CS 444/544
Operating Systems II

Lecture 2

BIOS, Booting, and CPU

4/3/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang

Odds and Ends

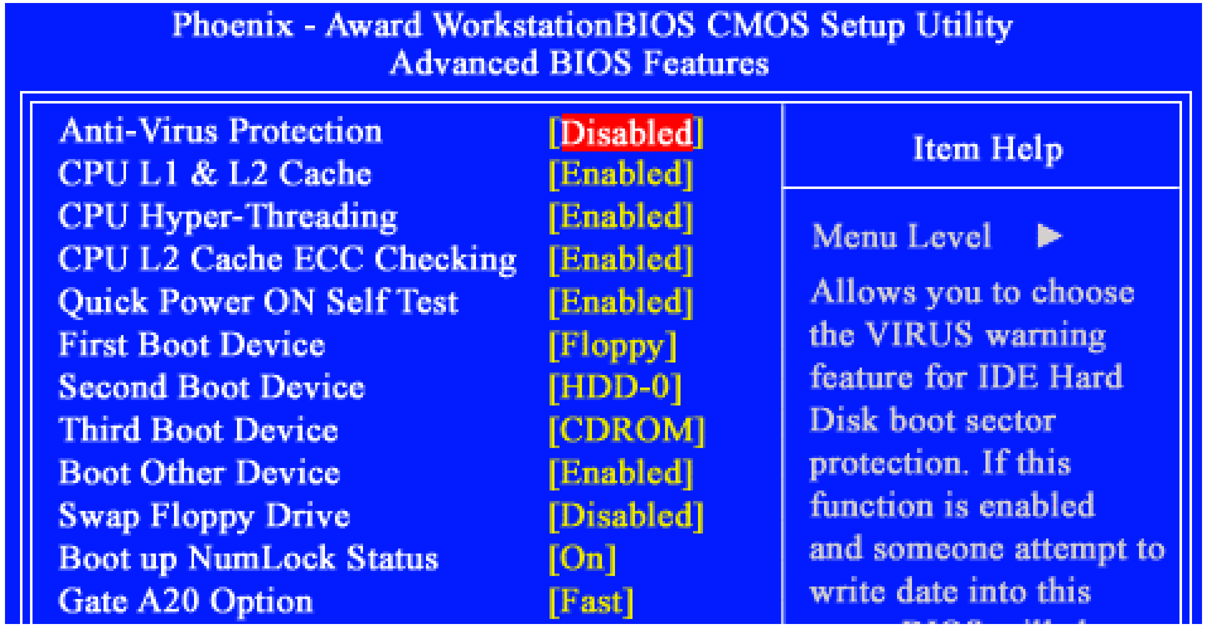
- Office hours starts next week (default: via discord)
  - Check Canvas → Office Hours

Topics for Today

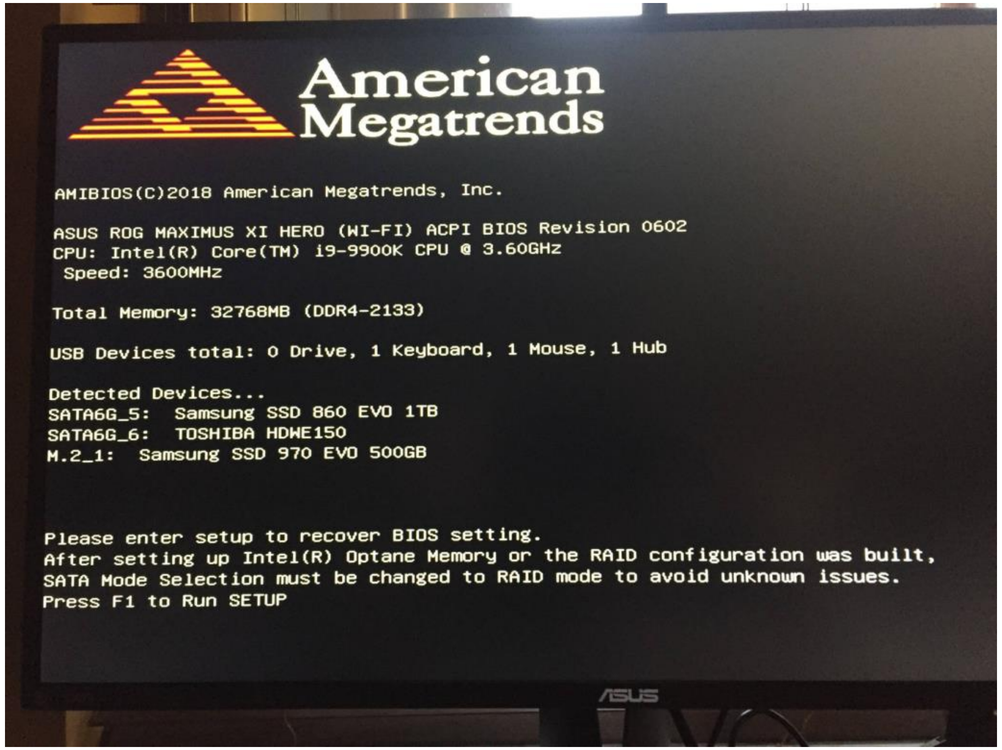
- Booting
  - BIOS
  - Bootloader
  - Kernel
- Others
  - History of x86 CPUs
  - Real mode
  - Protected mode
  - Memory Segmentation in x86
  - A20

What does your computer do if you press the power button?

- BIOS
  - Basic Input Output System
  - Enables basic device access



```
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
Quick Power ON Self Test       [Enabled]      the VIRUS warning
First Boot Device              [Floppy]       feature for IDE Hard
Second Boot Device             [HDD-0]        Disk boot sector
Third Boot Device              [CDROM]        protection. If this
Boot Other Device              [Enabled]      function is enabled
Swap Floppy Drive              [Disabled]     and someone attempt to
Boot up NumLock Status         [On]           write date into this
Gate A20 Option                [Fast]
```

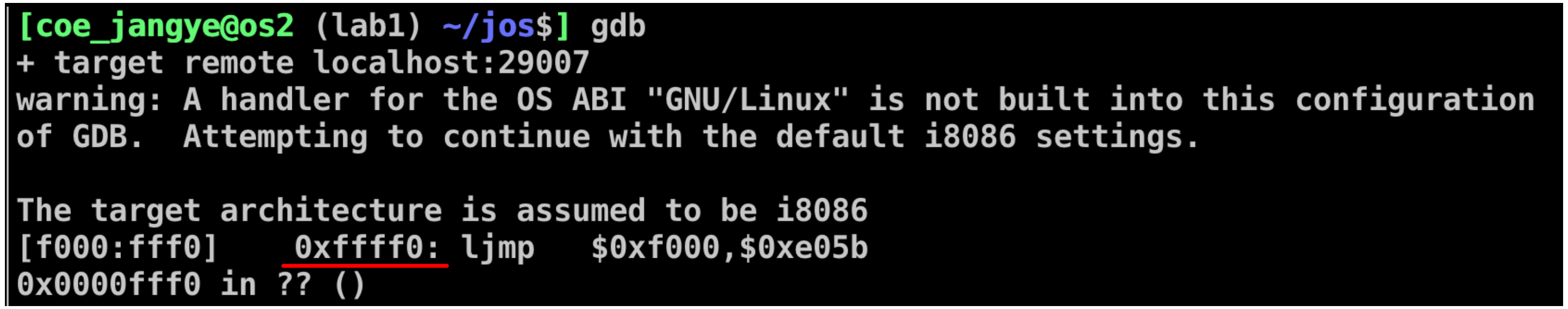


```
AMIBIOS(C)2018 American Megatrends, Inc.

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
```

