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 (which we are going to call the *main kernel code* or *main kernel* in short) 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.

[1] We 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 `0` [2] and jumped to our bootloader, that means the data segment in the system now starts from the physical memory address `0` [3], 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 accesses [5] 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

[2] It can be any other value

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

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

[5] By 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.

Writing the Starter

The starter is the first part of 539kernel which runs after the bootloader which means that the starter runs in 16-bit real-mode environment, exactly same as the bootloader, and due to that we are going to write the starter by using assembly language instead of C. The main job of the starter is to prepare the environment for the main kernel to run in. To prepare the proper environment for the main kernel the starter switches the current operating mode from the real-mode to protected-mode which, as we have said earlier, gives us the chance to run 32-bit code. Before switching to protected-mode, the starter is going to initialize and load the GDT table, furthermore, to be able to use the video memory correctly in protected-mode a proper video mode should be set [6]. Finally, the starter will be able to switch to protected-mode and gives the control to the main kernel. Let's start with the prologue of the starter's code which reflects the steps that we have just described.

```
bits 16
extern kernel_main

start:
    mov ax, cs
    mov ds, ax

    call load_gdt
    call init_video_mode
    call enter_protected_mode

    call 08h:( 0x09000 + ( start_kernel - start ) )
```

The code of the starter begins from the label `start`, from now on I'm going to use the term *routine* for any callable assembly label [7]. You should be familiar with the most of this code, as you can see, the routine `start` begins by setting the proper memory address of data segment depending on the value of the code segment register `cs` [8] which is going to be same as the beginning of the starter's

[6] We are going to discuss the matter of video in more details later in this chapter

[7] The term routine is more general than the terms function or procedure, if you haven't encounter programming languages that make distinctions between the two terms (e.g. Pascal) then you can consider the term *routine* as a synonym to the term *function* in our discussion.

[8] As you know from our previous examination, the value of `cs` will be changed by the processor once a far jump is performed.

code. After that, the three steps that we have described are divided into three routines that we are going to write during this chapter, these routines are going to be called sequentially. Finally, the starter preforms a far jump to the code of the main kernel. But before examining the details of those steps let's stop on first two line of this code that could be new to you.

```
bits 16
extern kernel_main
```

The first line uses the directive `bits` which tells `NASM` that the following code is a 16-bit code, remember, we are in a 16-bit real-mode environment, so our code should be a 16-bit code. Knowing this information, `NASM` is going to assemble [9] any code that follows this directive as a 16-bit code. You may wonder, why didn't we use this directive in the bootloader's code? The main reason for that is how `NASM` works, when you tell `NASM` to generate the output in a flat binary format [10], it is going to consider the code as a 16-bit code by default unless you use `bits` directive to tell `NASM` otherwise, for example `bits 32` for 32-bit code or `bits 64` for 64-bit code.

The second line uses the directive `extern` which tells `NASM` that there is a symbol [11] which is external and not defined in any place in the code (e.g. as a label) that you are currently assembling, so, whenever you the code uses this symbol, don't panic, and continue you job, and the address of this symbol will be figured out latter by the linker. In our situation, the symbol `kernel_main` is the name of a function that will be defined as a C code in the main kernel code and it is the starting point of the main kernel.

[9]The process of translating an assembly code to a machine code.

[10]That's exactly what we have done with bootloader, refer back to chapter 2 and you can see that we have passed the argument `-f bin` to `NASM`.

[11]A symbol is a term that means a function name or a variable name. In our current situation `kernel_main` is a function name.