ◇ or check Vim Bootsrap for easy configuration set-up
- Keep a master copy of your `.vimrc` file in a Dropbox folder or git repo and symbolically link (`ln -s`) to it on all your UNIX machines so you always have the most up-to-date version

vim plugin for extra features

We first need a plugin manager to install and update plugins. I use vim-plug but there are others (pathogen and vundle)

To start using it

- Download plug.vim and put it int the "autoload" directory

```
curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

- Append the following in `~/.vimrc`

```
call plug#begin('~/.vim/plugged')
Plug 'preservim/nerdtree'
call plug#end()
```

- Install all plugins in `ex` mode

```
:PlugInstall
```

- Source `.vimrc` file

Further resources

There are some excellent resources on the internet to master vim such as:

- Learn Vimscript the Hard Way for general vim script needs
- Writing Vim Plugins for plugins, specifically

Shell offerings

- How to list available valid shells? (check `/etc/shells`)
- Two main categories
  - The Bourne family
    - Bourne (`/bin/sh`), Kron (`/bin/ksh`), Bash (`/bin/bash`), and Zsh (/bin/zsh)
  - The C Shell (/bin/csh)
    - Tsch (`/bin/tcsh`)
- Bash and C are the most common
  - Bash is default on Linux
  - To list your shell invoke `echo $SHELL`

Check this post for more info on different shells

Customizing Bash

Customizing your `.bashrc` file can greatly improve your workflow and increase your productivity. The main benefits of configuring the `.bashrc` file are:

- Permanantly modify **Env**iroment **Var**iables
- Adding **aliases** allows you to type command faster
- Adding **functions** allows to save and return complex codes
- Display useful system information
- Custimomize the Bash prompt (colors, etc)

Add Env Variables

Environment variables are known to the shell and can be used by all programs run by the shell. Three important commands are:

common env variable	significance
HOME	home directory
PATH	list of dir searched by shell to locate command
LOGNAME	login name of user
TERM	type of terminal
PWD	absolute pathname of current dir
PS1	primary prompt string
SHELL	user's login shell

- To add new entries to the existing `PATH` use the notation:
  - `PATH=/my/new/path:$PATH`
    which adds `/my/new/path` to front of existing `PATH`
  - or: `PATH=$PATH:/my/new/path`
    adds `/my/new/path` behind existing `PATH`

Environment variables

- `which` ... shows the command full path (handle different versions?)
- `env` ... list all environment variables/settings
- Use `export VARIABLE=value` to set a new env variable

More on `export`

Type `a=20` ; this defines a variable a which will be known to all subsequent commands you issue in this shell (but not to a new shell). how to make it accessible to every bash? (use `export` command)

```
export a=20
echo $a
```

```
/bin/bash #start a new shell
echo $a
exit
```

→ If we want to make our changes permanent, we can add them to our shell startup files. The startup file name will vary based on OS and shell, but for the bash shell, it is
`~/.bashrc` on Linux or `~/.bash_profile` on MacOS.

Aliases and functions

- `alias <myAlias>='longer command'` assigns shorthand names for common commands. Some useful aliases:

```
alias c='clear'
alias h='history 20'
alias ..='cd ..'
alias ...='cd ../..'
```

- Functions are great for more complicated codes

```
function find_largest(){
    du -h -x -- * | sort -r -h | head -20;
}
```

check this page out to get colored bash and this one for more fun scripts

The initialization scripts

We can set or change the default behavior of the env variables, aliases, etc to persist from *login-to-login* or when a new sub-shell (e.g., `/bin/bash`) is launched. We do this with two files

- A *login script* (also called a *profile*) which is executed only once.
- One of these three files in this order:
  - `~/.bash_profile`
  - `~/.profile`
  - `~/.bash_login`
- A *run command script*, which is executed every time an interactive sub-shell is launched.
  - `~/.bashrc`

→ Never forget to source the scripts

- ◇ `source ~/.bashrc`
- ◇ `. ~/.bashrc`

The login script

In your login script:

- Environment variable changes or additions
- Custom shell variables
- Startup messages
- If you make changes to this file, you will have to source it from the current shell or logout and login
  - ◇ e.g. `source ~/.profie`

Terminal management

tmux is a very useful terminal multiplexer (similar to GNU screen)

- `tmux` ...start tmux
- **prefix key** + `%` or **prefix key** + `"` ...split panes vert or horiz
  - default prefix is `Crtl+b`; but some people map it to `Crtl+a`
- `exit` ...close a pane
- prefix + `c` ...create a new windows
- prefix + `z` ...fullscreen a pane
- prefix + `w` shows all windows

session handling with tmux

- `tmux ls` ... list all running sessions
- prefix + `d` ... detach from tmux (process runs in the bg)
- `tmux attach -t 0` ... reattach to session "0"

→ You can download my ~/.tmux.conf from website

