Operating System Development

An Introduction to OS Development

based on the os-tutorial by cfenollosa

February 14, 2026

Contents

Introduction

Welcome to the journey of creating your own operating system. This text serves as a modern successor to previous tutorials, integrating rigorous engineering practices with the hands-on "learn by doing" philosophy.

We will start from the absolute basics—a computer booting up—and build our way up to a functional kernel with a shell, memory management, and multitasking.

Prerequisites

You should be comfortable with:

- Command-line interfaces (Linux/Unix shell).
- Basic C programming (pointers, memory).
- Basic understanding of computer architecture (registers, stack, memory).

The Environment

Before we write a single line of code, we must understand the tools of our trade. Operating system development requires a specific set of tools that are often different from standard application development.

The Philosophy of the Toolchain

In the early days of computing, a programmer wrote code specifically for the machine they were sitting at. There was no concept of "cross-compiling" because the hardware was the OS. As systems became more complex, we needed a way to build software for a "target" machine that might not even be running yet. This is the essence of OS development: you are building the foundation upon which everything else rests.

The Toolchain

To build an operating system, we need three primary tools:

1. **A Compiler (GCC):** To translate our C code into machine code.
2. **A Assembler (NASM):** To translate our assembly code into machine code.
3. **A Emulator (QEMU):** To run and test our operating system safely.

Why a Cross-Compiler?

One of the most common mistakes for beginners is trying to use their system's default compiler (e.g., ‘gcc‘ on Ubuntu) to build their kernel. This is problematic because your system's ‘gcc‘ assumes it is building programs for your current OS (Linux, macOS, etc.). It tries to link against standard libraries (‘glibc‘) and use headers that rely on system calls that *don't exist yet* in your OS.

Therefore, we use a **Cross-Compiler**. This is a version of GCC compiled specifically to generate code for a generic target (like `i686-elf`) without assuming any underlying operating system. This ensures that the generated binary is "freestanding"—it contains nothing but your code and a few essential support routines.

Linux

Kbuild and the Kernel Toolchain
In the modern Linux kernel, the build system is known as **Kbuild**. It handles the complex dependency tracking and compilation flags required for the massive codebase.
Linux developers also use cross-compilers extensively. For example, an x86_64 developer might build a kernel for an ARM-based Android device using a command like:
`make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-`

This ensures that the generated machine code is correct for the target processor, regardless of the host machine.

Setting Up

We will be using Linux as our host environment. If you are on macOS or Windows, it is highly recommended to use a Linux Virtual Machine (VM) or WSL2 (Windows Subsystem for Linux).

Installing Dependencies

On a Debian/Ubuntu-based system, you can install the necessary tools with:

```
sudo apt-get update
sudo apt-get install build-essential nasm qemu-system-x86 gdb
```

- **nasm:** The Netwide Assembler. We use this for our assembly files (`.asm`).
- **qemu-system-x86:** A fast emulator that simulates an x86 computer.
- **gdb:** The GNU Debugger, essential for inspecting what went wrong when your kernel crashes.

Building the Cross-Compiler

As discussed, we need a cross-compiler. Since pre-built binaries can be outdated or incompatible with your system, we will build `i686-elf-gcc` and `binutils` from source. We will install them in your **home directory** to avoid requiring `sudo` and to keep your system clean.

1. Preparation

First, install the host build dependencies. On Ubuntu/Debian:

```
sudo apt install build-essential bison flex libgmp3-dev libmpfr-dev
    libmpc-dev texinfo libisl-dev
```

Now, set up your environment variables. Add these to your `.bashrc` or `.zshrc` later:

```
export PREFIX="$HOME/opt/cross"
export TARGET=i686-elf
export PATH="$PREFIX/bin:$PATH"
```

2. Binutils

Binutils provides the assembler (`as`), linker (`ld`), and other tools. We'll use a modern version (e.g., 2.40+).

```
mkdir -p $HOME/src
cd $HOME/src
wget https://ftp.gnu.org/gnu/binutils/binutils-2.40.tar.gz
tar -xzf binutils-2.40.tar.gz
mkdir build-binutils
cd build-binutils
../binutils-2.40/configure --target=$TARGET --prefix="$PREFIX" --
   with-sysroot --disable-nls --disable-werror
make
make install
```

3. GCC (The Compiler)

Now we build GCC itself. We only need the C compiler for now. We will use a modern version (e.g., 13.1.0+).

```
cd $HOME/src
wget https://ftp.gnu.org/gnu/gcc/gcc-13.1.0/gcc-13.1.0.tar.gz
tar -xzf gcc-13.1.0.tar.gz
mkdir build-gcc
cd build-gcc
../gcc-13.1.0/configure --target=$TARGET --prefix="$PREFIX" --
   disable-nls --enable-languages=c,c++ --without-headers
make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

You may want to add the `i686elfgcc/bin` folder to `$PATH` by exporting in `.bashrc` (or `.zshrc`) to be persistent on other sessions.

Linux

Self-Hosting and Toolchains

A compiler is "self-hosting" when it can compile its own source code. When you build the Linux kernel, you are typically using a compiler that was itself built by another compiler.

In professional environments (and Linux distribution build farms), toolchains are often managed using tools like `crosstool-NG`, which automates the complex process of matching GCC, Binutils, and C library versions to ensure a stable and reproducible build environment.

Testing Your Setup

Let's verify that QEMU is working. We don't have an OS yet, but we can verify the emulator launches.

