Project 1: Boot-Up Mechanism

Loading a kernel and switching to 32-bit mode

Mike Murphy

Spring 2024

Contents

Project 1

Project Overview

Project 1

Project 1 is a demo

- You do not have to turn anything in
- Get the code. Build it. Run it.
- Set up your build environment
- Familiarize yourself with the code and the tools

We will be building an OS

- Each project will build on the previous
- Project 1: We give you a “hello world”
- Project 2: You start building a kernel
- Projects 3-6: You add more and more features

Project 1: “Hello World”

The code contains

- Bootloader
- “Hello world” kernel
- Utility to create a bootable disk image

The image

1. Boots
2. Loads the kernel
3. Prints a hello message

Architecture

- We’re going back to the 90s: 32-bit Intel x86 (aka i386)
- The OS can boot on some old hardware, but it’s getting harder
- We will develop primarily in an x86 emulator: Bochs

Boot Basics

What happens when a PC boots?

First steps

1. CPU: starts executing at 0xfffffff0 (*reset vector*)
2. Motherboard: makes sure there’s something to execute: a jump to 0xf0000
3. BIOS ROM: starts at 0xf0000

BIOS: Basic Input Output System

- Firmware in ROM
- Knows how to talk to disks, keyboard, display
- Knows how to look for a bootable disk
- Provides interface for software to talk to disks, keyboard, display

Next steps

4. BIOS: looks for a bootable disk
5. BIOS: loads bootloader from disk
6. Bootloader: loads kernel from disk
7. Kernel: starts

UEFI is the New BIOS

UEFI: Unified Extensible Firmware Interface

- Began around 1998
- Replacement for BIOS
- Much more sophisticated

For example, the bootloader:

- BIOS:
  - Reads first 512 bytes of disk
  - Those 512 bytes contain simple partition table and bootloader code
  - First bootloader typically needs to load a *second stage* bootloader that understands the filesystems and can find the kernel
- UEFI:
  - Understands filesystems and executable file formats
  - Loads bootloader like an OS would load a normal program

We Don't Use UEFI

UEFI BIOS Compatibility

- UEFI replaced BIOS, but it kept *Compatibility Mode Support* (CSM) …until ~2020
- New computers are UEFI-only

Our OS still uses the old BIOS boot system

- This is why it's getting harder to boot our OS on real PCs
- This is why we're giving you this code instead of making you write it
- Writing a BIOS bootloader requires knowledge of obsolete hardware modes

That said, let's talk about some obsolete hardware modes…

Some History

x86 Family History

Year	CPU	Regs	Addresses	Hottest feature
1978	Intel 8086	16-bit	20-bit (1 MiB)	Segmented addressing
1982	Intel 286	16-bit	24-bit (16 MiB)	Memory protection
1985	Intel 386	32-bit	32-bit (4 GiB)	Virtual memory paging
…				
2003	AMD Opteron	64-bit	52-bit (4 PiB)	64 bits!

Note the shifting bottlenecks

- 16-bit CPUs: registers. Harder to make wider registers than more memory
- 32-bit CPUs: memory, then regs. 4 GiB of RAM was a lot … until it wasn't
- 64-bit CPUs: memory again. 4 PiB of RAM is a lot

Why is History Important?

Because every boot is a walk through history

- x86 CPUs are aggressively backwards compatible
- Even new CPUs act like an 8086 at boot, just in case
- Software must start with 16-bit code and bootstrap its way to 32- or 64-bit
  - BIOS-style: bootloader and OS must do this (our OS does this)
  - UEFI-style: firmware does this before handing off
- Understanding history helps you understand the present

i8086 Segmented Memory Addressing

Address Space	Registers
20-bit (1 MiB)	16-bit (64 KiB)

- Divide memory into 64 KiB *segments*
  - Instruction pointer (IP) $\rightarrow$ Code Segment (CS)
  - Stack pointer (SP) $\rightarrow$ Stack Segment (SS)
  - Data pointers $\rightarrow$ Data Segments (DS, ES, FS, GS)
- CS, SS, DS, ES, FS, GS are 16-bit segment registers
  - Specificy base of segment (shift by 4)

