Project 4: Writing a Bootloader from Scratch

15–410 Operating Systems

April 23, 2007

1 Introduction

1.1 Overview

In this project, you (and your partner) will be writing a bootloader that is capable of booting your 410 kernel. In doing so, you will complete a full stack kernel+bootloader that is ideally capable of booting on commodity hardware (although such isn't a strict requirement of this project). Furthermore, by writing a bootloader, you will be exposed to a different style of systems programming in a slightly unusual environment. Fortunately there will be no concurrency, no paging, and no protection issues to worry about. Unfortunately, there is also no memory manager and no libc—the only support code available are functions provided by the BIOS, and the ones you write yourself.

Due to the constrained environment that the bootloader operates in, this project will heavily rely on services provided by the PC BIOS. Most of this project will involve low level programming in 16-bit real mode. As GCC does not emit real mode compliant code, most of this project will be programmed in assembly. However, portions of the bootloader will also be programmed in 32-bit protected mode with paging disabled, an environment that much resembles the operating environment of Project 1.

1.2 Goals

- To learn (a small amount of) programming in x86 real mode, including the use of PC BIOS services.
- To become familiar with reading sectors from a floppy disk, including the BIOS interface for interacting with the floppy disk controller as well as the floppy disk CHS (cylinder, head, sector) layout.
- To understand the process of bootstrapping on an x86, including switching into protected mode and loading a Multiboot compliant kernel.

1.3 Outline

The bootloader you will be writing will consist of two distinct stages (boot0 & boot1), both of which are responsible for accomplishing key steps required to load and boot your kernel:

- The bootloader's zeroth stage (boot0) is responsible for loading the first stage (boot1) from the boot floppy disk and transferring execution to it.
- The bootloader's first stage (boot1) is responsible for loading the kernel from the boot floppy disk, preparing the system environment for executing the kernel, and transferring execution to it.

Since accomplishing these tasks requires significant programming in x86 real mode, a brief introduction on real mode programming follows.

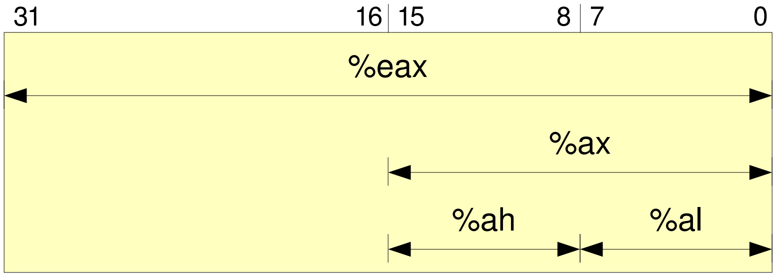
2 Programming in x86 Real Mode

The original IBM PC (model 5150) utilized an Intel 8088 CPU with a single operating "mode," which has been retained for software backwards compatibility in later processors. This primeval 8086-compatible operating mode is now known as real address mode (or real mode), and contrasts with protected mode which has been used thus far in this course.

All descendants of the original IBM PC boot in real mode, and it is often the duty of the bootloader to enter protected mode and transfer control over to the kernel. In addition, bootloaders often rely on services provided by the PC BIOS, most of which are available only while in real mode. Fortunately programming in real mode is similar to programming in protected mode without paging, except for a few key differences.

2.1 Default Operand Size

In order to emulate an 8086 processor with 16-bit registers, real mode instructions default to a 16-bit operand size. This means that all memory accesses use 16-bit offsets, and the %EIP and %ESP registers may contain only 16-bit values. Registers are generally referred to by their word-size counterparts (%AX, %BX, %CX, %DX, %SI, %DI, %BP, and %SP). Finally, some instructions have different semantics in real mode than in 32-bit protected mode (i.e., `PUSHA` which, in real mode, pushes only the lower 16 bits of each register on the stack). Here is an illustration of a 32-bit register (%EAX) with its word-size (16-bit) and byte-size (8-bit) counterparts:



It is possible to use 32-bit (register & immediate) operands while in real mode. While real mode code is typically written with 16-bit operands "by default," there is at least one place in the bootloader code where you may wish to use a 32-bit register for arithmetic computation. When using a 32-bit operand instruction (e.g., `ADDL %EAX, %EBX` instead of `ADDW %AX, %BX`) the assembler will properly emit 32-bit code.

While 32-bit register & immediate operands are allowed in real mode, 32-bit memory accesses are not allowed as each memory segment has a 64 kB limit, as is explained below.

2.2 Real Mode Segmentation

The 8086 processor was designed to support a 20-bit (1 MB, 1 megabyte) address space. Since the 8086 consists of only 16-bit registers, all memory accesses support only 16-bit operands. Thus, real mode uses segmentation in order to address the entire 20-bit address space.

