Operating Systems!

OS Tech Bootcamp
(Part 2)

Prof. Travis Peters

Montana State University

CS 460 - Operating Systems

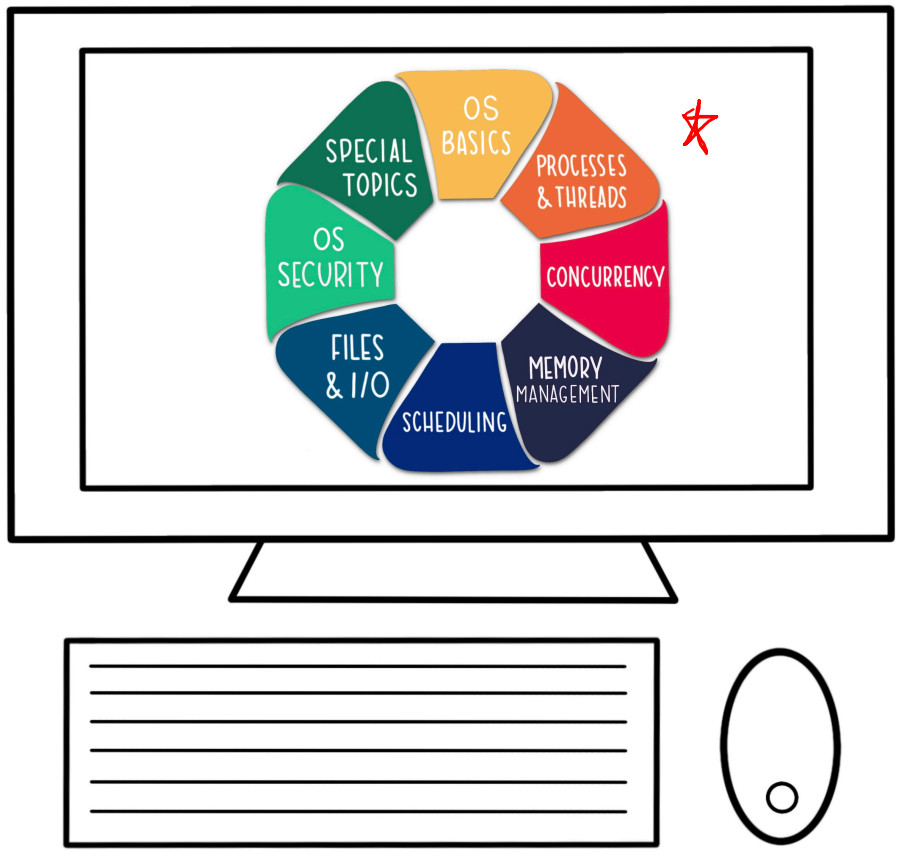
Fall 2020

https://www.cs.montana.edu/cs460

OS Tech Bootcamp (Overview)

- ~~VirtualBox & Vagrant~~
  - ~~A sandbox~~
  - ~~standardized work environment~~
- Basic Command Line
  - An interface to the computer / OS
- Working in C
  - Automating compilation w/ make
  - Debugging w/ gdb
- Teamwork & Communication
  - READMEs ~~& Markdown~~
  - Git & GitHub
- *Coming Up:*
  - System programming (fork(), exec())
  - OS internals (e.g., procs, threads)

Meet Reese!



Throughout "bootcamp" we will work through (many) parts of PA0.

Markdown Activity (Recap? Breakout?)

.md = markdown

➔ *See Task 2: READMEs & Markdown (PA0)*

Projects?

pandoc

mbedTLS

⋮

Observations?