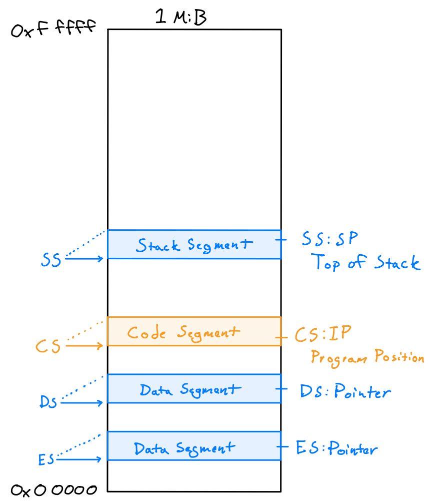

Figure 1: Segmented addressing

- CS 0x1234 → Address 0x12340
- IP 0x5678 → Address 0x12340 + 0x5678 = 0x179B8

- This allows a form of software relocation
  - Link your executable so that offsets start at 0x0000
  - Put it anywhere in memory, then set CS and DS
  - Our bootloader code does this

i286 Protected Mode Addressing

Address Space	Registers
24-bit (16 MiB)	16-bit (64 KiB)

- Add a layer of indirection
  - Move segment information into memory
  - Set up a table of 8-byte *segment descriptors*
- 8-byte segment descriptor in memory
  - 24-bit base address
  - 16-bit limit (end of segment)
  - various flags, including privilege level
- 16-bit segment register
  - 13 bit index into descriptor table
  - 3 bits for flags
- Diagram convention: addresses count up from bottom
  - "Higher" addresses are literally higher on the page

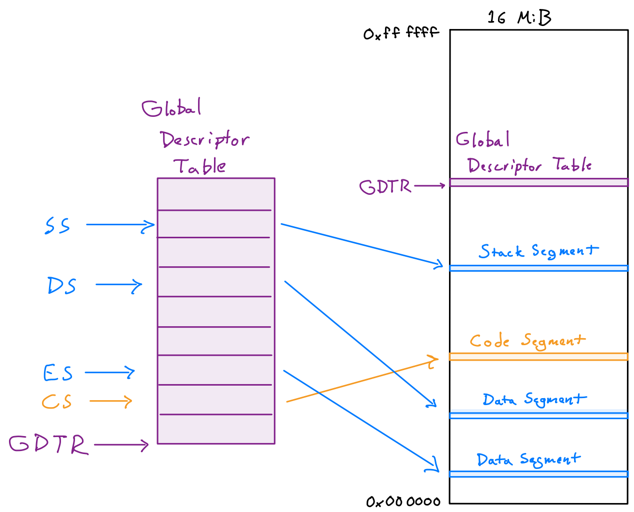

Figure 2: Protected mode addressing

Protected Mode Privilege Levels

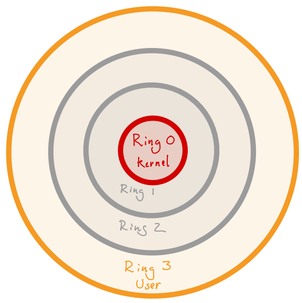

Figure 3: Privilege levels

- 2 bits for privilege level → 4 privilege levels
  - Ring 0 (innermost): operating system kernel
  - Rings 1 & 2: intended for device drivers
  - Ring 3 (outermost): user applications
- Segment descriptor determines current priv level
  - CS register selects descriptor
  - Descriptor priv level determines current priv level
  - Certain operations can only be executed with priv 0
  - Attempts to change CS are checked against current priv level vs requested priv level
- In practice, most operating systems use only kernel and user