Segmentation in real mode is quite unlike segmentation is protected mode. In real mode, the segment registers contain a segment base address that is shifted left by 4 bits and *added to* a 16-bit offset to form a 20-bit linear (and physical) address. This is perhaps best explained by example:

```
movw    $0x1234, %ax
movw    %ax, %ds

movw    $0x5678, %bx

# The following instruction is the same as "movw $0x1337, (%bx)".
movw    $0x1337, %ds:(%bx)  # Places 0x1337 into memory word 0x179b8.

# Segment Base: %ds << 4:   12340
# Offset:       %bx:      +  5678
#                         -------
# Linear Address:           179b8
```

and by illustration:

Segment Base (%ds):	1234
Offset (%bx):	+ 5678
Linear Address:	179B8

As an alternative to specifying the computed linear address, many texts refer to real mode memory addresses using a segment:offset pair notation. For example, the above linear address (0x179b8) can be written as the segment:offset pair 1234:5678.

As twelve bits of the segment base and the offset address overlap, it is possible to represent any (< 1 MB) linear address in real mode by many segment/offset combinations (e.g., 3000:1337 and 3123:0107 both equal 0x31337).

Since segment registers contain segment base addresses, and not protected mode segment selectors, real mode does not use the Local Descriptor Table nor the Global Descriptor Table.

2.3 Physical Memory Layout

The PC address space is primarily divided into two major sections. The first section is the lower 1 MB that corresponds to the entire 20-bit address space of the original IBM PC and is fully addressable in real mode. The second section is the memory above the first 1 MB known as "high" (or "extended") memory that was first featured on the IBM PC AT and is addressable only in protected mode.[1]

Since lower memory corresponds to the entire address space of the original IBM PC, lower memory itself is divided into subsections. Historically, the first 640 kB of lower memory was backed by RAM, and most of it was made available to the operating system and user programs.[2] In contrast, the last 384 kB of lower memory was not backed by RAM, but rather reserved for video memory, memory mapped I/O, and the BIOS ROM. In real mode, this reserved area appears at the top of the real mode address space. However in protected mode, the reserved area forms a memory hole between 640 kB and 1 MB that must not be disturbed (except for video memory) by either the bootloader or the operating system.

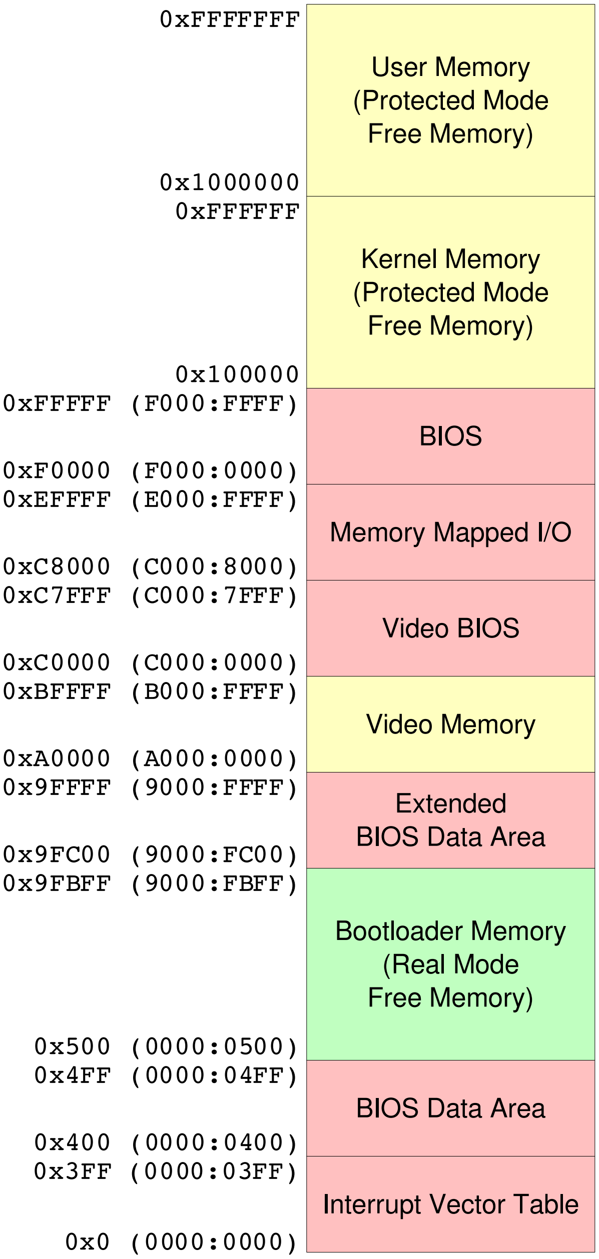
While the first 640 kB of lower memory is backed by RAM, the BIOS reserves the first 1.25 kB and the last 1 kB for its own use. Since the Pebbles kernel does not use any memory below 1 MB, the remainder of the unreserved lower memory space is available for use by your bootloader.

