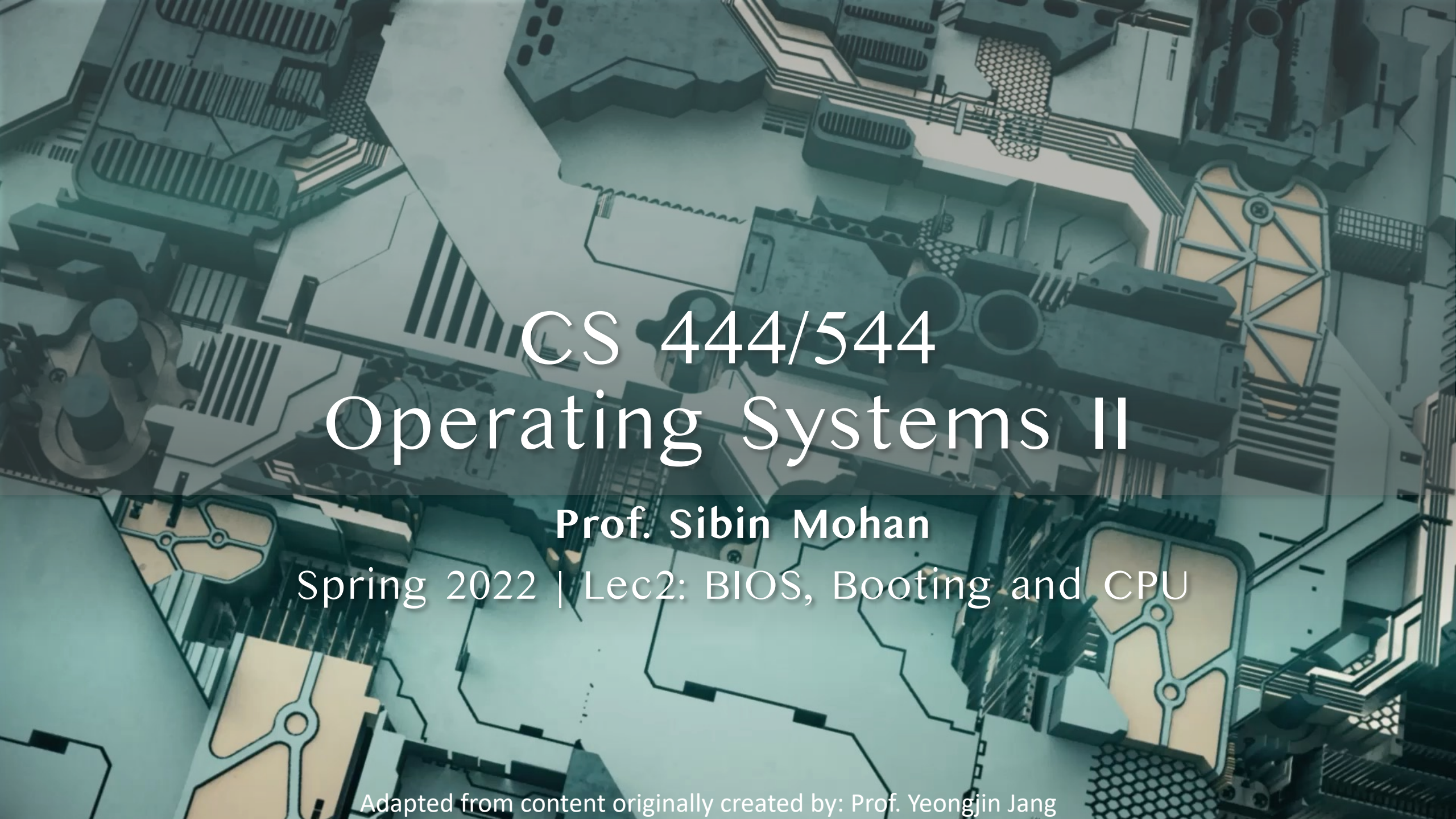


CS 444/544
Operating Systems II

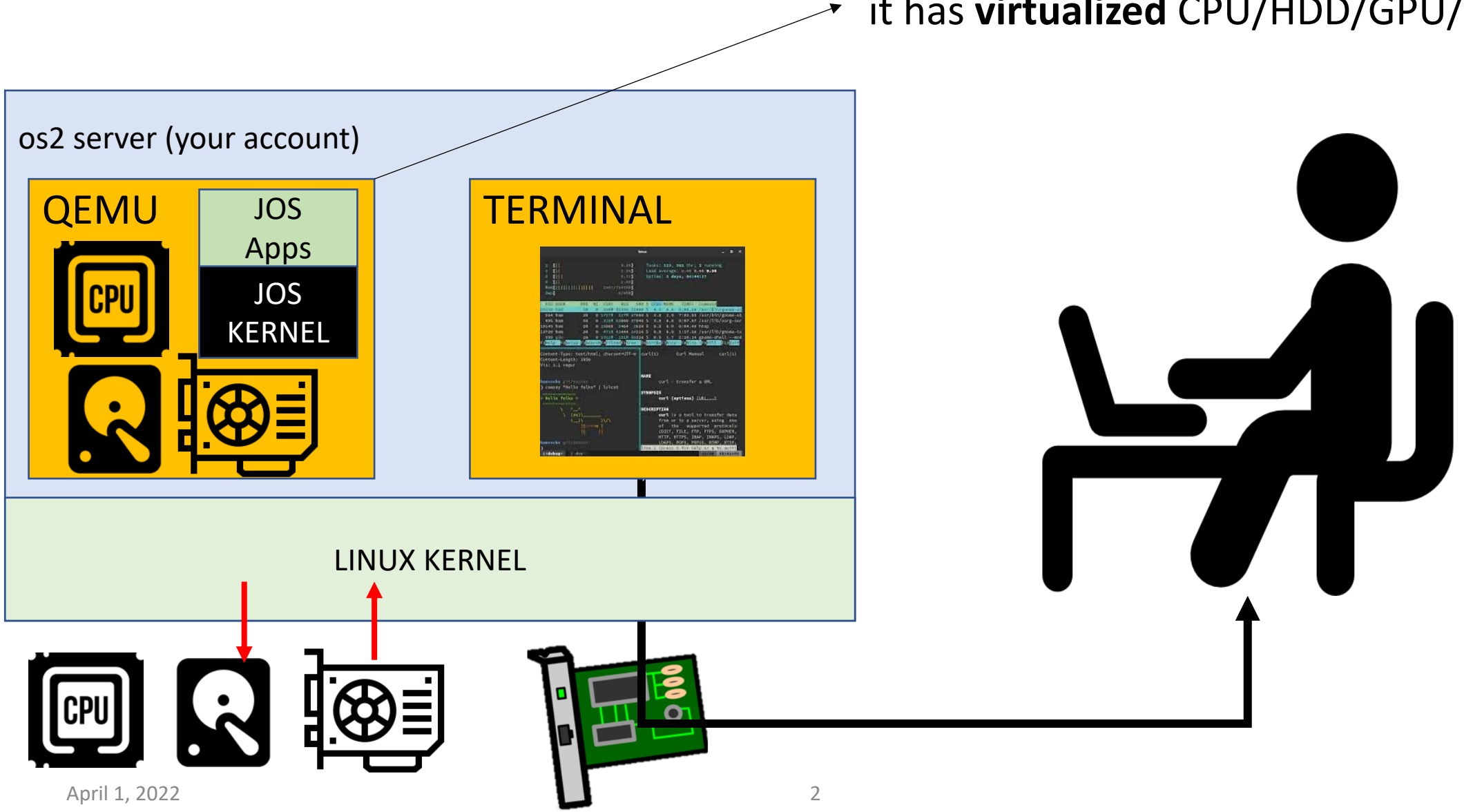
Prof. Sibin Mohan

Spring 2022 | Lec2: BIOS, Booting and CPU

Adapted from content originally created by: Prof. Yeongjin Jang

How JOS Works?

QEMU: an **emulator**,
it has **virtualized** CPU/HDD/GPU/NIC, etc.



Starting Lab Assignments

- Follow the guidelines from **lab1 tutorial**
- Setup your lab environment on os2 server

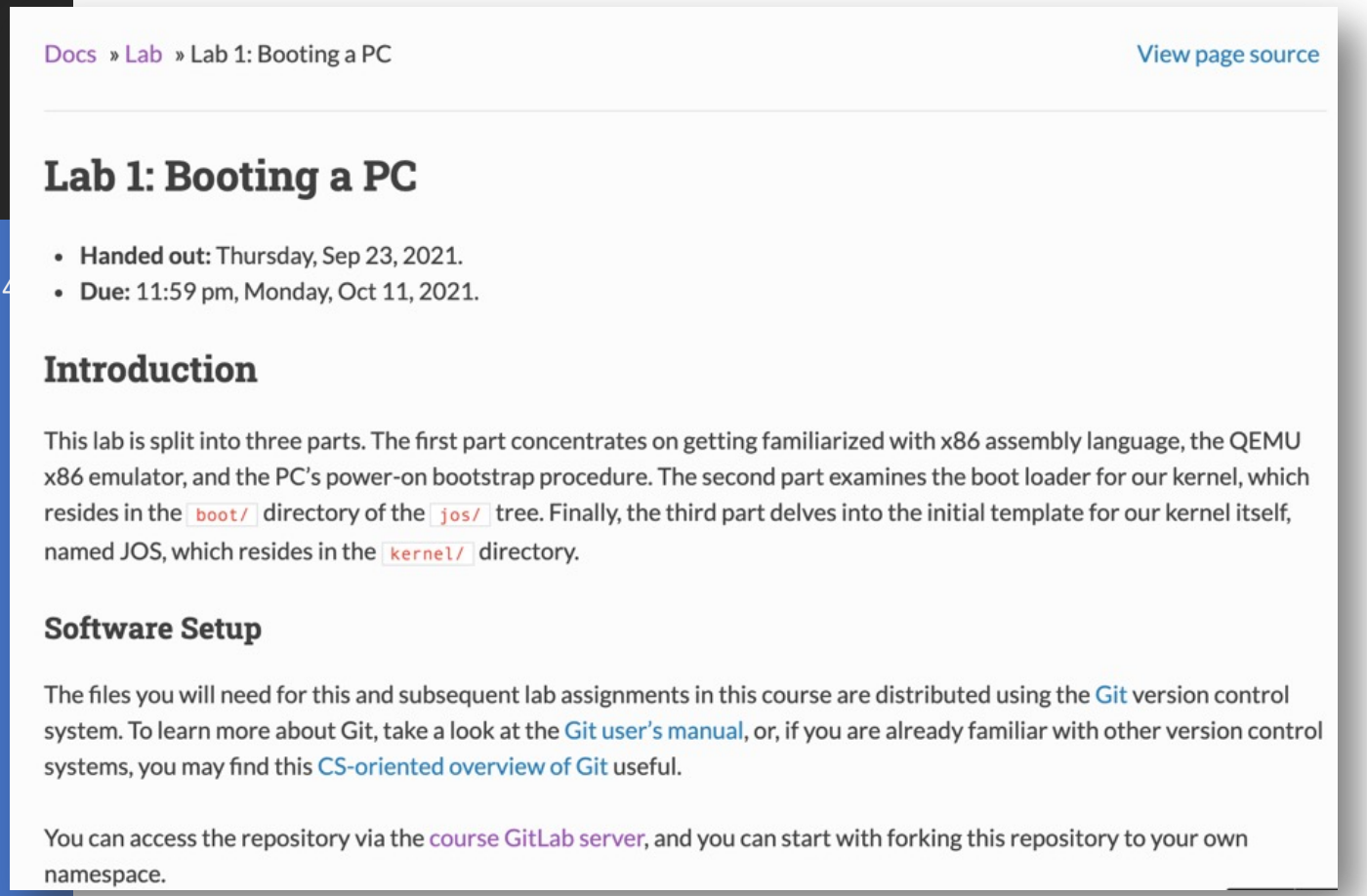
- Register at: **gitlab.unexploitable.systems**
  - Register an SSH key
  - Fork repository and Change visibility → private

- Get familiar to tools such as
  - GDB
  - TMUX

https://sibin.github.io/teaching/cs444-osu-operating-systems/spring_2022/lab.html

Solving Lab Assignments

Read lab guidelines **thoroughly** and **follow the instructions**



Docs » Lab » Lab 1: Booting a PC View page source

Lab 1: Booting a PC

- **Handed out:** Thursday, Sep 23, 2021.
- **Due:** 11:59 pm, Monday, Oct 11, 2021.