i386 Flat Memory Model

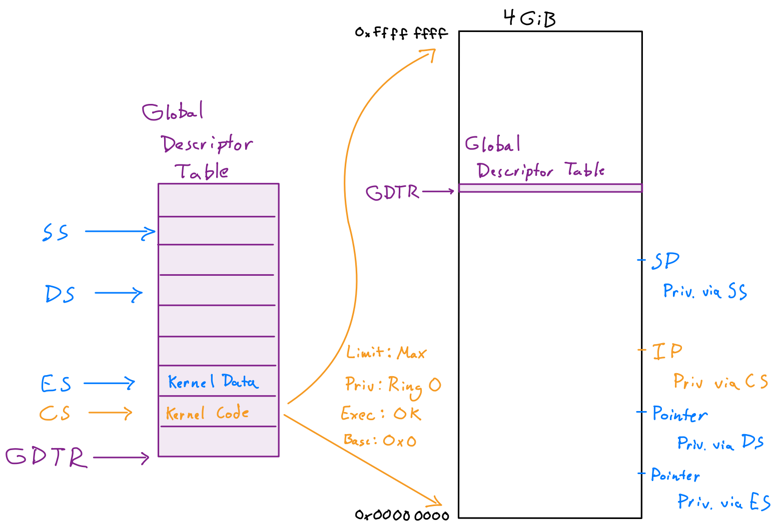

Figure 4: Flat addressing

Address Space	Registers
32-bit (4 GiB)	32-bit (4 GiB)

- Do we still need segment descriptors?
  1. To reach whole address space? — Not needed
  2. To mark off protected regions of memory? — No, paging is better
  3. To set current privilege level? — Still used
- Paging

  The 386 also introduces memory paging, which provides a much finer-grained way to mark memory for protection (per 4 KiB page). So most operating systems use that instead. More on paging in Project 4.
- Vestigial descriptor table
  1. Kernel-level code: priv 0, executable
  2. Kernel-level data: priv 0, no-execute
  3. User-level code: priv 3, executable
  4. User-level data: priv 0, no-execute

x86-64 Canonical Addressing

Address Space	Registers
52-bit (4 PiB)	64-bit (16 EiB)

- Pointers can address more memory than we can make RAM for

    - CPU designs don't bother having 64 address lines
    - Only 52 lines, which can address up to 4 PiB
- Canonical address form
    - High bits must be all 0 or all 1

      ```
      0xFFF0 0000 0000 0000 -- 0xFFFF FFFF FFFF FFFF
      0x0000 0000 0000 0000 -- 0x000F FFFF FFFF FFFF
      ```

    - This prevents software from trying to use unused bits for other purposes

Moving Through History on Boot

- CPU starts in 16-bit *Real Mode* (8086-style)
- To get to 32-bit *Protected Mode* (386-style)
    1. Create a Global Descriptor Table (GDT) in memory
    2. Set the Global Descriptor Table Register (GDTR)
    3. Enable the A20 line
    4. Enable Protected Mode
    5. Jump into 32-bit code

  $\rightarrow$ Our OS gets to here $\leftarrow$
- The A20 line is a backwards-compatibility hack
    - Segmented addressing can overflow

      ```
      Segmented address       Linear address
      0xffff:ffff         -> 0xffff0 + 0xffff = 0x10ffef
                                Overflows to   0x 0ffef
      ```

    - Some 8086 software started relying on this overflow
    - So 8086 mode needs to keep this overflow behavior by disabling address line 20 (A20)
    - To actually reach 0x10ffef, you need to enable A20
    - The switch to do this got tied to the keyboard controller
- To get to 64-bit *Long Mode* (x86-64)
    1. Set up paging data structures in memory
    2. Set the appropriate paging registers
    3. Enable Long Mode
    4. Jump into 64-bit code

  $\rightarrow$ UEFI gets here before handing off $\leftarrow$

Side Note: AMD and x86-64

Why was it AMD who had the first 64-bit x86 system? Because they developed the architecture

Intel went in a different direction at first.

Intel Itanium and Intel Architecture 64 (IA-64)

- All-new architecture, not backwards compatible
- Based on *Very Long Instruction Word (VLIW)* concept
- VLIW:
  - Instructions that can be executed in parallel can be bundled together into one very long instruction word
  - Requires a VLIW-aware compiler that can analyse instructions, decide which ones can execute in parallel, and bundle them together
- Backwards compatibility with x86 through emulation layer, very slow
- You buy an expensive new processor, and it makes the software you have *slower*? Nobody wants that.
- This is why backwards compatibility is so important.
- Commercial flop. Tech press called it "The Itanic".
- AMD extended the x86 architecture instead. That's what stuck.
- Side-side note: UEFI was developed for Itanium
- Intel needed a firmware that could have both x86 code and Itanium code, to support both processors.
  - This includes hardware drivers that are bundled into the firmware.
  - This the *unified* and *extensible* part of Unified Extensible Firmware Interface

