These boots are made for walking

Johan Montelius

HT2018

Acknowledgment: The later part of this tutorial is based on the first edition of Philipp Oppermann's excellent blog "Writing an OS in Rust".

1 Introduction

In this tutorial you're going to boot your own computer (or virtual machine) from scratch. Your program will hopefully be able to print "OK" on the screen, so it's probably the most meaningless program that you have ever written; you will however be very proud and live your life with the comfort of knowing that there is no magic to operating systems.

The examples in this tutorial are built on an Ubuntu system. You will need an editor to write some code but the programs are very small. You will need some tools listed in the Appendix to compile and run the programs.

2 The boot process

To understand what you are about to do, you need to understand the boot process of a typical computer. We describe this process in three stages: firmware, boot loader and operating system.

2.1 Firmware

When you turn on the power of your computer the CPU will start to execute instructions on a well defined address. The motherboard is built in a way that the CPU will find a pre-installed program in a read-only or flash memory. This program is called the firmware of the system but is often referred to as the BIOS (Basic Input/Output System) since this was the name used on IBM compatible PC:s back in the times when "The Terminator" was the best movie ever. On your laptop today, you will probably find a firmware system that follows the UEFI standard (Unified Extensible Firmware Interface). Most people that do not know the difference simply refer to any firmware program as the BIOS (and so will I, if it does not matter).

The role of the firmware is to detect memory modules and external devices, set clock speeds and monitor fans, perform a system test to verify that all hardware is functioning (Power-On Self-Test, POST) etc; a modern UEFI firmware also include a graphical user interface. If you interrupt the start-up procedure (typically by holding down the F2-key) you can enter a user interface and change any of the setting.

When the system is good to go, several things might happen depending on which firmware we run. The traditional BIOS firmware was very limited in what it could do while a modern UEFI compatible firmware is close to a small operating system. In this tutorial we will follow the path of a traditional BIOS; this is simpler while still giving you the basic understanding of what needs to be done to get a machine started.

When the BIOS has performed the initial testing of the system it will locate the primary disk and read the first segment of 512 bytes. This segment is called the *Master Boot Record* (MBR) and this is the first thing that you will write today. The BIOS does not know anything about file systems so one can only place the master boot record at a well defined position on disk.

The master boot record is a very small program that will almost immediately start to read a larger program from disk. This could be an embedded program or an operating system but in most cases it is something called a *boot loader*.

2.2 Boot loader

The boot loader (also called boot manager) is a program that is compiled and stored on disk. Its role is to again examine what the machine looks like; the BIOS will provide some information but the boot loader knows more about how the file system is arranged on the disk etc.

On a Linux machine the boot loader is typically GRUB (*GNU GRand Unified Boot loader* but there are dozen of boot loaders to choose from and they are not directly linked to any operating system. If you are using GRUB then you can configure it to boot one of several operating systems, this is called dual boot. In this way you can run both Windows and Linux on the same machine (popular among gamers).

The boot loader will prepare many things that all operating systems need or can make use of. They also often provide a GUI that you can invoke by holding down a key during the boot process (for GRUB the right shift key). You will then be able to select which system to start, perform memory test etc.

Once the boot loader has determined which operating system to start, it will read the kernel from disk and jump to its first instruction. Once this is done the operating system is in control.

2.3 Operating System

The operating system will start by taking control over the hardware. It will move to 64-bit mode (if that is not already done), create a stack, set up the *interrupt descriptor table*, initialize virtual memory etc. It will then organize the task scheduler, the file system and hundreds of other things before it launches services for users to access.

A large operating system is divided into modules and not all modules need to be provided in the kernel when the system boots. Kernel modules can be brought in at a later stage depending on how the user wishes to use the system. This modularity allows the system to be modified to handle new hardware etc. The largest part of an operating system is all the modules or drivers that are needed to cope with various hardware. If all hardware was identical, life would be so much simpler.

