CS 444/544
Operating Systems II

Lecture 2
BIOS, Booting, and CPU
10/3/2023

Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Odds and Ends

- Lab Setup & Lab 1: Document, slides, and video tutorial posted
- Recitations start this week (attendance are optional)
  - Week 1 Recitation: Go through setup guidelines and tools

- 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 messages!!

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

Warning: Permanently added the ECDSA host key
Permission denied (publickey).
fatal: Could not read from remote repository.

• 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

***
*** Use Ctrl-a x to exit qemu
***
quem-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

***
*** 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

X11 forwarding request failed on channel 0

• 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 2023 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
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)!
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., \([\text{f000:fff0}] = 0xf000 \times 16 + 0xfff0 = 0xf0000 + 0xfff0 = 0xffff0\)
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]
  • $0xf800 \times 16 + 0x8001 = 0xf8000 + 0x8001 = 0x100001$ OVER 1MB!!!
Real Mode Segmentation

• SEGMENT:OFFSET
• SEGMENT * 16 + OFFSET!

• Where does this code jump to?

\[
\text{The target architecture is assumed to be i8086}
\begin{array}{c}
[f000:fff0] 0xffff0: \text{ljmp} \quad \$0xf000,$0xe05b
\end{array}
\]

• 0xf000:0xe05b
  • 0xf0000 + 0xe05b == 0xe05b
Real Mode Segmentation

• Compare to what??

• \[0x0000:0x6ac8\]

• \(0x0000:0x6ac8 = 0xf6ac8\)

• \(0x6ac8\)
  • \(0x0000:0x6ac8 = 0xf6ac8\)
Boot from Disk

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

• Boot sector (Master Boot Record)
  • The 1\textsuperscript{st} sector of the disk partition
  • Ends with 0x55AA

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

Image from:
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!
In Lab 1

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

  ```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);
  }
  ```

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

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

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, \(2^{32} = 4,294,967,295 = 4\text{GB} \text{ Space!}\)
  • 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...

![Global Descriptor Table](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)
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

- **Low Memory**: 0x00000 ~ 0xa0000 (0 ~ 640KB)
- **VGA**: 0xa0000 ~ 0xc0000 (640KB ~ 768KB)
- **Devices**: 0xc0000 ~ 0xf0000 (768KB ~ 960KB)
- **BIOS**: 0xf0000 ~ 0x100000 (960KB ~ 1MB)

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)

- **Extended Memory**: Over 1MB
  - 4GB for 32bit
  - 256TB for 48bit on amd64
  - 128PB for 57bit on amd64

Load kernel and run!

Enabling Protected Mode

- **30**
What is A20?

Breakpoint at 0x7c00

```asm
.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
```

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

Breakpoint 1, 0x00007c00 in ?? ()

Registers
eax 0x000000a5  ecx 0x00000000
esp 0x000006f20  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 access the both end of the memory
    • 0xfffff0 (BIOS), f000:0xfffff0
    • 0x7c00 (Bootloader), 0000:7c00
    • 0xf800:7ff0 == 0xf8000 + 0x7ff0 = 0xfffff0
    • 0xf800:fc00 == 0xf8000 + 0xfc00 = 0x107c00 == 0x7c00
  • **DO NOT have to change Segmentation!**
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

gdt:
  SEG_NULL  # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)      # data seg

l gdt gdt desc
mov l %cr0, %eax
or l $CR0_PE_ON, %eax
mov l %eax, %cr0

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

Control Register (CR)
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!
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!