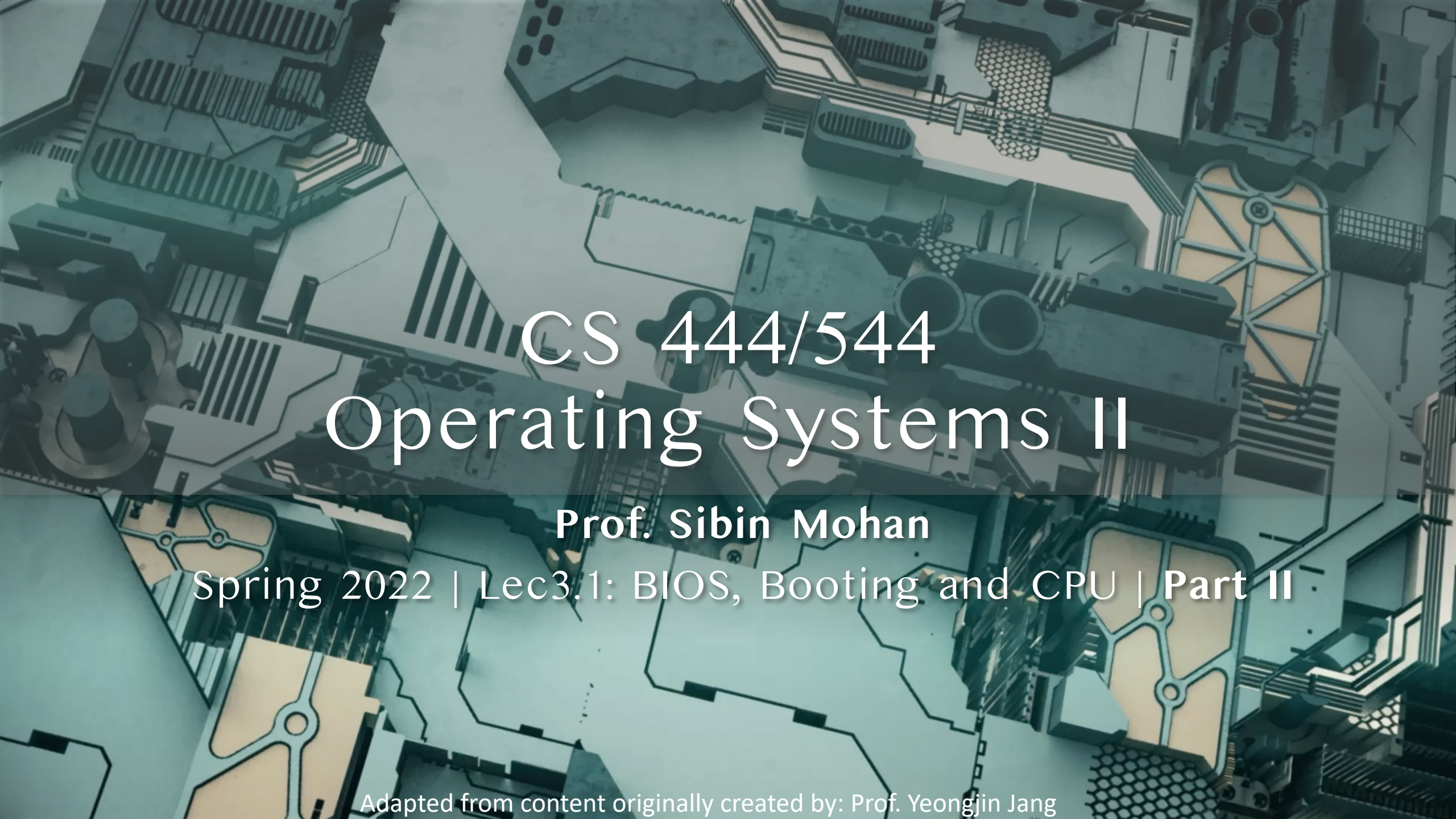


# CS 444/544
# Operating Systems II

**Prof. Sibin Mohan**

Spring 2022 | Lec3.1: BIOS, Booting and CPU | **Part II**

Adapted from content originally created by: Prof. Yeongjin Jang

# Boot from Disk

- Load boot sector from disk
  - **512 bytes**
- Boot sector, Master Boot Record [**MBR**]
  - The 1st sector of the disk partition
  - Ends with **`0x55AA`**
- Load MBR at **`0x7c00`** and run
  - **Now the OS takes the control**!