We will not have time to write a full operating system in one course, but one could write a small kernel in a couple of months, that works for a particular hardware. To write a complete operating system is of course a huge task but it's not something that's undoable. If you want to follow a new operating system in development you can take a look at Redox; an operating system written in Rust.

3 Let's go

So now that you have a basic understanding of how the boot process works let's do some coding. The first thing we will do is to create a minimal master boot record and learn how to boot a virtual machine (or your own laptop) using our code.

3.1 Some tools to get you going

In order to write a small master boot record we need an assembler. The *nasm* assembler, that you can install using *apt*, will serve our purpose. We also need a machine to run on and although you could use our own laptop this would be too cumbersome. It is much easier to use a virtual machine or emulator. One popular choice is the QEMU emulator, a system that can not only virtualize a machine but can also be configured to virtualize just about any type of hardware.

QEMU would for example allow us to develop and run programs compiled for an ARM processor, even though we have done the development on a X86 architecture. This would of course require that we compile our programs for the ARM architecture - this is often called cross compilation. We will not do this in this assignment, rather we will assume that you're developing on a X86 architecture and let the QEMU emulate a X86 architecture.

```
> sudo apt install nasm
  :
> sudo apt install qemu-system-x86
```

3.2 The master boot record

The master boot record is a 512 byte sequence that is the program that we will run. We will simply implement a program that writes "Hello" to the screen and then loops. Since we need to work with explicit addresses and will call procedures of the BIOS we need to do this in assembly. Create a file called *boot.asm* and start hacking.

Note that in the program listing below we use *Intel-syntax* as opposed to *AT&T-syntax*. The most striking difference is that commands are written: operation, destination, source. This could be very confusing if you have only used AT&T-syntax that uses: operation source destination.

The program needs to be compiled by the assembler (*nasm*) for a 16-bit Intel 8086 processor. Although you probably have an Intel iCore or AMD Ryzen processor, a million times more powerful than a 40-year old 8086 processor, it will boot up pretending to be a 16-bit 8086 processor (I know this sounds absurd but it is very backwards compatible). Therefore we tell the assembler that it should produce 16-bit code; at the top of the page write this:

```
bits 16
```

The next thing we need to do is to set up the *stack segment* and *data segment* register. If you have not read about memory segmentation and how this is done on a 8086, this is almost incomprehensible. The idea is that all memory references are done relative a segment register. We know that our code will be loaded into memory address *0x07c00* and thus load *0x07c0* into the data segment register. The value will be shifted four bits before added to an address. The stack segment register is given a value *0x07c0* plus *0x20* since our master boot record is *0x200* (512) bytes long. The stack pointer is then set to *0x1000* which means that the stack will go from memory address *0x08E00* to *0x07E00* (it grows down).

For almost all instructions that we have, this does not mean anything but we need a stack and we will make use of one pointer to a text string and we will have to somehow reference this.

Let's just leave this for now and we will try to understand what is happening later. The piece of code goes like this:

```
start:
        mov ax, 0x07C0        ; 0x07c00 is were we are
        add ax, 0x20          ; add 0x20 (when shifted 512)
        mov ss, ax            ; set the stack segment
        mov sp, 0x1000        ; set the stack pointer

        mov ax, 0x07C0        ; set data segment...
        mov ds, ax            ; more about this later
```

So far so good - it's now time to set up the arguments for the BIOS print procedure. We have left the BIOS but the BIOS has a set of procedures that we can make use of and one of them is the *print character procedure.* We will print our message, character by character, and we therefore place a pointer to it in register *SI* (Source index). We also set the identifier for the print procedure (*0x0e*) in the *AH* register. Then we are set to go.

```
        mov si, msg          ; pointer to the message in SI
        mov ah, 0x0E         ; print char BIOS procedure
```

Now for the printing of the message; we will load one byte from what ever *SI* is pointing to and copy it to register *AL*, check that it is not equal to zero and print it to the screen.

```
.next:
        lodsb                ; next byte to AL, increment SI
        cmp al, 0            ; if the byte is zero
        je .done             ;    jump do done
        int 0x10             ; invoke the BIOS system call
        jmp .next            ; loop

.done:
        jmp $                ; loop forever
```

This looks a bit strange but we're using a special instruction here, *lodsb*, that does several things. It will by default use *SI* to fetch a byte, store it in *AL* and increment *SI* by one. Since we have set up *SI* to point to our message we will take one character after the other until we find a byte that is zero.