The following memory map summarizes the entire physical memory address space that Simics makes available to your bootloader and kernel. The ad-

[1] Actually, the first 64 kB of high memory is also addressable in real mode due to real mode's odd segmentation mechanism. This fact lead to an incompatibility with the original IBM PC which resulted in the A20 gate hack (see Section 6.6).

[2] "640k ought to be enough for anybody"—Bill Gates, 1981.

dress range marked “Bootloader Memory” corresponds to the memory available exclusively for your bootloader’s use.



Bootloader & kernel physical memory map.

2.4 Real Mode Interrupts

In protected mode, interrupts are identified by interrupt gate & task gate entries in an Interrupt Descriptor Table (IDT). In real mode, an Interrupt Vector Table (IVT) is used instead of the IDT. The IVT serves the same purpose as the IDT, but it uses a different format. As you will not be writing interrupt handlers for this project (they are already installed by the BIOS) it is not necessary to know the details of the IVT. However, you should be aware that the table is located at address 0x0 in memory. Thus, the first 1.25 kB of memory should be considered as reserved and not used by your bootloader.

3 Using BIOS Services

The Basic Input/Output System or BIOS consists of code stored in non-volatile memory on a PC motherboard. The BIOS is responsible for initially booting a PC from power-on or reset. In addition, the BIOS provides a number of system-call–like "interrupts" (traps) that are available to real mode programs. These services provide a hardware-independent way of accessing common PC hardware including the video hardware and disk controllers. Real mode operating systems such as DOS rely on these services in order to communicate with PC hardware. As most BIOS calls are not available in protected mode, protected mode operating systems often feature hardware detection and hardware drivers, and thus, rarely rely on BIOS calls to communicate with hardware. However, since bootloaders typically do not contain driver code (and usually can't due to size restrictions), bootloaders often make extensive use of BIOS calls in order to load a protected mode kernel.

BIOS calls are made by issuing an `INT` instruction similar to Pebbles system calls, but use only registers and condition flags (not an argument packet) to communicate arguments and return values. When issuing BIOS calls, %SS:%SP must point to a valid stack, and interrupts must be enabled.

4 Bootloader: Overview

4.1 boot0: Zeroth Stage

Recall from the Bootstrapping lecture that the BIOS loads a valid boot sector (which must be the first sector of the boot device) to linear address 0x7c00 and jumps to the instruction located there. As the boot sector contains only 500 bytes (approximately) of executable code, many bootloaders are divided into stages. The bootloader for this project will consist of two stages, boot0 and boot1.

The zeroth stage bootloader (boot0) resides in the boot sector of our boot floppy disk and is automatically loaded by the BIOS. Its sole purpose is to read the first stage bootloader (boot1) from the floppy disk, load it into memory, and jump to it.

4.2 boot1: First Stage

The purpose of the first stage bootloader is to load the kernel from the boot floppy disk into memory, prepare the system for executing the kernel, and jump to the kernel entry point.

In order to prepare the system for executing the kernel, the first stage must switch into protected mode and load the kernel into high ($>$ 1 MB) memory. Unfortunately, the BIOS services used to read sectors from the boot floppy disk are available only in real mode and real mode cannot access high memory where the kernel is to be loaded. Many bootloaders work around this problem by repeatedly switching back and forth between real mode and protected mode while reading the kernel from the floppy disk.

Fortunately, the kernels you are responsible for loading have a maximum size restriction. They can be no larger than 576 kB, which allows your bootloader to preload the kernel entirely into contiguous low memory ($<$ 640 kB) before transitioning into protected mode.

4.2.1 Multiboot Loader

Once in protected mode, boot1 invokes a Multiboot loader routine in order to load the preloaded kernel from low memory into its proper location in high memory. Since the Multiboot loader executes entirely in protected mode, it is recommended that it be written in C and not assembly language.

5 Bootloader: Step-by-step

This section presents a step-by-step outline of the bootloader stages and their respective tasks. The details of each task is covered in Section 6.

5.1 boot0

boot0 minimally must accomplish the following tasks:

1. Disable interrupts.
2. Canonicalize %CS:%EIP (see Section 6.1 below).
3. Load segment registers (%DS, %ES, %FS, %GS, %SS).
4. Set the stack pointer.
5. Enable interrupts.
6. Reset the floppy disk controller.
7. Read boot1 sectors from the floppy.
8. Jump to boot1 code.

Should any of the above steps fail, boot0 should fail as follows:

1. Notify the user of a failure condition.
2. Disable interrupts.
3. Permanently suspend progress in execution.

5.2 boot1

boot1 minimally must accomplish the following tasks:

1. Set the stack pointer.
2. Query the BIOS for the size of lower memory.
3. Query the BIOS for the size of upper memory.
4. Read kernel sectors from the floppy into lower memory.
5. Enable the A20 gate.
6. Disable interrupts.
7. Load the Global Descriptor Table.
8. Switch to protected mode.
9. Invoke the Multiboot loader.
10. Begin execution of the kernel.

