x86 Operating Modes, Segmentation and Interrupts

Introduction

In our situation, and by using modern terminology, we can view the processor as a *library* and *framework*. A library because it provides us with a bunch of instructions to perform whatever we want, and a framework because it has general rules that organize the overall environment of execution, that is, it forces us to work in a specific way. We have seen some aspects of the first part when we have written the bootloader, that is, we have seen the processor as a library. In this chapter, we are going to see how the processor works as a framework by examining some important and basic concepts of x86. We need to understand these concepts to start the real work of writing 539kernel.

x86 Operating Modes

In x86 an operating mode specifies the overall picture of the processor, such as how does it perform its tasks, the maximum size of available registers, the available advanced features for the running operating system and the restrictions.

When we developed the bootloader in the previous chapter we have worked with an *x86 operating mode* named *real mode* [1] which is an old operating mode which is still supported by modern x86 processors for the sake of backward compatibility and when the computer turned on, it initially runs on real mode, for the same reason of backward compatibility.

Real mode is a 16-bit operating mode which means that, maximally, only 16-bit of register size can be used, even if the actual size of the registers is 64-bit [2]. Using only 16-bit of registers has consequences other than the size itself [3] for example, in real mode the size of the main memory is limited, even if the computer has 16GB of memory, real mode can deal only with 1MB. Furthermore, any code which runs on real mode should be 16-bit code, for example, the aforementioned 32-bit registers (such as `eax`) cannot be used in real mode code, their 16-bit counterparts should be used instead, for example, the 16-bit `ax` should be used instead of `eax` and so on.

Some core features of modern operating systems nowadays are multitasking, memory protection and virtual memory [4] and real mode provides nothing to implement these features. However, in modern x86 processors, new and more

[1] Its full name is *real address mode*.

[2] 64-bit CPUs.

[3] These consequences are considered as disadvantages in modern days.

[4] If some of these terms are new for you don't worry about them too much, you will learn them gradually throughout this book.

advanced operating modes have been introduced, namely, *protected mode* which is a 32-bit operating mode and *long mode* which is a relatively new 64-bit operating mode. Although the long mode provides more capacity for its users [5] we are going to focus on protected mode since it provides the same basic mechanisms that we need to develop a modern operating system kernel with the aforementioned features, hence, 539kernel is a 32-bit kernel which runs under protected mode.

Since protected mode is a 32-bit operating mode then 32-bit of registers can be used, also, protected mode has the ability to deal with 4GB of main memory, and most importantly, it provides important features which we are going to explore through this book that helps us in implementing modern operating system kernel features.

Modern Operating Systems and the Need of Protection

As we have said before, *multitasking* is one of core features that modern operating systems provide. In multitasking environment more than one software can run at the same time, at least illusionary, even if there is only one processor or the current processor has only one core in the current machine, for the sake of making our next discussion easier we should mention the term *process* which is used to describe a software which is currently running in a system, that is, a process is a running software. For example, if your web browser is currently running then this running instance of it is called a process, its code is loaded into the main memory and the processor is currently executing it. Another property of general-purpose operating systems that is allows the user to run any software from unknown sources which means these software cannot be trusted, they may contain code that intentionally or even unintentionally breach the security of the system or cause the system to crash.

Due to these two properties of modern general-purpose operating systems, the overall system needs to be protected from multiple actions. **First**, either in multitasking or monotasking [6] environment the kernel of the operating system, which is the most sensitive part of the system, should be protected from current processes, no process should be able to access any part of the kernel's memory either by reading from it or writing to it, also, no process should be able to call any of kernel's code without kernel's consent. **Second**, the sensitive instructions and registers that change the behavior of the processor (e.g. switching from real-mode to protected-mode) should be only allowed for the kernel which is the most privileged component of the system, otherwise, the stability of the system will be in danger. **Third**, in multitasking environment the running processes should be protected from each other in the same way the kernel is protected from them, no process should interfere with another.

[5]For example, long mode can deal with 16 **exabytes** of memory.

[6]That is, only one process can run at a given time. DOS is a an example of monotask operating system.

