Hands-on Introduction to Bootloaders

Software Engineering, NTUA, 9th semester

Nikos Gavalas

March 2018

Sections

- Booting Process Overview
- Rolling our own bootloader
- BIOS Interrupts
- 32-bit Mode
- Global Descriptor Table
- Interacting with h/w
- Beyond the bootloader

Booting Process Overview

What happens when we press the power button on our computer? - 4 basic stages

1. BIOS/UEFI
2. Bootloader
3. Kernel
4. Init Process

BIOS (Basic Input/Output System)

- Firmware that performs basic diagnostic checks and initialization of the hardware during the booting process
- Searches for the “boot block”
  - MBR Partitioning → Looks for the “**Master Boot Record**” - The first 512 bytes of the first sector of the disk
  - The MBR contains the code that loads the bootloader afterwards.
- Limitations of the MBR:
  - Can’t handle disks with more than 2 TB of space
  - Only supports up to four primary partitions

UEFI (Unified Extensible Firmware Interface)

- Successor of BIOS
- Most modern computers use it
- GPT disk partitioning
  - However GPT is backwards compatible, it’s first sector is reserved for MBR code → booting is possible from computers with no UEFI.

Bootloader

- Responsible for passing parameters and loading the OS Kernel
- Famous Bootloaders: GRUB, U-Boot...
- The Kernel can’t “see” the hardware initially (missing drivers)
    - Solution → mounts a **root filesystem** which contains all the needed modules to communicate with the hardware
    - this root fs was “initrd”, now “initramfs”
    - the kernel creates a “root device” and mounts the “root partition” in read-only mode, then verifies the integrity of the other partitions with “fsck”, and remounts the root partition in rw mode.
    - finally executes the “**Init**” process (systemd → initializes OS services) - every other process it a fork of “init”.

Rolling our own bootloader

We'll be using the following tools:

- **nasm** to assemble our code
- **qemu** to emulate a pc (arch: 32-bit i386 - x86)

and as bootable storage media for qemu we’ll emulate a **floppy disk**→ (way) simpler

Rolling our own bootloader

So when we power up our computer:

- BIOS is loaded from some flash memory on the motherboard
- Performs h/w checks
- Tries to find bootable storage devices (according to a boot sequence)
- Checks their first 512 bytes stored (MBR)
- If those bytes are **`bootable`**, it jumps to memory address **`0x7c00`** and starts executing, that's where we'll be placing the code for our bootloader

But what code is "bootable"?

- If the last 2 bytes of the MBR data are **`0xaa55`**, then the code is considered bootable

Rolling our own bootloader

Notice the address - 0x7c00 → 16-bit

It’s because the processor at this point runs at **16-bit mode** (Real Mode)

- Only 16-bit registers are available (ax, bx, ...)
- 20-bit segmented memory address space → 1 MiB of addressable memory
- No memory protection, access to all addressable memory

Later we will see how to enter the 32-bit “Protected Mode”

Getting our bootloader to print a message

Wait a second… print *where, how?*

We will use the BIOS interrupt calls, which provide basic I/O functionality.

We can see that we can call BIOS code to print a character (`al` $\leftarrow$ char) in tty mode, using the interrupt vector 0x10 (and `ah` $\leftarrow$ `0x0e`)

But first, a quick refresher on asm x86

All x86 registers [here]

Control Registers

CR0

bit	label	description
0	pe	protected mode enable
1	mp	monitor co-processor
2	em	emulation
3	ts	task switched
4	et	extension type
5	ne	numeric error
16	wp	write protect
18	am	alignment mask
29	nw	not-write through
30	cd	cache disable
31	pg	paging