Should any of the above steps fail, boot1 should fail as follows:

1. Notifying the user of a failure condition.
2. Disable interrupts (if not already disabled).
3. Permanently suspend progress in execution.

5.3 Multiboot Loader

The Multiboot loader must accomplish the following tasks:

1. Locate the Multiboot header in the preloaded kernel image.
2. Verify the Multiboot header & flags.
3. Load the kernel image into high memory.
4. Write the Multiboot information structure.
5. Return success or failure to boot1.

6 Bootloader: Details

6.1 Canonicalizing %CS:%EIP

As previously stated, the BIOS loads the boot sector of the boot disk at linear address 0x7c00. In real mode, this is logically and most often represented as the segment:offset pair 0000:7c00. However, certain odd BIOSes actually begin execution at 07c0:0000. To deal with this discrepancy, the first task of any bootloader is to canonicalize %CS:%EIP to a known segment:offset pair that the rest of the code depends on.

Simultaneously setting %CS:%EIP is accomplished with an absolute long jump instruction to a label that represents the next line of code. A typical example looks like:

```
        ljmp    $0, $the_next_line_of_code
the_next_line_of_code:
        # Code ...
```

6.2 Loading Segment Registers

The recommended memory map (see Section 8) places all code and data, except the preloaded kernel, in the first 64 kB of memory. By using only the first 64 kB as the primary working area, the recommended memory map avoids the use of "long" jumps and "far" pointers[3] in bootloader code. Thus, all segment registers should be initially set to the segment base address representing the first 64 kB of memory.

Since the kernel you are loading is much larger than 64 kB, you will have to write kernel data to memory in multiple segments outside the first 64 kB segment. When doing so, it is suggested to use %ES to refer to the segment where you are currently loading a portion of the kernel, and %DS to refer to the first 64 kB segment. This way, memory accesses default (using %DS) to memory in the primary working area, while %ES may be moved around freely as needed.

```
movw   myvar, %ax     # fetch myvar from somewhere in %ds (low 64k)
movw   %ax, %es:(%bx) # store the value up in high memory
```

6.3 Interacting with the Floppy Disk Controller

Floppy disk services are made available by the BIOS interrupt 0x13, which is documented in Section 3.4 of [5]. All of the BIOS functions corresponding to INT 0x13 require register %DL to contain the drive number of the boot floppy disk. When the BIOS launches boot0, %DL is already initialized to the drive number (0x00 for the first floppy disk drive, 0x01 for the second, 0x80 for the first hard disk drive, etc.). Thus, %DL will contain the correct value so long

[3] "long" jumps and "far" pointers are jumps/pointers to a different segment.

as registers %DL or %DX aren't modified. If registers %DL and %DX are to be modified, it is the bootloader's responsibility to record and restore the boot drive number appropriately.

Before interacting with the floppy disk drive the first time, the floppy disk controller must be reset to a known state using the BIOS call: INT 0x13, %AH=0x00.

Once the controller is reset, sectors may be read from the floppy disk using the BIOS call: INT 0x13, %AH=0x02. Sectors are addressed on the floppy disk using CHS (cylinder, head, sector) addressing. For example, the boot sector is the first sector of the floppy disk with CHS address (0, 0, 1).

While one INT 0x13 call can read multiple sectors into memory, a single read call is not permitted to cross a cylinder or head boundary. Thus, a single INT 0x13 call can read at most 18 sectors (a full track) for a 1.44 MB floppy disk (see table in Section 7). Furthermore, it is an unspoken rule that no transfer can cross a 64 kB boundary in memory. Thus, when reading more than 64 kB (128 sectors) in total you may want to read a smaller number of sectors at a time into a disk buffer, then copy the contents of the disk buffer to its proper place in memory.

6.4 Establishing the boot1 Execution Environment

To allow any compliant boot0 code to execute any compliant boot1 code, we require a strict execution environment to be established before jumping to boot1 code:

- boot1 must be loaded at the address specified in the memory map (see Section 8).
- All segment registers must be set to the segment base address representing the first 64 kB of memory.
- Register %DL must contain the boot drive number.
- Interrupts must be enabled.

Note that this execution environment does not place any restrictions on the value of the stack pointer (although %SP must point to a valid stack). Thus, it is the responsibility of boot1 code to set its own stack pointer.

6.5 Querying the BIOS for the Size of Lower/Upper Memory

Due to the hole that exists in the upper portion of the first megabyte of memory (video memory, MMIO, BIOS, etc), PC physical memory is typically divided into two usable regions. The lower region of memory typically ends around 640 kB, the actual value of which must be determined by the BIOS call: INT 0x12 (see Section 3.3 of [5]). Since most of low memory is required for preloading the kernel, your bootloader may die if the low memory size is less than 636 kB.

