Precept 1: Bootloader

COS 318: Fall 2021

Project 1 Schedule

- **Design Review:**
  - Mon 9/13, 8:30-10:30pm
  - Wed 9/15, 3-7pm

If these times do not work, let us know & we can figure out another time

Project 1 Schedule

- **Precept:**
  - Monday 9/13, 7:30-8:20 PM
  - Tuesday 9/14, 7:30-8:20 PM

No precept next week 9/20 & 9/21!

Project 1 Schedule

- **Project Due:**
  - Sunday 9/26, 11:55 PM

Project 1 Overview

1. Write a bootloader: **bootblock.s**
   - Write bootloader that sets up and starts running the OS
   - Written in x86 assembly (AT&T syntax)
2. Implement a tool to create a bootable OS image: **createimage.c**
   - Build OS image w/ bootloader and kernel
   - Understand how executable files are structured (ELF format)

General Suggestions

- Read **assembly_example.s** in starter code *//u/318/code/project1*
- Get **bootblock.s** working before starting on **createimage.c**
- Read documentation on ELF format
- If you haven’t already started, *start asap*

Segment Registers

- Set **%cs** as needed in bootloader, zero in kernel
- Bootloader linked with offset of 0x0
  - use **%ds** when accessing items in bootloader
- Kernel linked with offset of 0x1000
  - **%ds** must be set to 0x0 before tfer control

Bootloader

What is a bootloader?

The entire OS starts on USB, but need to move it into main memory on the machine.

To do that, BIOS loads the 1st sector (512 bytes) of the USB into memory and runs it, presumably to load the rest of the OS.

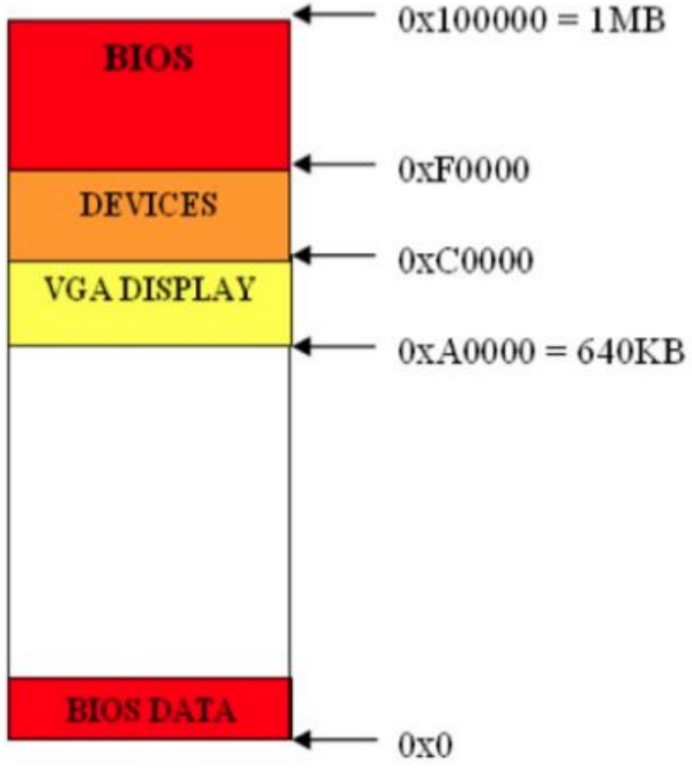
One-time step to load rest of kernel, and some other setup like changing from 16 bit to 32 bit (You won't need to do this)

Then, switches control to the rest of the kernel to actually do kernel things.