The print procedure is invoked by raising an *interrupt* with the argument *0x10.* The processor will then push a return address on the stack and use the *interrupt descriptor table* to jump to the BIOS call handler. The *interrupt descriptor table* determines what should happen when interrupts are generated. The operating system will when it is ready fill this table with routines thus taking control over all execution.

The BIOS interrupt handler will look in register *AL*, find a byte and print that character to the screen. It will then return from the call and we can continue with the next byte. When all characters have been printed we just loop forever.

We are now almost done but we need our welcome message. The *db* instruction will add a sequence of bytes (the ASCII of "Hello") at this location and terminate it with a zero.

```
msg:    db 'Hello', 0        ; the string we want to print
```

Now for the magic, we need to turn this into a 512-byte sequence with a precise master boot record signature (0̂xAA55) at the end. The *times* directive will repeat the assembly instruction *db 0* (set byte to zero) a number of times. The *$* and *$$* symbols are macros for the current position and start of segment position. We can then calculate how many zeros wee need to have to fill this segment up to position 510. We then have two bytes left and there we write the magic signature.

```
        times 510-($-$$) db 0    ; fill up to 510 bytes
        dw 0xAA55                ; master boot record signature
```

That's it, we now have a master boot record.

3.3 Boot a machine

To now boot a machine with this boot record we first need to assemble it to a binary. We use the *nasm* compiler and call the output *boot.bin*. The *-f bin* tells the assembler that we want a plain binary file without any bells nor whistles.

```
> nasm -f bin -o boot.bin boot.asm
```

You can verify that the file is exactly 512 bytes long using the *ls* command.

```
> ls -l boot.bin
```

Now we start the QEMU emulator and provide our binary as the master boot record. QEMU will behave as a regular x86 processor and start to execute our program. Give it a try:

```
> qemu-system-x86_64 -drive file=boot.bin,index=0,media=disk,format=raw
```

If everything works (does it ever on the first try), you will see a terminal window with the print out by the QEMU system first, followed by our "Hello" message. You might not believe that this would actually work on a real machine but trust me it does.

4 The GRand Unified Bootloader - GRUB

Now that we know that we can boot a machine from scratch we are going to make use of an existing boot loader. This will make life easier for us since it will do part of the work that we would otherwise have to do.

We will now create an *ISO image* of a disk that holds the GRUB master boot record in its first segment. An ISO image is the format of a CD-ROM disk that we can tell QEMU or VirtualBox to boot from. When booting using this disk, GRUB will be loaded.

To help us with this we will use a tool called *grub-mkrescue*, a tool that is used to create bootable rescue disks. The procedure is slightly more complicated but you will have your own operating system (that writes OK on the screen) up in no-time. We will now call our program the *kernel* since it is how a regular kernel is loaded.

You need *grub-common*, *grub-pc-bin*, *xorriso* and *mtools*.

```
> sudo apt install grub-common
  :
> sudo apt install grub-pc-bin
  :
> sudo apt install xorriso
  :
> sudo apt install mtools
```

The ISO image will also hold our kernel and a configuration file that instructs GRUB that we want our kernel to be loaded. Let's start by building a small kernel.

4.1 the multiboot header

Since we wrote the master boot record in a very special way, we need to write the kernel in a special way. In order for GRUB to recognize it as a kernel it must start with a magic *multiboot header* (actually, it does not have to be in the beginning but we will keep it simple). To make the development process easier, we will divide our kernel into different parts. These parts will then be *linked* together to form a final binary. We begin with the multiboot header.