The upper region of memory starts at 1 MB and extends for the amount of system DRAM (for sub 4 GB machines). The actual size of upper memory is most easily determined by the BIOS call: INT 0x15, %AX=0xe801 (documented in [1]).

6.6 Enabling the A20 Gate

The original 8086 processor featured a 20-bit address bus capable of addressing 1 MB of memory. However, the odd segmentation mechanism of the 8086 allowed, theoretically, slightly more than the first megabyte of memory to be addressed (e.g., ffff:ffff == 0x10ffef). On the 8086 these address overflows would wrap back to the beginning of memory (e.g., ffff:ffff would wrap to 0x00ffef), a "feature" that many 8086 programs depended on. Thus, when IBM designed the PC AT with a 80286 processor featuring a 24-bit address bus, many 8086 programs would fail to execute properly.

To maintain 8086 compatibility, IBM added an AND gate to the 21st address line (A20) that was signaled by an extra pin on the PC AT's keyboard controller. Unfortunately, this gate persists today on PC hardware and must be enabled in order to allow protected mode programs to see all of memory beyond the first 1 MB.

There exists three ways to enable the A20 gate, (i) via the keyboard controller, (ii) via System Control Port A, and (iii) with the BIOS call: INT 0x15, %AH=0x24. Unfortunately none of the three methods are supported on all hardware, although most "modern" hardware supports a combination of two or three of them. The most reliable method for enabling A20 is via the BIOS call (which implements the proper hardware specific method internally). Unfortunately the Simics BIOS appears not to support this call, so the bootloader must fall back to one of the alternate methods should the BIOS call fail. Sample code and documentation illustrating the three methods of enabling A20 is available at [2].

6.7 Loading the Global Descriptor Table

As discussed in lecture and in the segmentation guide, protected mode execution requires the segment registers to refer to a segment descriptors in the Global Descriptor Table (GDT). For your kernel, OSKit loads a GDT as part of its initialization phase. However, since the bootloader executes before OSKit, your bootloader must load a "temporary" GDT before it can switch to protected mode. The procedure for loading the GDT is completed in two steps:

1. Create an eight-byte aligned GDT with three segment descriptors (see Section 3.4.3 of [6] for the GDT segment descriptor format):

   - A null segment descriptor (required).
   - A 4 GB, 32-bit, DPL 0, "non-conforming" type, code segment descriptor.

- A 4 GB, 32-bit, DPL 0, "expand-up" type, data segment descriptor.

2. Load the Global Descriptor Table Register (GDTR) with the LGDT instruction. The GDTR contains the size (in bytes) of the GDT, and the GDT linear base address (see Section 2.4 of [6] for the GDTR format).

6.8 Switching to Protected Mode

The procedure for switching to protected mode is explained in detail in Section 8.9 of [6]. Once the GDTR is loaded, the procedure is as follows:

1. Set the Protection Enable bit in %CR0.

2. Immediately long jump to the next line of code, specifying the code segment selector corresponding to the code segment descriptor loaded in the temporary GDT. This step is similar to, but different from, what you did in Section 6.1.

3. Load each of the segment registers (%DS, %ES, %FS, %GS, %SS) with the data segment selector corresponding to the data segment descriptor loaded in the temporary GDT.

4. Set the stack pointer to the protected mode stack.

As the bootloader does not establish a protected mode IDT nor any protected mode interrupt handlers, interrupts must be disabled the entire time while in protected mode.

Finally, the assembler must be directed to emit 32-bit instructions in protected mode with the `.code32` directive. It must be placed immediately after the long jump that loads the 32-bit code segment selector.

6.9 Parsing the Multiboot Header

Once in protected mode, the bootloader has access to all 256 MB of memory. The job of the Multiboot loader is to "load" the kernel executable from low memory to high memory using load addresses contained in the Multiboot header. Parsing the Multiboot header is simpler than ELF, but the .bss section must still be handled properly. As mentioned earlier, it's probably easiest to write this code in C.

According to the Multiboot specification [4], the Multiboot header contains a flags field of which bits 0–15 must be properly handled by the bootloader if they are present (otherwise the bootloader should fail to load the kernel image), and bits 16–31 may be optionally handled, but the bootloader should proceed even if it doesn't understand them.

It is a requirement that your bootloader properly handle a flags field with flags 1 & 16 set. Your bootloader does not need to support the functionality of the other flags, but it must follow the specification in regard to booting vs. not booting when any of bits 0–15 are set and the bootloader doesn't understand them.

The presence of flag 16 in the Multiboot header indicates that the Multiboot header also contains the load addresses for the kernel image. You must implement a Multiboot loader that uses these load addresses to load the kernel image into upper memory.

The Multiboot specification states that compliant boot loaders must be able to load images that are in the ELF format (by parsing the ELF headers), and that flag 16 may not be present for ELF images. Your bootloader does not need to parse the ELF headers. Any kernel image that we expected your bootloader to properly boot (ELF format or otherwise) will always contain flag 16 set, and we expect the load addresses present in the Multiboot header to be used to load the kernel image.