```
qemu-system-i386
```

You should see a window pop up looking for a bootable disk. Since we haven't provided one, it will likely say "No bootable device." This is good! It means the emulator is working.

0.1 The Roadmap

Our development process will follow these stages:

1. **Bootloader:** Write x86 assembly to instruct the BIOS to load our kernel.
2. **Real Mode to Protected Mode:** Transition from the 16-bit legacy environment to the 32-bit world.
3. **Kernel Entry:** Jump from assembly to C.
4. **Drivers:** Write code to talk to hardware (Screen, Keyboard, Timer).
5. **Interrupts:** Handle hardware events.
6. **Memory Management:** Allocate and free memory.
7. **Filesystem & Shell:** A user interface for our OS.

0.2 Knowledge Check

1. Why can't we just use the standard 'gcc' installed on our Linux machine to compile our kernel?
2. What is the role of 'nasm' in our toolchain?
3. What does QEMU provide that allows us to develop without constantly rebooting our physical machine?

Lab Challenge

Challenge: Toolchain Inspection
Verify your cross-compiler is working by creating a dummy C file and checking its target architecture:

```
echo "int main() { return 0; }" > test.c
i686-elf-gcc -ffreestanding -c test.c -o test.o
readelf -h test.o
```

Confirm that the "Machine" field in the output says `Intel 80386`. If it says `Advanced Micro Devices X86-64`, your cross-compiler is not correctly configured!

Chapter 1

The Boot Sector (Bare Bones)

1.1 The Logic: The 512-byte Constraint

In the 1970s and 80s, storage was incredibly expensive and limited. When the original IBM PC was designed, the BIOS designers decided that the very first sector of a disk (512 bytes) would be the "boot sector." This tiny space was all you had to load the rest of your operating system. This forced early programmers to be extremely clever with their assembly code—a tradition of optimization that continues in modern kernels where every byte of memory still counts.

1.2 The Theory: How a Computer Boots

When you press the power button, the computer doesn't know how to run an operating system. It relies on a piece of software built into the motherboard called the **BIOS** (Basic Input/Output System).

The BIOS performs a Power-On Self-Test (POST) and then looks for a bootable device (Floppy, CD, Hard Disk, USB). When it finds one, it looks for the specific signature `0x55` and `0xAA` in the last two bytes of the first 512-byte sector.

If found, it loads those 512 bytes into memory at address `0x7C00`. Why `0x7C00`? It was a strategic choice by IBM engineers: it's at the end of the first 32KB of RAM, leaving enough room for the OS to grow downwards without hitting the BIOS data area at `0x0000`.

Linux

UEFI vs. BIOS
Modern computers (post-2010) largely use **UEFI** (Unified Extensible Firmware Interface) instead of the legacy BIOS. UEFI is capable of reading file systems (like FAT32) and loading executable files directly.
However, legacy BIOS booting is still supported via CSM (Compatibility Support Module) and is excellent for learning because it forces you to understand the raw hardware constraints without the massive complexity of the UEFI specification.

1.3 Our First Boot Sector

The simplest boot sector is one that does nothing but loop forever. Open a file named `boot_sect_simple.asm` and enter the following:

```
; A simple boot sector program that loops forever
loop:
    jmp loop

; Padding and magic number
times 510-($-$$) db 0
dw 0xaa55
```

1.3.1 Breaking it Down

- `loop: jmp loop`: This is an infinite loop. `jmp` is the assembly instruction for "jump", and `loop` is a label pointing to the current instruction.
- `times 510-($-$$) db 0`: This is a NASM directive. It tells the assembler to fill the rest of the 510 bytes with zeros. `$` is the current position, and `$$` is the start of the section.
- `dw 0xaa55`: This writes the 2-byte magic number `0xAA55` (in little-endian format, so it's actually `55 AA` in the file).

1.4 Compiling and Running

To compile this into raw binary format (no headers, just machine code), use:

```
nasm -f bin boot_sect_simple.asm -o boot_sect_simple.bin
```

Then, run it with QEMU:

```
qemu-system-i386 boot_sect_simple.bin
```

1.5 Knowledge Check

1. What is the exact memory address where the BIOS loads the boot sector?
2. What happens if the last two bytes are `0x1234` instead of `0xAA55`?
3. What is the purpose of the `times` directive in this context?

Lab Challenge

Challenge: The BIOS Handover

Run QEMU with the `-d cpu` flag to see the state of the CPU at boot time:

```
qemu-system-i386 -d cpu -D qemu.log boot_sect_simple.bin
```

Open `qemu.log` and find the initial `EIP` (Instruction Pointer). It should be near `0x7C00`. Notice how the registers are mostly empty—the BIOS has just handed over the keys to the kingdom to your code.

Chapter 2

Printing to the Screen

Now that we can boot, let's make the computer say something. In the early stages of OS development, we don't have a screen driver, so we rely on the BIOS.

2.1 The Logic: The BIOS as an API

Before operating systems like DOS or Windows existed, the BIOS provided the only "API" for programmers. By triggering software interrupts, you could ask the motherboard to do complex things like print text, draw pixels, or read from a disk. This was the first "Hardware Abstraction Layer" (HAL). In this chapter, we use the BIOS as our library to communicate with the display hardware.

2.2 BIOS Interrupts

The BIOS provides a set of services accessible via **Interrupts**. An interrupt pauses the current execution and jumps to a predefined handler in the BIOS memory.

For video services, we use `int 0x10`. To print a character, we must:

1. Set the register `ah` to `0x0e` (Teletype Mode).
2. Set the register `al` to the ASCII character we want to print.
3. Call `int 0x10`.

2.3 Registers: The CPU's Scratchpad

Registers are small, lightning-fast storage locations inside the CPU. In 16-bit real mode, the general-purpose registers are 16 bits wide: `ax`, `bx`, `cx`, and `dx`.

Each can be split into two 8-bit registers. For example, `ax` consists of `ah` (high byte) and `al` (low byte).

2.4 Hello, World! in Assembly

Create `boot_sect_hello.asm`:

