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:start_kernel
```

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.

Entering Protected-Mode

The code of `load_gdt` routine is the following.

```
load_gdt:
    cli
    lgdt [gdtr - start]

    ret
```

According to Intel's x86 manual, it is recommended to disable the interrupts before starting the process of switching to protected-mode, so, the first step of `load_gdt` routine is to disable the interrupts by using the instruction `cli` [12].

[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.

[12] In fact, `cli` disables only maskable interrupts but I use the general term interrupts here for the sake of simplicity.

The second step of load_gdt is to set the value of GDTR register. First, you should note that both gdtr and start are labels in the starter code, we have already defined start as a label for the main routine of the starter, but the label gdtr is a one that we are going to define later, what you need to know right now about this label is that is contains the value that we would like to load into the register GDTR, that is, it contains the memory address of the 539kernel's GDT table and the size of the table.

From our previous discussions, you know that when we mention any label through the assembly code it will be substituted with the memory address of this label, so, what is going on with the operand [gdtr - start] of lgdt? and why do we need to subtract the memory address of the label start from the memory address of label gdtr? First we need to understand the meaning of the brackets [] in NASM. Those brackets are used to refer to the content of a memory location inside the brackets, for example, assume we have a label named foo and we store the value bar in this label [13], then, [foo] in NASM means take the memory address of foo then get the content of the memory inside this memory location, the value bar. In other words, mov eax, foo means put the memory address of the label foo inside the register eax while mov eax, [foo] means put the value bar inside the register. In C, this concept is same as the pointers, assume foo is a pointer in C, then *foo expression is an equivalent to mov eax, [foo] while foo expression is equivalent to mov eax, foo.

After this explanation we now know that [gdtr - start] means subtract the memory address of start from the memory address of gdtr and use the result as a memory address and take the content inside this new address and load it to the register GDTR, but the current question is why do we need to perform the subtraction? isn't it enough to just get the memory address of the label gdtr and get its content and load it into the GDTR? The problem is when we refer to any label, this label will be substituted with the **full memory address** of that label, and if we tell NASM to get the content of the label gdtr through the brackets [gdtr] it will be considered as a refer to the memory (because it is) to get some data and as we have said earlier, with any refer to the memory the processor, in real-mode, is going to consult the corresponding segment register, in our case ds, and consider the referred memory address in the code as an offset inside the segment which starts from the memory address which is stored in the segment register. So, when we refer to the location of the label gdtr we need to make sure that we are referring to the offset of gdtr and not the full physical address, otherwise, the referred address will not be correct. To get the offset of gdtr instead of its full memory address we simply subtract the start memory address of the data segment from the memory address of gdtr, and we can get this value of that memory address in many ways, one of them is by referring to the start label. Let's take an example to make the matter of getting the offset of a label more clear, assume that the memory address of start is 1000d while the memory address of gdtr is 1050d, based on the beginning code of start

[13] In the same way of the labels title_string and message_string in the bootloader.

routine, the value of `ds` will be `1000d`, then `gdtr - start = 1050d - 1000d = 50d`, when the processor refers to the memory location by using the starting address of the data segment which is in `ds` the final generated address will be `ds:(gdtr - start) = 1000d:50d = 1050d` which is exactly the same as the memory address of `gdtr`.

Now, let's take a look to the value of the label `gdtr`, for the sake of organizing the code, I've dedicated a separated file for the values of `gdtr` and `gdt` under the name `gdt.asm`. To make the starter able to reach the labels `gdtr` and `gdt` which reside in a different assembly file than `starter.asm` we can use NASM's directive `%include` which will be substituted with the content of the file which is passed to this directive, so, in the end of `starter.asm` we need to add the line `%include "gdt.asm"` so the starter can reach `gdtr`. Now let's see content of `gdt.asm`.

```
gdt:
    null_descriptor             :   dw 0, 0, 0, 0
    kernel_code_descriptor      :   dw 0xffff, 0x0000, 9a00h, 0x00cf
    kernel_data_descriptor      :   dw 0xffff, 0x0000, 0x9200, 0x00cf
    userspace_code_descriptor   :   dw 0xffff, 0x0000, 0xfa00, 0x00cf
    userspace_data_descriptor   :   dw 0xffff, 0x0000, 0xf200, 0x00cf

