MyOS

Rationale

“How hard could it be to make my own OS?” was a thought that struck me after attending an OS lecture one Thursday. Spurred by the idea, I quickly set forth trying to figure out the exact steps I would need to take.

At the same time, after many years of using MacOS and Windows as my main operating systems, I had always wanted to explore Linux after hearing about how much easier it made setting up your “own” environment and installing tools and dependencies.

Hence, I started on this journey of making MyOS in Linux.

Installing Prerequisites

Installing Ubuntu

What I assumed would be a simple process of making a bootable drive using Rufus to dual-boot Ubuntu on my system turned into a several-hour ordeal. All because of a quirk in my Lenovo device.

After restarting my laptop and navigating to the boot menu, I realized immediately that there was no sign of the connected bootable drive. First, i thought it was a problem with the drive. I ensured that the drive was GPT-partitioned and even verified the image’s SHA256 checksum file by running `gpg --keyid-format long --verify SHA256SUMS.gpg SHA256SUMS`. No luck. So I connected it to a different device, and, voila, it worked.

So, the problem was definitely with my laptop. After an hour of tweaking with the BIOS and system settings, I was unable to fix the issue. So I decided to just reset my laptop, assuming that the issue was caused due to a corrupted file somewhere. No luck again.

Exhausting all other options, I started choosing random devices to boot from (I was quite frustrated at this point). And, for some reason, booting from Linpus Lite,

a lightweight Linux-based OS, took me to Ubuntu instead. My laptop was representing my bootable drive with Ubuntu as Linpus Lite? After some research, **I figured out that some UEFI/BIOS implementations mistakenly identify any GRUB-based bootloader, including Ubuntu, as Linpus Lite due to a preloaded identification string in the firmware database.**

To fix this issue I manually updated the EFI boot entry by booting into the UEFI settings, checking the boot entries, and then using the `efibootmgr` tool in Ubuntu to update the boot entry name (XXXX is the boot entry number of Linpus Lite):

```
sudo efibootmgr -v
sudo efibootmgr -b XXXX -L "Ubuntu"
```

Installing QEMU and NASM

```
sudo apt-get install build-essential qemu-system nasm
```

QEMU is an emulator where we'll run MyOS, while NASM is an assembler used to write assembly language programs.

Installing a Cross-Compiler and Cross-Linker

I need to install a **cross-compiler** to compile my code specifically for my target OS kernel. Using a standard compiler like `gcc` is not suitable because it assumes that the machine code it generates will run on the same architecture and environment as the one on which it was compiled (the host system).

In my case, the OS kernel will run on **QEMU**, which is configured to emulate an **x86 CPU architecture**. To ensure compatibility, I use a cross-compiler specifically designed to generate machine code for the x86 architecture. This allows me to compile the kernel code on my development machine (host) but produce a binary that will execute correctly on the target system (emulated x86).

Additionally, I needed a **cross-linker** because linking the kernel requires combining object files (such as those generated by the cross-compiler) into a

single binary targeted for the **x86 architecture**. For instance, the kernel must be linked to start at a specific address in memory (e.g., `0x1000`) as specified by the bootloader, and a standard linker designed for the host system would not generate the correct output for the x86 target architecture.

I set up the cross-compiler and cross-linker by running a shell script from **Github user Mell-o-tron's MellOS project**, as it had exactly what I was looking for and allowed me to save time during setup (https://github.com/mell-o-tron/MellOs/blob/main/A_Setup/setup-gcc-debian.sh).

New Things I Learnt

How a Computer Boots Up

The **CPU starts in 16-bit real mode after power-on** for compatibility with older systems. As a result, the **BIOS firmware** also operates in **16-bit real mode**, which means it can only execute code compatible with the **x86 instruction set.** Upon startup, the BIOS initializes hardware through the **Power-On Self-Test (POST)**, prepares basic I/O services, and identifies a bootable device based on the configured boot priority. Once this basic hardware initialization is complete, the BIOS loads the **bootloader** from the first sector of the bootable device into memory at **0x7C00** and transfers control to it (note that the BIOS firmware is still responsible for executing the bootloader code). Why 0x7C00? It's because the first 0x7C00 bytes are occupied by the Interrupt Vector Table and BIOS Data Area. The bootloader, written in **x86 assembly code**, operates in real mode initially but can perform critical tasks like loading additional bootloader stages or transitioning the CPU to **32-bit protected mode**. This transition involves disabling interrupts, setting up a **Global Descriptor Table (GDT)**, and enabling the **Protection Enable (PE)** bit in the `CR0` control register, allowing access to advanced CPU features such as memory protection and multitasking. Finally, the bootloader loads the operating system kernel into memory and hands control to it, enabling the kernel to initialize drivers, hardware, and system processes to complete the boot process.