Image from: https://www.biostar.com.tw/app/en/mb/introduction.php?S_ID=952

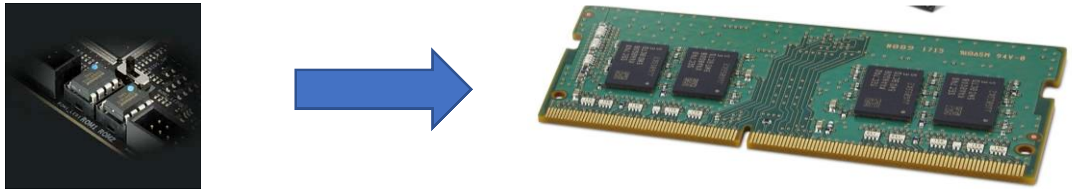
Boot Sequence

- Power up
- BIOS initialize basic devices



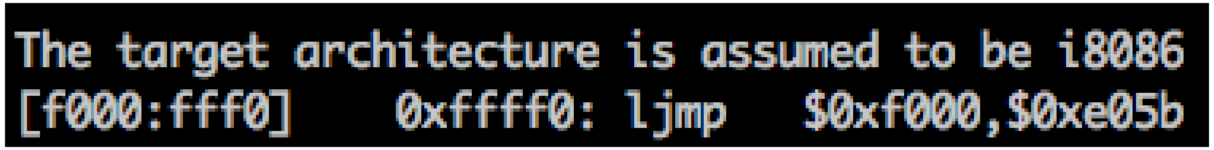
```
[coe_jangye@os2 (lab1) ~/jos$] gdb
+ target remote localhost:29007
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

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

- After initializing peripheral devices, it will put some initialization code to
  - DRAM physical address 0xffff0 ([f000:fff0])
  - Copy the code from ROM to RAM
  - Run (RAM)!



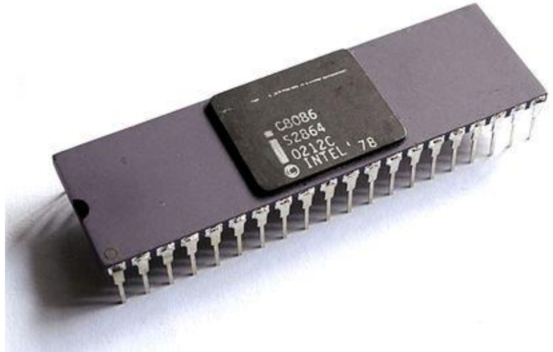
- What does the code do?: BIOS load and run the boot sector from disk
  - Read the 1st sector from the boot disk (512 bytes)
  - Put the sector at 0x7c00
  - Run it! (set the instruction pointer = 0x7c00)

What is i8086?



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

- Intel 8086 (1978, ~46 years old, runs @ 5MHz)
  - 16-bit processor; all registers are 16-bits.

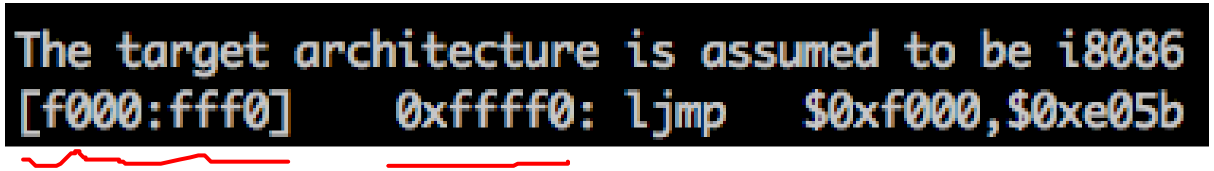


- BIOS assumes our processor is i8086
  - We are living in 2024 and Intel Xeon on the os2 server

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

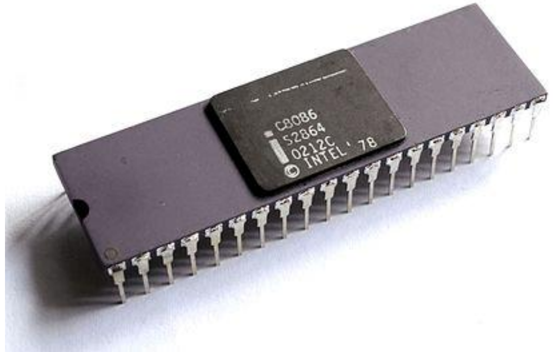
- Why?
  - Backward Compatibility
  - Use the same code for all CPUs!

What is [f000:fff0]?



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

- Intel 8086 (1978, ~46 years old)
  - 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



- f000:fff0
  - It points to 0xffff0, which is 1MB - 16

