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.

The Basic Code of The Progenitor

In this section we are going to start writing the most of 539kernel's progenitor code but one part which is related to the interrupts which will be examined in another section of this chapter. To be able to compile and run the code that we write in this section you need to update the `Makefile` of 539kernel, the changes of `Makefile` also will be examined in another section of this chapter. The following is the `Makefile` which presume that both `starter.asm` and `main.c` are available.

```
ASM = nasm
CC = gcc
BOOTSTRAP_FILE = bootstrap.asm
INIT_KERNEL_FILES = starter.asm
KERNEL_FILES = main.c
KERNEL_FLAGS = -Wall -m32 -c -ffreestanding -fno-asynchronous-unwind-tables -fno-pie
KERNEL_OBJECT = -o kernel.elf

build: $(BOOTSTRAP_FILE) $(KERNEL_FILE)
    $(ASM) -f bin $(BOOTSTRAP_FILE) -o bootstrap.o
    $(ASM) -f elf32 $(INIT_KERNEL_FILES) -o starter.o
    $(CC) $(KERNEL_FLAGS) $(KERNEL_FILES) $(KERNEL_OBJECT)
    ld -melf_i386 -Tlinker.ld starter.o kernel.elf -o 539kernel.elf
    objcopy -O binary 539kernel.elf 539kernel.bin
    dd if=bootstrap.o of=kernel.img
    dd seek=1 conv=sync if=539kernel.bin of=kernel.img bs=512 count=2
    dd seek=3 conv=sync if=/dev/zero of=kernel.img bs=512 count=2046
    qemu-system-x86_64 -s kernel.img
```

Beside the `Makefile`, the bootloader also need a modification to make the progenitor code works. In the previous version of the bootloader, we were loading only one sector from the disk [6] to memory, and that was more than enough for simple code such as `simple_kernel.asm` of chapter 2 . In most practical cases, the size of the kernel will be more than one sector and the 539kernel's progenitor is not an exception, therefore, the bootloader should load more than one sector in order to load the whole code of the kernel. First we need to add two new data labels in the bootloader, say below the definition of the label `load_error_string`, as the following.

[6] Remember, the size of a sector is `512` bytes.

```
number_of_sectors_to_load   db  3d
curr_sector_to_load         db  2d
```

The first one, as it is obvious from its name, indicates the number of sectors that we would like our bootloader to load from the disk, the current value is `3`, which means `1.5 KB` from the disk will be loaded to the memory, if you kernel's size is larger than `1.5 KB` we can simply modify the value of this label and increase the number of sectors to load.

The second label indicates the sector's number that we are going to load now, as you know, sector `1` of the disk contains the bootloader, and based on our arrangement [7] the code of the kernel will be there starting from sector `2` of the disk, therefore, the initial value of this label is `curr_sector_to_load`. The modified version of `load_kernel_from_disk` which loads more than one sector is the following.

```
load_kernel_from_disk:
    mov ax, [curr_sector_to_load]
    sub ax, 2
    mov bx, 512d
    mul bx
    mov bx, ax

    mov ax, 0900h
    mov es, ax

    mov ah, 02h
    mov al, 1h
    mov ch, 0h
    mov cl, [curr_sector_to_load]
    mov dh, 0h
    mov dl, 80h
    int 13h

    jc kernel_load_error

    sub byte [number_of_sectors_to_load], 1
    add byte [curr_sector_to_load], 1
    cmp byte [number_of_sectors_to_load], 0

    jne load_kernel_from_disk

    ret
```

The first difference in this new version of `load_kernel_from_disk` is the first `5` lines of this routine. As you may recall, the BIOS service `10,03` loads the required

[7] On `Makefile` of 539kernel.

sector into the memory address `es:bx` and due to that we set the value `0900h` to `es` because it is the starting memory address that we chose of kernel's segment. In the previous version of the bootloader it was enough the set `0` to `bx` since we were loading only one sector, that means the code will reside from offset `0` to offset `511` of the segment. Now we are loading more than one sector by executing `load_kernel_from_disk` multiple times (`number_of_sectors_to_load` times) with different `curr_sector_to_load` each time, so, if we keep the value of `bx` fixed to `0`, each sector will overwrite the previously loaded sector and only the last sector of the kernel will be there in memory, which is, of course, not what we want. The first five lines of `load_kernel_from_disk` ensures that each sector is loaded in the correct memory location, the first sector is loaded starting from offset `0` (`(2 - 2) * 512 = 0`), the second sector is loaded starting from offset `512` (`(3 - 2) * 512 = 512`) and the third sector is loaded starting offset `1024` (`(4 - 2) * 512 = 1024`).