Real Mode vs Protected Mode

The **16-bit real mode** and **32-bit protected mode** are two operational modes of x86 processors, each with distinct features and purposes. Real mode is the processor's default startup mode, designed for backward compatibility with early Intel 8086 systems. It uses a segmented memory model, limited to 1 MB of addressable memory, and supports 16-bit registers and instructions. Real mode lacks memory protection, multitasking, and advanced features, making it suitable for simple, legacy tasks like BIOS initialization and running bootloaders. In contrast, protected mode, introduced with the Intel 80286, supports 32-bit registers, flat or segmented memory models, and up to 4 GB of addressable memory. It enables advanced functionalities like memory protection, preemptive multitasking, and virtual memory via paging. While real mode is used during the initial stages of system boot, the bootloader transitions the CPU to protected mode to take advantage of these features. Protected mode also supports privilege levels (e.g., Ring 0 for the kernel) for security and stability.

Bios Routines in Real Mode

In real mode, I learnt that BIOS routines are used to perform essential hardware operations like screen output, keyboard input, and disk access. These routines are accessed through BIOS interrupts, where I pass parameters via CPU registers.

For video output, I can print characters on the screen by first switching to teletype mode. To do this, I move `0x0E` into the `AH` register and place the character I want to print into `AL`. Then, I call the video interrupt using `int 0x10`.

When I need to handle keyboard input, I use the BIOS keyboard interrupt. I set `AH = 0x00` and then call `int 0x16`. After this, the BIOS waits for me to press a key. Once a key is pressed, the ASCII value of the key is stored in the `AL` register, and its BIOS scancode is stored in `AH`.

Reading from a disk is another task I can accomplish using BIOS routines. BIOS relies on CHS (Cylinder-Head-Sector) addressing to locate data on the disk. To read data, I set up the required registers: `AH = 0x02` specifies the disk read function, `AL` contains the number of sectors to read, `CH` is the cylinder number, `CL` is the sector number, `DH` is the head number, `DL` is the disk number, and `ES:BX` specifies where to load the data in memory. For instance, if I want to read from the boot disk, I assume `Cylinder = 0`, `Head = 0`, and `Sector = 2`, and I load the data to

memory location `0x7E00`. After setting these values, I call the disk interrupt using `int 0x13`. When working with CHS addressing, sectors start from 1, whereas cylinders and heads start from 0.

To make BIOS calls, I use the `int` instruction, which invokes the corresponding interrupt routine. For example, I use `int 0x10` for video operations, `int 0x13` for disk operations, and `int 0x16` for keyboard input.

To manage temporary data during BIOS calls, I rely on the stack. I use `PUSH` and `POP` to store and retrieve data, ensuring my operations don't overwrite important values. Additionally, I can use `PUSHA` to push all general-purpose registers onto the stack and `POPA` to restore them afterward. The registers I commonly use include `AX`, `BX`, `CX`, `DX`, `SP`, `BP`, `SI`, and `DI`.

Memory Models in Real Mode

I learnt that segmentation is the primary memory model used in real mode. In real mode, the system can access 2^{16} memory locations, and segmentation splits this memory into segments with a maximum size of 64KB (2^{16} bytes). These segments provide a way to organize memory for different purposes, such as the stack, data, and code.

Each segment has a special segment register to define its base address. For example, the **DS (Data Segment)** register is used for data, the **CS (Code Segment)** register is used for instructions, and the **SS (Stack Segment)** register is used for stack operations. There's also an **ES (Extra Segment)** register for additional uses. These segment registers define the starting address of a segment, which is combined with an **offset** (stored in general-purpose registers like `BX`, `SI`, or `DI`) to calculate the physical address.

$$\text{physical address} = \text{segment register} \times 16 + \text{offset}$$

The physical address can also be represented as `segment:offset`.

I also learnt about the **tiny memory model**, a specific memory model used in **x86 real-mode programming**. In this model, all code, data, and the stack are confined to a single 64KB memory segment. This means the **CS**, **DS**, and **SS** registers all point to the same segment.