```
wget -O ~/.tmux.conf https://msarrafj.github.io/CAAM519-FA21/Files/dotfiles/tmux.conf
```

Connect to other machines

- Once you know the `username`, `password`, and `IP address` of the remote computer you can connect to it via SSH:

```
ssh <username>@<ip-addr>
```

  - To get `IP address` of a Windows, type `ipconfig` in **powershell** and for any ***nix** machine use `ifconfig` command.
  - use `-X` option for ssh to enable X11 forwarding (access to GUI). It is going to be annoying slow.

practice ssh

Task Connect to Clear, a Linux cluster available to Rice students for class room purpose `<riceID>@ssh.clear.rice.edu` Get the number of CPU and CPU model of this machine

Task Connect from your host machine to your Ubuntu VM.
You may follow theses steps:

i. get the ip address of your VM machine
ii. set an unused port to your VM
iii. enable SSH on VM machine by installing `openssh-server`
iv. start VM in your terminal

```
VBoxManage startvm Ubuntu --type headless
```

v. ssh to your VM

```
ssh -p 3022 <username>@127.0.0.1
```

read more here

To set up passsword-less SSH login (passwd needed only once) via `ssh-keygen` command follow this link.

Data transfer in Linux Systems

- Use `scp` (**S**ecure **C**o**p**y) for file and folder transfers to/from a remote server where you have a user account:

```
scp ./HostFile <username>@<server>:<path/to/file>
scp <username>@<server>:<path/to/file/RemoteFile> .
```

- first command transfers a HostFile in current dir to remote server at `path/to/file` directory
- second command transfers RemoteFile in remote server to current directory (i.e. `.`).
- Use `-r` option (**r**ecursive) if you want to transfer a directory

Archiving/pack/unpack files

Use `tar` command

1. **C**reate an archive from foo directory

```
tar -cvf target.tar file1 file2 file3
```

   ◇ `-z` **compresses** the resulting archive with gzip(1)

```
tar -cvzf target.tar.gz file1 file2 file3
```

2. **Un**packing/**ex**tracting a file in verbose mode:

```
tar -xvzf /path/to/foo.tar.gz
tar -xvzf /path/to/foo.tar.gz -C /path/to/directory
```

3. List the contents of a tar file:

```
tar -tvf source.tar.gz
```

practice tar and scp

Task

- download "hw1_materials.tar.gz" from course website
- unpack it
- transfer "us-states.csv" to home directory of your virtual machine

Write a shell script

- The convention is to use the ".sh" extension
- Can pass command line arguments to the script
- Loop and choice constructs are available
  - if . . . then . . . else . . . fi
  - while . . . do . . . done
  - for . . . do . . . done
  - Combine above to create powerful tools

bash scripting: hello world

```
1 #!/bin/bash
2 echo "Hello world"
3
4 # Each command starts on a new line, or after a semicolon:
5 echo 'This is the first line'; echo 'This is the second line'
```

Note: when you create your bash script, you will have to change the "permissions" of it to run, via

```
chmod +x hello_world.sh
```

and for running it you use the command

```
./hello_world.sh
```

declare variables

- first line is the shebang which tells the system how to execute the script
- comments start with #
- spaces (unlike indentation) are very important

```
#!/usr/bin/env bash
# Declaring a variable looks like this:
Variable="Some string"
# But not like this:
Variable = "Some string" # => returns error "command not found"
# Nor like this:
Variable= 'Some string' # => returns error: "command not found"