```
>>> 0xffff0
1048560
```

Photo from https://en.wikipedia.org/wiki/Intel_8086

Memory Segmentation

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

- How?
  - Use two registers
  - [Segment register]:[regular register]
  - e.g., $cs:$ip, $cs = 0xf000, $ip = 0xfff0, then it will be 0xf000:0xfff0
- Address calculation
  - A:B
  - A * 16 + B
  - Add one 0 at the end of A and then add B
    - In decimal numbers, multiplying 10 is adding one zero at the end
    - Likewise, in hexadecimal numbers, multiplying 16 is adding one zero at the end

Memory Segmentation

- Address Calculation
  - A:B
  - A * 16 + B
- f000:fff0
  - 0xf000 * 16 + 0xfff0
  - Multiplying 16 for a hexadecimal number is just shifting one digit left…
  - 0xf0000 + 0xfff0
  - 0xffff0 (becomes 5-digit address!)
- Each digits in hexadecimal number represents 4-bits
  - 4 * 5 == 20 bits!
  - A 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 processor boots in Real Mode
  - We need to switch it to a Protected Mode and enabling Paging, etc…
  - We will do all those initialization in JOS lab1 and lab2.

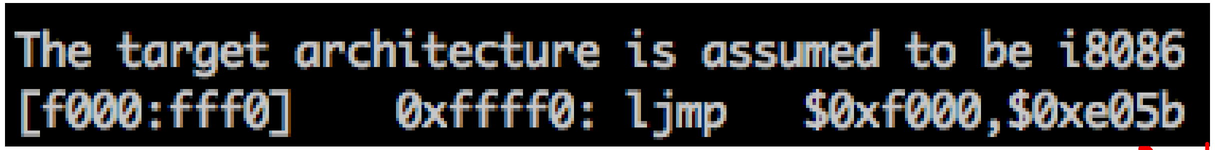
- Uses segmentation to access 1MB memory
  - [seg:offset] = seg * 16 + offset
  - e.g., [f000:fff0] == 0xf000 * 16 + 0xfff0 == 0xf0000 + 0xfff0 == 0xffff0

Quick Quiz

- What is the address of the following [seg:offset]?
- [1000:3333]
  - 0x1000 * 16 + 0x3333 = 0x10000 + 0x3333 = 0x13333
- [b000:b7ff]
  - 0xb000 * 16 + 0xb7ff = 0xb0000 + 0xb7ff = 0xbb7ff
- [0001:0101]
  - 0x0001 * 16 + 0x0101 = 0x00010 + 0x0101 = $\overset{0}{0}$x00111
- [f800:8001] **0xf8 + 8 = 0x100**
  - 0xf800 * 16 + 0x8001 = 0xf8000 + 0x8001 = 0x100001 **OVER 1MB!!!**

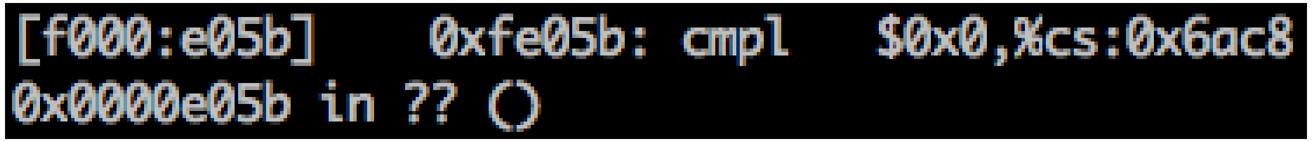
Real Mode Segmentation

- **SEGMENT:OFFSET**
  - **SEGMENT * 16 + OFFSET!**
- **Where does this code jump to?**



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

cs : offset

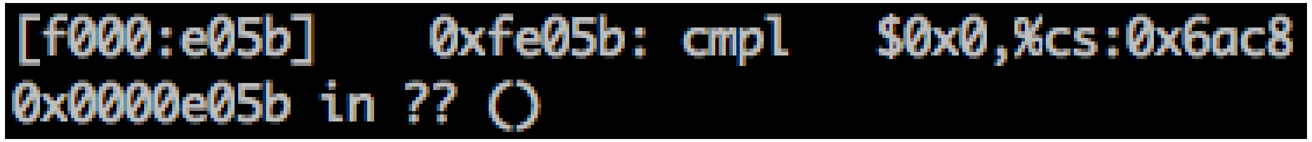


```
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
0x0000e05b in ?? ()
```

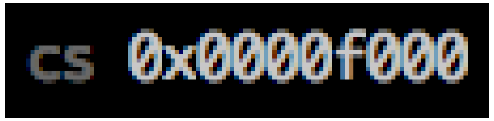
- **0xf000:0xe05b**
  - **0xf0000 + 0xe05b == 0xfe05b**

Real Mode Segmentation

- **Compare to what??**

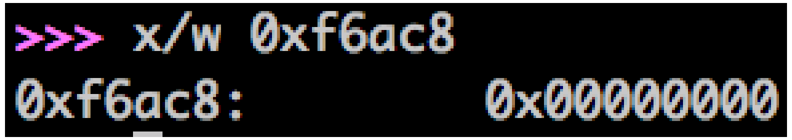


```
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
0x0000e05b in ?? ()
```



```
cs 0x0000f000
```

- **cs:0x6ac8**
  - **f000:6ac8 == 0xf6ac8**



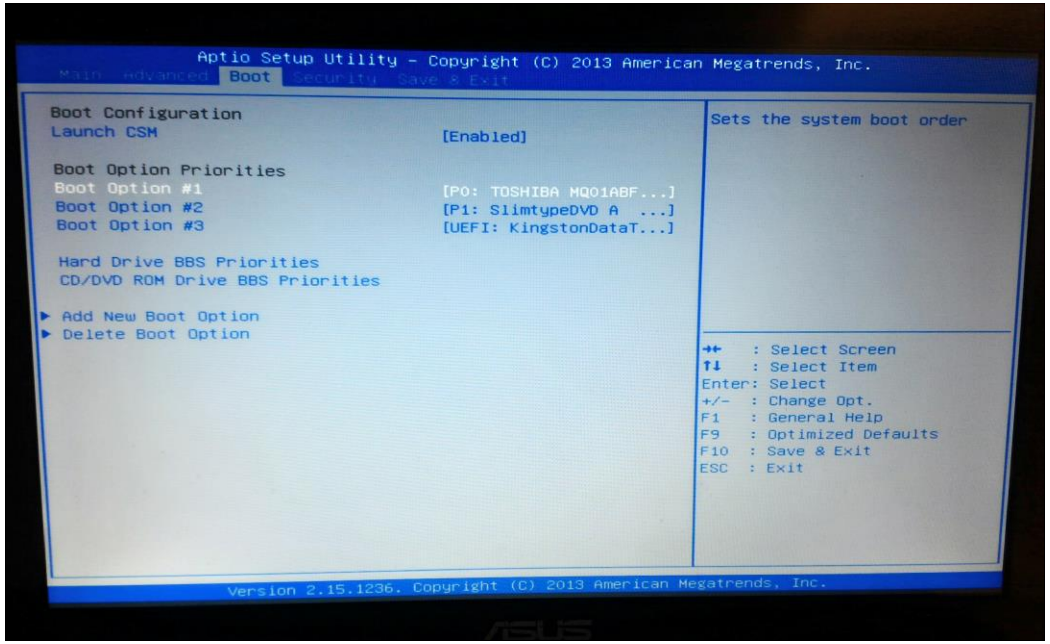
```
>>> x/w 0xf6ac8
0xf6ac8:        0x00000000
```

Boot from Disk

- Load the boot sector (512 bytes) from the boot disk

- Boot sector (Master Boot Record)
  - The 1st sector of the disk partition
  - Ends with 0x55AA

- Load that at 0x7c00, and run
  - Now the OS takes the control!

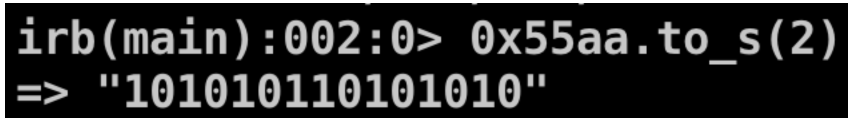


```
           Aptio Setup Utility - Copyright (C) 2013 American Megatrends, Inc.

Boot Configuration                                   Sets the system boot order
Launch CSM                  [Enabled]

Boot Option Priorities
Boot Option #1              [P0: TOSHIBA MQ01ABF...]
Boot Option #2              [P1: SlimtypeDVD A  ...]
Boot Option #3              [UEFI: KingstonDataT...]