gdtr:
    gdt_size            :   dw ( 5 * 8 )
    gdt_base_address    :   dd gdt
```

As we have said before , the label `gdt` is the GDT table of 539kernel, while the label `gdtr` is the content of the special register `GDTR` that should be loaded by the stater to make the processor use 539kernel's GDT, the structures of both GDT table and GDTR register have been examined in details in the previous chapter . As you can see, the GDT table of 539kernel contains 5 entries, the first one is known as *null entry* which is a requisite in x86 architecture, in any GDT table, the first entry should be the null entry that contains zeros. The second and third entries represent the code segment and data segment of the kernel, while the fourth and the fifth entries represent the code segment and data segment of the user-space applications. The properties of each entries is shown in the table and as you can see, based on the base address and limit of each segment, 539kernel employs the flat memory model. Because the values of GDT entries are set by bits level we need to combine these bits at least as a set of bytes (or larger as in our current code), by combining them into units that are larger than a bit the values will be unreadable for the human, as you can see, a mere look at the values of each entry cannot tells us directly what is the properties of each of these entries. I've written a simple script by using Python 3 that generates the proper values as double words by taking the required entries in GDT as JSON input. The following is the code of the script if you would like to generate a different GDT table than the one which is presented here. And the JSON input of 539kernel's GDT table is .

The second label gdtr has the same structure of x86's register GDTR since we want to load the content of this label to the register directly. As you can see, the first part of gdtr is the size of the GDT table, we know that we have 5 entries in our GDT table and we already know from previous chapter that each entry in the GDT table has the size of 8 bytes. That means the total size of our GDT table is 5 * 8 = 40 bytes. The second part of gdtr is the full memory address of the label gdt, which is, once again, 539kernel's GDT table. As you can see here, we didn't subtract the memory address of start from gdt memory address here, and that's because we need to load the full physical memory address of gdt into the GDTR table and not just its offset inside a given data segment, as we know, when the processor tries to reach the GDT table it doesn't consult any segment register [14], it assumes that the full physical memory address of GDT is stored in the register GDTR, and to get the full memory address of a label in NASM we need to just mention the name of that label.

Till this point, we have examined the routine load_gdt, let's know examine the routine enter_protected_mode which does the real job of switching the operating mode of the processor from real-mode to protected-mode. Its code is the following.

```
enter_protected_mode:
    mov eax, cr0
    or eax, 1
    mov cr0, eax

    ret
```

To understand what this code does we need first to know what is a *control register*. In x86 there is a bunch of control registers, and one of them has the name cr0 [15]. The control registers contain values that determine the behavior of the processor, for example, the last bit of cr0, that is, bit 31 indicates that paging is currently enabled when its value is 1, while the value 0 means paging is enabled. The bit of our concern in this memory if the first bit (bit 0) in cr0, when the value of this bit is 1 that means protected-mode is enabled, while the value 0 means protected-mode is disabled. To switch the operating mode to protected-mode we need to change the value of this bit to 1 and that's exactly what we do in the routine enter_protected_mode. Because we can't manipulate the value of a control register directly, we copy the value of cr0 to eax in the first line, note that we are using eax here instead of ax and that's because the size of cr0 is 32-bit. We need to keep all values of other bits in cr0 but bit 0 which we need to change to 1, to perform that we use the Boolean operator OR that works on the bit level, what we do in the second line of the routine enter_protected_mode is a bit-wise operation, that is, an operation in bits level, the value of eax, which is at this point is the same value of cr0, will be *ORred* with the value 1, the

[14] Otherwise it is going to be a paradox! to reach the GDT table you need to reach the GDT table first!

[15] The others are cr1 till cr7.

binary representation of the value `1` in this instruction will be the following `0000 0000 0000 0000 0000 0000 0000 0001`, a binary sequence of size `32-bit` with `31` leading zeros and one in the end. Now, what does the Boolean operator `OR` do? It takes to parameters and each parameter has two possible values `0` or `1` [16], there are only four possible inputs and outputs in this case, `1 OR 1 = 1`, `1 OR 0 = 1`, `0 OR 1 = 1` and `0 OR 0 = 0`. In other words, we are saying, if one of the inputs is `1` then the output should be `1`, also, we can notice that when one of the inputs is `0` then the output will always be same as the other input [17]. By employing these two observations we can keep the all values from bit `1` to bit `31` of `cr0` by ORring their values with `0` and we can change the value of bit `0` to `1` by ORring its current value with `1` and that's exactly what we do in the second line of the routine. As I've said, the operation that we have just explained is known as a *bit-wise operation*, if you are not familiar with this kind of operations that work on bit level, please refer to Appendix. Finally, we move the new value to `cr0` in the last line, and after executing this line the operating mode of the processor with be protected-mode.

[16] Also, can be considered as **true** for `1` and **false** for `0`.

[17] Boolean operators are well-known in programming languages and they are used mainly with `if` statement.