# Using the variable:
echo $Variable # => Some string
echo "$Variable" # => Some string
echo '$Variable' # => $Variable

# Parameter expansion ${ }:
echo ${Variable} # => Some string
# String substitution in variables
echo ${Variable/Some/A} # => A string
# This will substitute the first occurrence of "Some" with "A"
echo ${#Variable} # => 11 (which is string length)
```

declare arrays

```
#! /bin/bash
# Declare an array with 6 elements
array0=(one two three four five six)
# Print first element
echo $array # => "one"
# Print first element
echo ${array[0]} # => "one"
# Print all elements
echo ${array[@]} # => "one two three four five six"
# Print number of elements
echo ${#array[@]} # => "6"
# Print number of characters in third element
echo ${#array[2]} # => "5"
# Print 2 elements starting from forth
echo ${array[@]:3:2} # => "four five"

# Print all elements. Each of them on new line.
for i in "${array0[@]}"; do
    echo "$i"
done
```

built-in variables

- There are some useful built-in variables like

```
1 echo "Last program's return value: $?"
2 echo "Script's PID: $$"
3 echo "Number of arguments passed to script: $#"
4 echo "All arguments passed to script: $@"
5 echo "Script's arguments separated into different variables: $1
  ↪  $2..."
```

- we can also get access to built-in (or env) variables

```
1 echo "I'm in $(pwd)" # execs `pwd` and interpolates output
2 echo "I'm in $PWD" # interpolates the variable
```

more bash

- reading a value from input

```
1 echo "What's your name?"
2 read Name # Note that we didn't need to declare a new variable
3 echo Hello, $Name!
```

Task Write a script that reads your first and last name (two variables) from input and print

```
1 START oF FILE
2 my name is:
3 END oF FILE
```

bash conditionals

- There is also conditional execution

```
1 echo "Always executed" || echo "Only executed if 1st command
  ↪  fails"
2 # => Always executed
3 echo "Always executed" && echo "Only executed if first command
  ↪  does NOT fail"
4 # => Always executed
5 # => Only executed if first command does NOT fail
```

- To use && and || with if statements, you need multiple pairs of square brackets:

```
1 if [ "$Name" == "Steve" ] && [ "$Age" -eq 15 ]
2 then
3     echo "This will run if $Name is Steve AND $Age is 15."
4 fi
5
6 if [ "$Name" == "Daniya" ] || [ "$Name" == "Zach" ]
7 then
8     echo "This will run if $Name is Daniya OR Zach."
9 fi
```

- `=` and `==` are for string comparisons, -eq is for numeric ones. -eq is in the same family as -lt, -le, -gt, -ge, and -ne

for loop

- **For** loops iterate for as many args givens

```
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done
# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
done
```

- They can also be used to act on files..

```
# This will run the command `cat` on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done
# This will `cat` the output from `ls`.
for Output in $(ls)
do; cat "$Output"; done
```

while loop

- **While** the condition is met keep iterating

```
# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done
# => loop body here...
```

bash functions

```
# Definition:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    return 0
}
# Call the function `foo` with two arguments, arg1 and arg2:
foo arg1 arg2
# => Arguments work just like script arguments: arg1 arg2
# => And: arg1 arg2...
# => This is a function

# or simply
bar ()
{
    echo "Another way to declare functions!"
    return 0
}
# Call the function `bar` with no arguments:
bar # => Another way to declare functions!

# Calling your function
foo "My name is" $Name
```

some gotchas

- never use `test` as the name of a variable or shell script file
- when using `=` as an assignment operator, do not put blanks around it
- when using `=` as a comparison operator, you must put blanks
- when using `if [ ]` put spaces around the brackets

→ see https://devhints.io/bash for bash scripting syntax.

final example

Task Can you explain what does this program do

```
#! /bin/sh
# list names of all files containing given words
if [ $# -eq 0 ]
    then
    echo "enter word1 word2 word3 ..."
    echo "lists names of files containing all given words"
    exit 1
fi
for fyle in *; do
    bad=0
    for word in $*; do
        grep $word $fyle > /dev/null 2> /dev/null
        if [ $? -ne 0 ]
            then
            bad=1
            break
        fi
    done
    if [ $bad -eq 0 ] then
        echo $fyle
    fi
done
exit 0
```