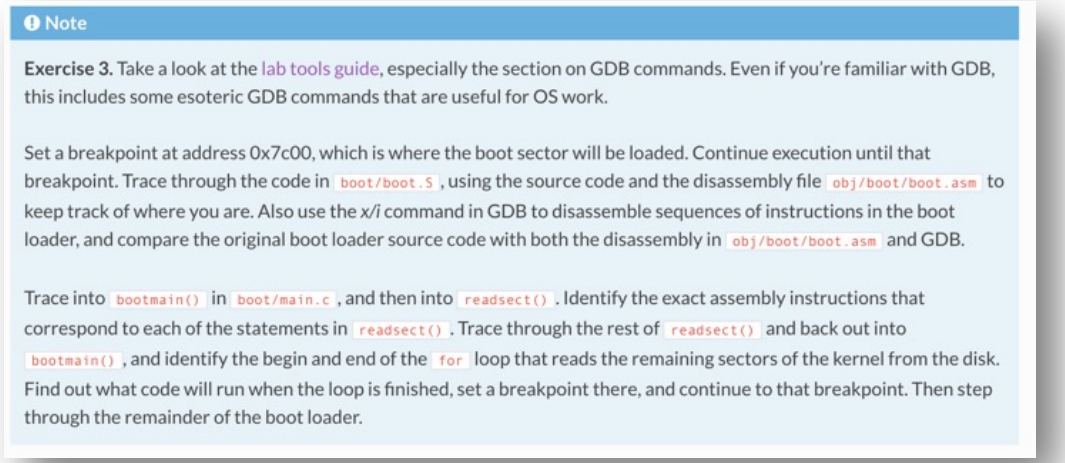
Introduction

This lab is split into three parts. The first part concentrates on getting familiarized with x86 assembly language, the QEMU x86 emulator, and the PC's power-on bootstrap procedure. The second part examines the boot loader for our kernel, which resides in the `boot/` directory of the `jos/` tree. Finally, the third part delves into the initial template for our kernel itself, named JOS, which resides in the `kernel/` directory.

Software Setup

The files you will need for this and subsequent lab assignments in this course are distributed using the Git version control system. To learn more about Git, take a look at the Git user's manual, or, if you are already familiar with other version control systems, you may find this CS-oriented overview of Git useful.

You can access the repository via the course GitLab server, and you can start with forking this repository to your own namespace.

