Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Odds and Ends

• Lab Setup & Lab 1: Document, slides, and video tutorial posted

• To-do:
  • Watch the tutorial video using the updated version of slides
  • Read the document multiple times, and follow instructions step by step
  • Check discord pinned messages!!

• Office hours starts next week (default: via discord)
  • Check Canvas homepage
Public Key Error

• It means that you did not setup your ssh keys correctly

• To solve it:
  • Generate ssh key pair using Lab1_slides (slide 11 & 12)
  • Add your public key to your GitHub account
Failed to bind socket: Address already in use

```bash
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=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
```

Run `$ kill-qemu`
kill-qemu

• This command will kill all running qemu instances that is owned by your account

• Please ignore the error message
  • It tries to kill qemu that is not owned by you, and has no effect to them
Add ~/bin to PATH in your .*shrc

• For students who typed ‘n’ on .bashrc installation,
• Please add ~/bin to your PATH environmental variable. E.g.,
  • export PATH=$PATH:~/bin

• Alternatively, you can modify the conf/env.mk file, and set
  QEMU=~/.cs444/bin/qemu-system-i386

• This will remove the errors like

```plaintext
*** 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?
***
```
Device or Resource Busy...

- This occurs when your tmux/vim/other apps working on some of the files that is required to be deleted by our ‘make’ script
- Kill all tmux/vim sessions would remove the problem
  - Make sure that you saved all your work!
Killing tmux

• RUN
  • $ kill-all-tmux

Killing vims
• $ ps aux | grep vim | grep your_username_here
• The command above will show your instance of vim
• You can kill it selectively by running
  • $ kill -9 [pid of vim]
• Or,
  • $ pkill vim
• to kill all vim instances...
Some other error messages

- Please ignore this error
  - It’s about forwarding GUI applications from the server to the client
  - We don’t use GUI applications on the server
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
Boot Sequence

• Power up

• BIOS initialize basic devices

• 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?

• 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 2021 and Intel Xeon on the os2 server

• Why?
  • Backward Compatibility
  • Use the same code for all CPUs!
What is \([f000:fff0]\)?

- Intel 8086 (1978, ~45 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

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 = 0xffff0, then it will be 0xf000:0xffff0

• 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
  • 0xffffffff (becomes 5-digit address!)

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

16
Segmentation in 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
  • \([\text{seg}:\text{offset}] = \text{seg} \times 16 + \text{offset}\)
  • e.g., \([f000:fff0] = 0xf000 \times 16 + 0xfff0 = 0xf0000 + 0xfff0 = 0xffff0\)
Quick Quiz

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

• [1000:3333]
  • $0x1000 \times 16 + 0x3333 = 0x10000 + 0x3333 = 0x13333$

• [b000:b7ff]
  • $0xb000 \times 16 + 0xb7ff = 0xb0000 + 0xb7ff = 0xbb7ff$

• [0001:0101]
  • $0x0001 \times 16 + 0x0101 = 0x0010 + 0x0101 = 0x0111$

• [f800:8001]
  • $0xf800 \times 16 + 0x8001 = 0xf8000 + 0x8001 = 0x100001$

  $0xf8 + 8 = 0x100$  OVER 1MB!!!
Real Mode Segmentation

- **SEGMENT:OFFSET**
- **SEGMENT * 16 + OFFSET!**

- Where does this code jump to?
  - 0xf000:0xe05b
    - 0xf000 + 0xe05b == 0xfe05b
Real Mode Segmentation

• Compare to what??

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

• 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!
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?

  ```iron
  irb(main):002:0> 0x55aa.to_s(2)
  => "101010110101010"
  ```