```
mov ah, 0x0e ; BIOS tele-type output
mov al, 'H'
int 0x10
mov al, 'e'
int 0x10
mov al, 'l'
int 0x10
int 0x10 ; 'l' is still in al
mov al, 'o'
int 0x10

jmp $ ; Loop forever ($ means current address)

times 510-($-$$) db 0
dw 0xaa55
```

Linux

System Calls vs. BIOS Interrupts
In modern Linux, user programs don't use BIOS interrupts to print text. Instead, they use **System Calls** (`syscall` on x86_64 or `int 0x80` on x86).

2.5 Knowledge Check

1. What register is used to select the BIOS video sub-function?
2. Why did we only set `ah` once in the "Hello" example?
3. What does the `$` symbol represent in the `jmp $` instruction?

Lab Challenge

Challenge: The Infinite Hello
Modify your bootloader to print a single character repeatedly in a loop. Use the `jmp` instruction to go back to the start of your print routine. How fast does it fill the screen? This demonstrates the raw power of direct hardware control before any OS overhead is added. Run it in QEMU and observe.

Chapter 3

Memory Organization

In the previous chapters, we printed characters by moving literals directly into registers. However, real programs need to store and retrieve data from memory. Understanding how the x86 processor addresses memory in Real Mode is crucial.

3.1 The Logic: The "Offset" Reality

In assembly, labels are just nicknames for numbers—specifically, the number of bytes from the start of the file. However, because our code is loaded at `0x7C00`, every label's "true" address is its position in the file **plus 0x7C00**. If we forget this, we'll try to read data from the wrong part of memory. This "absolute vs relative" addressing is a core concept in systems programming.

3.2 The 0x7C00 Problem

We know the BIOS loads our 512-byte boot sector at address `0x7C00`. This means that if we define a label in our code, its actual memory address will be its offset from the start of the file *plus* `0x7C00`.

Consider this code:

```
1 the_secret:
2     db "X"
3
4 ; ... later in the code ...
5 mov al, [the_secret] ; This will likely fail!
```

If `the_secret` is at the 10th byte of our file, the assembler thinks its address is `0x0A`. But in RAM, it's actually at `0x7C0A`. The CPU will look at address `0x0A` (which contains BIOS data) instead of our "X".

3.3 The [org] Directive

We can solve this in two ways:

1. Manually add `0x7C00` to every memory access (tedious and error-prone).
2. Use the `[org 0x7C00]` directive at the top of our assembly file.

The `org` (origin) directive tells the assembler to add a specific offset to all internal label calculations.

```
[org 0x7c00]
mov ah, 0x0e
mov al, [the_secret] ; Now this works!
int 0x10

jmp $

the_secret:
    db "X"

times 510-($-$$) db 0
dw 0xaa55
```

Linux

Virtual Memory and Relocation
In modern Linux, processes don't worry about where they are loaded in physical RAM. This is thanks to **Virtual Memory**. Every process thinks it starts at a standard address (e.g., `0x400000` for 64-bit ELF binaries).
The Memory Management Unit (MMU) of the CPU, managed by the Linux kernel, translates these virtual addresses into physical ones on the fly using **Page Tables**. Furthermore, Linux uses **ASLR** (Address Space Layout Randomization) to load programs at random offsets to make it harder for attackers to exploit memory-based vulnerabilities. This is like a dynamic version of the `org` directive!

3.4 Knowledge Check

1. Why doesn't `mov al, the_secret` (without brackets) work to print the character 'X'?
2. What is the difference between `mov al, [0x7c10]` and `mov al, 0x7c10`?
3. How does the `[org]` directive change the generated machine code? (Hint: It doesn't!)

Lab Challenge

Challenge: Memory Scavenger Hunt
Write a boot loader that prints the byte stored at memory address `0x0000:0x0475`. This specific address in the BIOS Data Area contains the number of disk drives detected by the system. Use your knowledge of memory addressing to retrieve and print this value (you'll need to convert the numeric value to an ASCII digit by adding `0x30`).

Chapter 4

The Stack

As we write more complex programs, we need a way to store temporary data and manage function calls. This is where the **Stack** comes in.

4.1 The Logic: Why Downwards?

One of the most confusing things for beginners is that the stack grows **downwards** (towards lower memory addresses). This design choice dates back to the earliest computers where memory was extremely limited. By having the "Heap" (dynamic data) grow upwards and the "Stack" (function data) grow downwards from the opposite end of memory, they could both expand until they met in the middle. This maximized the use of available RAM.

4.2 How the Stack Works

The x86 stack is a LIFO (Last-In, First-Out) data structure. Two registers manage it:

- **bp (Base Pointer):** Points to the "bottom" (highest address) of the stack.
- **sp (Stack Pointer):** Points to the "top" (lowest address) where the next value will be pushed.

When you `push` a value:

1. `sp` is decremented (by 2 bytes in 16-bit mode).
2. The value is stored at the address in `sp`.

When you `pop` a value:

1. The value at `sp` is copied to a register.
2. `sp` is incremented.

4.3 Setting Up the Stack

In the bootloader, we must manually initialize `bp` and `sp`. We should pick an address far away from our code (`0x7C00`) so they don't collide. `0x8000` is a safe choice.

```
mov bp, 0x8000
mov sp, bp

push 'A'
push 'B'
push 'C'

pop ax
mov al, ah ; Note: push/pop work on 16-bit words (ax), not 8-bit (al)
; ... print al ...
```

Linux

Kernel Stacks and Guard Pages
In Linux, every task (thread/process) has its own kernel stack (usually 8KB or 16KB). If a function calls itself too many times (infinite recursion), the stack will "overflow."

4.4 Knowledge Check

1. If `bp` is `0x8000` and you push a 16-bit value, what is the new value of `sp`?
2. Why does the stack grow downwards instead of upwards?
3. Can you push a single 8-bit register like `al` onto the stack?

Lab Challenge

Challenge: Stack Reversal
Write a boot sector that pushes the characters of your name onto the stack one by one. Then, use a loop to pop them off and print them. Because of the LIFO nature of the stack, your name will be printed in reverse! This is a classic exercise in understanding stack mechanics.

Chapter 5

Functions and Strings

As our bootloader grows, we need to organize our code into reusable blocks. In assembly, we use **labels** and the **stack** to implement functions.

5.1 The Logic: The Birth of Modularity

Early programmers wrote "spaghetti code"—one long sequence of instructions with frequent jumps. As programs grew, this became impossible to manage. The invention of the `call` and `ret` instructions allowed for the creation of "subroutines." This was a revolutionary step that led directly to modern modular software architecture.

5.2 Strings in Assembly

A string is just a sequence of bytes in memory. By convention (borrowed from C), we terminate strings with a null byte (`0x00`) so our code knows where the string ends.