This is the *multiboot_header.asm*, it contains: a *magic number*, an identifier for the architecture, the length of header and a *checksum*. The checksum is calculated using the magic number, architecture and the length of the header. The header also holds a sequence of *tags*. We will not have any tags so we simply provide the termination tag.

```
section .multiboot_header

magic   equ 0xe85250d6      ; multiboot 2
arch    equ 0               ; protected mode i386

header_start:
        dd magic                        ; magic number
        dd arch                         ; architecture
        dd header_end - header_start    ; header length
        dd 0x100000000 - (magic + arch + (header_end - header_start))

        dw 0                    ; type
        dw 0                    ; flags
        dd 8                    ; size
header_end:
```

We now compile this using *nasm* but we will not turn it into a binary immediately. We will create an *object file*, *multiboot_header.o*, that we will later link with the kernel. The object file holds the compiled code but is structured with tags so that a *linker* knows how to create one final binary from many object files. We do this by providing the flag *-f elf32* that will generate an ELF-file (*Executable and Linkable Format*).

```
> nasm -f elf32 multiboot_header.asm
```

4.2 the kernel

The kernel will now be a small piece of code that will run in 32-bit protected mode. The GRUB boot loader has already gone through the steps of starting in 16-bit real mode and gradually move up to 32-bit mode. When our kernel starts, it will not have a stack but the *segment registers* are set so that we can address the memory with out any problems.

We will now try an alternative way of accessing the terminal, we will not use the BIOS systems calls but instead use a memory area that is directly mapped to the terminal. The view we have is that the 80x25 characters on the screen are directly mapped to a memory area called the VGA buffer. This buffers starts on location *0xb8000* and maps $80 * 25 = 2000$ characters. Each character is represented by two bytes where the lower byte is the character number (basically ASCII) and the higher byte holds the attributes (low to high): color (4 bits), background (3), blinking (1),

Our first kernel will write "OK" in the upper right corner in white on green. Note how we write the characters in reverse order (*0x4b* is K and *0x4f* is O); has this anything to do with little-endian? The attribute *0x2f* is when written as a bit sequence equal to *0 010 1111* which is interpreted as: non-blinking, green background, white font. Look up *VGA-compatible text mode* to learn more.

```
bits 32
global start

section .text

start:
        mov dword [0xb8000], 0x2f4b2f4f     ; print ‘OK‘ to screen
        hlt
```

The program, small as it is, has a few things that are worth mentioning. We start by declaring the label *start* as global. This will tell the assembler to leave an entry for this tag in the object file. We also declare the section as *.text* which is the segment where we will have code. These labels are used when we will link the program into one executable binary. The *bits 32* directive tells the assembler to generate code for 32 bit protected mode and not the 16 bit mode that we used in our boot program. As for multiboot header we assemble the kernel and turn it into an object file.

```
> nasm -f elf32 kernel.asm
```

4.3 linking the object files

To understand what we will do now now, you can take a look at the object files using the *readelf* program. The program will read an ELF object file and display tons of information that's mostly mostly Greek (unless you're from Greece, then it is mostly Chinese).

```
> readelf -a kernel.o
```

You will however identify things like the *.text* segment, *start* label, *X86_64* architecture etc. Don't ask me to explain all this - the take away is that the object file contains information for the linker to create an executable binary.

In order to link the kernel and multiboot header correctly we write a linker script file that will tell the linker what should be done. We call this file *linker.ld* and it looks like follows:

```
ENTRY(start)

SECTIONS {
    . = 1M;

    .boot :
    {
        *(.multiboot_header)
    }

    .text :
    {
        *(.text)
    }
}
```

The linker will take the two object files and put them together into one executable file. The executable file is still in ELF format since this is what GRUB expects. The format describes what segments should be loaded and where execution is to start. We also specify that the segments should be loaded at address 1M (0x100000). This is safely above all addresses where the BIOS and GRUB have allocated their data.

```
> ld -m elf_i386 -o kernel.bin -T linker.ld multiboot_header.o kernel.o
```

You can now inspect the file using *readelf*: what is the *entry point address*? Does it make sense?

```
> readelf -a kernel.bin
```

4.4 creating an ISO image

So now for the last step before we can boot our machine. We should create an ISO image (this is a standard for CD-ROM discs and will probably survive for decades although no one has seen a CD-ROM for years). In doing so we need to set up a directory structure to use *grub-mkrescue*.