BIOS Booting and Our OS

BIOS Memory Map

- 1 MiB, divided into sixteen 64 KiB blocks

Addr	Block	Desc
`0xf0000`	F block	System ROM BIOS
`0xe0000`	E block	PCjr cartridges
`0xd0000`	D block	PCjr cartridges
`0xc0000`	C block	ROM expansion
`0xb0000`	B block	CGA memory
`0xa0000`	A block	EGA memory
`0x00000`	0-9 block	"conventional memory" / "low memory"

- Blocks 0-9: 640 KiB, this is the origin of the classic 640k limit

Where to Load Our OS

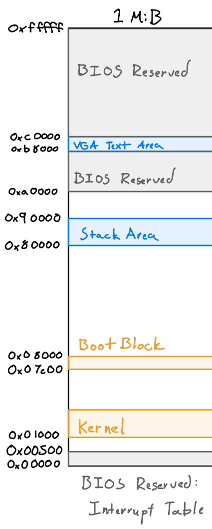

Figure 5: P1 memory map

- 0xa0000: Reserved: High memory
  - 0xb8000: VGA text buffer
- 0x80000: Stack
- 0x07c00: Boot block
- 0x01000: Kernel
- 0x00000: Reserved: Interrupt vector table

Disk Image: Normal OS for BIOS-Based System

Disk

- First 512 bytes: bootloader code + partition table
- Rest of disk is divided into partitions
- Partitions have filesystems
- Executables are ELF files

Boot

1. BIOS loads bootblock
2. Bootloader loads second-stage bootloader
3. Second-stage bootloader loads kernel ELF

Disk Image: Our OS

Disk

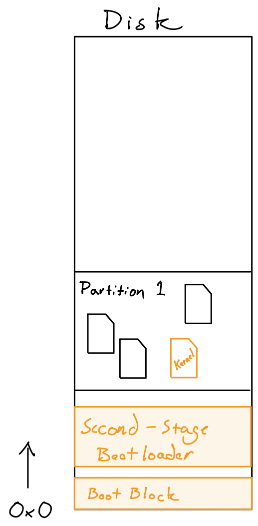

Figure 6: Typical OS disk

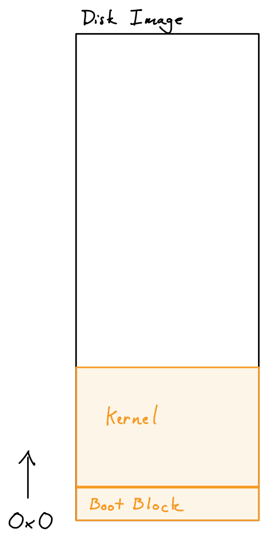

Figure 7: Our OS disk image

- First 512 bytes: bootloader code (no partition table)
- Rest of disk is raw kernel data

Boot

1. BIOS loads bootloader
2. Bootloader loads kernel

`createimage` program

- Read the kernel ELF file at build time
- Copy to image instead of memory

Disk Image to Memory

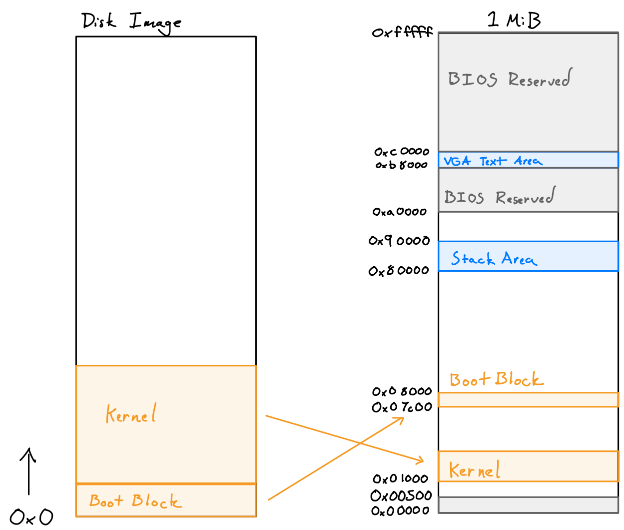

Figure 8: Loading boot block and kernel from disk to memory

Reading a Disk, BIOS Style: CHS Addressing

- CHS: Cylinder / Head / Sector
- Floppy disks
  - Move arm to select *track*
  - Select one of two read *heads*, one per side
  - Time read to rotation to select a *sector*
- Spinning hard disks
  - Stack of disk platters
  - Move arm to select *cylinder* of stacked tracks
  - Select head to select platter and side