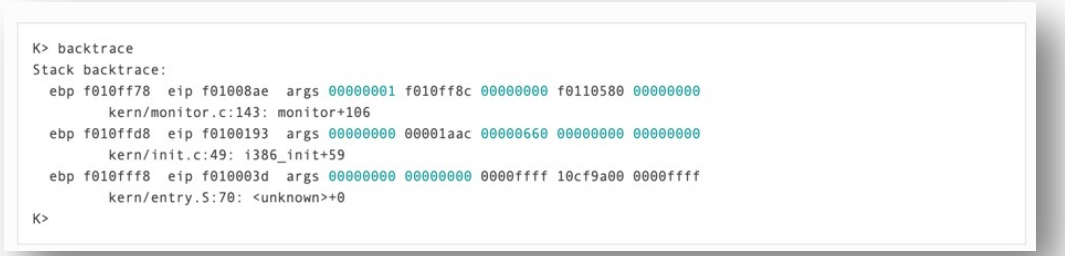


Note

Exercise 3. Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the *x/i* command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.



```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

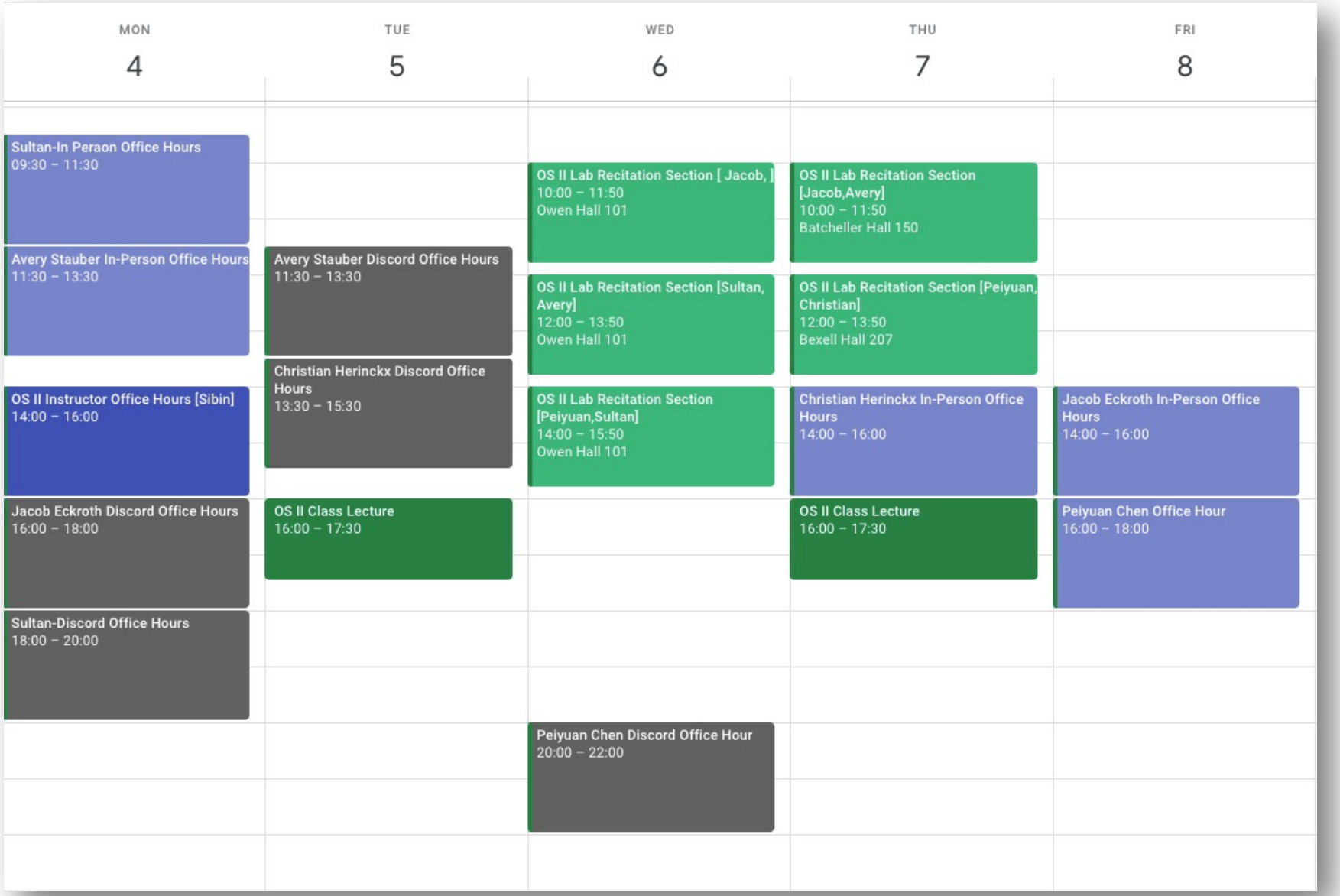
Lab Assignments

- Lab tutorial videos/slides are supplementary to the lab guideline text
  - VIDEO: https://www.youtube.com/watch?v=rj3pVybg2CA
  - Slides: https://sibin.github.io/teaching/cs444-osu-operating-systems/spring_2022/l/tutorial1_labsetup.pdf

- CODE/DEBUG
  - Write your code by following the guideline/tutorial videos
  - Debug your code using **gdb**

- **ENGAGE → questions and help others during office hours and on Discord**!

Office Hours



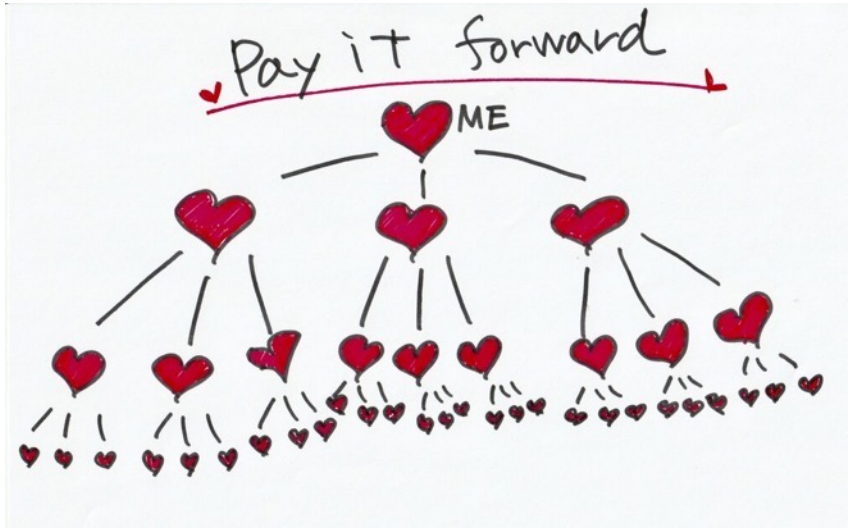
MON 4	TUE 5	WED 6	THU 7	FRI 8
Sultan-In Peraon Office Hours 09:30 – 11:30		OS II Lab Recitation Section [Jacob,] 10:00 – 11:50 Owen Hall 101	OS II Lab Recitation Section [Jacob,Avery] 10:00 – 11:50 Batcheller Hall 150	
Avery Stauber In-Person Office Hours 11:30 – 13:30	Avery Stauber Discord Office Hours 11:30 – 13:30	OS II Lab Recitation Section [Sultan, Avery] 12:00 – 13:50 Owen Hall 101	OS II Lab Recitation Section [Peiyuan, Christian] 12:00 – 13:50 Bexell Hall 207	
OS II Instructor Office Hours [Sibin] 14:00 – 16:00	Christian Herinckx Discord Office Hours 13:30 – 15:30	OS II Lab Recitation Section [Peiyuan,Sultan] 14:00 – 15:50 Owen Hall 101	Christian Herinckx In-Person Office Hours 14:00 – 16:00	Jacob Eckroth In-Person Office Hours 14:00 – 16:00
Jacob Eckroth Discord Office Hours 16:00 – 18:00	OS II Class Lecture 16:00 – 17:30		OS II Class Lecture 16:00 – 17:30	Peiyuan Chen Office Hour 16:00 – 18:00
Sultan-Discord Office Hours 18:00 – 20:00				
		Peiyuan Chen Discord Office Hour 20:00 – 22:00		

Pay it Forward

- We only have 5 full GTAs for 200+ students
  - There could be long latency to get your queries answered

- There are many ways you can get help quickly
  - 1. Post your questions on the 'labX' channel [e.g., lab1 for lab1 questions]
  - 2. DM TAs for their office hours [**LOTS of office hours!**]
  - 3. Post your questions on CANVAS 'discussion'

We need your help, so please help others if you know how to handle the questions posted on Discord/CANVAS



Some issues and Problem Solving

Failed to bind socket: Address already in use