The second change of the routine is the value that we set to the register `cl`, for BIOS's `10,03` the value of this register is the sector number that we would like to load. Now, this value depends on `curr_sector_to_load` which starts with `2` and increases by `1` after each sector being loaded. The last `4` lines before `ret` ensures that the value of `curr_sector_to_load` is increased to load the next sector from disk in the next iteration of the routine, the value of `number_of_sectors_to_load` is decreased by `1` after loading each sector and finally the new value of `number_of_sectors_to_load` is compared with `0`, when it is the case then the routine `load_kernel_from_disk` will return, otherwise, the routine will be called again with the new values for both `curr_sector_to_load`, `number_of_sectors_to_load` to load a new sector and so on.

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 and setting the interrupts up, furthermore, to be able to use the video memory correctly in protected-mode a proper video mode should be set [8]. 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.

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

```
bits 16
extern kernel_main

start:
    mov ax, cs
    mov ds, ax

    call load_gdt
    call setup_interrupts
    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 [9]. 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` [10] which is going to be same as the beginning of the starter's code. After that, the four steps that we have described are divided into four 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 [11] 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 [12], 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 [13] which is external and not defined in any place in the code (e.g. as a label)

[9] 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.

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

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

[12] 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`.

[13] A symbol is a term that means a function name or a variable name. In our current

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.

As I've said earlier, the stuff that are related to interrupts will be examined in another section of this chapter. To get a working progenitor we are going to define the routine `setup_interrupts` as an empty routine temporarily until we reach the interrupts section. Its code will be the following.

```
setup_interrupts:
    ret
```

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` [14].

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 [15], then, `[foo]` in NASM means take the memory address of `foo` then get the content of the memory inside this memory

situation `kernel_main` is a function name.

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

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

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` 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_in_bytes   :   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 descriptor* 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 entry 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 [16], 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

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

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` [17]. 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 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` [18], 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 [19]. 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

[17] The others are `cr1` till `cr7`.

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

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

mode of the processor with be protected-mode.

Setting Video Mode

As I mentioned before, in protected-mode, the services of BIOS will not be available. Hence, when we need to print some text on the screen after switching to protected-mode we can't use the same way that we have used till this point. Instead, the video memory which is a part of VGA standard should be used to write text on the screen or even drawing something on it. To be able to use the video memory, a correct *video mode* should be set, and there is a BIOS service that we can use to set the correct video mode. That means, before switching to protected-mode the correct video mode should be set first because we are going to use BIOS service to perform that and that's why the routine `init_video_mode` is being called before the routine `enter_protected_mode`. Now let's take a look at the code of `init_video_mode`.

```
init_video_mode:
    mov ah, 0h
    mov al, 03h
    int 10h

    mov ah, 01h
    mov cx, 2000h
    int 10h

    ret
```

This routine consists of two parts, the first part calls the service `0h` of BIOS's `10h` and this service is used to set the video mode which its number is passed in the register `al`. As you can see here, we are requesting from BIOS to set the video mode to `03h` which is a *text mode* with `16` colors. Another example of video modes is `13h` which is a *graphics mode* with `256` colors, that is, when using this video mode, we can draw whatever we want on the screen and it can be used to implement graphical user interface (GUI). However, for our case now, we are going to set the video mode to `03h` since we just need to print some text.

The second part of this routine uses the service `01` of BIOS's `10h`, the purpose of this part is to disable the text cursor, since the user of 539kernel will not be able to write text as input, as in command line interface for example, we will not the cursor to be shown. The service `01` is used to set the type of the cursor, and the value `2000h` in `cx` means disable the cursor.

Giving the Main Kernel Code the Control

According to Intel's manual, after switching to protected-mode a far jump should be performed, and the protected-mode of dealing with segments (via segment

selectors) should be used. Let's begin with the routine `start_kernel` which is the last routine to be called from `start` routine.