More info: https://wiki.osdev.org/Bootloader

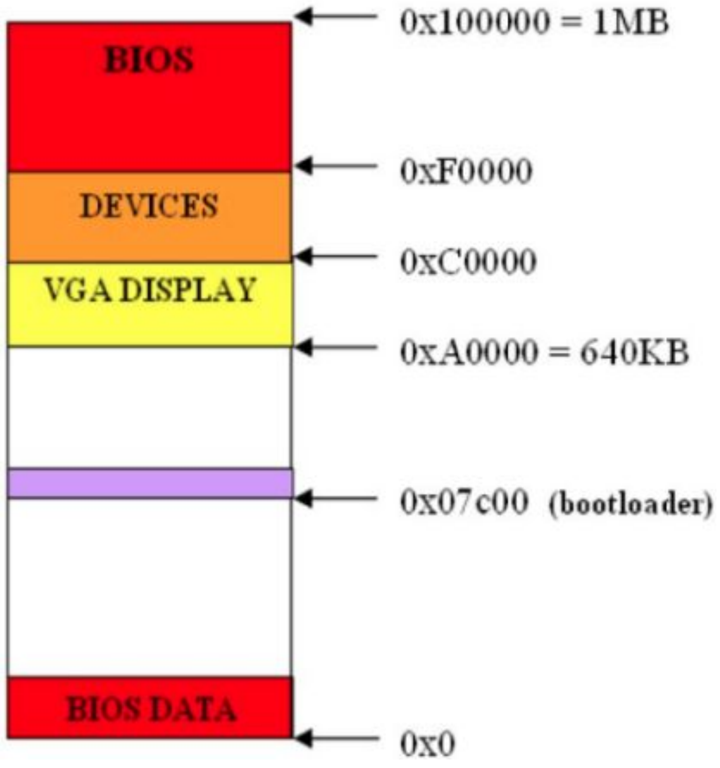
Boot Process

- Nothing in RAM on startup:
  - Load BIOS from ROM
  - BIOS loads bootloader from disk
  - Bootloader loads the rest



Loading the Bootloader

- BIOS finds bootable storage device (HDD, USB, etc.)
- Load first disk sector (MBR) into RAM at 0x7c00
- Switch control to this location

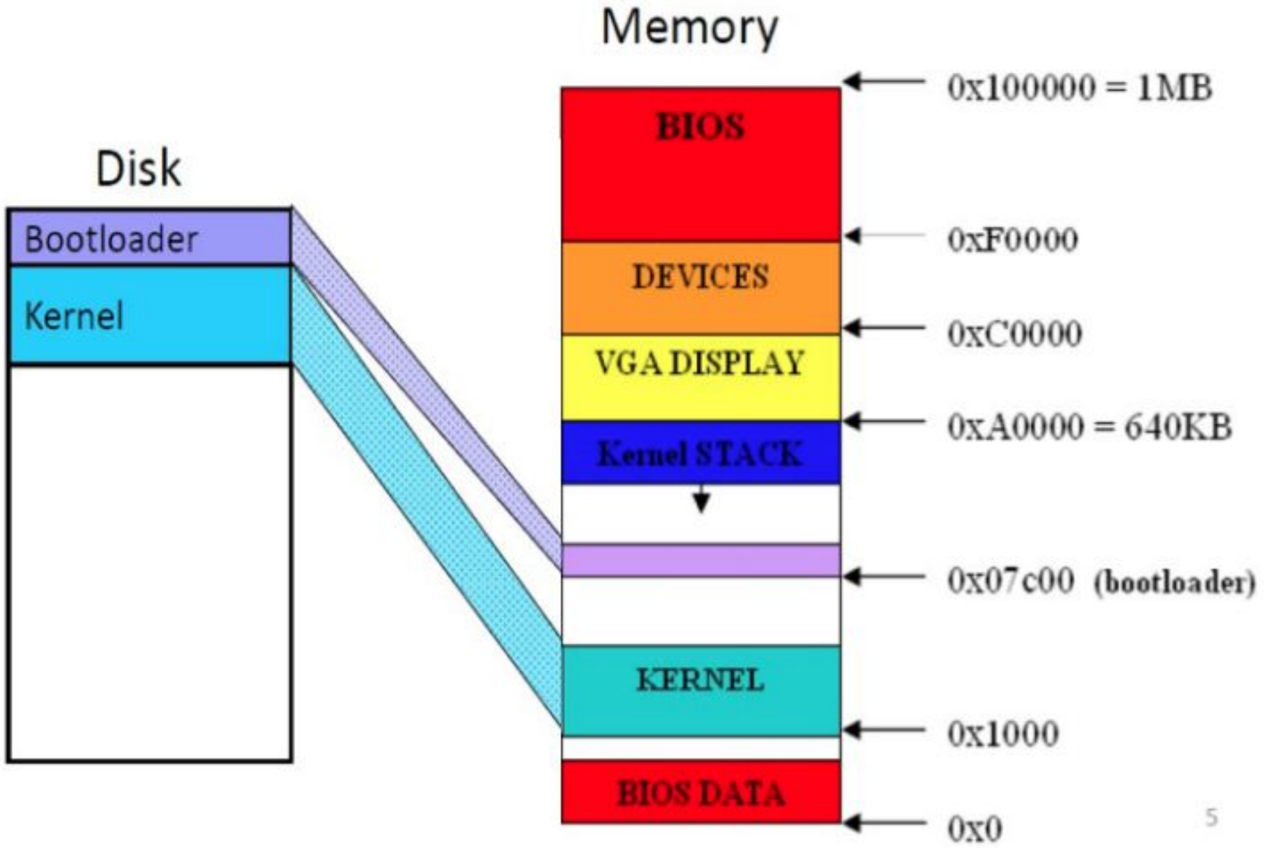


Master Boot Record

- First sector of a hard disk
    - Beginning: bootloader code
    - Remaining part: partition table
- BIOS sets %dl to drive number for bootloader
- For more info: see MBR and Partition Table