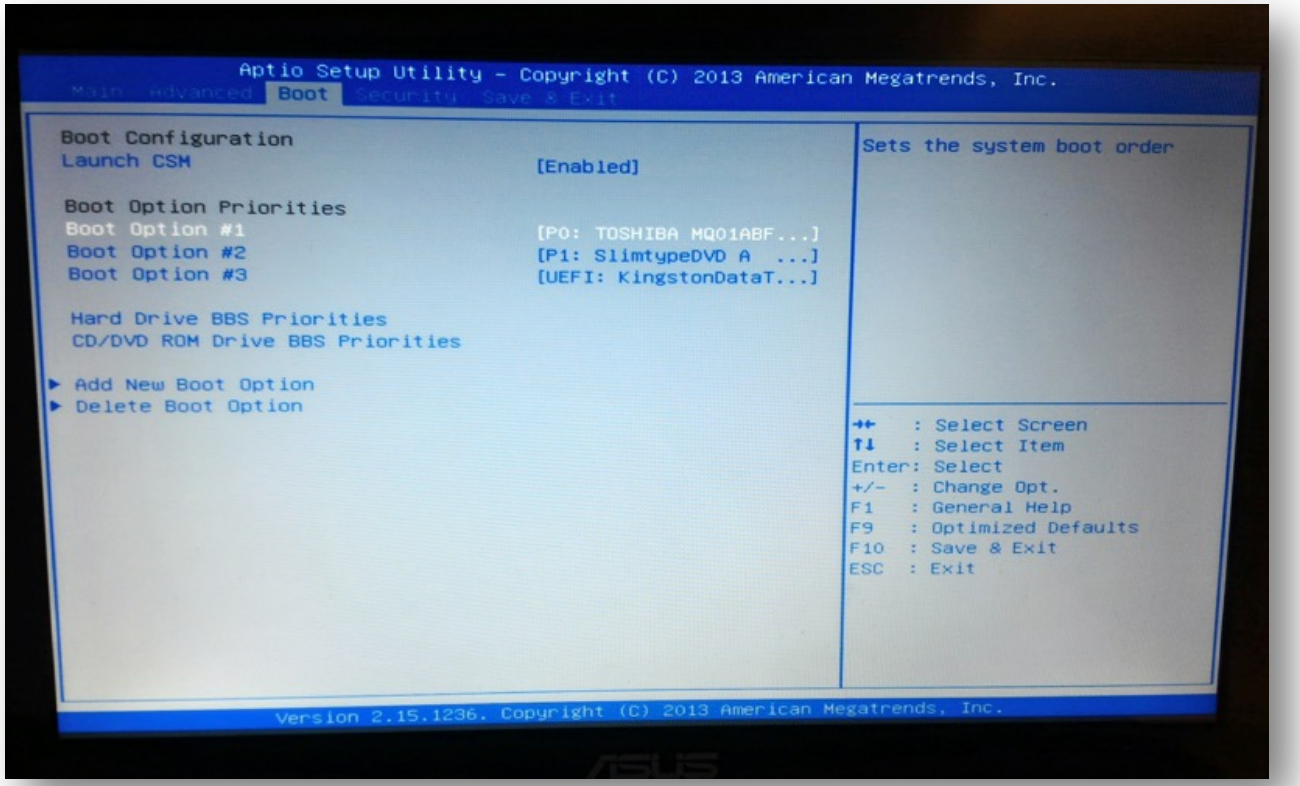



Image from:

https://support.endlessm.com/hc/en-us/articles/210527103-How-do-I-start-boot-my-computer-from-a-USB-device-or-DVD-with-Endless-OS-?mobile_site=true

# JOS Boot Sector

- Boot sector [MBR]
  - Check **obj/boot/boot**
    - After running make!
  - The 1st sector of the disk partition
    - Ends with **0x55AA**
- Why **0x55AA**?

```
irb(main):002:0> 0x55aa.to_s(2)
=> "1010101101010101"
```

