



Project 1: Bootloader

COS 318

Fall 2013



Project 1 Schedule

- Design Review
 - Monday, Sep 23
 - 10-min time slots from 11am to 7:40pm
- Due date: Sep 29, 11:55pm



General Suggestions

- Read *assembly_example.s* in start code pkg
- Get *bootblock.s* working before starting on *createimage.c*
- Read documentation on AT&T syntax x86 Assembly language
- Read provided documentation on ELF format
- Start as early as you can, and get as much done as possible by the design review



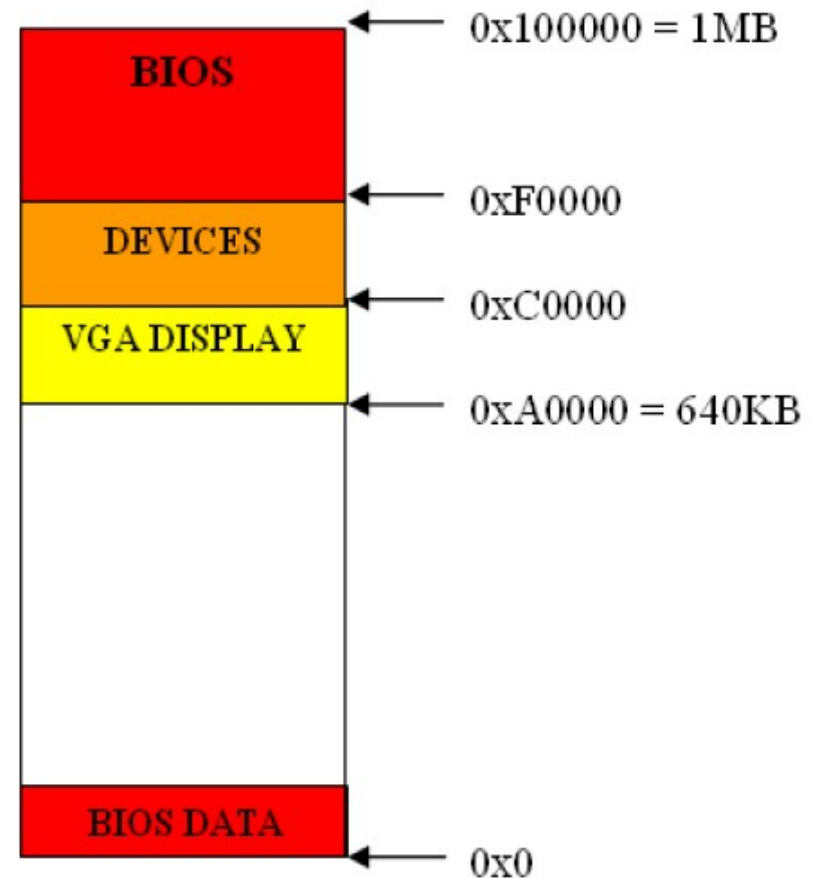
Project 1 Overview

- Write a bootloader: ***bootblock.s***
 - What happens during the boot process of a PC?
 - Written in x86 Assembly language (AT&T syntax)
- Implement a tool to create a bootable OS image: ***createimage.c***
 - How are executable files structured?
 - Become familiar with ELF format