Troubleshooting section

Setup (how to get code up & running)

Overview

Documenting features

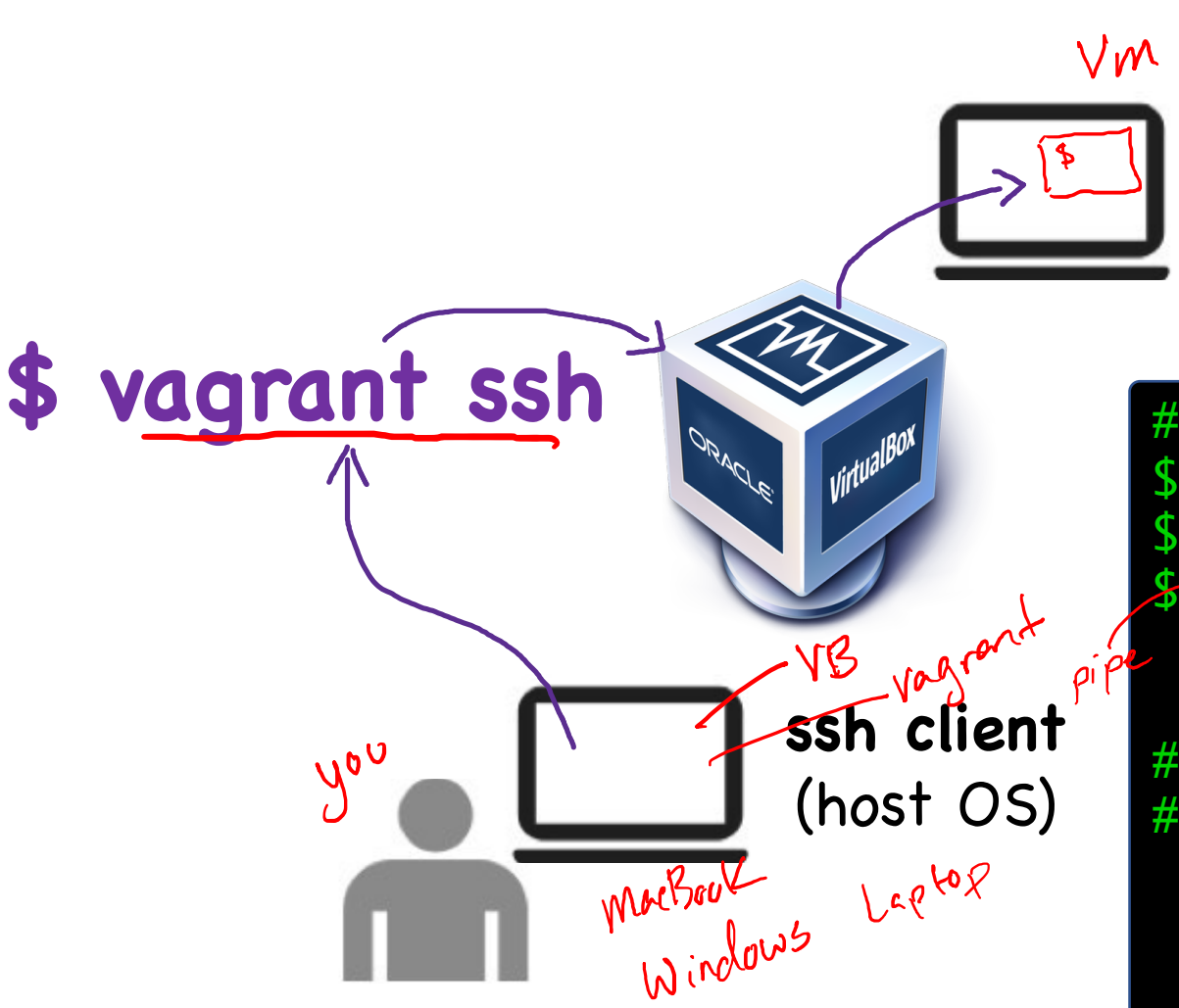
→ examples (code section)

Command Line (Task 4)

Because us humans need a way to control and interact with our computers!

