Let There Be Kernel: A Journey of Building an OS

Zeimpekis Vassilis

September 21, 2025

Contents

Chapter 1

Introduction

Welcome to this journey of writing an operating system kernel from scratch... and, with some optimism, not going insane in the process. This book encourages you to explore how kernels work, help you learn from a student's experience, and perhaps even avoid some common mistakes.

When starting studying operating systems the lack of resources becomes abundantly apparent. Most of the existing resources even warn that OS development is so challenging a subject that should be avoided by anyone lacking extensive experience in low-level systems programming. Optimistically ignoring the warnings and diving in the field we will immediately discover that they are, well... pretty much right.

However, the majority of obstacles of OS kernel development stem directly from that lack of resources. It is unclear whether this scarcity arises from financial incentives or the inherent difficulty of the field, but it undeniably prohibits many engineers from studying OS kernels and potentially gainning expertise.

This book presents the knowledge gain of someone who did try to write an OS kernel and explains the reasoning behind their approach. Following someone else's streamlined learning journey can enhance and accelerate your own understanding and help avoid common pitfalls.

That said, this book seeks to make OS development accessible to beginners while acknowledging and engaging with the field's inherent complexity.

1.1 Why Write Your Own OS?

Studying Operating Systems is challenging enough on its own but engineering one from the kernel upward adds vast layer of complexity. Determining an initial approach to the subject can be challenging enough to discourage many beginners. Ultimately, diving into OS development requires a willingness to get your hands dirty and learn by doing.

The low-level nature of OS development guarantees that many obstacles will inevitably arise. Yet beyond the theoretical knowledge, the field offers a wealth of engineering skills to be acquired. From understanding how a bootloader works to uncovering how hardware orchestrates basic tasks-such as handling signals from peripheral devices or performing arithmetic-the amount to learn is limited only by your determination. By the time you finish this project, you will emerge as a fundamentally different person: a low-level developer-or perhaps even an engineer.

1.2 What You'll Learn

- Bootloader basics

- Transitioning from real mode to protected mode
- CPU operating modes
- Interrupts and the Interrupt Descriptor Table (IDT)
- Memory management and paging
- Process management and context switching
- User and kernel modes

1.3 Who This Book Is For

By now, it should be clear that the purpose of this book is to make diving into OS development a bit more-beginner friendly. This book is written by the prespective of an Electrical Engineering and Computer Science student, and the readers are assumed to have a similar technical background.

Nevertheless, this project also aims to benefit engineers transitioning into the field, as well as professors seeking to incorporate OS kernel development into their courses. This is achieved by highlighting the challenges students are likely to face and by providing a simple prototype that can be understood, replicated, or built upon. That said, this book should make OS development more accessible to beginners, but it does not shy away from the inherent complexity of the subject.

1.4 What You Should Already Know

As was mentioned above, the readers should already be familiar with some technical concepts. Below you can find a brief summary of them:

- High-level C programming
- Basic understanding of assembly
- Introductory digital design and computer architecture
- Theoretical understanding of operating systems
- Tools such as Git, Make and gcc

1.5 How To Approach This Book

Most people approach complex subjects in fundamentally different ways. It should be clear that everyone will struggle in different areas, and this should never discourage you. Working through a structured resource can greatly help in overcoming those initial difficulties.

When-not if-times get tough, don't hesitate to reach out to the author, a contributor, or one of your professors for help in clarifying misunderstandings.

1.6 Book Structure

This book follows the narrative of the author's development journey. In each stage of building the operating system, we will face structural and architectural decisions. In the first chapters, we will walk through the reasoning behind certain choices — from setting up a simple bootloader and transitioning from 16-bit to 32-bit protected mode, to handling interrupts and implementing memory and process management. These early chapters emphasize on simplicity and clarity. After finishing the implementation of the designs decided we will explore more complex or alternative designs, compare architectures, and analyze the trade-offs they involve.