Hard Drive BBS Priorities
CD/DVD ROM Drive BBS Priorities

▶ Add New Boot Option
▶ Delete Boot Option
                                                     →←    : Select Screen
                                                     ↑↓    : Select Item
                                                     Enter: Select
                                                     +/-   : Change Opt.
                                                     F1    : General Help
                                                     F9    : Optimized Defaults
                                                     F10   : Save & Exit
                                                     ESC   : Exit

           Version 2.15.1236. Copyright (C) 2013 American Megatrends, Inc.
```

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

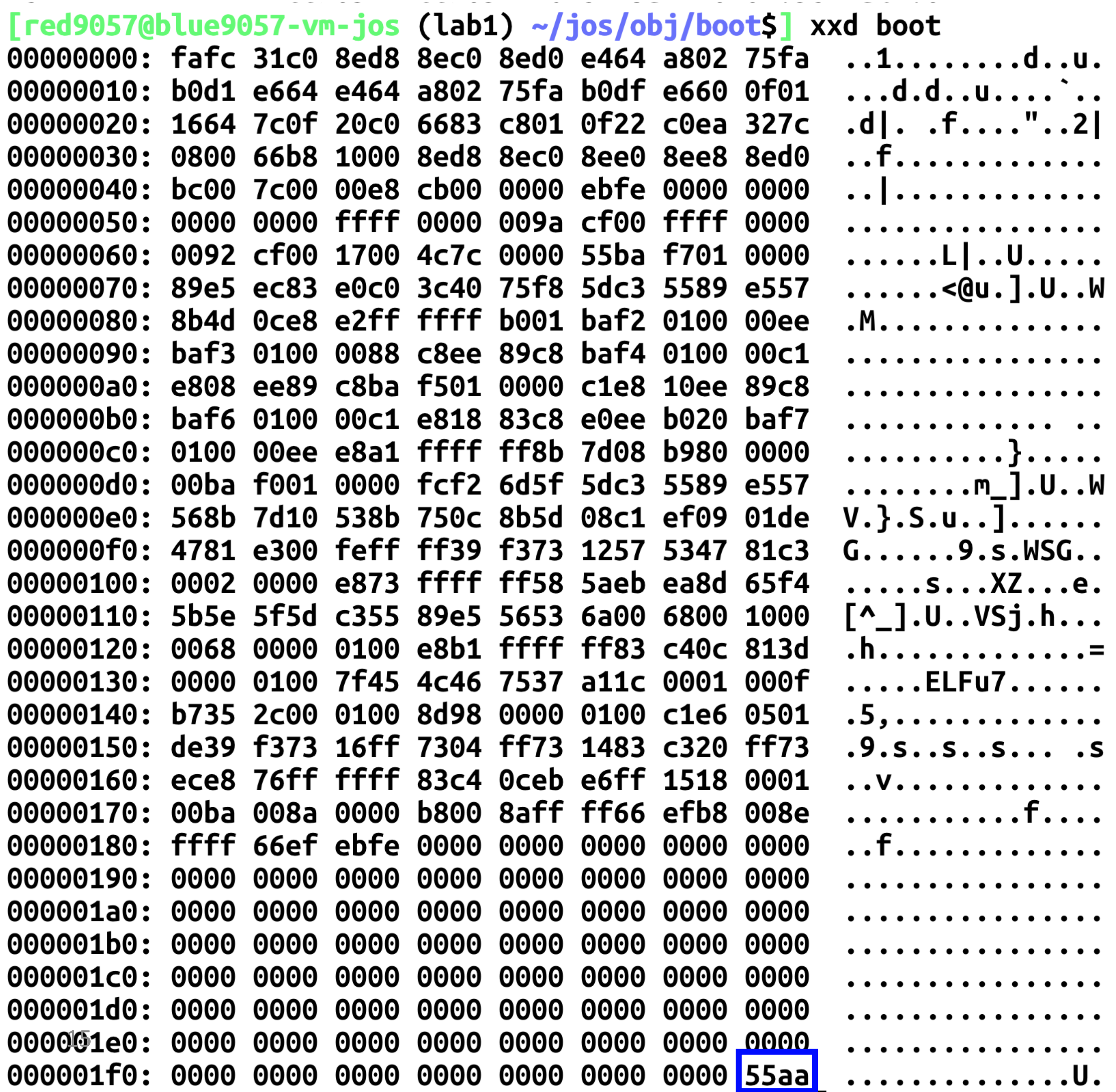
JOS Boot Sector

- Boot sector (Master Boot Record)
  - 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)
=> "101010110101010"
```

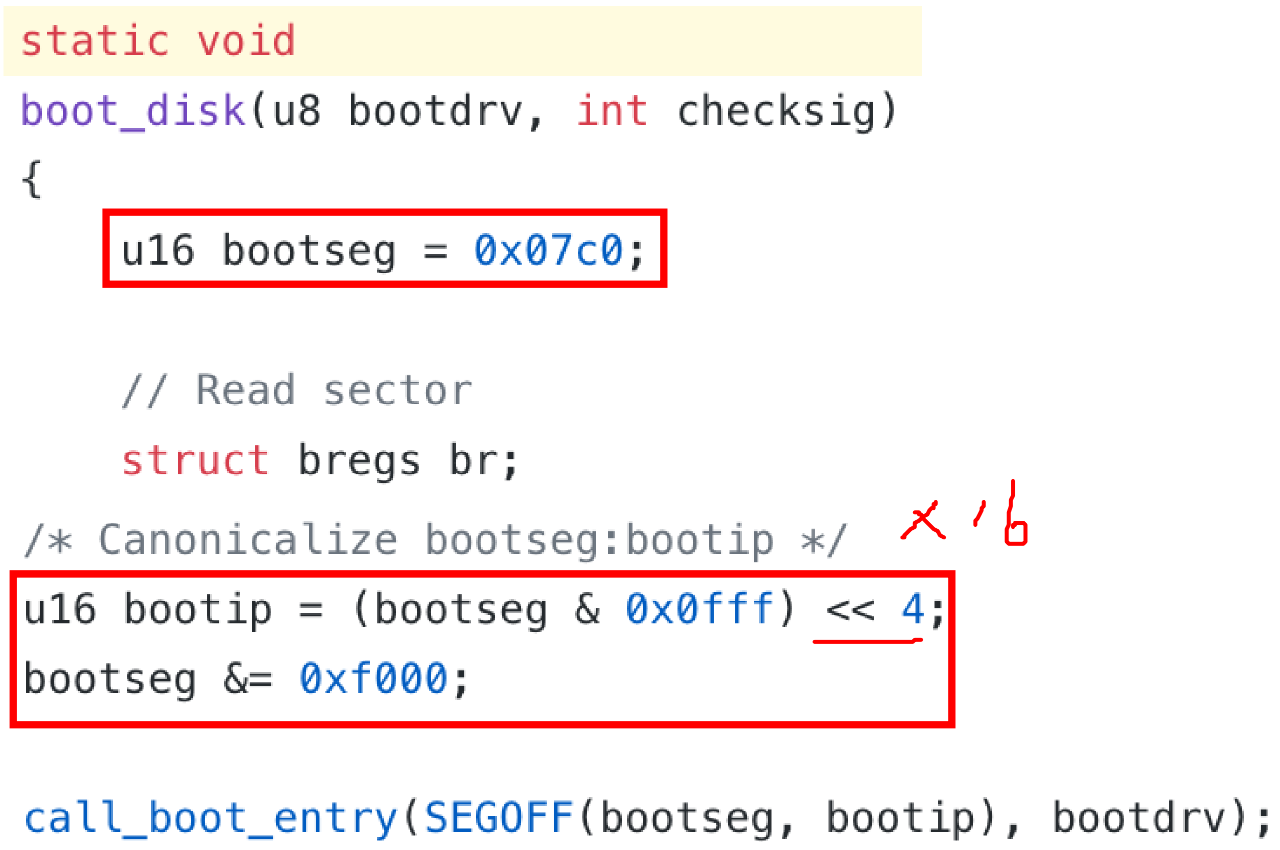
- Load that at 0x7c00, and run
  - Now the bootloader takes the control!



```
[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.
```