```
1 MY_STRING:
2     db 'Hello, OS World!', 0
```

5.3 Control Flow: Loops and Conditionals

Assembly uses the `cmp` (compare) instruction followed by conditional jumps like `je` (jump if equal), `jne` (jump if not equal), or `jl` (jump if less than).

5.4 The Function Mechanism: call and ret

To call a function, we use the `call` instruction. This does two things:

1. Pushes the address of the *next* instruction onto the stack (the return address).
2. Jumps to the function label.

The `ret` instruction at the end of the function pops that address from the stack and jumps back to it.

5.5 A Reusable Print Function

Here is a function that prints a null-terminated string. It expects the string's address in the `bx` register.

```
print:
    pusha            ; Save all registers to the stack
.loop:
    mov al, [bx]     ; Move char at [bx] into al
    cmp al, 0        ; Is it the null terminator?
    je .done

    mov ah, 0x0e     ; BIOS tty mode
    int 0x10         ; Print character

    add bx, 1        ; Move to next character
    jmp .loop
.done:
    popa             ; Restore registers
    ret
```

Linux

The C Calling Convention (ABI)
In modern Linux, when a C function calls another, it follows a strict **Application Binary Interface (ABI)**.

5.6 Knowledge Check

1. What would happen if a function forgot to call `ret`?
2. Why is `pusha` important at the beginning of a function?
3. How does the CPU know where to return to after a `call`?

Lab Challenge

Challenge: The Hexadecimal Bridge
Implement a function `print_hex` that takes a 16-bit value in `dx` and prints it in hex format (e.g., `0x12FE`). This requires you to master bitwise operations (`and`, `shr`) and ASCII conversion. Look at `boot/print_hex.asm` in the repository for the definitive implementation used in our kernel. Try to explain why we use the `ror` instruction in the loop.

Chapter 6

Segmentation

Real Mode has a major limitation: 16-bit registers can only address 2^{16} = 64 KB of memory. To access more (up to 1 MB), x86 uses **Segmentation**.

6.1 The Logic: The 8086 Hack

When Intel designed the 8086 processor, they wanted to address 1 MB of RAM, which requires 20 bits. However, they only had 16-bit registers. Their solution was "segmentation": combining two 16-bit registers to create a 20-bit address. While brilliant at the time, it became the "original sin" of x86 architecture, leading to decades of complexity that we are still dealing with today.

6.2 Segment:Offset Addressing

The CPU calculates a physical address using a **Segment Register** and an **Offset**:

$$\text{Physical Address} = (\text{Segment} \times 16) + \text{Offset}$$

For example, if the Data Segment (`ds`) is `0x7C0` and the offset is `0x0010`, the physical address is:

$$(0x7C0 \times 16) + 0x0010 = 0x7C00 + 0x0010 = 0x7C10$$

6.3 The Segment Registers

- **cs:** Code Segment (for instructions).
- **ds:** Data Segment (for variables).
- `ss`: Stack Segment (for the stack).
- **es, fs, gs:** Extra segments for general use.

6.4 Important Note on segments

You cannot move a literal value directly into a segment register. You must move it into a general-purpose register first:

```
mov ax, 0x7c0
mov ds, ax      ; Valid
; mov ds, 0x7c0 ; Invalid!
```

Linux

The Death of Segmentation
In 64-bit mode (long mode), Linux treats memory as one giant "flat" space. The segment registers `cs, ds, es, ss` are mostly ignored or forced to 0.

6.5 Knowledge Check

1. What is the physical address of `0x1000:0x1234`?
2. Why was segmentation invented instead of just using 32-bit registers from the start?
3. How does `[org 0x7c00]` relate to segmentation?

Lab Challenge

Challenge: The Segment Overlap
Because of the way physical addresses are calculated, multiple (Segment:Offset) pairs can point to the same physical address. Find three different pairs that all point to `0x7C00`. Write a bootloader that sets `ds` to one of these values and successfully reads data using an offset of 0. This experiment proves that memory is just one flat array, despite the segment "lenses" we use to look at it.

Chapter 7

Reading from Disk

Our OS will eventually grow much larger than 512 bytes. To load our kernel, we must use the BIOS to read subsequent sectors from the disk into memory.

7.1 The Logic: The Spinning Platter

In the 1980s, disks were literal spinning magnetic platters. To read data, a physical arm had to move to a specific "Cylinder," wait for the disk to spin to the right "Sector," and use a specific "Head" for the top or bottom of the platter. This physical reality defined the CHS addressing system. Even today, though we use SSDs with no moving parts, the legacy of CHS still exists in the BIOS and partition tables.

7.2 CHS Addressing

In the early days of PCs, disks were addressed by their physical geometry:

- **Cylinder:** The concentric circle on the platter.
- **Head:** Which side of the platter (top or bottom).
- **Sector:** The specific 512-byte slice of a track.

Crucial Note: Sectors are 1-indexed (they start at 1), while cylinders and heads are 0-indexed. Our bootloader is at Cylinder 0, Head 0, Sector 1.

7.3 BIOS Interrupt 0x13

To read from disk, we use `int 0x13` with `ah = 0x02`.

- `al`: Number of sectors to read.
- `ch`: Cylinder number.
- `cl`: Sector number (1-63).
- `dh`: Head number.

- `dl`: Drive number (BIOS sets this for us).
- `es:bx`: Buffer address where data will be stored.

7.4 The Carry Bit

If a BIOS disk operation fails, it sets the **Carry Flag (CF)** in the CPU's status register. We can check this using the `jc` (Jump if Carry) instruction.