```
$ kill-qemu
```

```
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,f
ormat=raw -serial mon:stdio -gdb tcp::29007 -D qemu.log
qemu-system-i386: -gdb tcp::29007: Failed to bind socket: Address already in use
make: *** [qemu-nox] Error 1
[coe_jangye@os2 (lab1) ~/jos$]
```

kill-qemu

- kill **all running qemu instances**
  - Only your instances! ☺
- Please ignore the error message
  - Trying to kill others' instances

```
[coe_jangye@os2 (lab1) ~/jos$] kill-qemu
pkill: killing pid 8876 failed: Operation not permitted
pkill: killing pid 27893 failed: Operation not permitted
pkill: killing pid 55242 failed: Operation not permitted
pkill: killing pid 55441 failed: Operation not permitted
pkill: killing pid 84173 failed: Operation not permitted
pkill: killing pid 89136 failed: Operation not permitted
pkill: killing pid 104933 failed: Operation not permitted
pkill: killing pid 112678 failed: Operation not permitted
pkill: killing pid 132572 failed: Operation not permitted
pkill: killing pid 211668 failed: Operation not permitted
pkill: killing pid 227128 failed: Operation not permitted
pkill: killing pid 245304 failed: Operation not permitted
pkill: killing pid 272435 failed: Operation not permitted
pkill: killing pid 272493 failed: Operation not permitted
pkill: killing pid 273868 failed: Operation not permitted
pkill: killing pid 292464 failed: Operation not permitted
pkill: killing pid 295167 failed: Operation not permitted
pkill: killing pid 308600 failed: Operation not permitted
pkill: killing pid 319294 failed: Operation not permitted
[coe_jangye@os2 (lab1) ~/jos$] make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,f
ormat=raw -serial mon:stdio -gdb tcp::29007 -D qemu.log
444544 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
```

Device or Resource Busy…

- Your tmux/vim/other apps are working on files that our make script is trying to delete
- **Killing all tmux/vim sessions** will resolve this
  - **Make sure that you saved all your work!**

```
[coe_jangye@os2 (lab1) ~/jos$] make grade
make clean
make[1]: Entering directory `/nfs/stak/users/coe_jangye/jos'
rm -rf obj .gdbinit jos.in qemu.log
rm: cannot remove ‘obj/boot/.nfs00000000b4434e8600000025’: Device or resource busy
make[1]: *** [clean] Error 1
make[1]: Leaving directory `/nfs/stak/users/coe_jangye/jos'
'make clean' failed.  HINT: Do you have another running instance of JOS?
make: *** [grade] Error 1
```

Killing tmux and vim instances

- To kill tmux, run:
  ```
  $ kill-all-tmux
  ```
- Killing vim (editor) instances
  ```
  $ ps aux | grep vim | grep your_u
  ```
  - This command will show your vim instances
  - You can kill it selectively by running:
    ```
    $ kill -9 [pid of vim]
    ```
  - Or
    ```
    $ pkill -9 vim
    ```
  - To kill all vim instances

Add ~/bin to PATH in your .*shrc

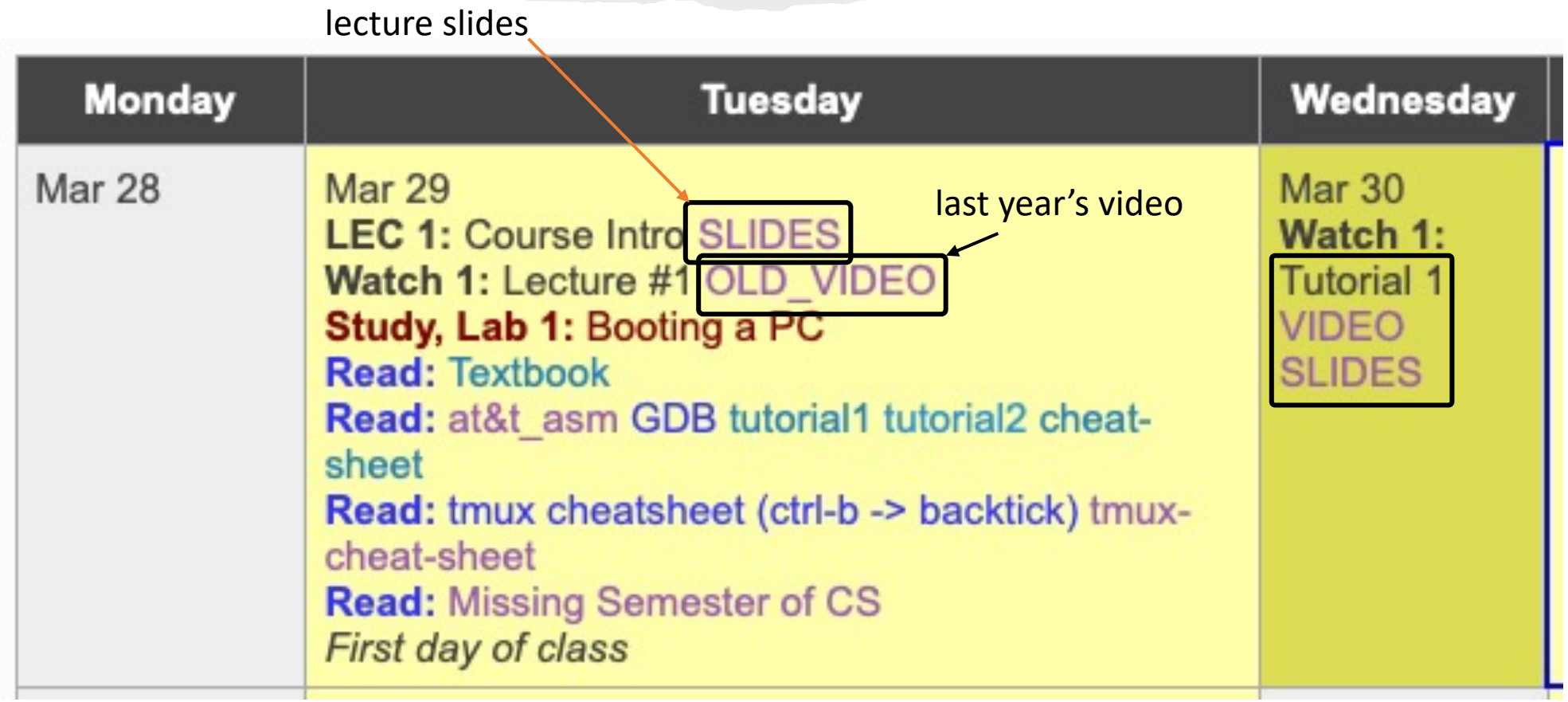
- We use a special version of qemu-system-i386
- For students who typed '**n**' during .bashrc installation,
- Please add ~/bin to your PATH environmental variable
  - export PATH=$PATH:~/bin

- This will remove the errors such as

