Lab 1 - Bootloader

Operating Systems, CS-UH 3010

Assigned:	Sep 1, 2022	Due:	Sep 15, 2022
Precept:	TBD	Design Review:	*(Coordinate with Miro)*

The start code is linked from the course book under source files.

1 Overview

This is the first of several course projects originally developed by Kai Li at Princeton and his colleagues at the University of Tromsø. The assignments have also been slightly modified by James Aspnes at Yale University.

Over the course of these projects, you are going to develop a simple operating system, which will cover most of the topics discussed throughout the course. In this first project, you will build the basic environment that will be needed to develop the operating system for the rest of the semester. Specifically, your job is to implement two programs: bootblock.s and creatimage.c.

The bootblock resides on the first sector of the booting device (USB flash disk for this project) and is automatically loaded and executed at system booting process. It is responsible for loading the rest of the operating system from the booting device into memory. Because GCC, the compiler used to compile the bootblock and kernel source, generates executable files in a specific format which needs operating system support (of course not available before the operating system itself is loaded) to run, we need the createimage tool to transform and pack them up into an image file that can be directly loaded and run on a bare machine. This file is called an image because it is loaded into the memory exactly as what it is like on a disk.

2 Background

To complete this project, you will need to learn about:

1. IA32 (a.k.a. x86) architecture and assembly language.
   - Brief & useful notes on IA-32 Assembly Programming

- Tutorial Slides on Assembly Language
- Read Section 10 below.
- Official IA32 Intel architecture software developer's manuals. (Use for specific instruction reference)

2. The booting process
   - The Class Lecture on Bootloading.
   - Information on MBR
3. BIOS functions needed to display messages and load disk data.
   - Read Section 10 below.
4. The ELF binary format.
   - Read the ELF Documentation
   - `man ELF`

3 The Start Code

The start code contains the following:

- man/: Manual for createimage. Invoke `man -M man createimage`
- bochsrc: Config for bochs
- bootblock.s: A template for you to write the bootloader
- bootblock_examples.s: A series of assembly language examples
- createimage.c: A template for you to write a Linux tool to create a bootable operating system image
- createimage.given: An executable Linux tool for you to test your bootblock.s code and validate your createimage.c
- kernel.s: A minimal kernel to test your bootup code and your tool. Larger kernels used for the extra credit will be provided later.
- Makefile

You are expected to only modify bootblock.s & createimage.c

4 Design Review

You should be prepared to demonstrate a stub bootloader that does something useful beyond the initial bootloader stub provided to you (e.g. prints a friendly message).

The intent of the review is not to force you to have fully-functional code or even a perfect design at this point; instead, we want to make sure that you are on the right track and thinking about the project well before the final deadline. The design review accounts for roughly 20% of the project grade.

You should be able to describe:

1. How to move the kernel from disk to memory?
   - Where to find it on disk?

   - Where to copy it in memory?
   - How to do the copying in assembly?
   - In what case must your bootloader be relocated?

2. How to create the disk image?
   - Given an executable (ELF), show how to use the header info to find where the first code segment begins.
   - Show how to determine where in the image file this segment must be placed.
   - Where to write padding?

5 Submission

Submit the code at `https://www.dropbox.com/request/89Wdp1n7FlPwJYi97qdg` as a single .zip folder with the following filename template: LAB1-netId1-netId2-netId3.zip. Make sure to include inside README.md with some explanation about the code, the decisions, and the weakness of your solution, etc.

6 Building

A Makefile is provided for you. Invoke `make` to compile everything a generate the image file. Invoke `make clean` to remove the files generated by the last make. The Makefile initially uses createimage.given but you want to change it to createimage for the second half of the project to compile and test createimage.c

7 bootblock

When compiled, bootblock.s, will be written to the first sector of your boot device (USB, an image, etc.). You must write it in IA32 assembly language and it cannot exceed 512 bytes (the size of one disk sector). Its function will be to:

1. load the kernel
2. set up the stack for the kernel
3. invoke the kernel

You are required to handle kernels of size up to 128 sectors. Reading up to 128 sectors can be done with one BIOS call, but you must beware that you cannot cross a physical segment (every 64K starting at address 0) in one read. In bochs, reading more than 64 sectors will fail if your image is used as a floppy disk instead of a hard disk.

When a PC boots the image/disk that you have prepared and the system has been initialized, the first sector of the boot disk (the bootloader) is loaded at address 0x07c0:0000 (0x7c00 in real addressing mode) and control jumps to that address.

Your bootloader can safely modify memory in the range [0x0a00, 0xa0000) without having to worry about overwriting video memory, the interrupt vector table, or BIOS. The kernel should be loaded at address 0x0000:1000.