```
disk_load:
    pusha
    mov ah, 0x02     ; BIOS read function
    mov al, dh       ; Read 'dh' sectors
    mov cl, 0x02     ; Start from sector 2
    mov ch, 0x00     ; Cylinder 0
    mov dh, 0x00     ; Head 0
    ; dl is set by the BIOS
    int 0x13
    jc disk_error    ; Jump if the Carry Flag is set
    popa
    ret
```

Linux

LBA and Block Devices
Modern operating systems like Linux have long abandoned CHS addressing.

7.5 Knowledge Check

1. Why do sectors start at 1 while everything else starts at 0?
2. What happens to the data already in memory at `es:bx` when you perform a disk read?
3. What is the purpose of the `dl` register in disk operations?

Lab Challenge

Challenge: The Sector Scrutineer
In our project, the disk loading routine is abstracted in `boot/disk.asm`. Use this routine to load 2 sectors starting from sector 2 into memory address `0x9000`. To verify success, use `db` to place a unique "secret" value at the very beginning of the second sector of your disk image (after the 512-byte boot sector), and have your bootloader print it. This is exactly how we will load our kernel later!

Chapter 8

32-bit Mode and VGA Memory

Until now, we have been working in **16-bit Real Mode**. This is the legacy mode that x86 CPUs start in for backward compatibility. To unlock the full potential of modern processors, we must transition to **Protected Mode**.

8.1 The Logic: Breaking the 1MB Barrier

In the mid-1980s, the 1 MB memory limit of the 8086 was becoming a bottleneck. The 80386 processor introduced "Protected Mode," which allowed for 32-bit addressing (up to 4 GB of RAM) and hardware-level memory protection. This transition marks the point where computers moved from single-tasking machines (like the early PC) to multi-tasking, multi-user systems. It is the "Rubicon" of OS development.

8.2 What is Protected Mode?

Protected Mode introduces several critical features:

- **32-bit Registers:** We can use `eax`, `ebx`, etc., allowing for 4 GB of direct memory addressing.
- **Memory Protection:** The hardware can prevent one program from accessing another's memory.
- **Virtual Memory:** Support for paging.

However, entering Protected Mode has a major downside: **we lose BIOS interrupts**. No more `int 0x10` for printing or `int 0x13` for disk access. We must write our own hardware drivers.

8.3 VGA Text Mode

The simplest way to print in Protected Mode is to write directly to **VGA Video Memory**. In text mode, this memory is mapped to physical address `0xB8000`.

The screen is an 80x25 grid. Each character on the screen takes 2 bytes of memory:

1. **ASCII Byte:** The character to display.
2. **Attribute Byte:** The foreground and background colors.

The formula to find the memory address of a character at (row, col) is:

$$\text{Address} = 0xB8000 + 2 \times (\text{row} \times 80 + \text{col})$$

```
[bits 32]
VIDEO_MEMORY equ 0xb8000
WHITE_ON_BLACK equ 0x0f

print_string_pm:
    pusha
    mov edx, VIDEO_MEMORY
.loop:
    mov al, [ebx]         ; ebx holds the string address
    mov ah, WHITE_ON_BLACK
    cmp al, 0
    je .done
    mov [edx], ax         ; Write char + attribute to VGA memory
    add ebx, 1
    add edx, 2
    jmp .loop
.done:
    popa
    ret
```

Linux

The Linux Framebuffer
Early in the boot process, Linux might use VGA text mode. However, modern Linux kernels use a **Framebuffer** (`fbdev` or DRM/KMS).

8.4 Knowledge Check

1. Why do we lose BIOS interrupts in Protected Mode?
2. What is the physical address of the character at the second row, third column?
3. What does the attribute byte `0x0F` represent?

Lab Challenge

Challenge: The 32-bit Scribe
Our 32-bit print routine is located in `boot/32bit_print.asm`. Study it and modify it to print a string with a different color attribute (e.g., green text on a blue background). You'll need to understand how the attribute byte is structured. Then, use this routine to print "Landed in Protected Mode" as soon as you switch CPU modes.

Chapter 9

The Global Descriptor Table (GDT)

Before we can flip the switch to 32-bit mode, we must define how memory is handled. In Protected Mode, segmentation is replaced (or rather, redefined) by the **Global Descriptor Table (GDT)**.

9.1 The Logic: From Fixed to Flexible

In Real Mode, segments were fixed at 64 KB and their base address was hardcoded as `Segment * 16`. This was simple but inflexible and offered no security. The GDT changed this by making segments "descriptors." A descriptor defines exactly where a segment starts, how long it is, and who is allowed to access it. This allowed the CPU to enforce "privilege levels" (Rings), which is why we call it "Protected Mode."

9.2 The GDT Structure

The GDT is an array of 8-byte **Segment Descriptors**. Each descriptor tells the CPU:

- **Base Address:** Where the segment starts (32 bits).
- **Limit:** How large the segment is (20 bits).
- **Access Byte:** Permissions (Read/Write, Executable, Privilege Level).
- **Flags:** Granularity (1B or 4KB units) and size (16-bit or 32-bit).

9.3 The Flat Memory Model

In modern OS development, we don't want to deal with segments overlapping or being restricted. We use a **Flat Memory Model**, where we define two segments (Code and Data) that both cover the entire 4 GB of addressable memory (Base 0, Limit 0xFFFFF).