```
bits 32
start_kernel:
    mov eax, 10h
    mov ds, eax
    mov ss, eax

    mov eax, 0h
    mov es, eax
    mov fs, eax
    mov gs, eax

    call kernel_main
```

As you can see, the directive `bits` is used here to tell NASM that the following code should be assembled as `32`-bit code since this code will run in protected-mode and not in real-mode. As you can see, the first and second part of this routine sets the correct segment selector to segment registers. As you can see in the first part, the segment selector `10h` (that is, `16d`) is set as the data segment and stack segment while the rest data segment registers will use the segment selector `0h` which points to the null descriptor which means they will not be used. Finally, the function `kernel_main` will be called, this function, as we have mentioned earlier, will be the main C function of 539kernel.

As you recall, the far jump which is required after switching to protected-mode has been performed by the line `call 08h:start_kernel` in `start` routine. And you can see that we have used the segment selector `08h` to do that. While it may be obvious why we have selected the values `08h` for the far jump and `10h` as segment selector for the data segment, a clarification of the reason of choosing these value won't hurt anybody. To make sense of these two values you need to refer to table , as you can see from the table, the segment selector [20] of kernel's code segment is `08`, that means any logical memory address that refer to kernel's code should refer to the segment selector `08` which is the index and the offset of kernel's code segment descriptor in GDT, in this case, the processor is going to get this descriptor from GDT and based on the segment starting memory address and the required offset the linear memory address will be computed as we have explained previously in chapter . So, when we perform a far jump to the kernel code we used the segment selector `08h` which will be loaded by the processor into the register `cs`. The same this happens for the data segment of the kernel, as you can see, its segment selector is `16d` (`10h`) and that's the value that we have loaded the data segment registers that we are going to use.

[20] We use the relaxed definition of segment selector here that we have defined in the previous chapter

Writing the C Kernel

Till this point, we are ready to write the C code of 539kernel, as mentioned earlier, the current C code is going to print some text on the screen after it gets the control from the starter. Before writing the code that prints text on the screen, we need to examine VGA standard.

A Glance at Graphics with VGA

Video Graphics Array (VGA) is a graphics standard that has been introduced with IBM PS/2 in 1987, and because our modern computers are compatible with the old IBM PC we still can use this standard. For our purpose, VGA is easy to use, at any point of time the screen can be in a specific *video mode* and each video mode has its own properties such as its resolution and the number of available colors. Basically, we can divide the available video modes into two groups, the first one consists of the modes that just support texts, that is, when the screen is on one of these modes then the only output on the screen will be texts we call this group *text mode*, while the second consists of the modes that can be used to draw pixels on the screen and we call this group *graphics mode*, we know that everything on computer's screen is drawn by using pixels, including texts and even the components of graphical user interface (GUI) which they usually called widgets, usually, some basic low-level graphics library is used by GUI toolkit and this library provides functions to draw some primitive shapes pixel by pixel, for instance, a function to draw a line can be provided and another function to draw a rectangle and so on. This basic library can be used by GUI toolkit to draw more advanced shapes known as widgets, a simple example is the button widget, which is basically drawn on the screen as a rectangle, and the GUI toolkit should maintain some basic properties that associated to this rectangle to convert it from a soulless shape on the screen to a button that can be clicked, fires an event and has some label upon it. When we have written the starter's routine `init_video_mode` we told BIOS to set the video mode to a text mode, and as we mentioned we can tell BIOS to set the video mode to graphics mode by changing the value `03h` to `13h`.

Whether the screen is in a text or graphics mode, to print some character on the screen or to draw some pixels on it. The values that you would like to the screen can be written to *video memory* which is just a part of the main memory that has a known starting memory address, for example, in text mode, the starting memory address of the video memory `b8000h` as we will see in a moment, note that this memory address is a physical memory address, not logical and not linear. Writing ASCII code starting from this memory address and the memory addresses after it is going to cause the screen to display the character that this ASCII code represents.

VGA Text Mode

When the screen it is in the text mode `03h` to print a character it should be represented in two bytes that are stored contiguously in video memory, the first byte is the ASCII code of the character that we would like to print, while the second bytes contains the information of the background and foreground colors that will be used to print this character. Before getting started in implementing `print` function of 539kernel, let's take a simple example of how to print a character, `A` for example, on the screen by using the video memory. From starter's code you known that the function `kernel_main` is the entry point of the main kernel code.