```
***
*** Error: Couldn't find a working QEMU executable.
*** Is the directory containing the qemu binary in your PATH
*** or have you tried setting the QEMU variable in conf/env.mk?
***
```

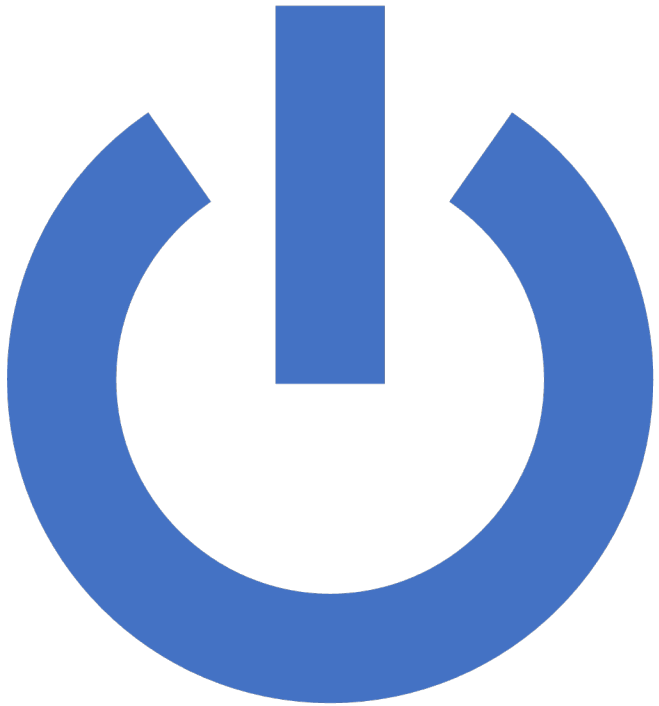
Readings, Slides, and Videos are Available from the course Website

https://sibin.github.io/teaching/cs444-osu-operating-systems/spring_2022/index.html



Booting

April 1, 2022



What happens when you turn on your computer/mobile phone/device?

Five [Basic] Steps of Boot Process

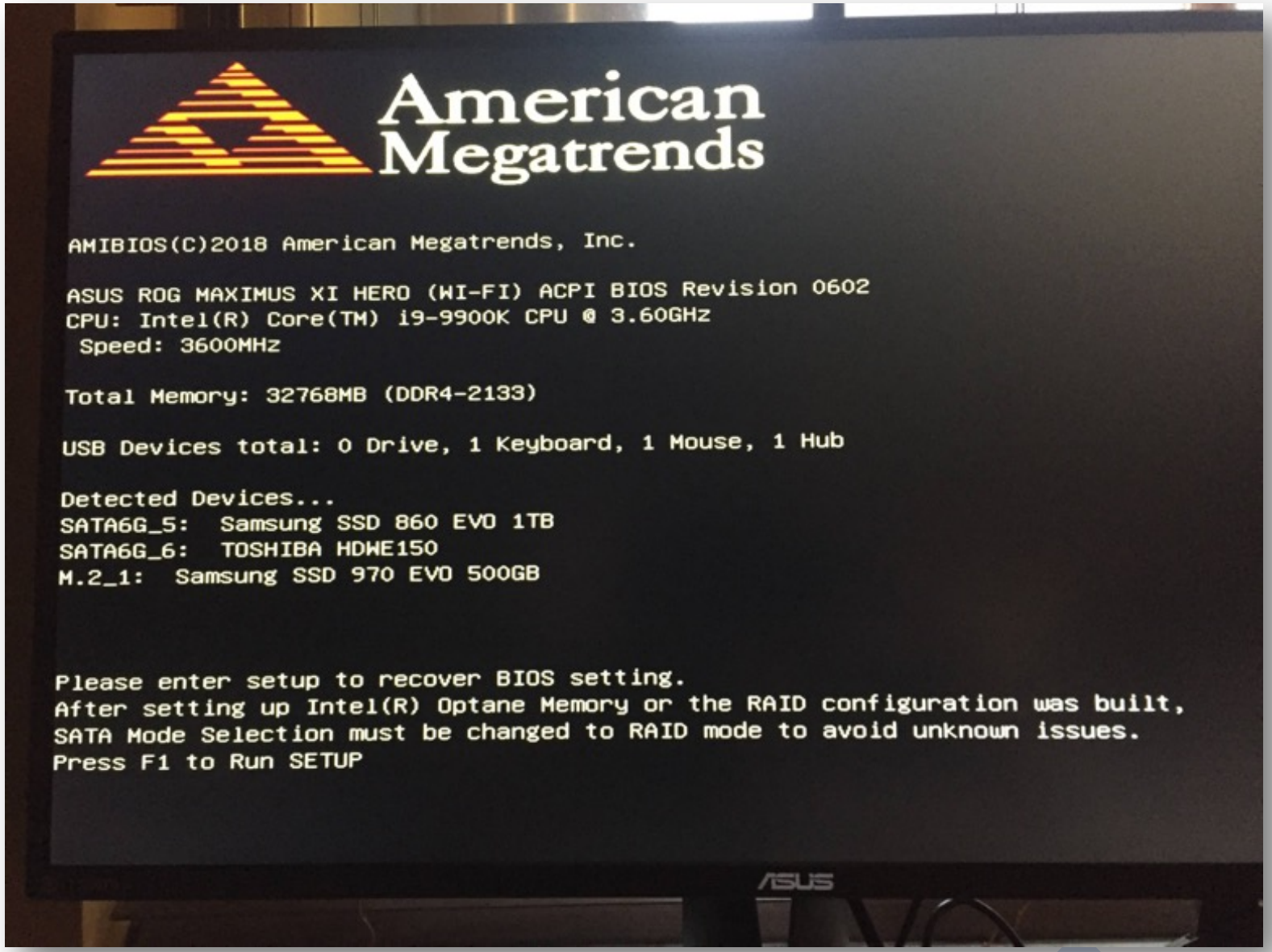
Power up

Power On Self Test [POST]

Finding a **boot device**

Load the **operating system**

Transfer control

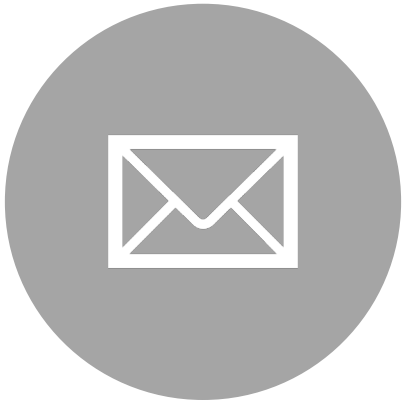

American Megatrends
AMIBIOS(C)2018 American Megatrends, Inc.
ASUS ROG MAXIMUS XI HERO (WI-FI) ACPI BIOS Revision 0602
CPU: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
Speed: 3600MHz
Total Memory: 32768MB (DDR4-2133)
USB Devices total: 0 Drive, 1 Keyboard, 1 Mouse, 1 Hub
Detected Devices...
SATA6G_5: Samsung SSD 860 EVO 1TB
SATA6G_6: TOSHIBA HDWE150
M.2_1: Samsung SSD 970 EVO 500GB
Please enter setup to recover BIOS setting.
After setting up Intel(R) Optane Memory or the RAID configuration was built,
SATA Mode Selection must be changed to RAID mode to avoid unknown issues.
Press F1 to Run SETUP
ASUS

POST
[Power On Self Test]

- **Hardware** portion of the boot process
- Nothing to do with the OS (same across any OS)
- Ensure that basic computer hardware is working correctly
  - Memory, disk, ROM, etc.
- If POST fails → **computer not usable and shuts down**!
- POST is part of the **BIOS**

BIOS

BASIC INPUT/OUTPUT SYSTEM



HANDLES THE ACTIVATION OF POST

INITIATES THE **BOOT SEQUENCE**

Phoenix - Award WorkstationBIOS CMOS Setup Utility
Advanced BIOS Features

Anti-Virus Protection	[Disabled]	Item Help
CPU L1 & L2 Cache	[Enabled]	
CPU Hyper-Threading	[Enabled]	Menu Level ►
CPU L2 Cache ECC Checking	[Enabled]	Allows you to choose the VIRUS warning feature for IDE Hard Disk boot sector protection. If this function is enabled and someone attempt to write date into this