Memory Models in Protected Mode

In protected mode, memory is structured using descriptors defined in the **Global Descriptor Table (GDT)**. Every segment in the system must have its descriptor in the GDT, which provides detailed information about how the memory is accessed and its privileges. Unlike real mode, where only segmentation is supported, protected mode introduces additional memory models, including the **flat memory model** and **paging**.

The memory model I'll be using for MyOS is the **flat memory model**, which simplifies memory addressing by treating the entire memory space as a single contiguous block. Instead of dealing with multiple distinct segments, the flat model allows me to think of memory as one large, seamless address space. One of the biggest simplifications in the flat memory model is that all segments share the same base address (`0x00000000`). This means the linear address is just the offset:

$$\text{physical address} = \text{offset}$$

In the flat memory model, all segments (`CS`, `DS`, `SS`, etc.) in the GDT are configured to cover the entire memory space. Each segment descriptor is set up with:

- **Base address**: `0x00000000`, pointing to the beginning of memory.
- **Limit**: `0xFFFFF`, covering up to 4 GB of memory.
- **Present**: Indicates whether the segment is active.
- **Privilege**: Specifies access levels (0 for kernel, 3 for user mode).
- Additional flags that define the segment's behavior:
  - **Type**: Defines whether the segment is for code or data.
  - **Access**: 1 bit to indicate whether the CPU is currently using this segment.
  - **Code**: Indicates if the segment contains executable code.
    - For code segments, there's also:
      - **Conforming**: Determines if lower privilege code can execute this segment.

        - **Readable**: Indicates whether the segment is read-only.
    - For data segments, there's also:
        - **Direction**: Specifies whether the segment grows downward.
        - **Writable**: Indicates whether the segment is writable.
- **Granularity**: Multiplies the segment limit by `0x1000`, allowing larger segments.
- **32 bits**: Enables 32-bit memory access.
- **64 bits:** Enables 64-bit memory access.
- **AVL**: general-purpose bit that can be set or cleared by the operating system or other software to mark segments or descriptors with custom information.

This is the format of the GDT entry format when configured in x86 assembly:

Byte(s)	Bits	Description
0-1 (16 bits)	0-15	**Segment Limit (15:0)**: Lower 16 bits of the segment limit.
2-3 (16 bits)	16-31	**Base Address (15:0)**: Lower 16 bits of the segment's base address.
4 (8 bits)	32-39	**Base Address (23:16)**: Middle 8 bits of the segment's base address.
5 (8 bits)	40-47	**Access Byte**: Defines the segment type and privilege level.
6 (4 bits)	48-51	**Segment Limit (19:16)**: Upper 4 bits of the segment limit.
6 (1 bit)	52	**Available (AVL)**: Unused in modern systems (set to 0).
6 (1 bit)	53	**Long Mode (L)**: Indicates 64-bit mode (set to 0 in 32-bit mode).
6 (1 bit)	54	**Default Operation Size (D)**: 0 for 16-bit, 1 for 32-bit segments.
6 (1 bit)	55	**Granularity (G)**: 0 for byte granularity, 1 for 4 KB granularity.

7 (8 bits)	56-63	**Base Address (31:24)**: Upper 8 bits of the segment's base address.

Building MyOS

Design Considerations with MyOS

MyOS will be written in x86 assembly and C. It will be designed for BIOS systems running on x86 processors. For now, MyOS will consist of the code for the bootloader and MyOS's main operating system kernel.

Making the Bootloader

The first step in making MyOS was writing the bootloader. Using everything I had learnt about the role of a bootloader and x86 assembly, I was able to write assembly code for the BIOS to load and execute in the boot.asm file located in the MyOS github repository (https://github.com/mentallyunstablefish/my-OS/blob/main/My_OS/boot.asm). Let me take you through each section.

Starting the Bootloader

I started by defining the **origin** of the code with `[org 0x7C00]` because the BIOS loads the bootloader to this address in memory. To make things organized, I set up a constant `KERNEL_LOCATION` as `0x1000`, which is where I plan to load my kernel. The `dl` register holds the drive number of the bootable device, so I saved it into a variable called `BOOT_DISK` for later use. This ensures that even if I use `dl` for something else, I still know which disk to read from.

Segment and Stack Initialization

Next, I initialized the segment registers and the stack. Since the CPU starts in **real mode**, all memory accesses are segmented. I cleared the `ax` register with `xor ax, ax` and loaded it into `es` and `ds` to point them to the base of memory (address `0x0000`). Then, I set up the stack by moving `0x8000` into `bp` and `sp`, making this

location the top of my stack. This ensures my stack operations like `push` and `pop` will work properly without overwriting critical areas of memory.

