Let There Be Kernel: My Journey to Building an OS

Zeimpekis Vassilis

July 17, 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 drag some of you along for this ride and help your learn from my mistakes.

When I started studying about operating systems (no earlier than 4 months ago) I found very few sources, most of which claim this dark path my helpless soul is about to stride along will spoil every bit of happiness a mortal can have and divert me of my final destination, eventually rendering me unable to finish this journey. 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 and am not fit to tell if this lack is pursued for financial purposes or genuinely created by the difficulties of the field, nevertheless, it halts possible advancement.

This book will contain what I have learned so far and explain my mental process. I have found reading along someones learning experience can ease the struggle of continuously emerging challenges by making available the thought process followed to overcome them.

1.1 Why Write Your Own OS?

Learing about Operating Systems is challenging but trying to implement one is a totally different beast. The struggle of finding out even how to start studying was what would for most people be a critical burden. Therefore, diving into OS development definitely requires some "getting your hands dirty" to understand.

In the process there is definitely a hilarious amount of details your learning journey could diverge towards learning, 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 writting this book I mainly give my own prespective, as an Electrical Engineering and Computer Science student, therefore I mainly imagine myself referring to people of virtually my technical background and knowledge.

However, I belive that hope 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 during studying this subject and providing a simple enough prototype which students can understand, replicate or iterate on.

1.4 What I Already Knew

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

More specifically I had experience on:

- High-level C programming
- Minimal understanding of assembly
- Hardware components and their job
- Introductory digital design

1.5 How To Approach This Book

While reading this book you will find that me and you have a slightly different way of understanding things, simply because that is statistically true. When I started implementing things at first nothing worked properly. My studying was not structured because of the lack of resources and most of my progress occured through trial and- most 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 contact me, another contributor or a professor of yours to clarify questions.

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, but 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 is technically still available the intended usage is the 16-bit registeres. This means that when writting our bootloader we should use 16-bit registers for any operations (with a small exception).

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:

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

Essentially, using this convention you can only represent a number if it can fit in a 5 digit hex. We can also see 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 accessing 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 does not allow us to protect memory and define its owenership because every process can simply see every physical address. 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 one. Since this mapping is not handled by the user program itself, 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.

However, this approach 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 and letting the CPU decide if a piece of code has the privilages it tries to claim, according to the privilage ring of the segment selected to access it. 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 our bootloader is and 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.

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 on. If the last two bytes of the MBR are the word 0xAA55 the BIOS knows the storage device is a bootable device.

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

Bibliography