Boot Process

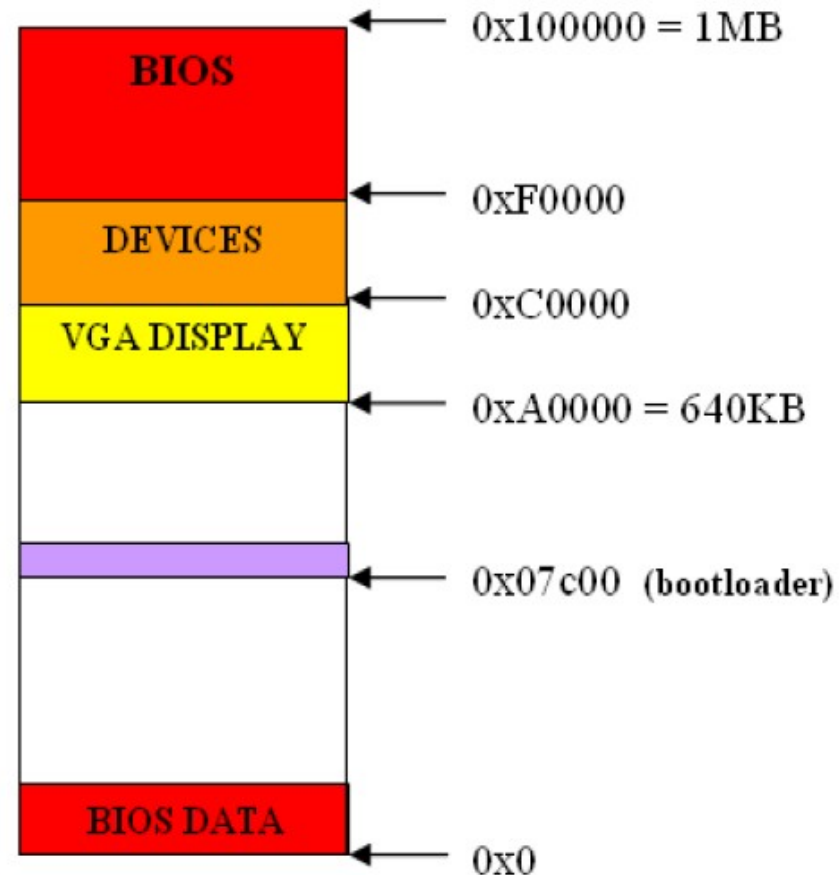
- When powered up, nothing in RAM, so how do we get started?
 - Resort to hardware
 - Load BIOS from ROM
- BIOS:
 - Minimal functionality
 - Initialization of I/O devices
 - Search for bootable devices





Loading the Bootloader

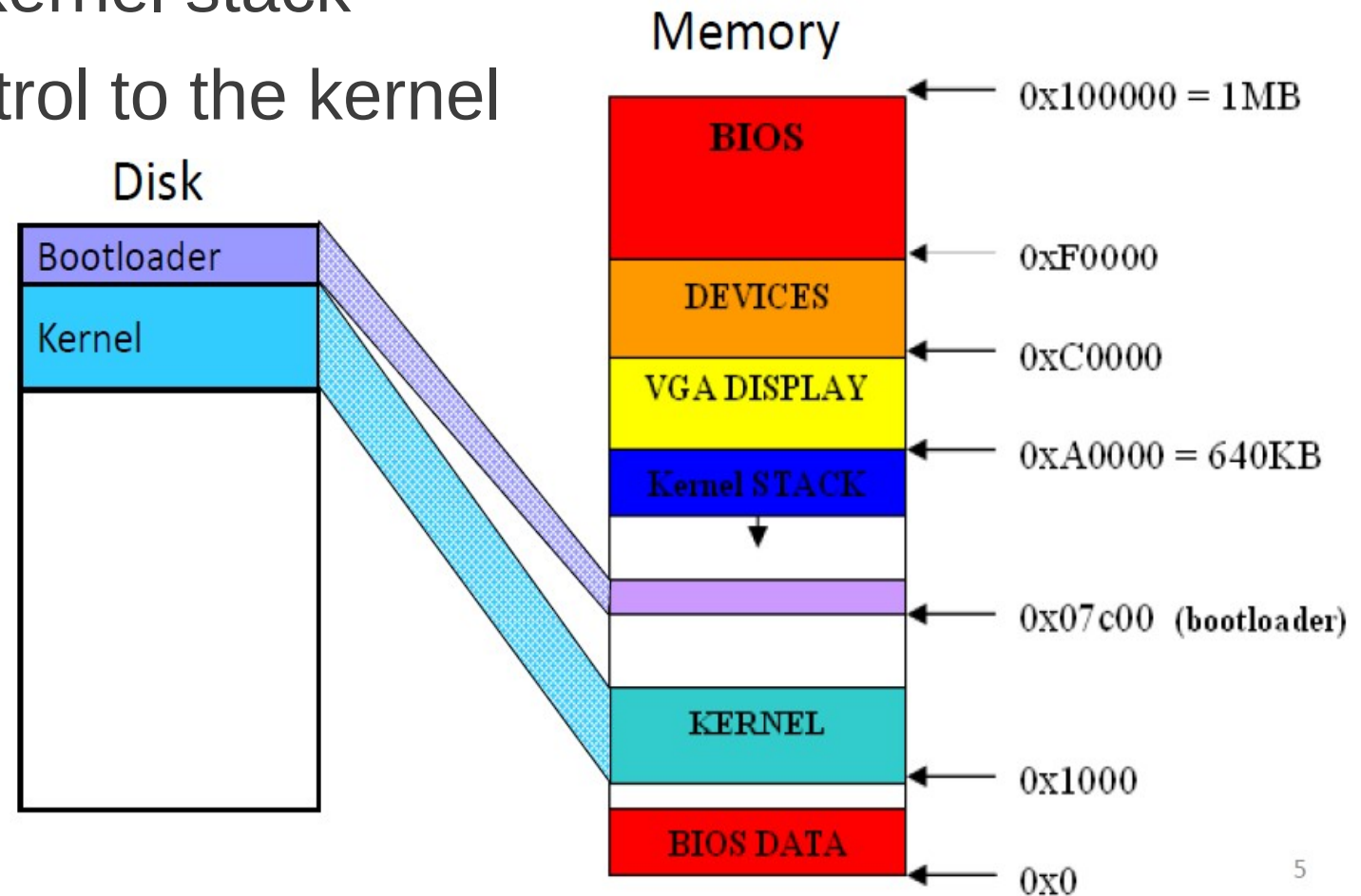
- Found bootable storage volume:
 - HDD, USB, Floppy
 - Load bootloader
- How is this done?
 - Load first sector (512 bytes)
 - Memory location: 0x7c00
 - Switch control to this location to launch the bootloader