Bootloader Tasks

1. Load kernel into memory
2. Setup kernel stack
3. Transfer control to kernel



BIOS Services

- Use BIOS services through INT instruction
  - Store parameters in the registers (usually AH/AX)
  - INT triggers a software interrupt
- **int $SERVICE_NUM**
  - int $0x10   # video services
  - int $0x13   # disk services
  - int $0x16   # keyboard services

BIOS INT 0x13

- Function 2 reads from disk
  - %ah: 2
  - %al: Number of sectors to read
  - %ch: Cylinder number (bits 0-7)
  - %cl: Sector number (bits 0-5); bits 6-7 are bits 8-9 of the cylinder number
  - %dh: Starting head number
  - %dl: Drive number
  - %es:%bx: Pointer to memory region to place data read from disk
- Returns
  - %ah: Return status (0 if successful)
  - Carry flag = 0 if successful, 1 if error occurred
- For more information:
  - https://en.wikipedia.org/wiki/Cylinder-head-sector
  - https://stanislavs.org/helppc/int_table.html

Createimage + ELF

Why do we need to make an image?

Our development process is writing/compiling/assembling the kernel/bootloader on our own machine. Thus, these programs are compiled to run on *our(courselab) machines* in userspace thru the existing OS (unix), so they have metadata which is not necessary for running as a barebones program.

Since we want to run on barebones hardware, we need to strip that away to just the machine code/instructions. We also need to add some extra magic so that BIOS knows it’s meant to be a OS and not random data.

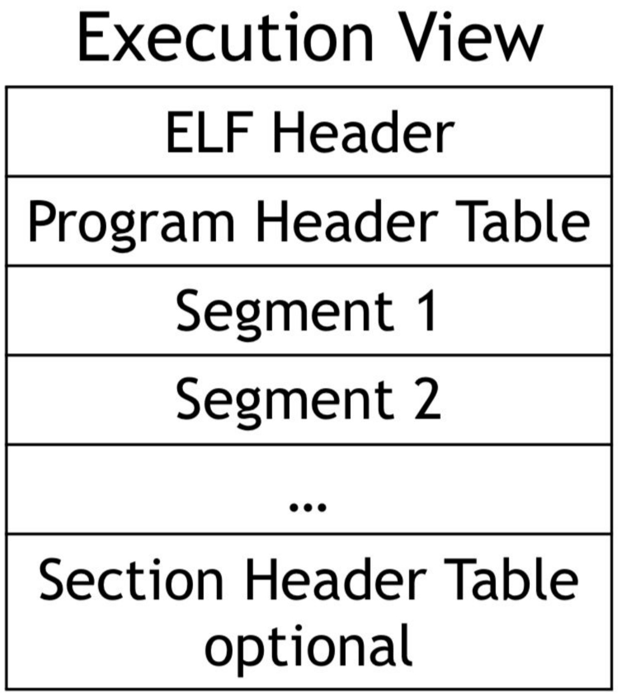
The compiled output on Unix is ELF, which is thankfully well-documented and easy to modify.

ELF Format

- ELF = Executable and Linkable Format
- Created by assembler (as) and linker (ld)
- Object File: Binary programs intended to be executed (.o files) by an OS
- ELF is supported by various processors/architectures
- Represents metadata in a machine-independent format

ELF Object File Format

- Header (pp. 1-3 to 1-5)
  - Beginning of file
  - Like a roadmap for file's organization
- Program Header Table (p. 2-2)
  - Array, each element describes a segment
  - Tells system how to create the process image
  - Files used to create an executable program must have a program header
- ELF Manual



p. 1-1 in the ELF manual

ELF Useful Tools

- **objdump**: Display information from object files
  - Read manual page (*man objdump*)
- **hexdump**: Display file contents in hexadecimal, decimal, octal, or ascii
  - Read manual page (*man hexdump*)

Questions?