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:

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

Essentially, using this convention you can only represent a number if it can fit in a 5 digit hex. 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 we 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.

```
[ORG 0x7C00] ; This is where the bootloader is loaded in memory
[BITS 16]    ; Bootloader code starts in 16-bit mode

start:
    cli
    mov ax, 0x0700
    mov ss, ax        ; Set stack segment to 0x0700
    mov sp, 0x0000    ; Set stack pointer below of bootloader
    xor ax, ax        ; Zero-out ax
    mov ds, ax        ; Set data segment to 0
    mov es, ax        ; Set extra segment to 0

    mov ah, 0x02
    mov al, 1         ; Number of sectors
    mov ch, 0
    mov cl, 2
    mov dh, 0
    mov dl, 0x80
    mov bx, 0x1000     ; Segment
    mov es, bx
    xor bx, bx         ; Offset
    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. As mentioned, there are other combinations that could address the same memory as well. 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 triggering the interrupt 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

Since interrupt `0x13` provides multiple storage device related function we can use the `ah` register to choose the `Read Sectors From Drive` function. In the `al` register we can define the number of sectors (512-byte unit) we want to read. For simplicity we will initially be reading just one.

Before understanding what registers `ch`, `cl` and `dl` we need to introduce a new addressing model called **Cylinder-Head-Sector** (or CHS). As the name states, this model mirrors the concept of physical cylinders, heads and sectors a traditional hard drive uses.

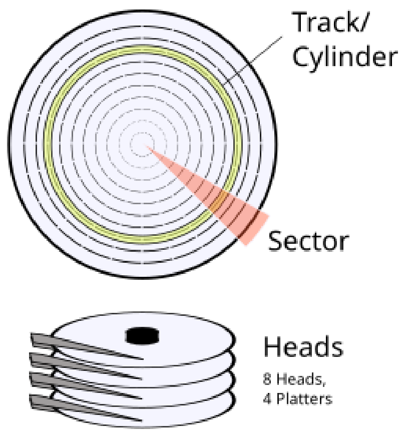

Figure 2.1: CHS addressing layout: cylinders, heads, and sectors

As shown above, hard drives are constituted of platters which are essentially circular disks. On both surfaces of these platters - both the top and bottom - data can be written magnetically by a device called head. So for 4 platters we have 8 heads.

Now imagine multiple cylinders with centers aligned with the platters'. The intersection of a platter's surface and a cylinder is called a track. Naturally we have $\mathbf{Cylinders} \cdot \mathbf{Platters} \cdot \mathbf{2}$ number of tracks. Each track can be accessed by only one head.

Each track is divided into sectors, which we can visualize as circular sectors. Each sector is the smallest addressable unit and has a size of 512 bytes.

When writting data to a hard drive we must specify the cylinder, the head and the sector along the track defined by the cylinder and the head's surface. Each combination of CHS maps to a 512-byte unit or more precisely a Logical Block Address (LBA). First, we write data in cylinder 0, head 0, and sector 1 (sectors start counting from number 1). As we fill in sectors until sector 63 we can start filling up heads. When head 255 is filled, we fill in cylinder. The last available cylinder is 1023.

- **1024**: Cylinders (0–1023, 10 bits)
- **256**: Heads (0–255, 8 bits)
- **63**: Sectors (1–63, 6 bits), *1-based indexing*
- **512**: Bytes per sector

Since CHS is limited at addressing 8.4 GiB of data it has been declared obsolete and replaced by direct LBA addressing. However, the BIOS still offers this function.

Also, note that modern hard drives do not expose their geometrical structure and instead use an emulation layer for CHS addressing.

For our case, since the first sector contains the bootloader, we want to start loading from sector number 2. So we can set our cylinder to 0, our head to 0 and our sector to 2. Be careful when addressing cylinders as their 2 high bits are defined in bits 6 and 7 of `cl`.

In the register `dl` we must specify the storage unit which we want to read data from. Options `0x00` to `0x7F` correspond to floppy dics, while options `0x80` and higher correspond to hard drives.

In registers `es:bx` we use the segment:offset addressing method to specify where in memory we want to store the data we read.

```
    mov ah, 0x02
    mov al, 1         ; Number of sectors
    mov ch, 0
    mov cl, 2
    mov dh, 0
    mov dl, 0x80
    mov bx, 0x1000     ; Segment
    mov es, bx
    xor bx, bx         ; Offset
    int 0x13
```

Listing 2.2: Assembly to load the kernel

Finally, after having specified our parameters we can call `int 0x13` and let the BIOS do its job.

Bibliography