Command Line Activity

See Task 4: Command Line (PA0)



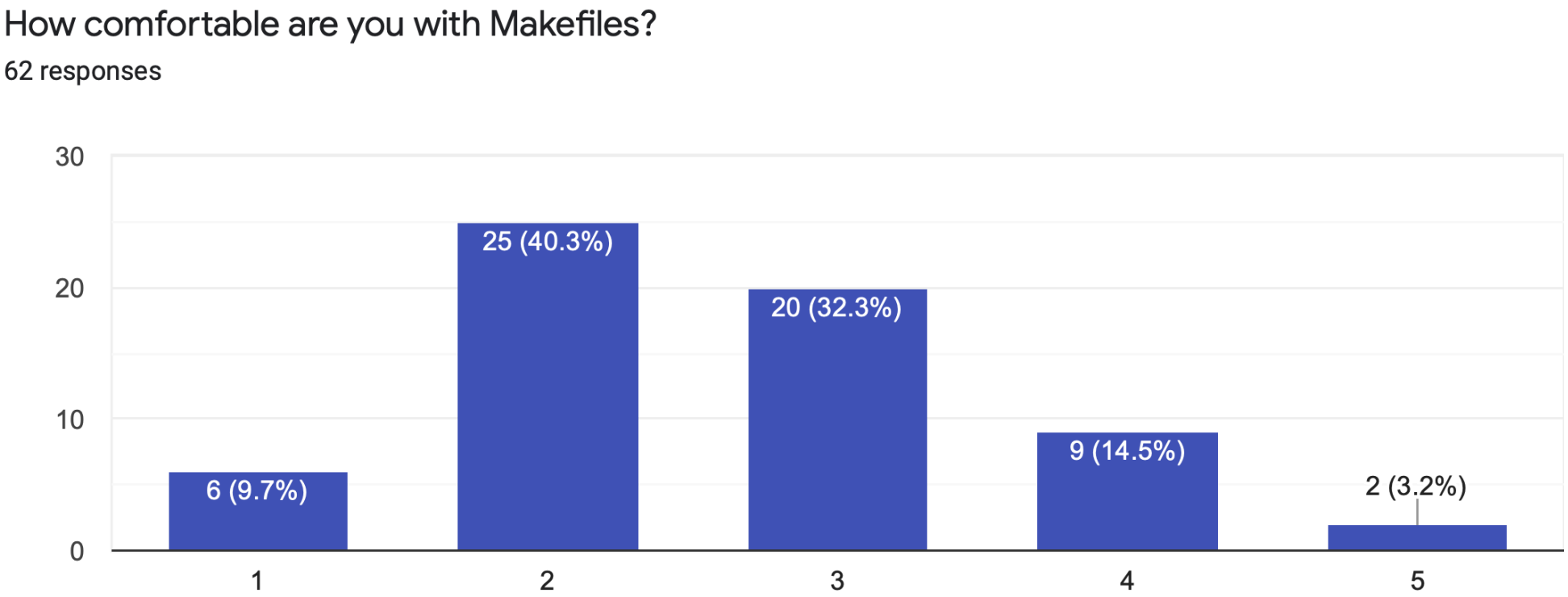

```
# Task 4, Part 1 (a few commands)
$ cd ~
$ env | grep PATH
$ sudo lshw -html > hardwareinfo.html

# Task 4, Part 3
# Some of your Favorite commands?
```

Makefiles (Task 5)

Because having a language/tool to automate more complex compilations (and other stuff) is a game changer!

You Tell Me: Why Do We Need Makefiles?



Automate Something *(Breakout Activity!)*

Your Objective:

How can you explain to me, with ZERO AMBIGUITY, how to get dressed before going outside.

* wear the same thing each day

↓

step by step commands / script

take into account
- preference
- weather
- social norms (socks)
- health concerns

Automating Builds

```
target: [dependencies]
    <shell command to execute>
    <shell command to execute>
```

- Make is a language for automating large compilations.
- You need to recompile a file ("target") only if it is older than one or more of its dependencies. *(Thus, you don't need to recompile if the target exists AND it is newer than all of its dependencies.)*

Put Simply:

Makefiles are just text files...

...with clear rules that express how to do something...