You may assume that the entire kernel is no more than 128 sectors long. Notice that a loaded kernel may overlap with the bootloader, so you will have to relocate the bootloader to a higher address that can't be overwritten before loading the kernel.

In order to load sectors, you must know the boot device number, which is stored in %dl. Common values are 0x0 for a floppy drive, 0x80 for the first hard drive, 0x81 for the second, etc. You should save this value for later use.

We have provided createimage.given which is a linux-compiled binary version of createimage so that you may test your bootloader independently of the next half of the project.

8 Createimage

A linux tool to combine the bootloader and kernel, and any number of programs in ELF format, into a bootable image file. Additionally, this tool must somehow let the bootloader know how many sectors to read in order to fully load the kernel. When a program is compiled in linux, an ELF executable file is generated (by default) that contains the program code and information telling the OS how to load the code into memory and what resources (dynamically-linked libraries, etc.) are required. Such an executable may contain multiple fragments of code, each expecting to be loaded to a particular offset in memory – as specified in the ELF header. When the OS loads the executable file, it will copy each code fragment to the correct offset in memory.

However, when booting a computer, there is no OS to load the bootloader or kernel. Thus, the bootloader's and kernel's code must be extracted from the generated ELF executables and carefully laid out in the image file as if it were memory. There should be no other required resources specified in the ELF executables because there will be no OS to prepare them when the code is loaded (this requirement is reflected in the compiler flags: `-nostartfiles -nostdlib -fno-builtin` which have been specified in the given Makefile).

Because the BIOS will load the bootloader to offset 0 (of segment 0x07c0), the compiler must be told that the starting address is 0 using the -Ttext flag. Similarly, the kernel's starting offset is set to 0x1000. Thus, when the BIOS loads the first sector of the image to offset 0, the bootloader will be ready to run. When the bootloader simply copies the kernel to 0x1000, it too, will be ready to run.

You are also required to implement the `--extended` option. In particular, when given this option, create image is expected to display the number of sectors used by the image, the specific sector numbers on the disk which will contain the image, the segments specified in the ELF headers, and the kernel size (os size) in the unit of sector so that you can have a sense of whether the bootloader will need to relocate itself to accommodate a large kernel.

Note:

1. The filesz and memsz properties of a segment in an executable may differ. This means that when writing the segment to memory (or writing to an image), you may have to add padding so that the segment occupies all of memsz
2. Two adjacent segments in the executable file may not be contiguous when loaded into memory. The simplest method to handle this case is to add padding between the segments, so that the two segments seem to be one contiguous segment.
3. A segment in a large kernel might occupy offset 0x7c00, which would overlap with the bootloader. (0x0000:0x7c00 is equivalent to 0x07c0:0x0000). Your bootloader will have to copy itself to a higher address that cannot be overwritten when it eventually loads the kernel.

For specific information on the functionality of createimage, refer to the man page provided in the code template by invoking `man -M man createimage`. You should ignore the '-vm' option for this project.

Useful Tools

```
readelf
objdump
od
diff
```

9 Testing

To test the image with Bochs:

Run `bochs` or `~/318/bin/bochsdbg` from the same directory as the given bochsrc and the generated image

Bochs will stop at a prompt if you run bochsdbg. Type c to continue execution. You'll probably want to set a breakpoint at your bootloader using b 0x7c00 before continuing. Take a look at some other debugger commands.

Bochs will create a logfile called bochsout [.txt] which may become quite large (up to 6 GB!). Delete this file when you're done so that you don't use up all the disk space!

9.1 Testing Extra Credit

Determine how to connect a USB disk to Virtual Box. You may need to install Oracle VM Extension to get this to work.

Locate your USB disk at /dev/sdf. In general, it may be accessible at /dev/sda, /dev/sdb, /dev/sdc, etc. Make sure you have identified the correct device before writing the image, otherwise, damage may be done to other devices on the machine.

When you are ready to test your image:

1. Mark the image so that the BIOS recognizes it as bootable. Write a signature consisting of two bytes, 0x55 0xaa, to the end of the first sector: address 0x1fe of image. Don't forget to do this in createimage.c
2. Copy it to your USB disk using `cat image > /dev/sdf`, assuming /dev/sdf is the USB disk you intend to boot from. You may also use `dd if=image of=/dev/sdf`.
3. Next, connect the USB disk to the test computer (e.g. a virtual machine or a simulator), restart, and boot off the USB disk. Don't try on your personal computer unless you are willing to bare the consequences!