```
volatile unsigned char *video = 0xB8000;

void kernel_main()
{
    video[ 0 ] = 'A';

    while( 1 );
}
```

Don't focus on the last line `while ( 1 );` right now, it is an infinite loop and it is not related to our current discussion. As you can see, we have defined a pointer of `char` (`1` byte) called `video` which points to the beginning of video memory in color text mode [21]. And now, by using C's feature that considers arrays accessing syntax as a syntactic sugar to pointer arithmetic [22] we write the ASCII code of `A` to the memory location `b0000h + 0` to make the screen shows the character `A` on the screen. Now, let's assume we would like to print `B` right after `A`, then we should add the line `video[ 2 ] = 'B';` to the code, note that the index that we write `B` on is `2` and not `1`, why? Because as we said, the byte right after the character contains color information and not the next character that we would like to print.

For sure, each character that we print has a specific position on the screen. Usually, in computer graphics a coordinate system is used to indicate the position of the entity in question (e.g. a pixel, or in our current case a character). The limit of `x` axis, that is, the last number in `x` axis and the limit of `y` axis is determined by the resolution of the screen. For example, in `03h` text mode the resolution of the screen is `80` of the width and `25` for the height. That means that the last available number on `x` axis is `80` and on `y` axis is `25` which means that the last point that we can use to print a character on is `(80, 25)` and its position will be on the bottom of the screen at the right side while the position of the point `(0, 0)` which is also known as *origin point* is on the top at the left side. In the previous example, when we wrote the character `A` on the location `0` of the video memory we actually put it on the origin point, while we have put `B` on the point `(1, 0)`, that is, on the second row and first column of the screen

[21] A monochrome text mode is also available and its video memory starts from `b0000h` instead.

[22] Thanks God!

and as you can see, each even location [23] of the video memory can be translated to a point on the coordinate system and vice versa.

Now, knowing what we know about text mode, let's write some functions for 539kernel that deal with printing stuff on the screen. The first function is `print`, which takes a string of character as a parameter and prints the whole string on the screen, the second function is `println` which prints a new line and the last function is `printi` which prints integers on the screen. Let's begin by defining some global variables that we will use later and writing the declarations of the three functions. These declarations should be on the top of `main.c` , that is before the code of `kernel_main`, and the code of those functions should be on the bottom of `kernel_main` [24].

```
int nextTextPos = 0;
int currLine = 0;

void print( char * );
void println();
void printi( int );
```

The global variable `nextTextPos` is used to maintain the value of `x` in the coordinate system which will be used to print the next character while `currLine` maintains the current value of `y` in coordinate system, in other words, the current line of the screen that the characters will be printed on. The following is the code of `print`.

```
void print( char *str )
{
    int currCharLocationInVidMem, currColorLocationInVidMem;

    while ( *str != '\0' )
    {
        currCharLocationInVidMem = nextTextPos * 2;
        currColorLocationInVidMem = currCharLocationInVidMem + 1;

        video[ currCharLocationInVidMem ] = *str;
        video[ currColorLocationInVidMem ] = 15;

        nextTextPos++;

        str++;
    }
}
```

Beside putting the characters in the correct location in the video memory, the

[23] As you know, in even locations of the video memory the character are stored, while in odd locations the color information of those characters are stored.

[24] Can you tell why?

function `print` has two other jobs to do. The first is iterating through each character in the string that has been passed through the parameter `str` and the second one is translating the value of `x` into the corresponding memory location [25]. For the first job, we use the normal way of C programming language which as we mentioned earlier, considers the type string as an array of characters the ends with the null character `\0`. For the second job, the two local variables `currCharLocationInVidMem` and `currColorLocationInVidMem` which, as I think, have a pretty clear names, are used to store the calculated video memory location that we are going to put the character on based on the value of `nextTextPos`. As we have said before, the characters should be stored in an even position, therefore, we multiply the current value `nextTextPos` with `2` to get the next even location in the video memory, and since we are starting from `0` in `nextTextPos`, we can ensure that we are going to use all available locations in the video memory and because the color information is stored in the byte exactly next to the character byte, then calculating `currColorLocationInVidMem` is too easy, we just need to add `1` to the location of the character. Finally, we increase the value `nextTextPos` by `1` since we have used the `x` position that `nextTextPos` is pointing to for printing the current character. The last point to discuss is the code line which put the color information `video[ currColorLocationInVidMem ] = 15;`, as you can see, we have used the value `15` which means white color as foreground, you can manipulate this value to change the background and foreground color of the characters. Next, is the code of `println`.