- Load MBR at **0x7c00** and run
  - **Now bootloader takes control!**

```
[red9057@blue9057-vm-jos (lab1) ~/jos/obj/boot$] xxd boot
00000000: fafc 31c0 8ed8 8ec0 8ed0 e464 a802 75fa  ..1........d..u.
00000010: b0d1 e664 e464 a802 75fa b0df e660 0f01  ...d.d..u....`..
00000020: 1664 7c0f 20c0 6683 c801 0f22 c0ea 327c  .d|. .f...."..2|
00000030: 0800 66b8 1000 8ed8 8ec0 8ee0 8ee8 8ed0  ..f.............
00000040: bc00 7c00 00e8 cb00 0000 ebfe 0000 0000  ..|.............
00000050: 0000 0000 ffff 0000 009a cf00 ffff 0000  ................
00000060: 0092 cf00 1700 4c7c 0000 55ba f701 0000  ......L|..U.....
00000070: 89e5 ec83 e0c0 3c40 75f8 5dc3 5589 e557  ......<@u.].U..W
00000080: 8b4d 0ce8 e2ff ffff b001 baf2 0100 00ee  .M..............
00000090: baf3 0100 0088 c8ee 89c8 baf4 0100 00c1  ................
000000a0: e808 ee89 c8ba f501 0000 c1e8 10ee 89c8  ................
000000b0: baf6 0100 00c1 e818 83c8 e0ee b020 baf7  ............. ..
000000c0: 0100 00ee e8a1 ffff ff8b 7d08 b980 0000  ..........}.....
000000d0: 00ba f001 0000 fcf2 6d5f 5dc3 5589 e557  ........m_].U..W
000000e0: 568b 7d10 538b 750c 8b5d 08c1 ef09 01de  V.}.S.u..]......
000000f0: 4781 e300 feff ff39 f373 1257 5347 81c3  G......9.s.WSG..
00000100: 0002 0000 e873 ffff ff58 5aeb ea8d 65f4  .....s...XZ...e.
00000110: 5b5e 5f5d c355 89e5 5653 6a00 6800 1000  [^_].U..VSj.h...
00000120: 0068 0000 0100 e8b1 ffff ff83 c40c 813d  .h.............=
00000130: 0000 0100 7f45 4c46 7537 a11c 0001 000f  .....ELFu7......
00000140: b735 2c00 0100 8d98 0000 0100 c1e6 0501  .5,.............
00000150: de39 f373 16ff 7304 ff73 1483 c320 ff73  .9.s..s..s... .s
00000160: ece8 76ff ffff 83c4 0ceb e6ff 1518 0001  ..v.............
00000170: 00ba 008a 0000 b800 8aff ff66 efb8 008e  ...........f....
00000180: ffff 66ef ebfe 0000 0000 0000 0000 0000  ..f.............
00000190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
```

# Lab 1

- QEMU uses SeaBIOS
  - Open-Source Software, so we can look at the source code!

```
static void
boot_disk(u8 bootdrv, int checksig)
{
    u16 bootseg = 0x07c0;

    // Read sector
    struct bregs br;
```

```
/* Canonicalize bootseg:bootip */
u16 bootip = (bootseg & 0x0fff) << 4;
bootseg &= 0xf000;

call_boot_entry(SEGOFF(bootseg, bootip), bootdrv);
```

- **bootseg = 0x7c0**
- **bootip = (bootseg & 0x0fff) << 4  == 0x7c00**
- **bootseg &= 0xf000  == 0**

**bootseg:bootip == 0000:7c00 == 0x7c00** → **runs 0x7c00!!**

# What does boot sector need to do?

Only **512 bytes**

Too small to load OS
Kernel Size = 6MB

**Real Mode**

Can only use **1MB** mem
[Can't load 6 MB even!]

Bootloader's TODO:

Enable protected mode
[4GB access]
Load other parts of OS

We must do this in the first **510 bytes**

512-2, because last 2 bytes are 0x55aa

```
[coe_jangye@os2 (lab1) ~/jos$] ls -l /boot/vmlinuz-3.10.0-1062.12.1.el7.x86_64
-rwxr-xr-x. 1 root root 6734016 Feb  4 15:07 /boot/vmlinuz-3.10.0-1062.12.1.el7.x86
_64
```

# Breakpoint at 0x7c00

```
+ symbol-file obj/kern/kernel
>>> b *0x7c00
Breakpoint 1 at 0x7c00
>>> c
```

```
── Output/messages ──
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
── Registers ──
   eax 0x0000aa55       ecx 0x00000000
   esp 0x00006f20       ebp 0x00000000
   eip 0x00007c00    eflags [ IF ]
    ds 0x00000000        es 0x00000000
── Assembly ──
0x00007c00 ? cli
0x00007c01 ? cld
0x00007c02 ? xor    %ax,%ax
0x00007c04 ? mov    %ax,%ds
0x00007c06 ? mov    %ax,%es
0x00007c08 ? mov    %ax,%ss
0x00007c0a ? in     $0x64,%al
── Source ──
── Stack ──
[0] from 0x00007c00
(no arguments)
── Memory ──
── Expressions ──

>>>
```