In Lab1

- QEMU uses SeaBIOS
  - It's an Open Source Software, so we can take a look into 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);
```

x16

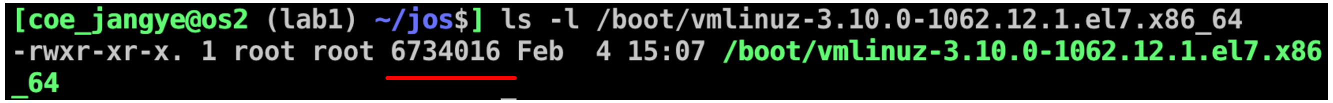
- bootseg = 0x7c0
- bootip = (bootseg & 0x0fff) << 4 == **0x7c00**

0x7c0

- bootseg &= 0xf000 == 0

Bootseg:bootip == 0000:7c00 == 0x7c00, Runs 0x7c00!!

What does the boot sector need to do?

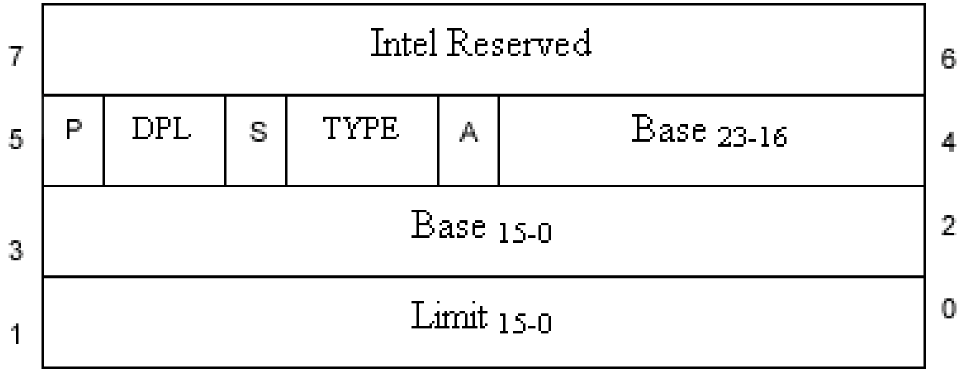