Index Registers

32 bit	16 bit	description
esi	si	source index
edi	di	destination index

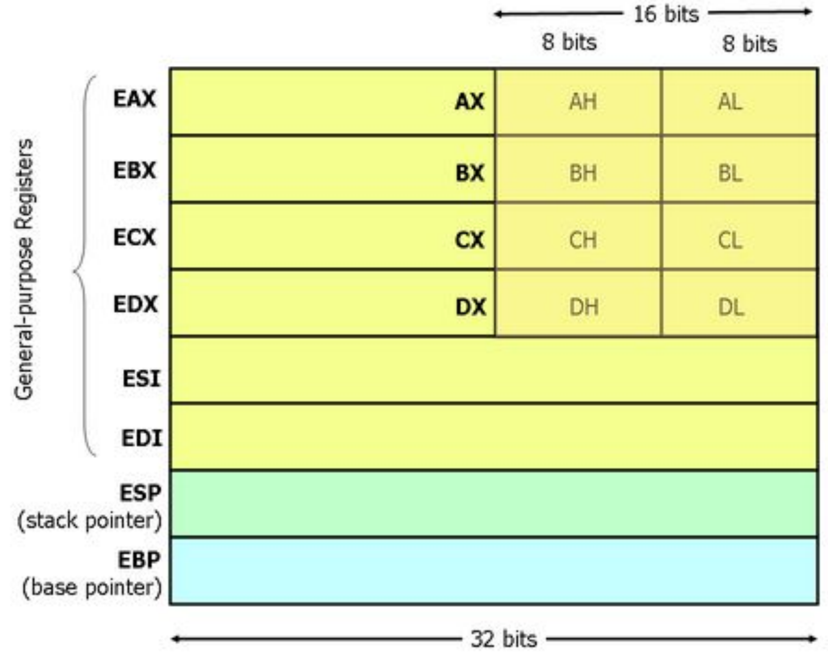


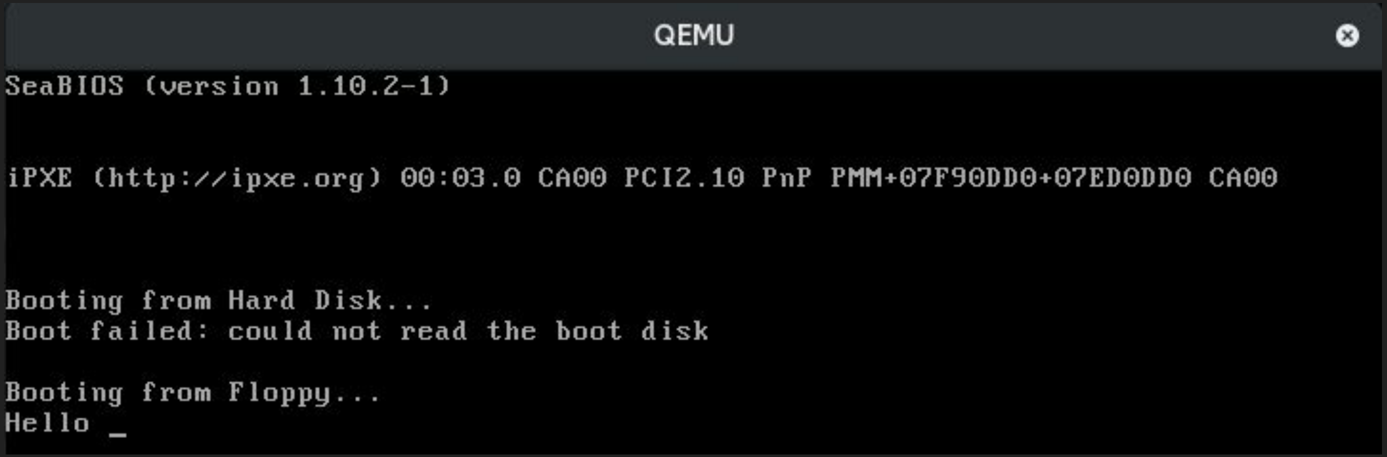
Figure 1. x86 Registers

First steps

```
bits    16
org     0x7c00

global  main

main:
    mov     bx, str                 ; load str address

loop:
    mov     byte ax, [bx]           ; load byte from where bx points to

    inc     bx

    push    bx
    mov     ah, 0x0e                ; print char in TTY mode
    int     0x10                    ; s/w intr -> BIOS video services
    pop     bx

    cmp     al, 0                   ; if we reached null
    jz      halt
    jmp     loop

halt:
    cli                             ; clear interrupts
    hlt                             ; halt

str:
    db  "Hello", 0

times   510 - ($-$$) db 0           ; pad remaining 510 bytes with zeroes
                                    ; (in nasm, $ -> current address,
                                    ; and $$ -> beginning of current section)
dw      0xaa55                      ; mbr magic number (2 bytes + 510 = 512 total)
```

Assemble and run

- Assemble with nasm:
  - `$ nasm bootloader.asm`
- Run with qemu:
  - `$ qemu-system-i386 -fda bootloader`



Let’s check the assembled 512-byte binary

```
bootloader

00000000 BB 14 7C 8B 07 43 53 B4 0E CD 10 5B 3C 00 74 02 EB F1 FA F4 48 65 6C 6C 6F 00 00 00 00 00 00 00 ..|..CS....[<.t.....Hello.......
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
000000c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
000000e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
000001c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
000001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA ..............................U.
00000200
```

We can see the magic number at the last two bytes

Something interesting

Let's check out our own computer's MBR code

- `$ dd if=/dev/sda bs=512 count=1 > mbr`

We can even disassemble it with `objdump` and take a closer look:

- `$ objdump -D -b binary -mi386 -Maddr16,data16 mbr`

Accessing more memory and 32-bit protected mode

To be able to access more memory we'll need to enable the A20 line

We can do that with another BIOS interrupt call → int 0x15

```
mov     ax, 0x2401
int     0x15            ; enable A20 bit
```

Now, to enter the Protected Mode and access the full 32-bit registers, we first need to set up a GDT (Global Descriptor Table)

- The GDT contains entries telling the CPU about memory segments
- Useful for memory protection

GDT

Define memory segments:

See here.

```
gdt_start:
    dq  0x0

gdt_code:                               ; code segment from 0 to 0xffff
    dw  0xFFFF
    dw  0x0
    db  0x0
    db  10011010b                       ; r/w/e
    db  11001111b
    db  0x0

gdt_data:
    dw  0xFFFF                          ; data segment from 0 to 0xffff
    dw  0x0
    db  0x0
    db  10010010b                       ; r/w
    db  11001111b
    db  0x0

gdt_end:

gdt_pointer:                            ; pointer structure needed from the cpu to know
                                        ; how big the gdt is
    dw  gdt_end - gdt_start
    dd  gdt_start


CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

32-bit mode

Special command to load the GDT:

```
lgdt [gdt_pointer]
```

Finally we can enable protected mode by setting the corresponding bit on the special cr0 register:

```
mov eax, cr0
or eax,0x1
mov cr0, eax
```

We can now use the full registers

Now let’s try accessing Hardware

Now that we are in protected mode, let’s write on the screen using the VGA buffer, because why not

First we have to set video mode from the corresponding BIOS call:

```
mov     ax, 0x3
int     0x10
```

Then we can write directly to address 0xb8000 where the directly mapped VGA buffer is located, according to this format:

Attribute								Character							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Blink[n 1]	Background color			Foreground color[n. 2][n. 3]				Code point							

Writing on the VGA buffer

```
    mov     bx, str                 ; load str address

    mov     esi, str
    mov     ebx, 0xb8000            ; vga buffer address
loop:
    lodsb                           ; load byte at address DS:(E)SI into AL
    or      al, al                  ; update flags
    jz      halt
    or      eax, 0x0200             ; green color
    mov     word [ebx], ax          ; write to the vga buffer
    add     ebx, 2                  ; go to the next cell
    jmp     loop

halt:
    cli                             ; clear interrupts
    hlt                             ; halt

str:
    db  "Hello", 0
```

It’s cleaner than before, isn’t it

How about we access more hardware?

Let’s go for the disk now

We don’t really have a choice, because BIOS only loads the first 512 bytes, therefore if we want to write bigger programs, we ‘ll have to load them from the disk

How? You guessed it, BIOS Disk Interrupts

With int 0x13, ah=0x02, we can read sectors from the drive:

```
    mov     [disk], dl          ; BIOS places the disk index on dl on startup, so we save it

    mov     ah, 0x2             ; read sectors
    mov     al, 1               ; sectors to read
    mov     ch, 0               ; cylinder index
    mov     dh, 0               ; head index
    mov     cl, 2               ; sector index
    mov     dl, [disk]          ; disk index
    mov     bx, copy_target     ; target pointer
    int     0x13
```

What’s next?

We can go further and start building a basic kernel

But how? We need a way to compile code (mainly C) for our new system

What we need is a *cross-compiling toolchain* → to compile software for our targeted arch from our host system

A toolchain includes everything that cross-compilation needs:

- Binutils (assembler, linker, …)
- C compiler, C lib and more

However...

Building a cross-compiling toolchain is a nightmare, too difficult task

We use pre-compiled ones… but they don’t offer much flexibility

The best choice is to go for a toolchain building tool, which makes the process of building a toolchain easier

The most popular toolchain building tools are crosstool-ng, OpenEmbedded, Buildroot etc

To sum up

… There are a lot of ways to print ‘Hello” on the screen…

Let’s go through the steps once more

- BIOS searches for bootable code
- This bootable code in the boot sector loads the bootloader
- The bootloader loads the kernel
- The kernel checks the disks partitions, then remounts the filesystems, and executes init

Resources

- https://deltahacker.gr/how-pcs-boot/
- https://wiki.osdev.org/Bootloader
- https://wiki.osdev.org/Boot_Sequence
- https://wiki.osdev.org/Rolling_Your_Own_Bootloader
- https://www.codeproject.com/Articles/664165/Writing-a-boot-loader-in-Assembly-and-C-Part
- https://osandamalith.com/2015/10/26/writing-a-bootloader/

Questions?