**boot/boot.S**

```
12 .globl start
13 start:
14   .code16                     # Assemble for 16-bit mode
15   cli                         # Disable interrupts
16   cld                         # String operations increment
17
18   # Set up the important data segment registers (DS, ES, SS).
19   xorw    %ax,%ax             # Segment number zero
20   movw    %ax,%ds             # -> Data Segment
21   movw    %ax,%es             # -> Extra Segment
22   movw    %ax,%ss             # -> Stack Segment
23
24   # Enable A20:
25   #   For backwards compatibility with the earliest PCs, physical
26   #   address line 20 is tied low, so that addresses higher than
27   #   1MB wrap around to zero by default.  This code undoes this.
28 seta20.1:
29   inb     $0x64,%al               # Wait for not busy
30   testb   $0x2,%al
31   jnz     seta20.1
32
33   movb    $0xd1,%al               # 0xd1 -> port 0x64
```

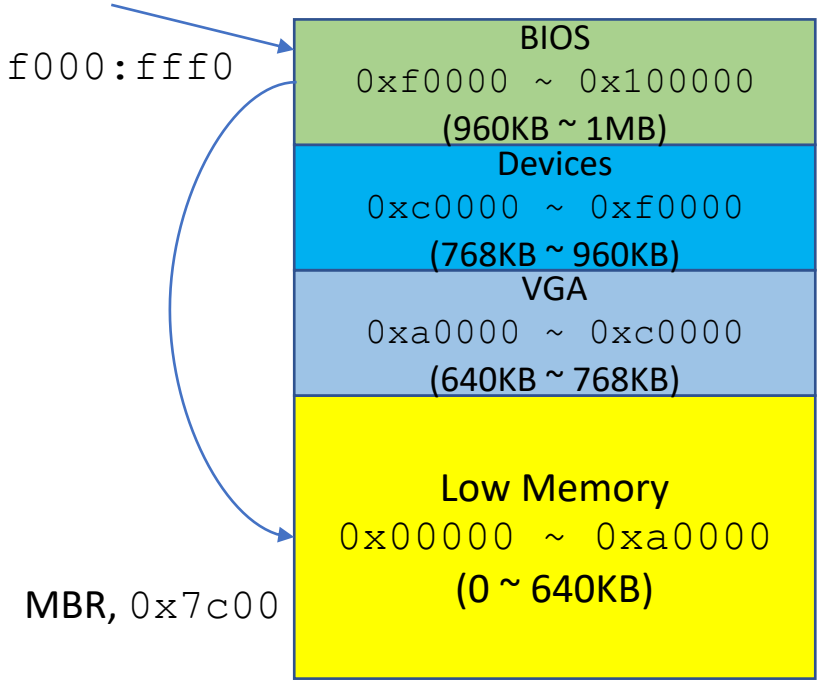
what is “**A20**”?

# Weird Segmentation: A20

- [f800:0001]
  - 0xf800 * 16 + 0x0001 = 0xf8001
- [f800:8001]
  - 0xf800 * 16 + 0x8001 = 0x100001
  - **More than 1MB range → overflow** in 8086!
- Why **20**?
  - A hexadecimal digit can represent **4 bits**
  - 0x100000 (1MB)
  - **0001** 0000 0000 0000 0000 0000
  - **20th** bit (indexing starting from 0)