```
void println()
{
    nextTextPos = ++currLine * 80;
}
```

The code of `println` is too simple, the width of the screen in `03h` text mode is `80`, as we have mentioned earlier, which means `80` characters can be printed on a specific line and each line in the screen has `80 * 2 = 160 bytes` of video memory. To print a character on a new line, we can get the first position (`x`) of this line by multiplying the number of the line (starting from `0`) in question with `80`. For example, when we were printing on the first line of the screen, that is, the value of `currLine` is `0`, the first `x` position of this line to print a character is `0 * 80 = 0`, and because the width of the screen is `80` characters, the range of `x` values which they represent a position on the first line is `0` to `79` and right after this range the characters will be printed on the second line, starting from position `80` to `159` and so on. The function `println` uses these facts to change the position of next character to print from the current line to the next one. The following is the code of `printi`.

```
void printi( int number )
```

[25] Please note that the way of writing this code and any other code in 539kernel is focuses on the readability of the code instead of efficiency in term of anything. Therefore, there is absolutely better ways of writing this code and any other code in term of performance or space efficiency

```
{
    char* digitToStr[] = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" };

    if ( number >= 0 && number <= 9 )
    {
        print( digitToStr[ number ] );
        return;
    }
    else
    {
        int remaining = number % 10;
        number = number / 10;

        printi( number );
        printi( remaining );
    }
}
```

The first problem with implementing `printi` is that it receives an integer as a parameter, and the ASCII code of this integer cannot be known until we convert it to the corresponding character, this will be easy with the numbers from `0` to `9` as you can see on the first part of this function, but it becomes tricky when the passed integer is greater than `9`. I'm not going to explain the details of converting an integer to a list of characters since that these problems aren't our focus now, but the algorithm here is obvious, and the following example shows how it does work. Let's assume the value of the parameter is `539`, it is a fact that `539 % 10 = 9` [26] which is the digit at the right of `539`, also, it is a fact that `539 / 10 = 53.9` and if we get the integer of the result we get `53`, so, by using these simple arithmetic operations, we managed to get a digit from the number and remove this digit from the number. This algorithm is going to split the digits in the reverse order, and due to that I have used recursion and a simple solution to print the number in the correct order. However, on its basis, `printi` depends on the first function `print` to print one digit on the screen and before that this digit is being converted to a character by using the array `digitToStr`.

VGA Graphics Mode

Although 539kernel is text-based, it is useful to take a glance on how graphics mode works on VGA. First, to use graphics mode instead of text mode, the value `03h`, which is passed to the register `al` in the routine `init_video_mode` of the starter of 539kernel, it should be changed to `13h` which gives us a graphics mode with `320x200` resolution and `256` colors. Also, the starting memory location of the graphics mode is different and it is `a0000h`.

[26] The operation which is represented by `%` is known as modulus, in other words, the remaining value of a division.

As explained in the section of text mode , the coordinate system is used to specify the position of an entity on the screen, in the text mode this entity is a character, while in the graphics mode this entity is a pixel which is a small dot on the screen that has a color. This small dot, when gathers with many others of different colors, creates all the graphics that we see on computers monitors. So, according to the provided resolution (`320x200`) and that the graphical entity is a pixel, we should know that we are going to have `200` lines (the height or `y`) on the screen and each line can have up to `320` (the width or `x`) of our graphical entity which is the pixel. The structure of video memory is even simpler in graphics mode, each byte represents a pixel on a specific position of the screen and numeric value which represents a color is stored in byte to be shown in the screen. By using this simple mechanism and a review for the basics of geometry you can draw the primitive shapes on the screen (e.g. lines, rectangles and circles) and by using these basic shapes you can draw even more complex shapes [27]. While we are not going any further with this topic since it is out of our scope [28] the following example is intended to give you a feel of how to draw pixels on the screen. It is going to draw blue pixels on all available positions on the screen which is going to perceived as a blue background on the screen.

```
volatile unsigned char *video = 0xA0000;


void kernel_main()
{
    for ( int currPixelPos = 0; currPixelPos < 320 * 200; currPixelPos++ )
        video[ currPixelPos ] = 9;

    while( 1 );
}
```

The Code of `kernel_main`

Now, everything is ready to write the function `kernel_main`, it does nothing but printing some text by using the functions that we have defined earlier.

```
void kernel_main()
{
    print( "Welcome to 539kernel!" );
    println();
    print( "We are now in Protected-mode" );
    println();
    printi( 539 );
```

[27] If you are interested on the topic of drawing shapes by using the pixels you can read about the basics of computer graphics and geometry. I recommend a tutorial named "256-Color VGA Programming in C" by David Brackeen as a good starter that combines the basics of both.

[28] Sorry for that! I, myself, think this is an interesting topic, but this book is about operating systems kernel!

