Let There Be Kernel: A Journey of Building an OS

Zeimpekis Vassilis

October 10, 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, helps you learn from a student's experience and perhaps even avoid some common mistakes.

While 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 expertise in low-level systems programming. By optimistically ignoring the warnings and diving in the field we immediately discovered 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 destills the knowledge of a student who did try to write an OS kernel and explains the reasoning behind their approach. Following someone else's streamlined learning process can enhance and accelerate your own understanding and help avoid common pitfalls.

The present work 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 a 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 complexity 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. Trying to challenge some of the architectural decisions of the present project is encouraged and it will be directly beneficial to your engineering skills.

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, structural and architectural decisions will need to be made. In the first chapters, the reasoning behind certain choices will be documented — 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 decided designs, emphasis will shift upon alternative designs, architectural comparisons, and analysis of the trade-offs they involve.

Chapter 2

Bootloader

In this chapter some fundamental concepts regarding the functioning of a bootloader will be explained and used in practice to load a placeholder kernel.

Before we start, it should be made clear that everything below applies to 32-bit, x86 architecture processors. The majority of the following concepts, however, are transferable to other common architectures.

2.1 Booting in Real Mode

Before we can understand the basics of a bootloader an introduction about the Real Mode of x86 processors is necessary.

When the computer turns on the x86 processor automatically enters **Real Mode**, which poses some significant limitations. Newer processors still support Real Mode to ensure backward compatibility, although it is now considered obsolete.

When in Real Mode the CPU is running by default in 16-bit mode, meaning we are expected to use 16-bit registers for our operations. While 32-bit registers are technically still available and can be utilized on newer machines, their usage is advised against for most beginner applications. Switching to 32-bit Protected Mode is an exception to this, as we will see later.

Another limitation of Real Mode is that we 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:

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

The CPU fetches the Segment value from a 16-bit segment register and the Offset value from a 16-bit address register.

Essentially, by using this convention we can only represent numbers that 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.

It should be made clear that the addresses used in Real Mode refer to physical addresses. This hinders us from protecting memory and defining its ownership because no process is prohibited from accessing any memory segments. This problem could be overcome by using a Global Descriptor Table in Protected Mode, however, for the purpose of this book we will adopt virtual memory via paging as a more modern approach. Virtual memory, along with paging, will be discussed later in this book but a brief explanation of how it helps with memory safety is given just to clarify why physical addressing is limiting.

When using virtual addresses, our Operating System is responsible of mapping every virtual address to a physical one. By this mapping, the operating system 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 they are translated into totally different physical addresses by the operating system, rendering both unable to access eachother's memory space.

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 there is no virtualization by default, we can take advantage of how memory is segmented to protect critical data from the user. This is achieved by defining different memory segments with different privileges (ring levels). The CPU decides if a piece of code has the privileges it tries to claim. This is determined by the segment through which we are accessing that memory (we will see that in practice). Memory management and CPU Operating Modes will have chapters of their own later in this book.

2.2 The Role Of The Bootloader

When the computer turns on the first program to take control is the BIOS. Among other things that do not concern us in this section, the BIOS transfers control to the bootloader which is then responsible for loading the OS. Modern systems use UEFI instead to overcome some limitations of the BIOS but for educational purposes 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 memory a bootloader conventionally resides upon. If the last two bytes of the MBR are the word 0xAA55 the BIOS identifies the storage device as a bootable device. The code below demonstrates we can sign the MBR by padding all bytes until byte 510 with zeros and then writing the word 0xAA55 to the last two bytes.

```
    times 510 - ($ - $$) db 0
    dw 0xAA55
```

Listing 2.1: MBR Signature

Once the bootable is determined control is transfered to the Stage-1 bootloader, located in the first 446 bytes of the MBR. In advanced operating systems the Stage-1 bootloader reads the partition table, located in the next 64 bytes of the MBR. This table contains four entries of 16 bytes each. Each entry describes a partition in the storage device. When the Stage-1 bootloader finds the active (bootable) partition it loads the volume boot sector of that partition (VBR) into memory and transfers control to it. The VBR contains information about the file system of that partition and the next stage of bootloading, which can differ between operating systems. However, for simplicity, in this chapter we will assume that the Stage-1 bootloader loads the kernel directly from the MBR.

Below is a table describing the layout of the MBR.