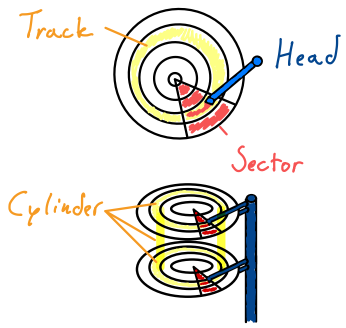

Figure 9: Spinning disk platters

- Mapping blocks ↔ CHS
  - Block 0 ↔ CHS 0/0/1
  - No circuitry in floppy controller for mapping
  - Software must do the mapping
  - Even solid-state drives provide CHS emulation
- Need to know the *geometry* of the disk
  - Number of sectors: when to roll over to next head
  - Number of heads: when to roll over to next cylinder
  - Number of cylinders: size of disk

Building the OS

OS Source Code

Repository tree

```
.
|-- src
|    |-- boot     Bootloader source
|    |-- kernel   Kernel source
|    `-- lib      Libraries source
|-- host          Output directory for host machine (your computer)
|-- target        Output directory for target machine (x86-32)
`-- thirdparty    BIOS images for emulator
```

Build process

```
1. Compile src/lib/**           -> target/lib/lib*.a
2. Compile src/boot/**          -> target/boot/bootblock
3. Compile src/kernel/**        -> target/kernel/kernel
4. Compile src/createimage.c    -> host/createimage
5. Run host/createimage         -> image
```

Make

Make Commands

```
make                 # Default, equivalent to 'make image test'
docker-run make      # Run the same command in a Docker container
```

Makefiles

```
Makefile             # Main Makefile, delegates to host or target Makefiles
host/Makefile        # Rules for compiling for host machine
target/Makefile      # Rules for compiling for target machine
src/Makefile.common  # Common rules, included by host and target Makefiles
```

Host System (Your Computer)

- x86-64 architecture
- Recommend Ubuntu 22.04 LTS (Jammy Jellyfish)
  - Other Linuxes should work, but may have GCC version issues
  - Windows Subsystem for Linux might work
  - Mac ???
- Can also use Docker
  - `Dockerfile` describes the system
  - `docker-run` script spins up a container and runs a command

  ```
  docker-run make
  ```

Emulator: Bochs

- Supported emulator: Bochs
  - x86 emulator that focuses on accuracy over speed
  - Sorry, no QEMU (yet)
- Bochs built-in debugger
  - Similar to GDB, but a little clunky
  - Advantage: 16-bit code (bootloader)
  - Install `bochs` from `apt`: configured for built-in debugger
- Connect Bochs to GDB (recommended)
  - Much more powerful
  - Weakness: 16-bit code (because GDB assumes flat memory model)
  - Requires building Bochs from source

Project Summary

Suggested Tasks

- Build: get code, set up environment, build OS

- Step through with a debugger
  - Bochs debugger? Step through bootloader (breakpoint: 0x7c00)
  - Bochs + GDB? Step through 32-bit code (breakpoint: `_start32`)
  - Get a feel for x86 ASM and how C maps to it
- Read code
  - Can you follow the bootblock ASM?
  - Can you follow the kernel C code?
  - Can you follow the C library code?
- Look at docs
  - `doc/x86`: x86 architecture and ASM programming
  - `doc/abi`: System V ABI docs (C calling convention, ELF format, etc.)
  - Download and flip through Intel manuals
  - Download and flip through AMD manuals (AMD may be easier to read)

Ask for Help

- The OS course is challenging
  - But it's supposed to be a fun challenge
- We're here to help
  - Ask questions on Discord (and please ask in the open chats)
  - Talk to your TA
  - Come to the colloquium sessions
- We also need feedback
  - The course is in transition, there will be hiccups
  - The code is in transition, there will be bugs
  - Let us know what is working and what is not