10 Assembly Tips, Tricks, and Code

Tips

- Do not optimize prematurely (i.e., before it works). It's fine to store all your variables on the heap instead of trying to use only registers (all the way until it must be faster).
- If it works in bochs but not on a real machine, then there is a bug in your program. In bochs, all memory is initially 0; this is not true on all computers. Registers, upon entering the bootloader, may not be the same. Even %dl (the boot device) may be different.
- The stack, upon entering your bootloader, might be good (the BIOS was just using it), but make sure you don't corrupt it unintentionally.
- The int instruction does, in fact, use the stack.
- rep mobsb is a quick and simply way (of many) to move data about
- rep scasb can be used to implement strlen

Examples

The file bootblock_examples.s also contains several x86 assembly language examples.

Loading a segment:offset

The project will require you to load a number into a segment register to setup the stack and data segments. The code segment register (CS) cannot be loaded directly, but instead only indirectly through a JMP type instruction. When loading a value into the stack segment register (SS), interrupts are disabled for the next instruction, thus allowing you to set the stack pointer (SP). As an example of setting up the segment registers for data, consider the following string copy:

```
# Setup the registers - see chapter 3 of Intel ISA reference volume 2
movw DATA_SEGMENT, %ax
movw %ax, %ds
movw OTHER_DATA_SEGMENT, %ax
movw %ax, %es
movw STRING_FROM_ADDRESS, %si
movw STRING_TO_ADDRESS, %di

# Move a byte from one string to the other - implictly DS:SI to ES:DI
movsb
```

The values in %si and %di are automatically incremented/decremented based on the DF flag.

Display Memory

For your design review you are required to implement routines that print to the screen. During booting, you can write directly to the screen by writing to the display RAM which is mapped starting at 0xb800:0000. Each location on the screen requires two bytes – one to specify the attribute (Use 0x07) and the second for the character itself. The text screen is 80x25 characters. So, to write to i-th row and j-th column, you write the 2 bytes starting at offset $((i-1)*80+(j-1))*2$.

So, the following code sequence writes the character 'K' (ascii 0x4b) to the top left corner of the screen.

```
movw $0xb800,%bx
movw %bx,%es
movw $0x074b,%es:(0x0)
```

This code sequence is very useful for debugging.

Useful BIOS Features

You are allowed to use these BIOS functions to complete this assignment.

BIOS Int 0x13 Function 2 Reads a number of 512 bytes diskette sectors starting from a specified location. The data read is placed into RAM at the location specified by ES:BX. The buffer must be sufficiently large to hold the data AND must not cross a 64K linear address boundary. (For our project, for simplicity, you can assume cylinder number bits 8&9, i.e. bits 6&7 of cl is zero).

Called with:

```
ah = 2
al = number of sectors to read,
ch = cylinder number, (lowest 8 bits of the 10-bit cylinder number,
0 based)
cl, bits 6\&7 = cylinder number bits 8 and 9.
bits 0-5 = starting sector number, 1 to 63
dh = starting head number, 0 to 255
dl = drive number. Use the value of dl that you stored at the
beginning of the bootloader.
es:bx = pointer where to place information read from diskette
```

Returns:

```
ah = return status (0 if successful)
al = burst error length if ah = 11; undefined on most BIOSes for other ah return status values.
carry = 0 successful, = 1 if error occurred
```

BIOS Int 0x10 Function 0x0e This function sends a character to the display at the current cursor position on the active display page. As necessary, it automatically wraps lines, scrolls and interprets some control characters for specific actions. Note: Linefeed is 0x0a and carriage return is 0x0d

Called with:

```
ah = 0x0e
al = character to write
bh = active page number (Use 0x00)
bl = foreground color (graphics mode only) (Use 0x02)
```

Returns:

```
character displayed
```

11 Other Useful References

- 80386 Programmer's Reference Manual
- PC Assembly Language's tutorial (uses Intel format, not AT&T)
- as (GNU Assembler)
- Official IA32 Intel architecture software developer's manuals. (Use for specific instruction reference)
- Bochs Tutorial.
- CHS addressing

12 Grading Rubric

This project is worth a total of 10 points, plus 1 point extra credit.

	Deliverables	**Points**
1	Attend design review and are prepared	2
2	Both bootblock.s and createimage.c compile with no errors or warnings	2
3	The bootloader must be properly relocated and control transferred to the kernel with CS=0x0000, IP=0x1000, and DS=0x0000	1
4	The bootloader must initialize sufficient space for the kernel stack. Sufficient means that if the kernel issues 256 pushw instructions, the machine should not crash	2
5	Output of createimage.c and createimage.given should match exactly. All padding must consist of zero bytes and files need to be padded properly (both before each program segment and at the end)	2
6	When createimage is given the –extended flag, it must output additional information from the ELF file. Specially, it is required to display the entry offset (from the ELF Header), and must display the Offset, Virtual Address, File Size and Memory Size fields from each ELF Program Header, i.e. roughly in the same format of the printing output of createimage.given	1
7	You can demonstrate that your bootloader boots off a USB disk	$+1$
8	Obfuscated, undocumented and messy code will be penalized	-1