```
[coe_jangye@os2 (lab1) ~/jos$] ls -l /boot/vmlinuz-3.10.0-1062.12.1.el7.x86_64
-rwxr-xr-x. 1 root root 6734016 Feb  4 15:07 /boot/vmlinuz-3.10.0-1062.12.1.el7.x86
_64
```

- Only 512 bytes
  - Too small for loading operating system
  - Our kernel on the OS2 server is around 6MB when it is compressed (vmlinuz)
- Real Mode
  - Can only use 1MB memory (Uh-oh? We cannot load even that 6MB!)
- Bootloader's TODO:
  - Enable protected mode (full 4GB memory access)
  - Load the other parts of OS
- We must do this in the first 510 bytes
  - 512-2, because the last 2 bytes are 0x55aa

More about Intel x86 memory

- 8086 (1978, 16-bit), 8088 (1979, 8-bit), and 80186 (1982, 16-bit)
  - Uses 20-bit addressing via ***Real Mode*** segmentation
- 80286 (1982), a 16-bit computer
  - Uses 24-bit (16MB) addressing via ***Protected Mode***
  - A different way of using segment registers (286 is also 16-bit computer)
  - Segment register points to Global Descriptor Table, which sets base (24-bit) and limit (16-bit)

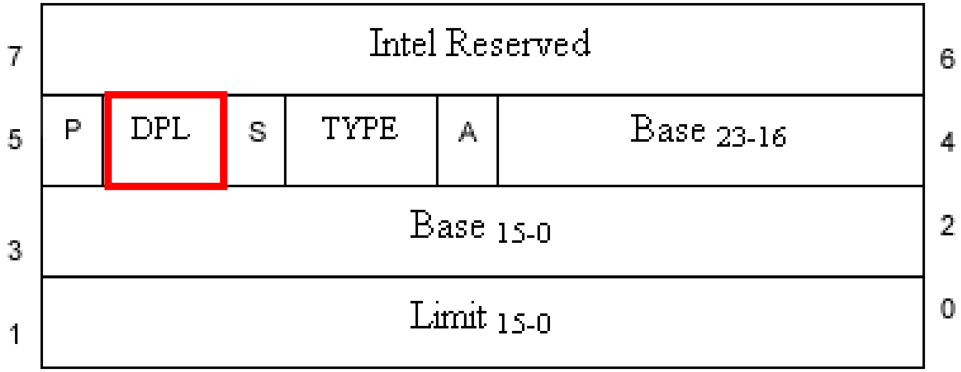


7	Intel Reserved						6
5	P	DPL	S	TYPE	A	Base $_{23-16}$	4
3	Base $_{15-0}$						2
1	Limit $_{15-0}$						0

The picture is from https://nptel.ac.in/courses/117104072/32

Why ‘Protected’?

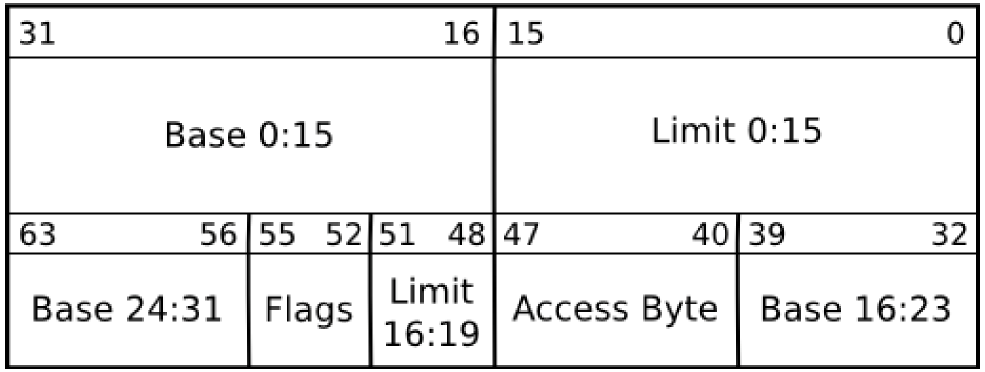
- DPL (Data Privilege Level)
  - We can set memory privilege!!!!



The picture is from https://nptel.ac.in/courses/117104072/32

i386 Protected Mode

- 80386 (1985, 32-bit)
  - 32-bit processor, all registers are 32 bits, 2^32 = 4,294,967,295 = 4GB Space!
  - Still major computers were equipped only with 4~16MB RAM…
  - Segment register now points 32bit base addressable by 32bit offset
- Supports paging (Lab2)
  - The virtual memory that we use now…



31 16	15 0
Base 0:15	Limit 0:15

63 56	55 52	51 48	47 40	39 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

The image is from https://wiki.osdev.org/Global_Descriptor_Table

i386 Protected Mode (cont'd)

- 80486, Pentium (P5), Pentium II (i686, P6), Pentium !!!
  - Uses the same protected mode with 80386

- Pentium 4 (Prescott, 2004)
  - Supports 64-bit (amd64)
  - Address space: 48-bit (256TB)
- Coffee Lake (2017)
  - Address space: 57-bit (128PB)
- Alder Lake (2021)
- Raptor Lake (2022)
- Meteor Lake (2023 Dec) …

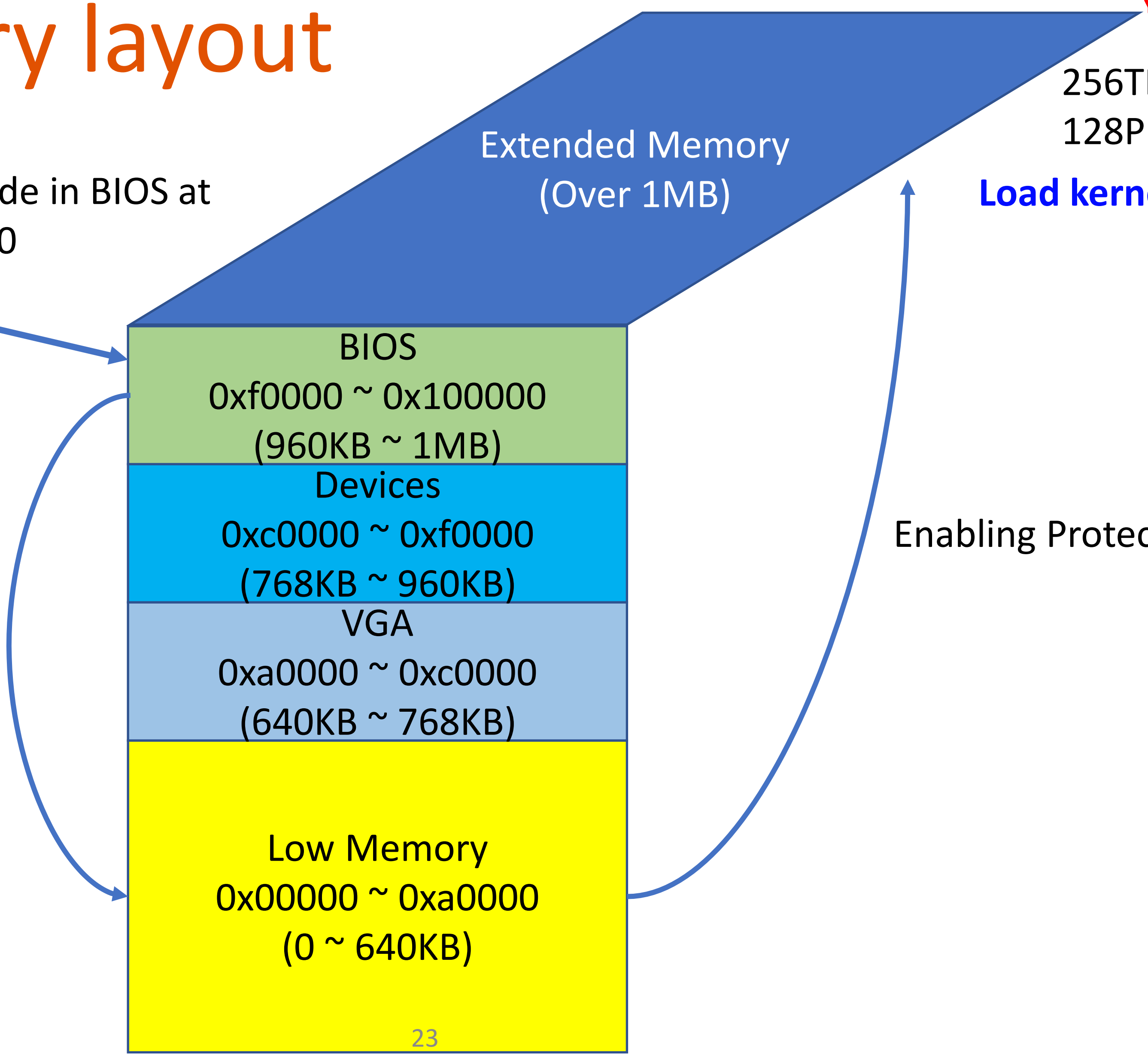
Intel CPU Codenames from Oregon

- Pentium 2
    - Deschutes
    - Klamath
- Pentium 3
    - Tualatin
- Pentium 4
    - Willamette
    - Cedar Mill (near Beaverton, OR)
- Core i7
    - Nehalem (Nehalem River)
- Core i9 / Xeon
    - Cascade Lake

Boot memory layout