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

```bash
[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...
00000020: 1664 7c0f 20c0 6382 c801 0f22 c0ea 327c ....dj.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 ...
00000070: 89e5 ec83 e0c0 3c40 75f8 5dc3 5589 e557 ...
00000080: 8b4d ece8 e2ff ffff b001 baf2 0100 00ee ...
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 b920 baf7 ...
000000c0: 0100 00ee e8a1 ffff ff8b 7d08 b980 0000 ...
000000d0: 00ba f001 0000 fc02 6d5f 5dc3 5589 e557 ...
000000e0: 568b 7d10 538b 750c 8b5d 08c1 ef09 01de ...
000000f0: 4781 e300 feff fff9 f373 1257 5347 81c3 ...
00000100: 0002 0000 e873 ffff ff58 5aeb ea8d 65f4 ...
00000110: 5b5e 5f5d c355 89e5 5653 6a00 6800 1000 ...
00000120: 0068 0000 0100 e8b1 ffff ff8a c40c 813d ...
00000130: 0000 0100 7f45 4c46 7537 a11c 0001 000f ...
00000140: b735 2c00 0100 8d98 0000 0100 c1e6 0501 ...
00000150: de39 f373 16ff 7304 ff73 1483 c320 ff73 ...
00000160: ece8 76ff ffff 83c4 0ceb e6ff 1518 0001 ...
00000170: 00ba 008a 0000 b800 8aff ff66 efb8 008e ...
00000180: ffff 66ef ebfe 0000 0000 0000 0000 0000 ...
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 Lab 1

- QEMU uses **SeaBIOS**
  - It’s an Open Source Software, so we can take a look into the source code!

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

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

```c
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);
}
```
What does the boot sector need to do?

• 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, bcz 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)

![Segment Register Diagram](https://nptel.ac.in/courses/117104072/3225)
Why ‘Protected’?

- DPL (Data Privilege Level)
  - We can set memory privilege!!!!
### i386 Protected Mode

- **80386 (1985, 32-bit)**
  - 32-bit processor, all registers are 32 bits, \( 2^{32} = 4,294,967,295 = 4\text{GB} \) 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...

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base 0:15</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>56</td>
<td>55</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>47</td>
<td>40</td>
</tr>
<tr>
<td><strong>Limit 0:15</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>32</td>
<td>39</td>
<td>32</td>
</tr>
<tr>
<td><strong>Access Byte</strong></td>
<td><strong>Base 16:23</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16:19</td>
<td>Base 16:23</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

• Latest (Coffee Lake, 2017 and ongoing)
  • Address space: 57-bit (128PB)
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

Map code in BIOS at f000:fff0

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

Extended Memory (Over 1MB)

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

Load kernel and run!

Enabling Protected Mode

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

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

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

Low Memory
0x00000 ~ 0xa0000 (0 ~ 640KB)
Breakpoint at 0x7c00

What is A20?

# 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
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)
  • 0000 1 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 access both ends of the memory
  - 0xfffff0 (BIOS), f000:0xfffff0
  - 0x7c00 (Bootloader), 0000:7c00
  - 0xf800:7ff0 == 0xf8000 + 0x7ff0 = 0xffff0
  - 0xf800:fc00 == 0xf8000 + 0xfc00 = 0x107c00 == 0x7c00
  - **DO NOT have to change Segmentation!**

---

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

<table>
<thead>
<tr>
<th>BIOS</th>
<th>0xf0000 ~ 0x100000 (960KB ~ 1MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devices</td>
<td>0xc0000 ~ 0xf0000 (768KB ~ 960KB)</td>
</tr>
<tr>
<td>VGA</td>
<td>0xa0000 ~ 0xc0000 (640KB ~ 768KB)</td>
</tr>
<tr>
<td>Low Memory</td>
<td>0x00000 ~ 0xa0000 (0 ~ 640KB)</td>
</tr>
</tbody>
</table>

f000:fff0

f000:fff0

Need to change the segment
From 0xf000 to 0x0000

MBR, 0x7c00
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)

```assembly
# Bootstrap GDT
daalign 2

; Set GDT
lgdt gdt; ; Load GDT descriptor

; Code segment
SEG(STA_X|STA_R, 0x0, 0xffffffff) ; Segment descriptor

; Data segment
SEG(STA_W, 0x0, 0xffffffff)
```

CR0? See this: [https://en.wikipedia.org/wiki/Control_register](https://en.wikipedia.org/wiki/Control_register)

Control Register (CR)
JOS Bootloader (boot/main.c)

• After enabling protected mode, boot.S will run ‘l jmpl’ (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)
  • Load binary program into memory
  • Read header, map memory, copy data...

• Then, run Kernel!
In Lab Tutorial...

• Following the boot sequence with ‘gdb’ in assembly and C code
  • Up to Exercise 6

• Learning how Intel x86 uses STACK to store a function’s local context
  • Exercise 10!