# Weird Segmentation: A20

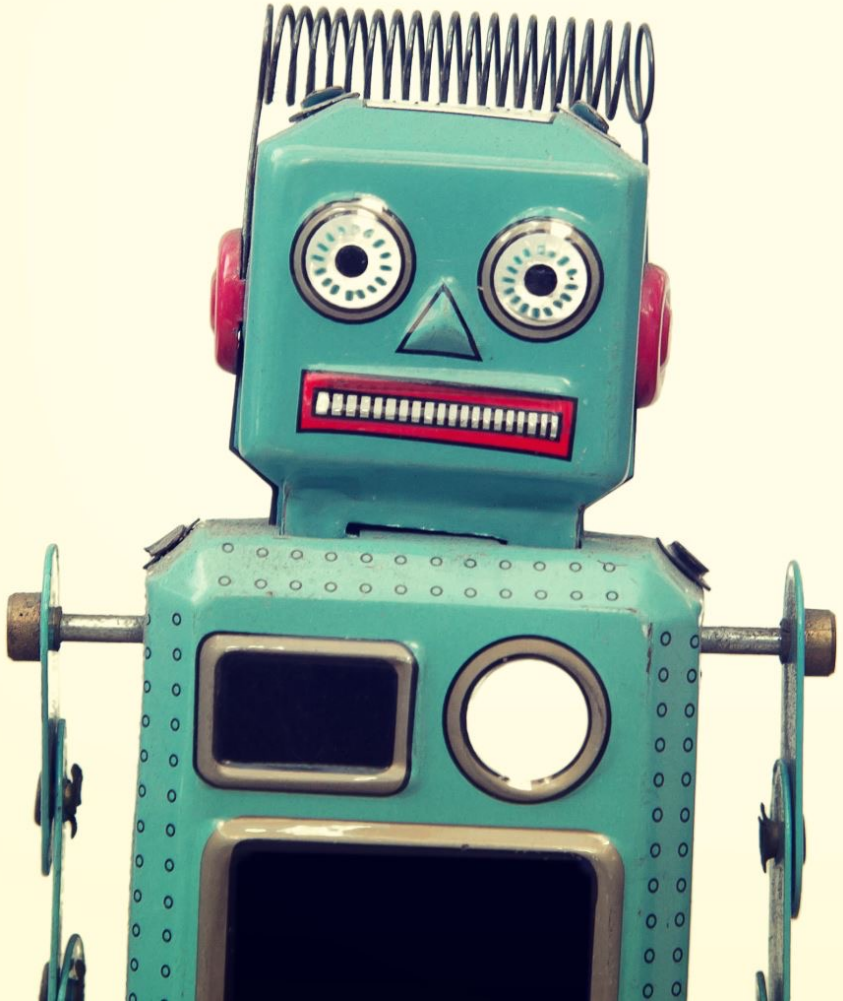
- A20: address line at bit 20
  - the **top bit right after 1MB range**
  - Software developers set **A20 as low** (**always zero**)
    - to make overflow condition be benign
  - [f800:8001] = 0x100001 == 0x000001 in A20 low




- Why?
  - Can access the **both ends** of the memory
  - 0xffff0 [BIOS], f000:0xfff0
  - 0x7c00 [Bootloader], 0000:7c00 Need to change the segment from **0xf000** to **0x0000**
  - 0xf800:7ff0 == 0xf8000 + 0x7ff0 = 0xffff0
  - 0xf800:fc00 == 0xf8000 + 0xfc00 = **0x107c00 == 0x7c00**

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

**DO NOT have to change Segment Register!**



# Weird Segmentation: A20

- In modern machines:
  - **Cannot use memory 1MB memory for every other 1MB**
    - `0x300000` and `0x200000` points to **same address!**
    - `0x400000` and `0x500000` will do so as well!
- **~50%** memory utilization? **Silly!**
- Need to turn it on

# JOS Bootloader (**boot.S**)

- Enable A20
- Enable **protected** mode
  - enabling 4GB memory access)
- **Read kernel ELF** [Executable Linkable Format]
- Do all these in **510** bytes
  - actually uses less than this!

will discuss soon

# JOS Bootloader (boot.S)

- Enable **protected** mode [enabling 4GB memory access]
  - Set Global Descriptor Table
  - Code segment from `0 ~ 0xffffffff` [full 4GB access
  - Data segment from `0 ~ 0xffffffff` [full 4GB access]

```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL                    # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)           # data seg
```

```
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

CR0? See this : https://en.wikipedia.org/wiki/Control_register

Control Register (CR)

```
10 .set CR0_PE_ON,      0x1
```

# JOS Bootloader (boot/main.c)

After enabling protected mode,

1. boot.S will run '**`ljmpl`**' [long jump, far jump]
   - apply new segment assigned by global descriptor table
2. Then, it will call **`bootmain()`** in **boot.c**
3. Read **kernel ELF** [Executable Linkable Format]
   - https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
   - Load binary program into memory
   - Read header, map memory, copy data
4. Then, **run Kernel**!

# In Lab Tutorial #2

01

Follow boot sequence with ‘gdb’ in assembly and C code

- Up to **Exercise 7**

02

Learn how Intel x86 uses STACK to store a function’s local context

- **Exercise 10**

# Additional Reading

- SeaBIOS