For the bootloader to run the BIOS loads it from the MBR to the memory address 0x7C00. This address has been traditionally used by BIOS software to load the bootloader. Bootloader developers assume this is where their bootloader will be loaded to ensure compatibility. The address 0x7C00 is ideal since it is way below the 1MiB accessible range and still leaves space below it for the interrupt vectors (they will be covered soon).

Region	**Offset (bytes)**	**Size**	**Notes**
Boot code & data	`0x000–0x1BD`	446 B	Stage-1 bootloader
Partition table entry #1	`0x1BE–0x1CD`	16 B	Status, CHS/LBA, sectors
Partition table entry #2	`0x1CE–0x1DD`	16 B	Status, CHS/LBA, sectors
Partition table entry #3	`0x1DE–0x1ED`	16 B	Status, CHS/LBA, sectors
Partition table entry #4	`0x1EE–0x1FD`	16 B	Status, CHS/LBA, sectors
Boot signature	`0x1FE–0x1FF`	2 B	`Contains the word 0x55AA`

Table 2.1: Master Boot Record (512 bytes) layout.

After control has been transferred to the bootloader it is now its job to load the kernel. In a later section we will suggest some techniques to overcome the 512-byte barrier and the limitations of the Real Mode.

To complete the picture, below is a table describing the layout of a partition table entry. Some concepts in the table will be explained later in this chapter.

Bytes	**Field**	**Description**
Byte 0	Status	`0x80` = bootable; `0x00` = non-bootable. Indicates if this partition is active (bootable).
Bytes 1-3	First CHS	Starting Cylinder-Head-Sector address (legacy BIOS format). Retained for backward compatibility; often ignored.
Byte 4	Type	Partition type identifier (e.g., `0x0B` = FAT32, `0x07` = NTFS).
Bytes 5-7	Last CHS	Ending Cylinder-Head-Sector address (legacy). Used by old BIOS systems.
Bytes 8-11	First LBA	Logical Block Address (LBA) of the first sector of the partition. Modern OSes use this instead of CHS.
Bytes 12-15	Total Sectors	Number of sectors in the partition (counted from First LBA). Determines partition size.

Table 2.2: Structure of a Partition Table Entry (16 bytes total).

2.3 Utilizing BIOS Interrupts

Before we can load our kernel we must understand how to interact with the BIOS. The BIOS exposes a set of interrupts that we can use to perform basic I/O operations.

Interrupts are usually defined as signals sent to the CPU to indicate that an event needs immediate attention. When an interrupt is triggered the CPU stops what it is doing and executes a handler function to deal with the event. Interrups are divided into software and hardware interrupts. Software interrupts are triggered by software instructions, while hardware are triggered by signals sent by hardware devices.

A very important advantage of using interrupts is that they expose a controllable interface to interact with critical hardware. This abstraction not only simplifies the development process and detaches the programming from the hardware specifics, but also allows for argument checking before execution. Since security is a major concern in such low-level operations, this is a very important feature. This concept will be extended after user and kernel modes are introduced.

In our case we will be using software interrupts to request services from the BIOS. Such services include reading from storage devices, printing to the screen and reading from the keyboard. For example the BIOS interrupt `0x10` offers video services.

In order to specify which function of the interrupt we want to use, we can pass parameters through CPU registers. For example, to print a character to the screen using BIOS interrupt `0x10` we can use the teletype output function by setting the `ah` register to `0x0E` and the `al` register to the ASCII value of the character we want to print. Finally, we can trigger the interrupt using the `int` instruction.

```
    mov ah, 0x0E        ; BIOS function: teletype output
    mov al, 'A'         ; Character to print
    mov bh, 0x00        ; Page number
    mov bl, 0x07        ; Text color (white)
    int 0x10            ; Call BIOS video interrupt
```

Listing 2.2: Simple BIOS interrupt call in assembly

2.4 LBA vs CHS Addressing

Before understanding how the bootloader will load our kernel 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 traditional hard drives use.

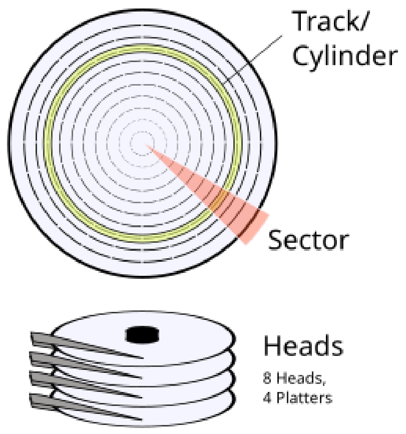

Figure 2.1: CHS addressing layout: cylinders, heads, and sectors

As shown above, hard drives are constituted of platters which are essentially circular disks. On both surfaces of these platters - both 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 **Cylinders·Platters·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 defines the smallest addressable unit upon a track which has a size of 512 bytes.

