Boot-up Mechanism

Alexander Einshoj

February 3, 2016

1 Introduction

This paper describes the creation of a bootable medium (USB disk) that loads a small OS kernel.

2 Technical Background

There are a lot of technical areas to understand in order to fully appreciate what lies behind completing the task in question.

2.1 POST

The Power-On Self-Test is a process performed immediately after the computer is powered on. The POST routines are part of the device's pre-boot sequence, and the bootstrap loader code is invoked only after the POST is completed successfully. It initializes and tests hardware. It looks for available RAM, detects CPU type and speed, verifies cache memory and CPU registers. It detects keyboard, as well as testing CMOS.
It detects adapters, such as video adapters and video memory. After POST, the BIOS will search for a bootable medium.

2.2 BIOS

The Basic Input-Output System is the motherboard's firmware. It was created to offer generalized low-level services to early PC system programmers. It is stored on a non-volatile ROM chip, which means that it contains a battery to save data even when the computer is turned off. The ROM chip(CMOS/EEPROM) stores the boot sequence, which devices to test in what order. The BIOS is responsible for detecting and initializing hardware, and provides a service interface for interacting with hardware. These services are often named BIOS functions, and can be accessed through software interrupts in Real Mode, using Assembly. BIOS will look for a valid boot sector(512 bytes at a known location with magic signature). If the BIOS is successful in finding a boot sector, it copies it into memory and executes it.

2.3 INT0x13

INT0x13 or INT 13H is a BIOS interrupt call in an x86-based computer system. It is one of the BIOS interrupt functions mentioned above, and is responsible for reading floppy/hard disk in CHS mode.

2.4 CHS

CHS, or Cylinder-head-sector is a method for giving addresses to each physical block of data on a hard disk drive. CHS values used to be directly in relation to the data stored on the disks, but no longer. Instead, virtual CHS values are used, which are translated by disk or software to LBA values. CHS addressing is the process of identifying individual sectors on a disk by their position in a track, where the track is determined by the head and cylinder numbers. The most typical sector size on todays hard disks is 512 bytes.. With CHS addressing, the lowest sector number is 1(as opposed to 0), while tracks and cylinders are counted from 0. Tracks are basically strips of sectors. At least one head is required to read a single track. The amount of cylinders of a disk drive is the same as the number of tracks on a surface in the drive. Cylinders are vertically formed by tracks.

2.5 Boot block

A boot block is a part of a data storage device that contains machine code to be loaded into RAM. The boot block is responsible for initializing registers, before copying the sectors with the kernel code from the boot device to memory. When this is done, it transfers control to the kernel code. The boot block is the first sector, and occupies exactly one sector. The kernel starts at the second sector. The boot sector(boot block) is copied by BIOS from the boot device to memory address 0x07c00. The boot block has to be written in assembly, and operate in 16-bit Real Mode.

2.6 16-bit Real Mode

Real Mode is a simplistic 16-bit mode that exists in all x86 processors. To ensure compability, all x86 processors begin their execution in Real Mode. Even though Real Mode is a 16-bit mode, the 32-bit GPRs are still accessible. There is about 1MB of addressable memory in Real Mode, whereas the usable amount will be less than 1MB. In Real Mode, we have a 20-bit linear addressing space, and a 16-bit logical addressing space.

2.6.1 Memory segmentation

Segmentation means that addressing is done within a segment. A segment is a limited address subspace, and logical memory addressing is relative to the linear base address of the segment instead of relative to 0x0. A Real Mode segment is defined by a segment selector, which is stored in a segment register. Segment base address = selector << 4 Logical memory addresses are 16-bit positive offsets to the base address of a segment. Linear addresses are corresponding with physical addresses. There are six segment registers, with different purposes of use. It is vital to keep in mind that segment selectors can not be set directly, their value must first be set to a register.

2.7 ELF

Executable and Linkable Format is an object file format, divided into three main categories: A relocatable file, an executable file and a shared object file, which all serve different purposes. Object files, which are created by the assembler and link editor, are binary representations of programs intended to run directly on a computer. An ELF file contains raw instruction code and data segments, address of where in RAM each segment should be loaded, metadata, and other information. In order to complete this assignment, extensive knowledge about the ELF format and its features were required.