The Bootloader

- 3 tasks:
 - Load the kernel into memory
 - Setup the kernel stack
 - Switch control to the kernel





Let's Review Assembly

- About numbers, need good bookkeeping
- Move data, perform simple arithmetic
- Need a lot of steps to do useful things
- KEY:
 - Understand memory addresses
 - Know where things are in memory



Memory Addressing

- 1MB of memory
 - Valid address range: 0x00000 - 0xFFFFF
- Real mode segmented model:
 - See full 1MB with 20-bit addresses
 - 16-bit segments and 16-bit offsets
- Addressing format: segment:offset
 - Actual address = $16 * \text{segment} + \text{offset}$
 - How would you write the address for the bootloader?



Registers

- 5 types of CPU registers:
 - General purpose: ax, bx, cx, dx (can address high or low-order byte via ah/al etc.)
 - Segment: cs, ds, es, ss
 - Pointer: ip, bp, sp
 - Index: di, si
 - Flags: df, zf (only 9 bits used)
- 32-bit registers have e prefix: e.g. eax



AT&T Syntax

- Prefix register names with % (e.g. %ax)
- Instruction format: instr src, dest
 - e.g. movw %ax, %bx
- Prefix constants, immediate values with \$
 - e.g. movw \$0x01, %ax
- Suffix instructions with size of data
 - b for byte, w for word (16 bits), l for long (32 bits)
 - **Keep the size of your registers in mind!**



Important Instructions

- **mov x, y**: moves data into a register
 - e.g. `movw %ax, %ds`
- **Jumps**:
 - **jmp imm**: `%ip ← imm`
 - e.g. `jmp $print_char`
 - **ljmp imm1, imm2**: `%cs ← imm1, %ip ← imm2`
 - e.g. `ljmp $0x7c0:0x00, $0x00`



Important Instructions

- Stack ops:
 - **push x**: $\%sp--$, $\text{Mem}[\%ss:\%sp] \leftarrow x$
 - **pop x**: $x \leftarrow \text{Mem}[\%ss:\%sp]$, $\%sp++$
- Function calls:
 - **call <label>**: push $\%ip$, jmp <label>
 - **ret**: pop $\%ip$
 - **Be careful not to override register values!**