```
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
```

Map code in BIOS at
f000:fff0



Extended Memory
(Over 1MB)

BIOS
0xf0000 ~ 0x100000
(960KB ~ 1MB)

Devices
0xc0000 ~ 0xf0000
(768KB ~ 960KB)

VGA
0xa0000 ~ 0xc0000
(640KB ~ 768KB)

Low Memory
0x00000 ~ 0xa0000
(0 ~ 640KB)

Read Master Boot Record (MBR)
from the boot disk
and load it at 0x7c00

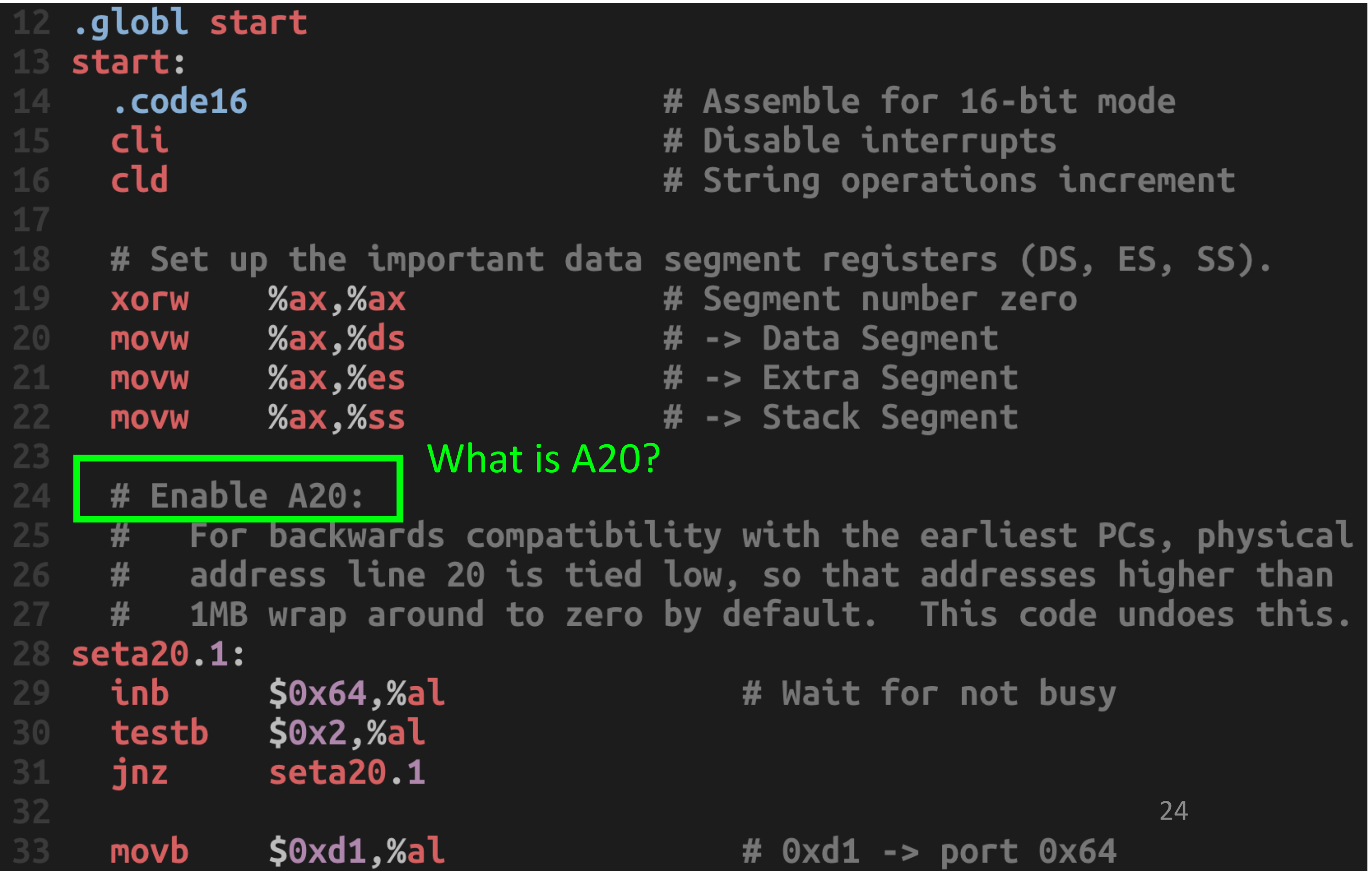
Enabling Protected Mode

4GB for 32bit
256TB for 48bit on amd64
128PB for 57bit on amd64

Load kernel and run!

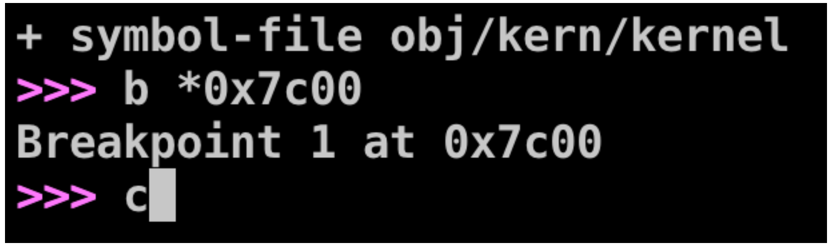
Breakpoint at 0x7c00

boot/boot.S

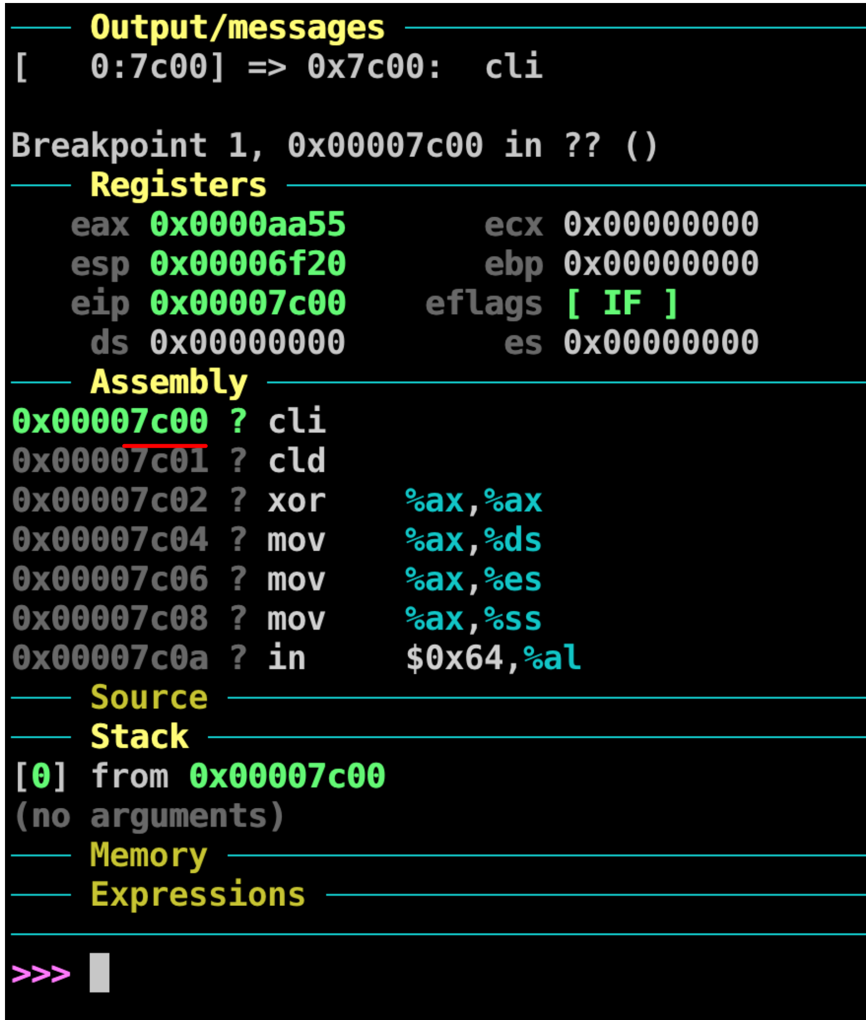


```
.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
```

What is A20?