Quick Power ON Self Test	[Enabled]	
First Boot Device	[Floppy]	
Second Boot Device	[HDD-0]	
Third Boot Device	[CDROM]	
Boot Other Device	[Enabled]	
Swap Floppy Drive	[Disabled]	
Boot up NumLock Status	[On]	
Gate A20 Option	[Fast]	

Boot Sequence

- Once BIOS POST completes successfully
- Initiate the **boot sequence**
  - Essentially start the software components
  - A first step to get the OS up and running
- Locate the "**boot sector**" on any attached bootable devices

INT 13H

- First valid boot sector found ➔ load into RAM [Note: BIOS in **ROM**]
- Boot sector is first part of boot loader

Linux Bootloader [Grub2]

- Grand Unified BootLoader v2
- Makes computer just smart enough to find OS and load into memory
- Stages of Grub:
  - Stage 1: load **boot record**
  - Stage 1.5: load a few common drivers, mainly the **filesystem** [EXT, NTFS, etc]
  - Stage 2: locate and load linux kernel into RAM

Grub2 Stage 1

Search for boot record (called “master boot record”) & load into memory

Start executing boot record

boot.img

Boot record is very small → must fit into fist 512 **bytes** of memory!

Grub2 Stage 1.5

Contains larger code such as filesystem drivers

Code is more complex than stage 1

core.img

Stage 2 can actually **be on the filesystem**!

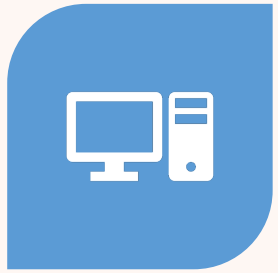
Grub2 Stage 2

LOCATED IN THE **/BOOT/GRUB2** DIRECTORY AND SUBDIRECTORIES

DOES NOT HAVE ITS OWN IMAGE FILE

(LIKE BOOT.IMG OR CORE.IMG)

KERNEL AND FILES ARE IN **/BOOT** DIRECTORY [VMLINUZ]



CAN BOOT FROM MULTIPLE KERNEL VERSIONS

Loads kernel into memory and turns control over to it!

Five [Basic] Steps of Boot Process

Power up

Power On Self Test [POST]

Finding a **boot device**

Load the **operating system**

Transfer control

April 1, 2022

Let’s Dive in!

Boot Sequence Details

- Intel Architecture + JOS as examples
- First step ➔ POST
- After testing/initializing peripheral devices
  - copy initialization code to DRAM [copy from ROM to RAM]

0xffff0 [f000:fff0]

