Let There Be Kernel: My Journey to Building an OS

Zeimpekis Vassilis

July 19, 2025

Contents

Chapter 1

Introduction

Welcome to my journey of writing an operating system from scratch... and hopefully not going insane in the process. My hope is that this book will bring some of you along for this ride and help you learn from — and hopefully avoid — my mistakes.

When I started studying about operating systems (no earlier than 4 months ago) I found very few sources, most of which claim OS development is a dark path mere mortals should avoid. And me being the optimist I am, I denied this fate and walked down that path only to find out they were pretty... well, right.

However, the majority of obstacles I have encountered are immediately correlated with the lack of resources. I am not sure whether this scarcity is due to financial incentives or the genuine difficulty of the field, but it undeniably slows down potential progress.

This book will contain what I have learned so far and explain my mental process. I have found following someone's learning experience can ease the beginner's struggle. The reader can overcome common challenges by following the thought process someone else used to tackle them. However, all material should be accompanied by hands-on experience for best understanding.

1.1 Why Write Your Own OS?

Studying Operating Systems is challenging but trying to implement one is a totally different beast. Even realizing how to start studying could critically discourage many people. Therefore, diving into OS development definitely requires some "getting your hands dirty" to understand.

In the process there is definitely an overwhelming amount of low-level detail you could encounter, but there is luckily as much to gain in knowledge. From how a bootloader works to how the hardware of your computer is wired together to perform basic tasks like handling signals of peripheral devices or doing math, how much there is to learn is only limited by your determination. After finishing this project you will walk out a totally different person: a low-level developer or even engineer, dare I say.

1.2 What You'll Learn

- How bootloaders work and how to build one

- What real mode and protected mode is

- Writing a memory manager

- Implementing syscalls

1.3   Who This Book Is For

By now you should know that the purpose of this book is to make diving into OS development a little more beginner friendly. When writing this book I mainly give my own perspective, as an Electrical Engineering and Computer Science student, therefore I mainly imagine myself addressing people of virtually my technical background and knowledge.

However, I believe that this project will also benefit hobbyists and professors looking to incorporate low-level OS development into their courses, by helping them understand what challenges students might face while studying this subject and providing a simple enough prototype which students can understand, replicate or iterate on.

1.4   What You Should Already Know

As was mentioned above I, myself, have some technical background. Although I am not an expert it would surely help if the readers of this book had a level of understanding similar to or greater than mine.

More specifically I had experience in:

- High-level C programming

- Minimal understanding of assembly

- Basic hardware knowledge and how components interact

- Introductory digital design

Understanding the material of this book should require some familiarity with the above.

1.5   How To Approach This Book

While reading this book you will find that you and I have a slightly different way of understanding things, simply because that is statistically true. When I started implementing things, nothing worked. My study process was chaotic due to the lack of structured resources. Most of my progress came from repeated trial — and more importantly — error.

Having said that, I hope it is clear that we will struggle in slightly different things for slightly different reasons and this should not discourage you at all. When (not if) times get tough feel free to reach out to me, a contributor, or one of your professors to help clarify misunderstandings.

1.6   Book Structure and Approach

This book follows the narrative of my own development journey. In each stage of building the operating system, I had to make structural and architectural decisions. In the first chapters, I'll walk you through the reasoning behind my choices — from setting up a simple bootloader and transitioning from 16-bit to 32-bit protected mode, to handling interrupts and implementing memory management. These early chapters favor simplicity and clarity. Later on, we'll explore more complex or alternative designs, compare architectures, and analyze the trade-offs behind them.

Chapter 2

Bootloader

If you are stil here... buckle up. In this chapter we will familiarize ourselves with some concepts necessary to understand bootloaders and then start writting our own.

Before we start it should be made clear that everything below applies to the Intel i-386 processor architecture (32-bit x86). However, transferring that knowledge to different architectures should not be that difficult

2.1   CPU Operating Modes

When the computer turns on the x86 processor is automatically in **Real Mode**, which has some significant limitations. However, it is kept even in newer processors for backwards compatibility reasons.

When in Real Mode the CPU is running by default in 16-bit mode. While 32-bit registers are technically still available they should not be used, except when changing to Protected Mode (32-bit mode).

Another limitation of Real Mode is that you only have access to a specific size of memory. When referring to a memory address in Real Mode we use a 20-bit physical addressing. The Physical Address is referred to using a Segment and Offset like this:

$$\textbf{PA} = \textbf{Segment} \cdot \textbf{16} + \textbf{Offset}$$

Essentially, using this convention you can only represent a number if it can fit in a 5 digit hex. It is also apparent that there are multiple ways to represent the same address. For example the physical address 0x12345 can be represented by 0x1234 and 0x0005, 0x1230 and 0x0045, 0x1200 and 0x0345... and so on. This way of addressing memory limits us to just below 1MiB of memory.

By now it should be clear that the addresses used in Real Mode are the actual physical addresses. This hinders us from protecting memory and defining its owenership because every process can simply access all memory segments. We will get into virtual memory later on this book but I will give a brief explaination of how it helps with owenership just to clarify why physical addresses are limiting.