When writing 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 (sector numbering starts at 1). As we fill up all sectors up to sector 63 we can start filling up heads. When head 255 is filled, we start filling up cylinders. 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.

2.5 Loading The Kernel

We can start by dumping some code. Please do not be overwhelmed by it; 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 2.3: Simple bootloader start in assembly

Let's analyze this line by line.

First of all, `cli` is used to prevent the system from triggering maskable interrupts, requests for the CPU to stop what it is doing and execute some other instructions. Since interrupt handling has not been set up 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 physical memory is addressed in Real Mode. In 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 our 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, being in a controllable environment we can load our kernel into memory. BIOS interrupts will prove useful once again. On this occasion by calling `int 0x13` we ask the BIOS to load data from our hard drive into our memory. However, before triggering the interrupt we must pass some parameters through CPU registers.

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

Since interrupt `0x13` provides multiple storage device related functions 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 units) we want to read. For simplicity we will initially be reading just one.

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 discs, 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
    jc disk_error    ; Jump if carry flag is set (error occurred)
```

Listing 2.4: Assembly to load the kernel

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

Now, we need to ensure the read operation was performed successfully. Errors can be detected by checking the carry flag. In case of an error the BIOS sets the carry flag. We can use the `jc` instruction to jump to an error handling routine if the carry flag is set. The label `disk_error` is defined below.

```
disk_error:
    mov ah, 0x0E
    mov al, 'E'
    int 0x10
    hlt
```

Listing 2.5: Assembly to handle disk read errors

This routine uses the teletype output function of the video BIOS interrupt we introduced earlier to print the character 'E' to the screen, indicating an error has occurred. After printing the error message, the system halts using the `hlt` instruction.

2.6 Entering Protected Mode

A pattern might have started emerging since this is an introductory chapter, but some concepts below are just referred to and not explained. Every one of those will be thoroughly explained in next chapters.

Before jumping to our kernel code we will switch our CPU operating mode from Real Mode to **Protected Mode**. Technically, this is not a requirement, however, modern C compilers output 32-bit binaries. To prepare the ground for our C kernel code properly we will enter Protected Mode before jumping. By doing so, we enable accessing 4GiB of memory as well.

To enter Protected Mode x86 processors require us to have described a memory layout. We can achieve this by defining a Global Descriptor Table (GDT). Segment selectors can use this table to find their corresponding memory segments. Since our approach will use paging for memory management we will set up a very simple GDT in which the code and data segments overlap just to satisfy the CPU requirement.

```
gdt_start:
    dq 0x0000000000000000      ; null descriptor
    dq 0x00CF9A000000FFFF      ; code segment
    dq 0x00CF92000000FFFF      ; data segment
gdt_end:

gdt_desc:
    dw gdt_end - gdt_start - 1    ; size = total bytes - 1
    dd gdt_start                    ; address of the GDT
```

Listing 2.6: Defining a GDT

To pass this table to the CPU we use a descriptor, a pointer to a piece of memory in which we write the size and the location of the GDT.

```
    lgdt [gdt_desc]

    mov eax, cr0
    or eax, 1
    mov cr0, eax

    jmp 0x08:protected_mode_start
```

Listing 2.7: Loading GDT and Enabling Protected Mode

In the snippet above we load the descriptor of the GDT into the CPU and enable the Protected Mode option. Immediately after, we jump to the `protected_mode_start` label which includes 32-bit mode. We can ignore how and why we use a far jump for now.

Finally, we can tell the assembly compiler we are in 32-bit mode using the `[BITS 32]` directive and jump to our kernel. The location of the kernel has been defined when we loaded it from the disk using the int 0x13 instruction.

```
[BITS 32]
protected_mode_start:
    ; Set up segment selectors
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    ; Jump to kernel entry point
    jmp 0x08:0x10000
```

Listing 2.8: Initializing Registers and Jumping to Kernel

2.7 Build and Run

TODO

2.8 Two-Stage Bootloaders

TODO

Bibliography