6.10 Writing the Multiboot Information Structure

The Multiboot information structure (MBI) exists to communicate information about the system environment from the bootloader to the kernel image. According to the Multiboot specification, the bootloader isn't required to write any of the MBI fields except the flags field (which determines which of the other fields are present), and the `mem_lower` & `mem_upper` fields if flag 1 in the Multiboot header is set.

In addition to the `mem_lower` and `mem_upper` fields, your bootloader must also write the `boot_device` field. These fields are described in Section 3.3 of [4].

Unfortunately, the current version of the Multiboot specification contains an error regarding the byte ordering of the `boot_device` field. The specification claims that "the first byte [of the `boot_device` subfield] contains the bios drive number as understood by the bios INT 0x13 low-level disk interface." However, the `drive` subfield is actually the *last* byte (i.e., most significant) of the `boot_device` field and the `part3` subfield is actually the *first* byte (i.e., least significant).

Here is an improved version of the subfield illustration from the Multiboot specification with byte offsets (relative to the start of the MBI) clearly depicted:

```
+-------+-------+-------+-------+
| part3 | part2 | part1 | drive |
+-------+-------+-------+-------+
   12      13      14      15
```

6.11 Establishing the Multiboot Execution Environment

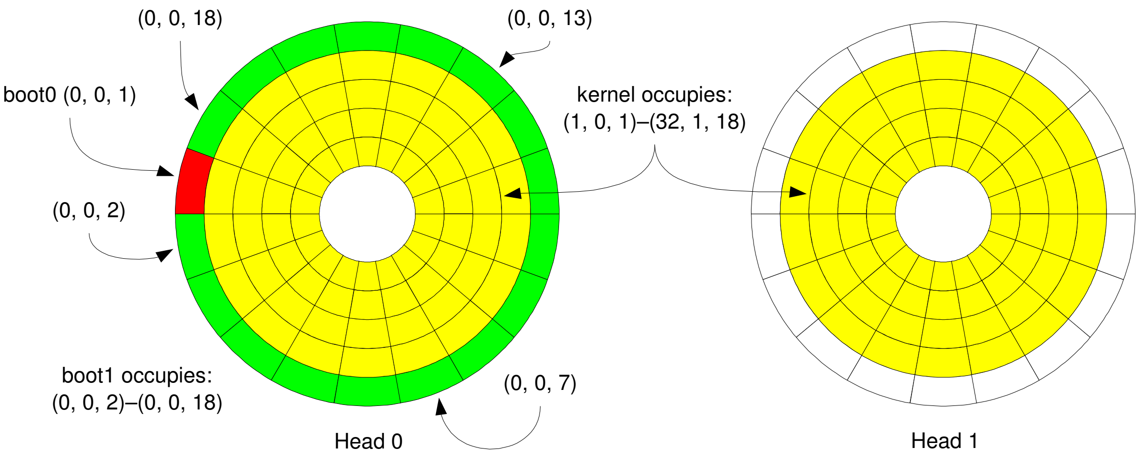
The Multiboot specification specifies a strict execution environment that must be established when transferring execution to the loaded kernel. Your bootloader must establish and provide this execution environment (see Section 3.2 of [4]).

7 Boot Floppy Disk Layout

Your bootloader may assume that all boot floppies are 1.44 MB double-sided disks with 80 tracks (per head), 18 sectors per track, and 512 bytes per sector. All boot floppies for this project conform the following disk layout:

C	H	S	Item
0	0	1	boot0
0	0	2	boot1 (start)
:	:	:	:
0	0	18	boot1 (end)
0	1	1	(unused)
:	:	:	:
0	1	18	(unused)
1	0	1	kernel (start)
:	:	:	:
32	1	18	kernel (end)
33	0	1	(unused)
:	:	:	:
79	1	18	(unused)

Here is a helpful illustration of the layout (there are also unused cylinders at the end (center) of the disk that are not represented):



Due to this layout, the maximum size of boot0 is 512 bytes. The maximum size of boot1 is 8.5 kB, and the maximum size of the kernel is 576 kB. boot0 should load the entirety of the maximum amount of boot1, and boot1 should preload the entirety of the maximum amount of the kernel. If your boot1 is ($<$ 8.5 kB) or if your kernel is ($<$ 576 kB) some garbage sectors will also be loaded, but this simplifies boot0 and boot1 dramatically.

The portions of the boot floppy disk layout marked unused are considered unused with respect to the bootloader. It is possible that those portions of the disk may actually contain extra kernel data and be loaded by the kernel itself.