When having virtual addresses, our Operating System is responsible of mapping every virtual address to a physical onen. When this mappning is done, the operating systems ensures that each program has its own range of physical addresses. When two programs refer to the same virtual memory, very little does it matter, since the mapping will make sure they will refer to totally different physical addresses, rendering both unable to access eachothers' memory.

Having said that, Real Mode is simply inadequate for modern systems. This is why engineers came up with another CPU Operation Mode called **Protected Mode**. In this mode, while we still do not have any virtualization, we can take advantage of how memory is segmented to protect critical data from the user. This can happen by defining different memory segments with different privileges (ring levels) and letting the CPU decide if a piece of code has the privilages it tries to claim. This is decided according to the segment through which we are accessing that memory (we will se that in practice). Memory management and CPU Operating Modes will have chapters of their own later in this book so we will not elaborate any longer here. For now let's focus on what a bootloader is and what it needs to do in its short but critical lifespan.

2.2   The Role Of The Bootloader

When the computer turns on the first program to take control is the BIOS. Among other things, the BIOS transfers control to the bootloader which is then responsible for loading the OS. Modern systems use UEFI but in the sake of simplicity we will focus on BIOS.

First, the BIOS scans the data storage devices. More precisely it checks the first 512 bytes of each one, namely the Master Boot Record (MBR), which is the place a bootloader conventionally resides upon. If the last two bytes of the MBR are the word 0xAA55 the BIOS classifies the storage device as a bootable device.

Once the bootable is identified the BIOS loads the bootloader from the MBR to the memory address 0x7C00. This address has been traditionally used by BIOS software to load the bootloader. Bootloader developers conventionally assume this is where their bootloader will is be loaded. The address 0x7C00 is way below the 1MiB accessible range and still leaves space below it for interrupt vectors (we will talk about them later).

After control has been passed to the bootloader it is now its job to load the rest of the operating systems. In later section we will discuss some techniques to overcome the 512-byte barrier and the limitations of the Real Mode. For now let's take a look in how we can load the kernel.

2.3   Loading The Kernel

Let's start by dumping some code. Please do not be overwhelmed by it as we will explain it thoroughly shortly after.

```
1  [ORG 0x7C00] ; This is where the bootloader is loaded in memory
2  [BITS 16]    ; Bootloader code starts in 16-bit mode
3
4  start:
5      cli
6      mov ax, 0x0700
7      mov ss, ax       ; Set stack segment to 0x0700
8      mov sp, 0x0000   ; Set stack pointer below of bootloader
9      xor ax, ax       ; Zero-out ax
10     mov ds, ax       ; Set data segment to 0
11     mov es, ax       ; Set extra segment to 0
12
13     mov ah, 0x02
14     mov al, 1        ; Number of sectors
15     mov ch, 0
16     mov cl, 2
17     mov dh, 0
```

```
18      mov dl, 0x80
19      mov bx, 0x1000    ; Segment
20      mov es, bx
21      xor bx, bx        ; Offset
22      int 0x13
```

Listing 2.1: Simple bootloader start in assembly

Let's analyze this line by line.

First of all `cli` is used to prevent the system from triggering interrupts. Since we have not dealt with setting up interrupt handling yet, they will just cause problems.

In lines 6 to 8 we are setting up a simple stack for our Real Mode. To understand these lines we must remember how we address physical memory in Real Mode. It our Stack Segment pointer (ss) we choose segment 0x0700 and in our Stack Pointer (sp) we define the offset 0x0000. Now the physical address is calculated as such:

$$\mathbf{PA = 0x0700 \cdot 16 + 0x0000 => PA = 0x7000 + 0x0000 => PA = 0X7000}$$

This address is chosen to be more than 512 bytes below 0x7C00, the range where out bootloader is placed. Note that the bootloader might still run without lines 6 to 11, however, without having the stack set up and the ax, ds and es registers zeroed-out the environment would be highly unpredictable.

Now that we are in a controllable environment we can load our kernel into memory. To do this we will must use a very useful feature of BIOS, BIOS interrupts. Since we have not explained interrupts yet, we can think of them as favors. More specifically when we trigger a BIOS interrupt we stop the execution of anything else, as the name states, and have BIOS do what we ask. In this occasion by calling `int 0x13` we ask the BIOS to load some data from our hard drive into our memory. However, before asking that we must pass some parameters using some registers to tell BIOS exactly what it should do.

Register	Purpose	Value Used
AH	BIOS function number	0x02 (read sectors)
AL	Number of sectors to read	1 sectors
CH	Cylinder number (part 1)	0
CL	Sector number (bits 0–5) and high bits of cylinder (bits 6–7)	2 (start at sector 2)
DH	Head number	0
DL	Drive number	0x80 (first hard drive)
ES:BX	Memory segment:offset where data is stored	0x1000:0x0000 (i.e., 0x10000)

Table 2.1: INT 0x13, AH=0x02 — Disk Read BIOS Call Parameters

```
1      mov ah, 0x02
2      mov al, 1         ; Number of sectors
3      mov ch, 0
4      mov cl, 2
5      mov dh, 0
6      mov dl, 0x80
7      mov bx, 0x1000    ; Segment
8      mov es, bx
```

```
9       xor bx , bx          ; Offset
10      int 0 x13
```

Listing 2.2: Assembly to load the kernel

Bibliography