```
1 gdt_start:
2     dq 0x0              ; The mandatory null descriptor
3 gdt_code:               ; Code segment descriptor
4     dw 0xffff           ; Limit (bits 0-15)
5     dw 0x0              ; Base (bits 0-15)
```

```
    db 0x0              ; Base (bits 16-23)
    db 10011010b        ; Access byte
    db 11001111b        ; Flags + Limit (bits 16-19)
    db 0x0              ; Base (bits 24-31)
gdt_data:               ; Data segment descriptor
    ; ... similar to code, but with different access byte (10010010
       b)
gdt_end:

gdt_descriptor:
    dw gdt_end - gdt_start - 1
    dd gdt_start
```

Linux

GDT in Modern Linux
Even though 64-bit Linux uses paging for almost everything, the GDT is still required by the x86 architecture. Linux sets up a minimal GDT with entries for Kernel Code, Kernel Data, User Code, and User Data.

9.4 Knowledge Check

1. Why is a "Null Descriptor" required at the start of the GDT?
2. What is the purpose of the "Granularity" bit in a segment descriptor?
3. In a flat memory model, what is the difference between the Code and Data segments?

Lab Challenge

Challenge: The Descriptor Architect
The GDT implementation we use is in `boot/gdt.asm`. It defines a simple flat memory model. Your task is to calculate the 8-byte value for a segment descriptor that covers only the first 1 MB of memory, is readable, but NOT executable. Compare your calculated bytes with the ones in `gdt.asm`. This manual calculation is essential for understanding how the hardware actually "sees" your memory configuration.

Chapter 10

The Switch to Protected Mode

We have our GDT ready. Now it's time to "flip the switch" and enter the 32-bit world. This process is delicate because the CPU changes how it interprets every instruction and memory address instantly.

10.1 The Logic: The Pipeline Paradox

Modern CPUs are incredibly fast because they "pre-fetch" dozens of future instructions and decode them in advance. This is called a **Pipeline**. When we switch from 16-bit to 32-bit mode, the instructions already in the pipeline are "garbage" because they were decoded as 16-bit code. The far jump (`jmp CODE_SEG:init_pm`) is more than just a jump; it is a "pipeline flush" that forces the CPU to throw away its pre-fetched instructions and start fresh in the new 32-bit mode.

10.2 The 7-Step Switch

To safely transition to Protected Mode, we follow these steps:

1. **Disable Interrupts:** We use `cli`. Since BIOS interrupts won't work in 32-bit mode, and we haven't set up our own Interrupt Descriptor Table (IDT) yet, an interrupt would crash the system.
2. **Load GDT:** Use the `lgdt` instruction to tell the CPU where our GDT is.
3. **Set CR0 Bit:** The CPU has control registers. Bit 0 of `cr0` (the PE bit) enables Protected Mode.
4. **The Far Jump:** This flushes the CPU pipeline.
5. **Update Segment Registers:** Set `ds, ss, es, fs, gs` to point to our Data Segment.
6. **Update Stack:** Move the stack to a safe, 32-bit memory location.
7. **Call Entry Point:** Jump to our first 32-bit code.

```
1 [bits 16]
2 switch_to_pm:
```

```
    cli                     ; 1. Disable interrupts
    lgdt [gdt_descriptor]    ; 2. Load GDT
    mov eax, cr0
    or eax, 0x1             ; 3. Set bit 0 in cr0
    mov cr0, eax
    jmp CODE_SEG:init_pm    ; 4. Far jump to 32-bit code

[bits 32]
init_pm:
    mov ax, DATA_SEG        ; 5. Update segment registers
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov ebp, 0x90000        ; 6. Update stack
    mov esp, ebp

    call BEGIN_PM           ; 7. Call 32-bit entry point
```

Linux

Pipelining and Pre-fetching
In the Linux kernel source, this transition happens in `arch/x86/boot/pm.c` and `arch/x86/boot/pmjump.S`.

10.3 Knowledge Check

1. What does the `cli` instruction do, and why is it necessary here?
2. Why can't we just use a regular `jmp` instead of a far jump?
3. What happens to the `cs` register after the `jmp CODE_SEG:init_pm`?

Lab Challenge

Challenge: The 32-bit Milestone
The switching logic is implemented in `boot/switch_pm.asm`. Study this file and integrate it into your main boot sector. Your goal is to print "Started in 16-bit Mode", perform the switch, and then print "Landed in 32-bit Protected Mode" using the VGA routine from Chapter 9. This is the ultimate test of your bootloader's stability.

Chapter 11

C for Kernels

Writing an OS entirely in assembly is possible but impractical. We want to use a higher-level language like C. However, "Kernel C" is different from "Application C".

11.1 The Philosophy: The Unix Revolution

When Unix was first written in the early 1970s, it was almost entirely in Assembly. Dennis Ritchie and Ken Thompson realized that to make Unix portable, they needed a language that was "close to the hardware" but expressive enough to manage complex logic. This led to the creation of the C language. Today, C is the "lingua franca" of systems programming because it allows us to control the hardware with the elegance of a high-level language.

11.2 The Freestanding Environment

When you write a normal C program, you have access to the **Standard Library**. In OS development, we use the `-ffreestanding` flag. This tells the compiler that the standard library might not be available. We only get a few headers like `<stdint.h>`.

11.3 Why kernel_main and not main?

The C standard expects a function named `main` to be the entry point. To avoid this and clearly distinguish our entry point, we use `kernel_main()`.

Linux

The No-Libc World
The Linux kernel does not use `glibc`. Instead, it implements its own versions of essential functions like `printk()`.

Lab Challenge

Challenge: The C-Side
Our main kernel entry point is in `kernel/kernel.c`. Your task is to implement a simple loop in this file that writes characters to the VGA memory (starting at `0xB8000`) one by one. This confirms that your C compiler is producing code that can correctly interact with hardware memory. Use pointers!

Chapter 12

The Linker and Object Files

12.1 The Logic: Resolving the Mystery

The compiler processes one file at a time. If ‘main.c‘ calls a function in ‘util.c‘, the compiler doesn’t know where that function will eventually live in memory. It leaves a "placeholder" or a "symbol" in the object file. The Linker’s primary job is to act as a matchmaker: it finds the actual memory addresses for all these symbols and stitches the object files together into a coherent whole.

12.2 The Linker’s Job

The Linker (`ld`) takes multiple object files and merges them. It resolves symbols and decides memory layout.

Linux

The vmlinux ELF
The final Linux kernel image is an ELF file called `vmlinux`.

Lab Challenge

Challenge: The Symbol Inspector
Compile your `kernel.c` but don’t link it yet. Use the command `i686-elf-nm kernel.o` to see the symbol table. Find your `kernel_main` symbol. This knowledge is invaluable when troubleshooting "Undefined Reference" errors during the linking phase.

Chapter 13

The Barebones Kernel

13.1 The Logic: The Glue Code

When a C function is called, it expects the stack to be set up in a specific way, and it expects to be called from code that knows the ABI. Our bootloader is raw assembly. The `kernel_entry.asm` file acts as the glue (or "shim") that provides a clean entry point for the CPU to transition from the raw world of the bootloader into the structured world of C.

13.2 The Entry Point

Create `kernel_entry.asm`:

```
[bits 32]
[extern kernel_main] ; Defined in kernel.c
call kernel_main     ; Jump to the C function
jmp $                ; Hang if kernel returns
```

13.3 Knowledge Check

1. Why is the `-ffreestanding` flag necessary?
2. What is an `extern` label in assembly?
3. Why do we link the kernel at `0x1000` instead of `0x0`?

Lab Challenge

Challenge: The Kernel Concatenator
In our project, we use a `Makefile` to handle the concatenation, but doing it manually once is instructive. Use `cat` to merge your `bootsect.bin` and `kernel.bin`. Use a hex editor to find the exact byte where your bootloader ends and your kernel begins. Verify that the kernel code starts at exactly byte 512.

Chapter 14

Automating with Makefiles

14.1 The Philosophy: Declarative Building

In the early days, programmers wrote long shell scripts to compile their projects. This was slow because it recompiled everything every time. Stuart Feldman created `make` in 1976 at Bell Labs to solve this. Instead of telling the computer *how* to build (imperative), you tell it what the *dependencies* are (declarative). Make then calculates the most efficient way to build the project, only recompiling files that have changed.

14.2 The Makefile Structure

A Makefile consists of **Rules**:

```
target: dependencies
    command
```

Linux

The Linux Makefile
The Linux kernel Makefile is one of the most complex pieces of software engineering in existence.

Lab Challenge

Challenge: The Makefile Mechanic
Our project's `Makefile` is in the root directory. Your task is to add a new rule called `disassemble` that uses `i686-elf-objdump` to show the assembly code of your linked `kernel.bin`. This is a crucial skill for verifying that your C code is being translated into the machine instructions you expect.

Chapter 15

Hardware I/O with Ports

15.1 The Logic: The Second Address Space

The x86 architecture is unusual because it has a completely separate address space for hardware devices, distinct from main RAM. This is called **Port-Mapped I/O**. Originally, this was done to simplify motherboard wiring—hardware could be controlled without interfering with the limited memory space available for programs. Today, while most high-speed hardware uses memory mapping, legacy devices like the keyboard controller and VGA registers still rely on these 16-bit ports.

15.2 Port-Mapped I/O

Each hardware device is assigned a "Port Number" (from 0 to 65535). We use two special assembly instructions to talk to these ports: `in` and `out`.

Linux

MMIO vs. PMIO
Most modern hardware (PCIe cards, NVMe drives) uses **Memory-Mapped I/O (MMIO)**.

Lab Challenge

Challenge: The Cursor Mover
Implement a function `set_cursor(int offset)` in `drivers/screen.c`. It should calculate the high and low bytes of the offset and write them to the VGA ports `0x3D4` and `0x3D5` to move the blinking cursor. This is your first real "driver" code that controls hardware behavior through software.

Chapter 16

The Video Driver

16.1 The Logic: Abstraction Layers

A kernel shouldn't have to worry about ASCII codes or memory addresses every time it wants to log a message. The Video Driver provides an **Abstraction Layer**. By creating functions like `kprint()`, we allow the rest of the kernel to treat the screen as a high-level console, hiding the messy details of VGA memory and attribute bytes.

16.2 The kprint API

Our driver provides `kprint_at`, `kprint`, and `clear_screen`.

Linux

Escape Sequences (ANSI)
Modern terminal drivers in Linux interpret **ANSI Escape Sequences**.

Lab Challenge

Challenge: The Colorized Kernel
Our video driver is in `drivers/screen.c`. Your task is to implement `kprint_at_color(char *message, int col, int row, char attr)`, which allows you to print strings in different colors. Use this to print "ERROR" messages in red and "SUCCESS" messages in green. This makes your kernel much more user-friendly.

Chapter 17

Screen Scrolling

17.1 The Logic: The TTY Legacy

In the days of physical Teletype (TTY) machines, text was printed on a continuous roll of paper. When the paper reached the bottom, the user just unrolled more. Computers mimicked this behavior by "scrolling" the screen content up. This is a purely software-driven illusion: we are moving bytes from one part of video memory to another to simulate motion.

17.2 The Scrolling Logic

Scrolling in VGA text mode is usually done by shifting every row of characters up by one and then clearing the last row.

Lab Challenge

Challenge: The Smooth Scroller
Update your `print_char` in `drivers/screen.c` to handle the newline character (`\n`). It should set the cursor to the beginning of the next row. If it was already on the last row, it should trigger the scrolling logic by calling `memory_copy` to shift the rows. This is essential for a working shell.

Chapter 18

Interrupts and the IDT

18.1 The Logic: The CPU as an Event-Driven Machine

Most of the time, an operating system does nothing—it waits. It waits for you to press a key, for a timer to tick, or for data to arrive from the network. This "waiting" is managed by **Interrupts**. An interrupt is a signal from hardware that demands the CPU's immediate attention. Instead of the CPU "polling" (checking hardware repeatedly), the hardware "interrupts" the CPU. This event-driven model is what makes multitasking possible.

18.2 The Interrupt Descriptor Table (IDT)

The IDT maps interrupt numbers to handlers. Gates 0-31 are for CPU exceptions.

Crucial Fix: In modern systems, we must pass the CPU state to our C handler as a **pointer** (`registers_t *`).

Linux

Top Halves and Bottom Halves
Linux splits interrupt handling into two parts to keep the system responsive.

Lab Challenge

Challenge: The Great Exception