Loading the Kernel

The heart of the bootloader is the section where I load the kernel. I used the BIOS **disk interrupt** (`int 0x13`) to read the kernel from the disk into memory. I prepared the registers:

- `ah = 0x02` to specify the read sectors function.
- `al = 2` to read two sectors (since my kernel is small).
- `ch = 0x00`, `cl = 0x02`, and `dh = 0x00` to set the Cylinder-Head-Sector (CHS) values.
- `dl` to the disk number stored in `BOOT_DISK`.
- Finally, I pointed `es:bx` to `KERNEL_LOCATION` (`0x1000`) where the kernel should be loaded. After setting everything up, I called `int 0x13` to perform the read. I didn't add error checking here, but I know it's something I need to improve.

Switching to Text Mode

Before transitioning to protected mode, I reset the screen to **text mode** (80×25 characters) by calling `int 0x10` with `ah = 0x00` and `al = 0x03`. This step ensures a clean and predictable output environment, which is helpful for debugging.

Setting Up the GDT

Now, to enable protected mode, I needed a **Global Descriptor Table (GDT)**. I created a small GDT in memory with three descriptors:

1. A **null descriptor** to satisfy the CPU's requirement for the first entry.
2. A **code segment descriptor** (`GDT_code`) that is executable and spans the full 4 GB memory space.
3. A **data segment descriptor** (`GDT_data`) that is writable and also spans the full memory space.
   The GDT is defined between

`GDT_start` and `GDT_end`, and its address and size are stored in `GDT_descriptor`. I loaded this descriptor using the `lgdt` instruction.

Switching to Protected Mode

To enter protected mode, I first disabled interrupts with `cli` to avoid any unexpected behavior during the transition. Then, I enabled protected mode by setting the least significant bit (PE bit) in the `cr0` control register. Once protected mode was enabled, I performed a far jump (`jmp CODE_SEG:start_protected_mode`) to reload the code segment register (`cs`) with the new GDT descriptor.

Protected Mode Initialization

In the `start_protected_mode` section, I initialized all the segment registers (`DS`, `SS`, `ES`, `FS`, and `GS`) with the **data segment descriptor** to ensure consistent memory access in the flat memory model. I also set up a **32-bit stack** by pointing `ebp` and `esp` to `0x90000`. Finally, I jumped to `KERNEL_LOCATION` (`0x1000`), where the kernel was loaded, handing control over to the kernel code.

Padding and Boot Signature

To ensure the bootloader was exactly 512 bytes, I added padding using the `times` directive to fill the space. At the end, I included the boot signature `0xAA55`, which the BIOS requires to identify the disk as bootable.

Making the OS Kernel

Setting Up the Kernel (kernel.cpp)

The function `main()` is marked with `extern "C"` to ensure that the C++ compiler uses **C-style linkage**, making it compatible with the bootloader and entry code. Inside `main()`, I directly wrote the character `'Q'` to the memory address `0xB8000`. This address corresponds to the start of **video memory** in text mode. Each character on the screen is represented by two bytes: the character itself and its color attribute. By setting `*(char*)0xb8000 = 'Q';`, I effectively displayed the letter `'Q'` in the top-left corner of the screen. For now, I kept the kernel minimal,

focusing on verifying that my bootloader successfully transitioned to protected mode and passed control to the kernel.

Setting Up the Kernel Entry Point (kernel_entry.cpp)

The kernel entry point serves as the **bridge between the bootloader and the main kernel code**. After the bootloader jumps to the kernel's location in memory, execution starts here.The code begins with the `[bits 32]` directive to indicate that all instructions from this point are in **32-bit protected mode**. I defined this as part of the `.text` section, which holds executable code. I used `[extern main]` to declare the `main` function from `kernel.cpp`. This tells the assembler that `main` exists in another file, allowing it to be linked properly during compilation. Then, I called the `main` function using the `call` instruction. This transfers control to the kernel's main function, which writes `'Q'` to the screen. After the `main` function returns, I added an infinite loop (`jmp $`) to prevent the CPU from executing unintended instructions. Since there's no multitasking or other programs running at this point, this is necessary to keep the system stable after the kernel completes its task.

Running MyOS

- The final step was to compile the individual object files and link them together. The detailed steps are located in the MyOS github repository's README.