*...like **build files** you want to build or **do some task** in a logical order!*

A Non-Technical Example

See Task 5: Makefiles (PA0)

Makefile

```
dress: trousers shoes jacket
    @echo "All done. Let's go outside!"
jacket: pullover
    @echo "Putting on jacket."
pullover: shirt
    @echo "Putting on pullover."
shirt:
    @echo "Putting on shirt."
trousers: underpants
    @echo "Putting on trousers."
underpants:
    @echo "Putting on underpants."
shoes: socks
    @echo "Putting on shoes."
socks: pullover
    @echo "Putting on socks."
```

Thanks! *Afraid of Makefiles? Don't be!*
https://endler.dev/2017/makefiles/

1. Download this file
(See our GitHub: /week02/makefile-intro/Makefile)

2. SSH into your VM and change to the directory where you put your Makefile. Run the Makefile:

```
$ make
```

3. What do you see? Does it make sense? Explain...

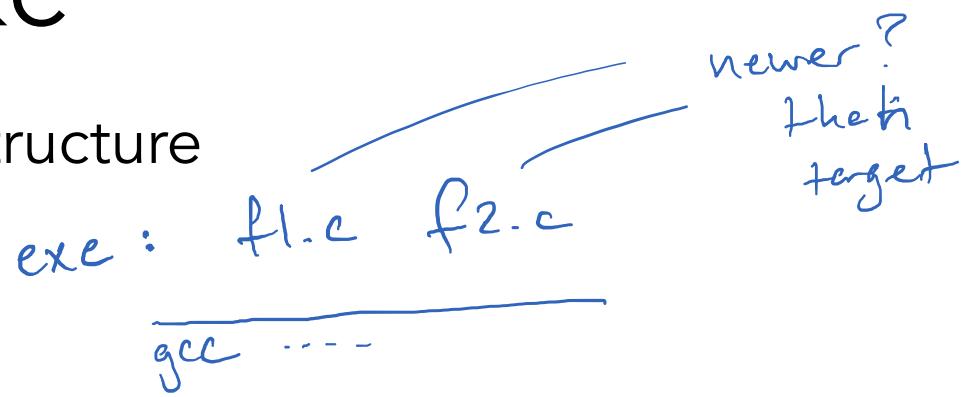
< Breakout Activity! >

```
Putting on underpants.
Putting on trousers.
Putting on shirt.
Putting on pullover.
Putting on socks.
Putting on shoes.
Putting on jacket.
All done. Let's go outside!
```

Automating Builds w/ Make

- In general, each build step has the following structure

```
target: [dependencies]
    <shell command to execute>
    <shell command to execute>
```



- Some good things to knows...
  - The first target in a Makefile will be executed by default if you run make w/out args
  - Shell commands MUST be indented with a TAB
  - .PHONY: *list targets here if target isn't a file you build (e.g., clean, install, run, check)*

***Congrats!* You now know 90% of what you really need to know about Make and Makefiles!**

Really, the rest is mostly just abbreviations, macros, and default rules that help you write shorter/simpler Makefiles

Automating Builds w/ Make *(cont.)*

- A few things that you might come across:

```
# CC is a default macro that you can override to set the default compiler
CC = gcc

# CFLAGS is often used to hold flags given to gcc for compilation.
# To use CFLAGS, you need to put it inside "$(...)" – see below.
CFLAGS = -g -DDEBUG

# Make supports pattern rules using '%' as a wildcard
%.o: %.c
    gcc $(CFLAGS) -c $<

# Here is an example from Makefiles we will use & look at (not shown here: definitions for PROG, OBJS, LIBS)
$(PROG): $(OBJS)
    $(CC) $(CFLAGS) $^ $(LIBS) -o $@
```

See also: Makefile-ideas

In our GitHub: /week02/makefile-intro/Makefile-ideas

- Oh and FYI
  - **@** = *put before a shell command; suppresses echoing command when running it*
  - **$@** = *full name of the target*
  - **$^** = *ALL of the files listed on the dependency line (everything after ':')*
  - **$<** = *the FIRST dependency listed on the dependency line*