Important Instructions

- Interrupts:
 - **int imm**: invoke a software interrupt
 - int 0x10 (console output)
 - int 0x13 (disk I/O)
 - int 0x16 (keyboard input)
 - Each interrupt offers several functions and parameters
 - Function indicated in %ah
 - Params in other regs

Assembly Program Structure



- Assembler directives:
 - Not instructions
 - Segment the program
- `.text` begins code segment
- `.globl` defines a list of symbols as global
- `.data` begins data segment
- `.equ` defines a constant (like `#define`)
 - e.g. `.equ ZERO, $0x00`
- `.byte`, `.word`, `.asciz` reserve space in memory



Read from Disk to Memory

- BIOS int 0x13, function 2:
 - Read disk sectors into memory
 - Parameters:
 - %ah = \$0x02 (disk read function)
 - %al = # of sectors to read
 - %ch = cylinder number
 - %cl = sector number
 - %dh = head number
 - %dl = drive number (already set)
 - %es:%bx address into which we want to read the data
 - Finally call the interrupt: int \$0x13
 - Refer to <http://en.wikipedia.org/wiki/Cylinder-head-sector> for more info



Design Review: Print Chars

- Refer to `assembly_example.s`
- BIOS int 0x10, function 14
 - Parameters:
 - `%ah = $0x0e` (disk read function)
 - `%al = character to be printed`
 - `%bh = active page number (use 0x00)`
 - `%bl = foreground color (use 0x02)`
 - call the interrupt: `int $0x10`



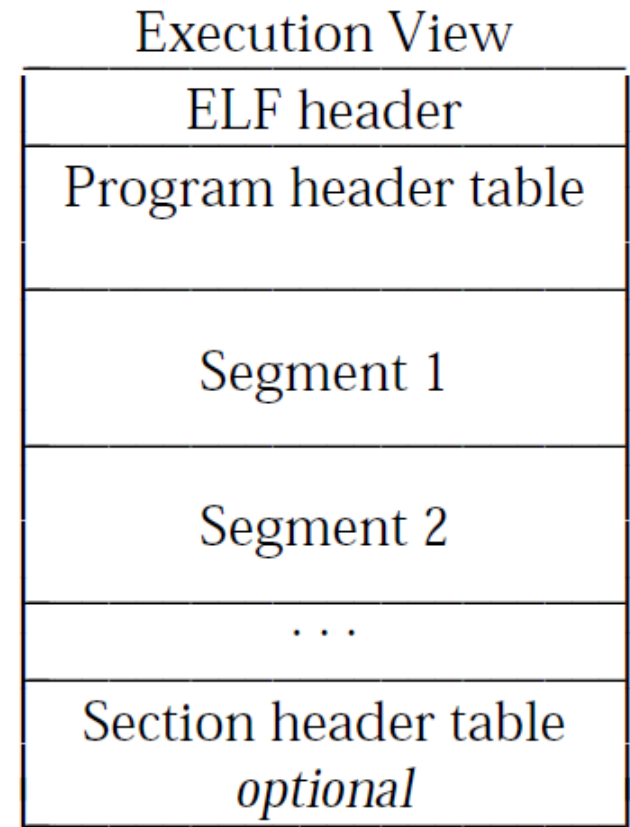
ELF Format

- Executable and linking format
- Created by assembler and link editor
- Object file: binary representation of programs intended to execute directly on a processor
- Support various processors/architectures:
 - Represent some control data in a machine-independent format



ELF Object File format

- Header (p. 9/10):
 - Beginning of file
 - Roadmap, file organization
- Program header table (p.33):
 - Array, each element describes a segment
 - Tells system how to create the process image
 - Files used to create an executable program must have a Phdr



p. 7 in ELF manual



Warm-up Exercise

- Hello World (hello-world.s)
- Download start code from Precept 1 webpage
- Specifications:
 - Define length of string using `.equ`
 - function `0x04` in `%eax` to indicate write
 - Value of `stdout` (`0x01`) in `%ebx`
 - Specify address of string in `%ecx`
 - Specify length of string in `%edx`
 - Invoke system call (`int 0x80`)
 - Define string using `.asciz`