

The Progenitor of 539kernel

Introduction

Till the point, we have created a bootloader for 539kernel that loads a simple assembly kernel from the disk and gives it the control. Furthermore, we have gained enough knowledge of x86 architecture basics to write the progenitor of 539kernel which is, as we have said, a 32-bit x86 kernel that runs in protected-mode. In x86, to be able to switch from real-mode to protected-mode, the global descriptor table should be initialized and loaded first. After entering the protected mode, the processor will be able to run 32-bit code which gives us the chance to write the rest of kernel's code in C and use some well-known C compiler ¹ to compile the kernel's code to 32-bit binary file. When our code runs in protected-mode, the ability of reaching BIOS services will be lost which means that printing text on the screen by using BIOS service will not be available for us, although the part of printing to the screen is not an essential part of a kernel, but we need it to check if the C code is really running by printing some text once the C code gains the control of the system. Instead of using BIOS to print texts, we need to use the *video memory* to achieve this goal in protected mode which introduces us to a graphics standard known as *video graphics array* (VGA). The final output of this chapter will be the progenitor of 539kernel which has a bootloader that loads the kernel which contains two parts, the first part is called *starter* which is written in assembly, this part initializes and loads the GDT table, then it is going to change the operating mode of the processor from real-mode to protected-mode and finally it is going to prepare the environment for the C code of the kernel which is the second part and it is going to gain the control from the starter after the latter finishes its work. In this early stage, the C code will only contains an implementation for a `print` function and it is going to print some text on the screen, in the later stages, this part will contain the main code of 539kernel.

And Now The Bootloader Makes More Sense

Before getting started with coding the new parts, let's revisit the code of the bootloader which we have written in chapter . You may recall that in that chapter we have written some code that I didn't explain and requested from you to take these lines on faith. With our current knowledge of x86 architecture these lines will now make sense and before explaining these lines first you need to remember that BIOS loads the bootloader to the physical memory address 07C0h, second, you need to recall that the register `ds` is one of registers that can be used to refer to a data segment. Now let's examine the first pair of these lines in our bootloader.

¹We are going to use GNU GCC in this book.

```
start:
    mov ax, 07C0h
    mov ds, ax
```

In the beginning of our bootloader, we set the value 07C0h to the register `ds`, the purpose of that should be obvious now. You know that our bootloader uses some x86 instructions that deal with data, the line `mov si, title_string` is an example of these instructions, and we have said before that any reference to data by the code being executed will make the processor to use the value in data segment register as the beginning of the data segment and the offset of referred data as the rest of the address, after that, this physical memory address of the referred data will be used to perform the instruction. Now assume that BIOS has set the value of `ds` to 0² and jumped to our bootloader, that means the data segment in the system now starts from the physical memory address 0³, now let's take the label `title_string` as an example and let's assume that its offset in the binary file of our bootloader is 490, when the processor starts to execute the instruction `mov si, title_string`⁴ it will, somehow, figures that the offset of `title_string` is 490 and based on the way that x86 handles memory accesses⁵ the processor is going to think that we are referring to the physical memory address 490 since the value of `ds` is 0, but in reality, the correct physical memory address of `title_string` is the offset 490 **inside** the memory address 07C0h since our bootloader runs from this address and not the physical memory address 0, so, to be able to reach to the correct addresses of the data that we have defined in our bootloader and that are loaded with the bootloader starting from the memory address 07C0h we need to tell the processor that our data segment starts from 07C0h and any reference to data should calculate the offset of that data starting from this physical address, and that exactly what these two lines do, in other words, change the current data segment to another one which starts from the first place of our bootloader. Let's move to the second pair of lines.

```
load_kernel_from_disk:
    mov ax, 0900h
    mov es, ax
```

These two lines will be executed before calling BIOS service 13,2 that loads sectors from disk, and they are going to tell BIOS to load the sector starting from the physical memory address 0900h, in other words, these lines are saying that the sector will be loaded in a segment that starts from the physical memory address 0900h, and the exact offset inside this segment that the sector will be loaded into is decided by the value of register `bx` before calling the service of BIOS, in our bootloader we have set `bx` to 0, which means the sector of the kernel

²It can be any other value

³And ends at the physical memory address 65535 since the maximum size of a segment in real-mode is 64KB.

⁴Which loads the physical memory address of `title_string` to the register `si`.

⁵By using segmentation.

will be loaded in the memory address `0900h:0000` and due to that when our bootloader finishes its job and decides to jump to the kernel code the operand of `jmp` instruction was `0900h:0000` which means that the value of `cs` register will be `0900h` and the value of `ip` register will be `0000` when the bootloader jumps to the loaded kernel.