8 Bootloader Memory Map

It is suggested that your bootloader implement the following memory map:

Linear Address	Item
`100000`	Top of memory hole
`0F0000`	Video memory, MMIO, BIOS
`0A0000`	Bottom of memory hole
`090000`	kernel preload (end: 0x9f000)
`010000`	:
`00F000`	kernel preload (start: 0xf000)
`00E000`	Protected mode stack (top: 0xf000)
`00D000`	:
`00C000`	Disk buffer (end: 0xc7ff)
`00B000`	:
`00A000`	:
`009000`	:
`008000`	Disk buffer (start: 0x8000)
`007000`	boot0 (0x7c00–0x7dff)
`006000`	Real mode stack (top: 0x7000)
`005000`	:
`004000`	
`003000`	boot1 (end: 0x31ff)
`002000`	:
`001000`	boot1 (start: 0x1000)
`000000`	Reserved (real mode IVT, BIOS data)

If you choose not to implement this exact memory map, you still must load boot1 in the indicated location, and must be able to load a kernel of the indicated size.

9 Useful BIOS Call List

The following is a list of BIOS calls you may find useful in programming your bootloader. Your bootloader is not obligated to use all of these calls, and may use calls not found in this particular list. However, if you're unsure of which BIOS call to use to achieve a certain task, this list may provide some guidance.

Call	Description
INT 0x10, %AH=0x00	Set Video Mode
INT 0x10, %AH=0x03	Read Cursor Position
INT 0x10, %AH=0x0e	Write Teletype to Active Page
INT 0x10, %AH=0x13	Write String
INT 0x12	Get Lower Memory Size
INT 0x13, %AH=0x00	Reset Disk System (FDC)
INT 0x13, %AH=0x02	Read Sectors
INT 0x15, %AH=0x24	A20 Gate Support
INT 0x15, %AX=0xe801	Get Upper Memory Size (EISA)
INT 0x18	Start BASIC

The above BIOS calls are fully documented (argument registers, return codes, etc) in [5], except *Write String* which is documented in [7], *Get Upper Memory Size (EISA)* which is documented in [1], and *Start BASIC*. The *Start BASIC* call takes no arguments and starts the ROM BASIC interpreter[4] (this call is not supported by Simics).

10 Build Environment

Here are the steps you need to follow to get started on Project 4.

1. Create a copy of your entire Project 3 kernel directory in a directory called `p4`.
2. Into that directory, download the Project 4 tarball, proj4.tar.gz.
3. Extract the tarball via

   ```
   tar xfz proj4.tar.gz
   ```

   You should have a new directory called "boot" and an updated version of `update.sh`.
4. Verify that you are happy with the filenames contained in `boot/config.mk`.
5. You can delete the tarball now.
6. Run "make update". You should now observe more files in the boot/ directory.
7. Run "make". This will *fail*, see below.

[4] Only on machines that contain BASIC in ROM, which is very few. Most BIOSes displays a `BOOT FAILURE` message with an option to reboot the machine (often after clearing the screen of any useful diagnostic output).

Your boot0 and boot1 and your kernel are installed in the disk image by a program called `mkimage.sh`. It will demand that you provide it with a boot0 which is exactly 512 bytes long and with a boot1 which is no larger than 8.5 kB. Since the template boot0.S we provide you with doesn't generate a valid boot sector, the build will fail.

You may find it useful to invoke `mkimage.sh` manually in order to mix and match various boot0, boot1, and kernel images. For example, you can try our special alternate kernel payload by running the command below.

```
gunzip payload.gz
boot/mkimage.sh bootfd.img temp/boot0 temp/boot1 payload
```

11 Plan of Attack

1. Before starting to write the bootloader, carefully read the provided source code, especially `sample_boot1.S` which provides a concrete example of making a BIOS call and demonstrates a number of useful GNU Assembler directives that will be helpful in writing assembly code for your bootloader.

2. Write a dummy boot0 that displays "Hello World!" using the *Write String* BIOS call in order to familiarize yourself with writing real mode assembly and making BIOS calls. This is harder than it sounds—thus it is *very important* to get something to work early on even if the particular thing isn't ultimately ultimately useful to the bootloading process.

3. Write a dummy boot0 which reads something from the floppy into a particular area in memory. One suggestion would be to read boot0 into the suggested disk buffer area and then compare a Simics memory dump against a hex dump (`od -x`) of your floppy image (`bootfd.img`).

4. Build `sample_boot1` and use your boot0 to load and execute it.

5. Now that boot0 is working, start work on boot1. Your goal is to load a kernel (such as yours) from the designated area on the floppy into the designated area in memory. You can then compare a Simics memory dump against a hex dump of your kernel.

6. Now enter protected mode. Try reading from and writing to a memory word somewhere around 16 megabytes. Also, the first line of output from the Simics `pregs` ("print registers") command will tell you whether you're in "16-bit legacy real mode" or "32-bit legacy protected mode".

7. Write your Multiboot loader. After you have loaded the parts of the binary into high memory, you can compare Simics hex dumps against gdb "memory" dumps (e.g., `x/i 0x100000`).

8. Launch your kernel!

9. Now boot the special alternate kernel payload we provide you with. Holding down the shift key and right-clicking in the graphical console window allows Simics to "capture" the mouse (and to relinquish it too).