Create the following directory structure and copy *kernel.bin* to its place.

```
cdrom/
   boot/
       grub/
         grub.cfg
       kernel.bin
```

The file *grub.cfg* is a file that GRUB will read during boot time to locate our kernel. This file holds information about which kernels are available (hence the name multiboot) and where they are found. We only have one kernel to choose from but we could have more than one version. We set a delay of 10 seconds before GRUB loads the default kernel.

```
set timeout=10
set default=0

menuentry "my os" {
    multiboot2 /boot/kernel.bin
    boot
}
```

We now use *grub-mkrescue* to create a ISO image that is bootable and contains both GRUB and our kernel.

```
> grub-mkrescue -o cdrom.iso cdrom
```

You can inspect the image using the *file* or *isoinfo* tools. As you can see the file is a CD-ROM image, and apparently it's a bootable disk image.

```
> file cdrom.iso
  :
> isoinfo -d -i cdrom.iso
  :
```

If you're really interested in what it looks like you can *mount* it. It will then look like any directory in you file system.

```
> mkdir iso
  :
> sudo mount -o loop cdrom.iso iso
```

You can now look in the *iso* directory and explore the file system in the image. You will of course find the *boot* directory with our kernel and the configuration file, but also everything that is needed for GRUB to run.

Let's see if it works:

```
> qemu-system-x86_64 -cdrom cdrom.iso
```

... 8, ... 9, ... 10 ... OK!

Congratulations, you have just loaded your first kernel using GRUB. Only thing needed to create a full-fledged operating system from here, is simply a ton of work - there is no magic to computers!

5 Summary

You have gone through the steps of booting a computer. The first method was the most basic where we built a small master boot record and booted directly from the BIOS. In the second method we booted GRUB and then booted our kernel from there. To build an operating system from here is of course a lot of work, but I still think it is valuable to see that operating systems are not created by magic.

BIOS, UEFI and more

We used the simpler form of firmware when we tinkered with the system. A regular computer will probably have a full UEFI firmware, but these can often be configured to pretend to be a regular BIOS (legacy mode).

If you're interested in knowing more about alternative firmwares you should take a look at Coreboot; this is an open source firmware that can be tailored to boot an operating system directly, instead of going through a boot loader. The start-up sequence can then optimized to bringing up a systems in seconds rather than a minute. Chromebooks use this firmware to quickly have the machine up and running.

Boot your own machine

If you're running a Linux system you can take a look at your own */boot/grub/grub.cfg*. This file should not be edited manually but by adding entries in */etc/grub.d/* and running *update-grub*, you can configure your own GRUB to be able to choose between a regular Ubuntu operating system and our *kernel.bin* that we have built. You will of course only see "OK" in the top left corner but it's a good experience - it takes the magic away.

Going further

If you intend to go further from here you need to get a programming environment where you can build a system using something better than assembler. One alternative is to look at what Phillip Opperman is doing on his blog "Writing an OS in Rust". You will then learn a lot about operating systems and a new language.

Tools that you need

These are the tools that you must have in your system in order to follow the tutorial. Assuming that you're running an Ubuntu machine they should be easily installed using *apt* if you have sudo rights.

```
> sudo apt install nasm
```

These are the packages that you will need:

- *nasm* : the assembler
- *qemu* : the emulator machine
- *qemu-x86* : the qemu-system-x86_64
- *grub-common* : grub-mkrescue to create the GRUB iso image
- *grub-pc-bin* : to make grub-mkrescue produce a i386 boot loader
- *xorriso* : to allow grub-mkrescue to create an ISO image
- *isoinfo* : to look at the ISO image