To parse an ELF document, you need to know what information you need and where it resides within the structure. In the beginning of an ELF file, you will find an ELF header, which can be described as a ”road map” to the file's organization. This header is followed by a program header table or a section header table, or both. The program header table and the section header table's offset in the file are defined in the ELF header. Those two tables describe the rest of the particularities of the file.

The ELF header is a struct of type ELFN(N = 32 or 64)_Ehdr, which contains information about the headers described above, and size of program header entries, size of the ELF header in bytes,number of entries in a program header, and the same for section header, in addition to other potentially useful data.

The ELF program header contains a lot of information too, such as offset from beginning of file to the first byte of the first segment(also for where it resides in memory). It tells us the number of bytes in the file image(also memory image) of the segment.

3 Design

In order to complete the implementation of the bootblock, very few lines of code is required as a minimum. You need to setup data segment for the boot block(code segment is already set). While doing this, it's important to remember that the value of the data segment can not be set directly, it must pass through a GPR first. Then, you have to setup stack segment and stack. The kernel needs to be copied to 0x01000, data segment needs to be set for the kernel as well, in the same matter as is the case for boot block data segment, through another register.

In this assignment, an important thing to think of, is setting the INT 0x13 parameters. INT 0x13 is a BIOS interrupt call that lets us read from disk, and it's parameters has to be set. The CHS values are basically the necessary values to be set, in addition to amount of sectors to read from kernel.

The boot block is not very flexible, as it should be exactly 1 sector large (512 bytes), and only the basic library functions are available. Then again, it's job isn't too complex either, which means it's work can be done with very limited space available.

The createimage has a lot more flexibility as it is written in C with all libraries available. The goal of createimage is to parse the ELF executable and read it into memory before writing it to an imagefile. At this point, knowledge about the ELF executable header and program header structures is everything. My solution is divided into multiple small functions that each serve a significant purpose towards the creation of the image file.

The code automatically becomes a lot less messy and hard to navigate through with the use of functions. It also makes debugging easier, and altogether a better understanding of what you're really doing. Something that's of significant importance, but not yet mentioned due to it's simplicity, is padding. Padding is very important for the code to work, and is needed both to make the boot sector exactly 512 bytes, and to make the kernel size dividable by 512 (because an entire sector is read at a time).

The functions ftell(), fseek() and fread() were taken extensively use of during this assignment, in the process of padding.

As it looks now, the create image function looks very clean because it consists of nothing but function calls and loops/conditions. This way, it's easily readable and easy to fix mistakes without it having consequences for the entirity of the code.

4 Implementation

The bootblock was written in 16-bit Real Mode Assembly, while the createimage was written in C.

5 Discussion

I will start off this part of the paper by saying that the assignment itself is in my opinion a bit hard to actually know if one has completed or not, due to the requirement saying that we have to make a solution that works for an arbitrary number of N files, while we are only given the opportunity to test for one file.

I've tried to divide the code into multiple, logical functions that are as softcoded as possible, doing my best effort to support multiple files, although there I'd anticipate encountering some problems if that were the case. To calculate the kernel size in sectors, instead of doing what I did, I could have saved everything I wrote from the beginning of the kernel and out, either by physical address or just size in bytes, before padding and such.

Doing so would solve the problem that would arise if the kernel consisted of multiple files that did not follow each other directly in physical address, or if the second file should write over some of the padding from the first file, then knowing how much you have padded and written to the image is crucial for it to be correct.

6 Conclusion

All the requirements for the assignment are met, with the potential exception of supporting multiple files(this is not tested). In my opinion, the code is well commented and written in a way that seems logical and is easy to grasp when looking at.

7 References

[1] https://en.wikipedia.org/wiki/INT_13H
[2] http://www.skyfree.org/linux/references/ELF_Format.pdf
[3] http://wiki.osdev.org/Real_Mode
[4] https://en.wikipedia.org/wiki/Boot_sector
[5] https://en.wikipedia.org/wiki/X86_memory_segmentation
[6] https://en.wikipedia.org/wiki/Cylinder-head-sector
[7] http://wiki.osdev.org/BIOS
[8] https://en.wikipedia.org/wiki/Power-on_self-test
[9] Assignment text from Fronter