```
    println();

    while( 1 );
}
```

Interrupts in Practice

In the previous chapter we knew that there are two sources of interrupts, the first source is software, while the second source is hardware. The send hardware interrupts, there is a device which is connected to the processor and is known as *programmable interrupt controller* (PIC)[29], the job of this device is to send the interrupts of other devices (e.g. hard disk) to the processor. In other words, PIC is a mediator between the machines I/O devices and the processor, when a device needs to interrupts the processor to handle some event (e.g. the disk has finished from copying some data to the main memory) it is going to send this interrupt to PIC which is going to send it to the processor, this type of interrupts is known as *interrupt request* (IRQ).

Only `8` devices can be attached to one PIC devices, therefore, `IRQ0` is the name of the interrupt which is emitted by the device which is attached to the first slot of PIC, `IRQ1` for the second and so on. Because `8` slots are not enough to attach all external devices to the processor, another PIC has been attached to the first one which is now known as *master PIC* which the second is known as *slave PIC*. Figure shows this arrangement, as you can see now, there are `15` slots in the whole system instead of only `8` slots. In the master PIC, the third slot (`IRQ2`) is connected to the slave PIC, that is, whatever interrupt received by slave PIC from the devices that attached to it, will be sent to the master PIC through `IRQ2`. All other slots in both master (`IRQ0` to `IRQ7` but `IRQ2`) and slave PICs (`IRQ8` to `IRQ15`) are connected to external devices. There is a standard which tells us each `IRQ` is dedicated for what types of devices exactly, for example, `IRQ0` is the interrupt which is received by a device known as *system timer*[30], table shows the use of each `IRQ`.

After receiving the `IRQ` from a device, PIC should send this request to the processor, in this stage each `IRQ` number is mapped (or translated, if you prefer) to an interrupt number for the processor, for example, `IRQ0` will be sent to the processor as interrupt number `8`, `IRQ1` will be mapped to interrupt number `9` and so on until `IRQ7` which will be mapped to interrupt number `15d` (`0Fh`). While `IRQ8` till `IRQ15` are mapped to interrupts number from `112d` (`70h`) to `119d` (`77h`). In the real-mode, this mapping will be fine, but in protected-mode it is going to cause conflicts between software and hardware interrupts, that is, one interrupt number will be used by both software and hardware which may causes some

[29] The newer technology is known as *advanced programmable interrupt controller* (APIC)

[30] The system timer is a device which sends an interrupt in each unit of time which makes it extremely useful for multitasking environment as we shall see later when we start discussing process management.

difficulties later in distincting the source of this interrupt, is it from the software or hardware? For example, in protected mode, interrupt number `8` which is used for system timer interrupt by PIC is also used by the processor when a software error known as *double fault* happens. The good thing is PIC is **programmable**, that is, we can send commands to PIC and tells it to change the default mapping (from `IRQs` to processor's interrupts number) to another mapping of our choice.

There are two well-known types of communicating with external devices by the processor, we have encountered one of them when we worked with video memory which causes the processor to communicate with the screen to write characters or draw pixels, this type of communication from the processor to a devices is known as *memory-mapped I/O* communication, that is, the main memory is used to perform the communication. There is another type which is used by PIC and this type is known as *port-mapped I/O* communication, in this method, each device (that uses this way) has *ports* that are dedicated for it and each port has its own distinct number and job and the size of each port is `1 byte` , for example, master PIC has two ports, the number of this first port is `20h` while the number of the second one is `21h`, the first port is used to send commands [31] to master PIC while the second port is used two write data on it so the master PIC can read it. The same is applicable to slave PIC with different port numbers, `a0h` and `a1h` respectively. PIC has no explicit command to remap `IRQs`, instead, there is a command to initialize PIC and one step of initializing it is to set the required mapping. Now, we can present the skeleton of `setup_interrupts` as following.

```
setup_interrupts:
    call remap_pic
    call load_idt

    ret
```

First, we are going to remap `IRQs` to different interrupt numbers by sending initializing command to both master and slave PICs, then we are going to initialize and load `IDT` and writing the necessary interrupts handlers which are also known as *interrupt service routines* (ISRs).

[31] Each device has its own set of commands.