```
+ 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 ───────────────────────────────
───────────────────────────────────────────────
>>> 
```

Weird Segmentation: A20

- [f800:0001]
    - 0xf800 * 16 + 0x0001 = 0xf8001

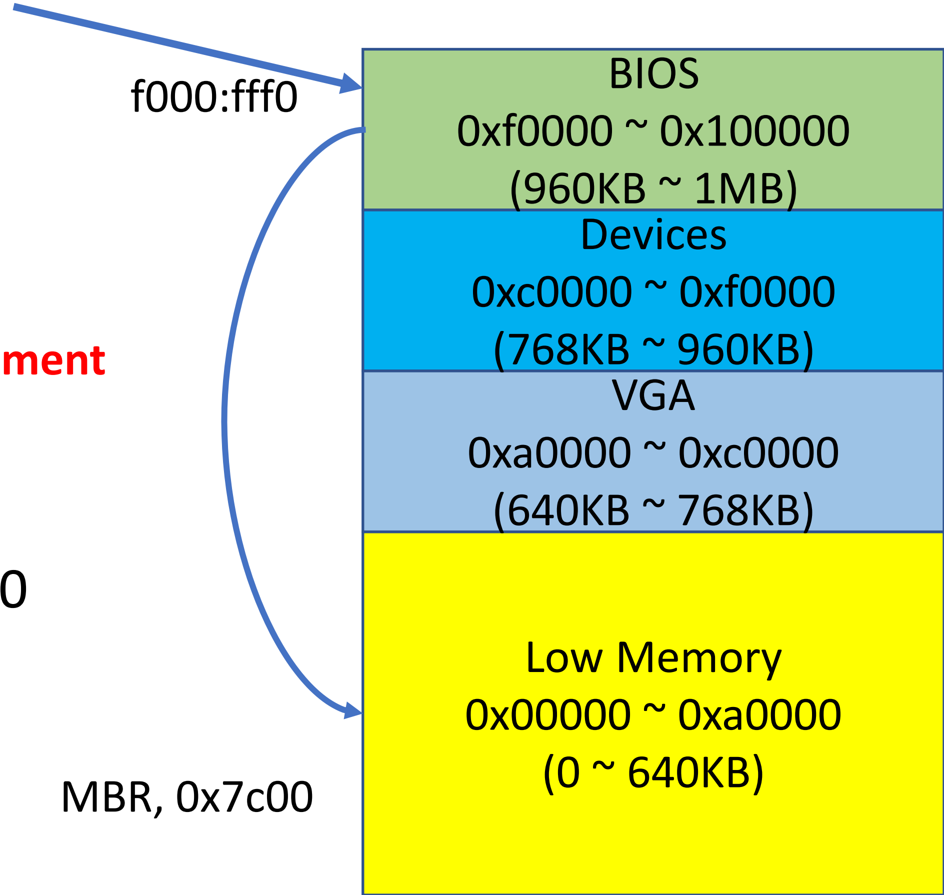
- [f800:8001]
    - 0xf800 * 16 + 0x8001 = 0x100001
    - More than 1MB range, an overflow in 8086!

- 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, which is 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 quickly access both end of the memory
  - 0xffff0 (BIOS), f000:0xfff0
  - 0x7c00 (Bootloader), 0000:7c00
  - 0xf800:7ff0 == 0xf8000 + 0x7ff0 = 0xffff0
  - 0xf800:fc00 == 0xf8000 + 0xfc00 = 0x107c00 == 0x7c00
  - **DO NOT have to change Segmentation!**

Need to change the segment
From 0xf000 to 0x0000



f000:fff0

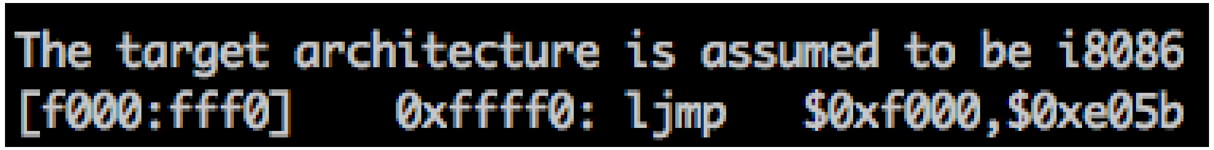
BIOS
0xf0000 ~ 0x100000
(960KB ~ 1MB)

Devices
0xc0000 ~ 0xf0000
(768KB ~ 960KB)

VGA
0xa0000 ~ 0xc0000
(640KB ~ 768KB)

Low Memory
0x00000 ~ 0xa0000
(0 ~ 640KB)

MBR, 0x7c00



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

Weird Segmentation: A20

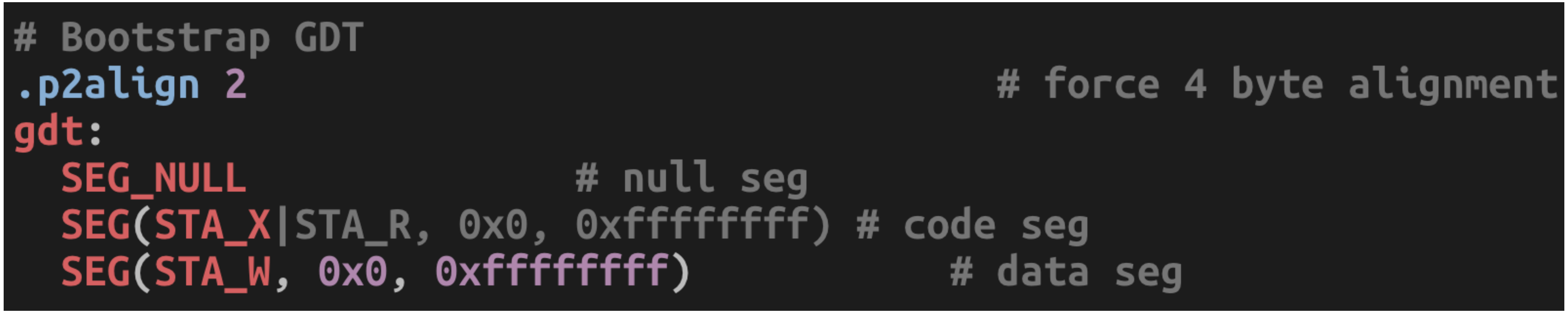
- In modern machines:
  - Cannot use memory 1MB ~ 2MB
  - Need to turn it on…

JOS Bootloader (boot.S)

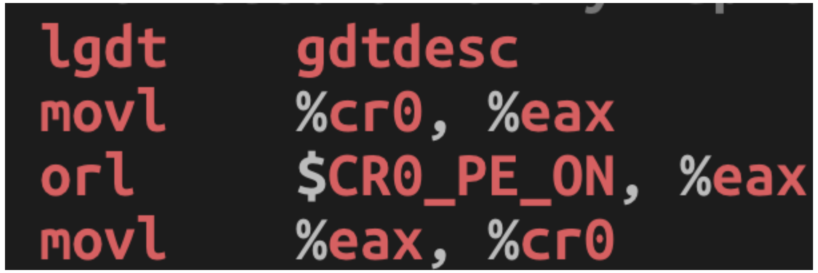
- Enable A20
- Enable protected mode (enabling 4GB memory access)
- Read kernel ELF (Executable Linkable Format)
- Do all these in 510 bytes.. (actually, uses less than this..)

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)

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

JOS Bootloader (boot/main.c)

- After enabling protected mode, boot.S will run ‘ljmpl’ (long jump, far jump) to apply the new segment assigned by the GDT.
- Then, it will call bootmain in boot.c

- 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…

- Then, run Kernel!