12 Frequently Asked Questions

1. *How do I make boot0 exactly 512 bytes in size?*

   See `sample_boot1.S` for an example use of the `.org` directive.

2. *Why does Simics report* `FATAL: Not a bootable disk` *when attempting to boot?*

   This is the Simics BIOS equivalent of the dreaded "No operating system present" message. Carefully reread the relevant slides in the Bootstrapping lecture. The `.org` directive may also be useful here.

3. *What does a "track" refer to in CHS addressing?*

   The original 3.5 in floppy disk was single-sided with one writable surface and one head. The concentric circles that divided a surface formed the disk *tracks*. Since "modern" floppy disks are double-sided with two writable surfaces and two heads, the matching tracks across the two surfaces are collectively called a *cylinder*.

   The API specification for BIOS call INT 0x13, %AH=0x02 [5] states that register %CH contains the "track" number parameter. The term "track" as it is used here is a carry-over from the days of single-sided disks, and refers to the same value as the disk cylinder. Thus, %CH should contain the cylinder number of the CHS address when reading sectors from the floppy disk.

4. *Is it possible to avoid bit-mangling the cylinder and head numbers in %CH and %CL when reading sectors from the floppy disk?*

   Since the largest addressable cylinder in a 1.44 MB floppy disk is cylinder 79 (a 7-bit quantity) the top two bits of the 10-bit track number will always be zero. Similarly, since the largest addressable sector in a 1.44 MB floppy disk is sector 18 (a 5-bit quantity), writing the byte value 18 to %CL always zeros the top two bits of %CL. Thus, the naive method of writing the cylinder number to %CH and the sector number to %CL results in the same quantities for %CH and %CL as if the bit-mangling were performed, and therefore, it is sufficient to naively write the cylinder number to %CH and the sector number to %CL without performing any bit mangling.

5. *Where is the partition table in boot0?*

   Partition tables exist only in the Master Boot Record (MBR) of a partitioned disk. As floppy disks are not partitioned, they do not contain an

MBR, and thus, also do not contain a partition table. Some literature refers to the boot sector of a disk partition, or the boot sector of an unpartitioned disk, as the Volume Boot Record (VBR), or Partition Boot Sector (PBS).

All that really matters it that the lack of a partition table means that there is more room for boot0 code.

6. *Why does the sample boot1 code contain only a .text section, and not a .data or any other section?*

   All assembled code implicitly has a .text section containing the code itself. However, since the "executables" being produced in this project are not ELF binaries, but rather straight machine code, it is not necessary to have multiple sections with different permission bits, etc. However, it is advantageous in the case of boot0 to have a single section passed to the linker to guarantee that the generated machine code fits exactly in the size constraints of the bootsector.

   For compiled C code, there will be multiple sections present which are merged together by the linker before producing the binary machine code output.

7. *Why does the floppy disk keep spinning?*

   Because spinning the Floppy Disk Drive (FDD) up to its breathtaking speed of 300 rpm takes many milliseconds, it is traditionally left on between read/write operations.

   The FDD must be explicitly told to spin down its motor, and apparently the *Reset FDC* BIOS call does not do this on all hardware. Thus, the FDC must be told manually to stop spinning FDD motors by zeroing the Digital Output Register [3].

   Since Simics and other emulators don't actually have FDD motors, spinning them down is not a required part of this project. However, you are welcome to do so.

References

[1] E. Boleyn. INT 15h, AX=E820h: Query system address map [online]. Available from: `http://www.uruk.org/orig-grub/mem64mb.html`.

[2] A. E. Brouwer. A20: A pain from the past [online]. Available from: `http://www.win.tue.nl/~aeb/linux/kbd/A20.html`.

[3] debs@savah.freeserve.co.uk. Programming the NEC μpd765 and Intel 82072/7 floppy disk controller [online]. Available from: `http://www.isdaman.com/alsos/hardware/fdc/floppy.htm`.

[4] B. Ford, E. S. Boleyn, and Free Software Foundation. *Multiboot Specification*. Free Software Foundation, Boston, MA, June 2006. Available from: `http://www.gnu.org/software/grub/manual/multiboot/`.

[5] General Software, Inc. *BIOS User's Manual with BIOS Interrupt Reference*, 1998. Available from: `http://www.embeddedarm.com/Manuals/EBIOS-UM.PDF`.

[6] Intel Corporation. *IA-32 Intel Architecture Developer's Manual: System Programming Guide*. Mt. Prospect, IL, 2001. Available from: `http://www.cs.cmu.edu/~410/doc/intel-sys.pdf`.

[7] International Business Machines Corporation. *BIOS Interface Technical Reference*, Sept. 1991. Other manuals: `http://www.mcamafia.de/pdf/pdfref.htm`. Available from: `http://www.mcamafia.de/pdf/ps2bios2.pdf`.