Our interrupt handlers are installed in `cpu/isr.c`. Your task is to intentionally trigger a "Divide by Zero" exception in `kernel/kernel.c`. Verify that your OS doesn't just crash silently, but instead prints "Division By Zero" and the register dump using your pointer-based `isr_handler`. This proves your IDT is correctly installed and accessible.

Chapter 19

Remapping the PIC

19.1 The Logic: Legacy Hardware Coordination

The 8259 PIC was designed in the late 1970s for the 8085 processor. When the IBM PC was designed, they used two of these chips to handle 15 IRQs. However, they chose interrupt vectors that overlapped with the CPU's internal exceptions. This design flaw persists in every PC to this day. Remapping the PIC is our way of cleaning up after the hardware engineers of 1981.

19.2 Communicating with the PIC

We remap the PIC so that its interrupts start at `0x20`.

Linux

APIC and MSI
Modern systems use the **APIC** and **MSI**, bypassing the legacy PIC entirely.

Lab Challenge

Challenge: The IRQ Verification
The PIC remapping code is in `cpu/isr.c`. Your task is to implement a temporary handler for IRQ 1 (the keyboard). Once your PIC is remapped, pressing a key should trigger your new handler at interrupt vector `0x21` (32 + 1). Use `kprint` inside the handler to confirm that the IRQ is being correctly routed to the new vector.

Chapter 20

The Hardware Timer

20.1 The Logic: The Pulse of the Machine

An operating system needs a sense of time. Without it, we can't tell how long a process has been running, we can't provide a `sleep()` function, and we can't implement preemptive multitasking. The **PIT** (Programmable Interval Timer) provides this pulse. By setting it to tick 100 times a second, we create a "system tick" that allows the kernel to maintain order.

20.2 Configuring the PIT

The PIT is configured via its command and data ports.

Linux

Tickless Kernels
Modern Linux supports **CONFIG_NO_HZ** to save power.

Lab Challenge

Challenge: The Heartbeat Monitor
Our timer implementation is in `cpu/timer.c`. Your task is to calculate the divisor required for a 1000 Hz timer (1 tick per millisecond). Update the `init_timer` call in `kernel/kernel.c` and implement a `get_ticks()` function that returns the total number of milliseconds since the OS started. Use this to print a message every 5 seconds.

Chapter 21

The Keyboard and Shell

21.1 The Logic: From Pulses to Commands

Keyboard input is surprisingly complex. When you press a key, the keyboard controller sends a "scancode"—a raw number representing the physical key. It doesn't know about Shift, Caps Lock, or "A" vs "a". Our driver must track the state of "modifier keys" and translate these scancodes into ASCII characters. Only then can our shell take these characters and turn them into commands that the OS understands.

21.2 Scancodes and Translation

Our keyboard driver translates scancodes to ASCII.

Linux

The TTY Subsystem
In Linux, the TTY layer handles "canonical mode" (line-buffering).

Lab Challenge

Challenge: The Command Handler
Our shell logic is in `kernel/shell.c`. Your task is to implement a new command called `"VERSION"` that prints the current version of your OS and the name of the professor (you!). This involves adding a string comparison in the command parser and a corresponding action. This is how you will eventually build a full suite of system utilities.

Chapter 22

Memory Management (kmalloc)

22.1 The Logic: The Finite Frontier

Physical RAM is a finite resource. In the early boot stages, we can just hardcode where things go. But once we have multiple processes, we need a way to dynamically allocate memory and, eventually, free it. Our `kmalloc` is the most primitive form of a memory manager—a "bump allocator" that just moves a pointer forward. While simple, it is the essential first step toward a full heap manager.

22.2 The Primitive kmalloc

Our allocator starts at a safe address and moves forward.

Linux

Buddy Systems and SLABs
Linux uses the **Buddy System** and **SLAB Allocator**.

Lab Challenge

Challenge: The Memory Logger
Our memory allocator is in `libc/mem.c`. Your task is to modify `kmalloc` to print a debug message every time it is called, showing the requested size and the physical address being returned. Then, use your shell's `"MALLOC"` command to allocate memory and verify the addresses are incrementing correctly. Does the 4KB alignment work as expected?

Chapter 23

The Road Ahead

23.1 The Philosophy: The Professional Edge

What separates a hobby project from a professional kernel is attention to detail and adherence to standards. By applying the "Fixes" (pointer-based register passing, ffreestanding, proper stack alignment), we have moved beyond simple tutorials and into the realm of real systems engineering. These practices are the same ones used by engineers at Microsoft, Google, and Apple every day.

23.2 The Importance of the "Fixes"

We have ensured ABI compatibility, stack alignment, and portable types.

23.3 What's Next?

Paging, Multitasking, File Systems, and User Mode.

Lab Challenge

Challenge: The Codebase Audit
Your final task is to perform a "Security Audit" of your codebase. Look at every function that takes a string input (like `kprint` or `append`). Does it check for buffer overflows? If you enter a 1000-character command in your shell, will it crash your OS? Fixing these vulnerabilities is the first step toward building a secure, production-ready operating system.

References

- *The Little Book About OS Development* (https://littleosbook.github.io)
- *OSDev Wiki* (https://wiki.osdev.org)
- *Intel 64 and IA-32 Architectures Software Developer Manuals*
- *The Linux Kernel Archives* (https://kernel.org)