```
[coe_jangye@os2 (lab1) ~/jos$] gdb
+ target remote localhost:29007
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
```

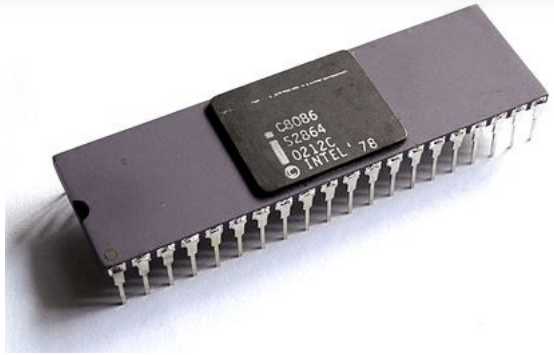
  - **RUN code** from the RAM

What Does Initialization Code Do?

- BIOS load and run boot sector from disk
  - Read the 1st sector from the boot disk (**512 bytes**)
  - Put the sector at **0x7c00**
  - **Run it!** (set the instruction pointer = **0x7c00**)

What is i8086?

```
The target architecture is assumed to be i8086
[f000:fff0]       0xffff0: ljmp   $0xf000,$0xe05b
```



- Intel 8086 (1978, ~45 years old, runs @ **5MHz**)
  - 16-bit processor; all registers are **16-bits**
- BIOS assumes our processor is **i8086**
  - We are living in 2022 and Intel Xeon on the os2 server

```
model name      : Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz
```

- Why?
  - **Backward Compatibility**
  - **Use the same code for all CPUs!**

What is [f000:fff0]?

```
The target architecture is assumed to be i8086
[f000:fff0]       0xffff0: ljmp   $0xf000,$0xe05b
```

- Intel 8086 (1978, ~45 years old, runs @ **5MHz**)
  - 16-bit processor; all registers are **16-bits**
- Intel 8086 can access **1MB of memory**
  - 1MB == **1048576 Bytes** == **2^{20} Bytes**
  - Requires **20-bits to address the 1MB** memory space

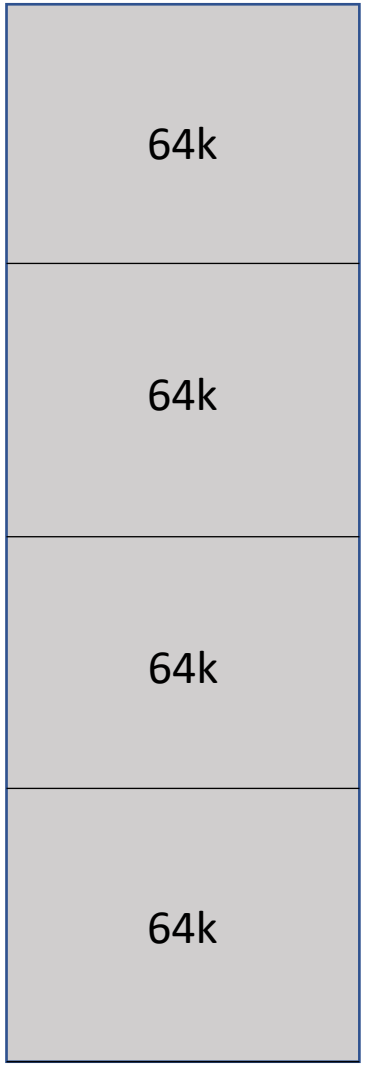
16 bits can address only 2^{16}-1 locations ➔ 64k!

Do we see a problem here?

Memory Segmentation to the rescue!

Memory Segmentation

Allows memory to be "**segmented**" into sections of 64k each



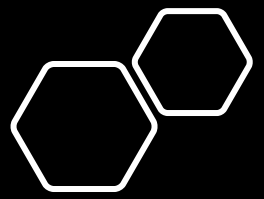
- Allows 16-bit processor to **access 20-bit address space**

Segment Register	Regular Register

- Written as: **[Segment Register Value]:[Regular Register Value]**

- e.g., **$cs:$ip, $cs = 0xf000, $ip = 0xfff0**

  then it will be **0xf000:0xfff0**



Address Calculation

[SEGMENT:OFFSET]

SEGMENT * 16 + OFFSET!

Memory Segmentation [contd.]

- f000:fff0
  - **0xf000 * 16 + 0xfff0**
  - Multiplying 16 for a hexadecimal number is just shifting one digit left
  - 0xf000**0** + 0xfff0 ➔ 0xffff0 [becomes equivalent of **5-digit address**!]

- Each digit in hexadecimal number represents 4-bits
  - 4 * 5 == 20 bits!
  - 8086 processor can access from **0x00000 ~ 0xfffff (1,048,576 bytes, 1MB)!**

Segmentation in Real Mode

- **Real mode** [https://en.wikipedia.org/wiki/Real_mode]
  - Mode that uses physical memory directly
  - **No memory protection**
  - MS-DOS (1981 ~ 2000) runs in this mode

- **Backward Compatibility**: **all x86 processors boots in Real Mode**
  - We need to switch it to a **Protected Mode** and enabling **paging**, etc.
  - We will do all these initializations in JOS labs, 1 and 2

Quick Quiz

What is the effective address of the following [seg:offset] values?

- [1000:3333]
- [b000:b7ff]
- [0001:0101]
- [f800:8001]

Quick Quiz Asnwers

What is the effective address of the following [seg:offset] values?

- [1000:3333]
  - 0x1000 * 16 + 0x3333 = 0x10000 + 0x3333 = **0x13333**
- [b000:b7ff]
  - 0xb000 * 16 + 0xb7ff = 0xb0000 + 0xb7ff = **0xbb7ff**
- [0001:0101]
  - 0x0001 * 16 + 0x0101 = 0x0010 + 0x0101 = **0x0111**
- [f800:8001]
  - 0xf800 * 16 + 0x8001 = 0xf8000 + 0x8001 = **0x100001**

Additional Reading

- Five steps of the boot sequence:

https://www.techwalla.com/articles/five-steps-computer-bootup-process

- Linux boot process and GRUB

https://www.learnitguide.net/2015/11/linux-boot-process-step-by-step.html