In x86, the *segmentation* mechanism provided a logical view of memory in real-mode and it has been extended in protected-mode to provide the needed protection which has been described in the third point [7], while segmentation can be used in x86 for this kind of protection, it is not the sole way to perform that, the another well-known way is called *paging*, but segmentation is the default in x86 and cannot be turned off while paging is optional, the operating system has the option to use it as a way of memory management and protection or not.

Also, in the protected-mode the concept of *privilege levels* has been introduced to handle the protections needed in the previous first and second points. The academic literature of operating systems separate the system environment into two modes, *kernel-mode* and *user-mode*, at a given time the system can either run on one of these modes. The kernel runs on kernel-mode, and has the privilege to do anything (e.g. access any memory location, access any resource of the system and perform any operation), while the user applications run on user-mode which is a restricted environment where the code that runs on doesn't have the privilege to perform sensitive actions. This kind of separation has been realized through privilege levels in x86 which provides **four privilege levels** numbered from `0` to `3`. The privilege level `0` is the most privileged level which can be used to realize the kernel-mode while the privilege level `3` is the least privileged which can be used to realized the user-mode. For privilege levels `1` and `2` it's up to the kernel's designer to use them or not, some kernel designs suggest to use these levels for device drivers. According to Intel's manual, if the kernel's design uses only two privilege levels, that is, a kernel-mode and user-mode, then the privilege level `0` and `3` should be used and not for example `0` and `1` [8]. In addition to protecting the kernel's code from being called without its consent and protecting kernel's data from being accessed by user processes (as both required by first point above), the privilege levels also prevent user processes [9] from executing *privileged instructions* (as required by second point above), only the code which runs in the privilege level `0`, that is, the kernel, will be able to execute these instructions since they could manipulate sensitive parts of the processor's environment [10] that only the kernel should maintain. When a system uses different privilege levels to run, as in most modern operating systems, the x86 processor maintains what is called *current privilege level* (CPL) which is, as its name suggests, the current privilege level of the currently running code. For example, if the currently running code belongs to the kernel then the current privilege level will be `0` and according to its the processor is going to decide allowed operations. In other words, we can say that the processor keep tracking the current state of the currently running system and one of the information in this state is in which privilege level (or mode) the system is currently running.

[7] Segmentation will be examined in details later on this chapter.

[8] According to my readings of Intel's manual, no reason has been presented for this restriction.

[9] That is, the processes which runs on privilege level greater than `0`

[10] For example, loading GDT register by using the instruction `lgdt` as we will see later in this chapter.

The Basic View of Memory

The basic view of the main memory is that it is an *array of cells*, the size of each cell is a byte, each cell is able to store some data (of 1 bytes of course) and is reachable by a unique number called *memory address* [11], the range of memory addresses starts from `0` to some limit `x`, for example, if the system has `1MB` of *physical* main memory, then the last memory address in the range is `1023` [12]. This range of memory addresses is known as *address space* and it can be *physical address space* which is limited by the physical main memory or *logical address space*. A well-known example of using logical address space that we will be discuss in later chapters is *virtual memory* which provides a logical address space of size `4GB` in 32-bit architecture even if the actual size of physical main memory is less than `4GB`. However, The address space starts from the memory address `0`, which is the index of the first cell (byte) of the memory, and it increases by `1`, so the memory address `1` is the index of the second cell of the memory, `2` is the index of third cell of memory and so on.

When we say *physical* we mean the actual hardware, that is when the hardware of the main memory (RAM) size is `1MB` then the physical address space of the machine is up to `1MB`. On the other hand, when we say *logical* that means it doesn't necessarily represents or obeys the way the actual hardware works, instead it is a hypothetical way of something that doesn't exist in the real world (the hardware). To make the *logical* view of anything works, it should be mapped into the real *physical* view, that is, it should be somehow translated for the physical hardware, this mapping is handled by the software or sometimes special parts of the hardware.

Now, for the following discussion, let me remind you that the memory address is just a numerical value, it is just a number. When I discuss the memory address as a mere number I call it *memory address value* or *the value of memory address*, while the term *memory address* keeps its meaning, which is a unique identifier that refers to a specific location (cell) in the main memory.

The values of memory addresses are used by the processor all the time to perform its job, and when it is executing some instructions that involve the main memory (e.g. reading a content from some memory location), the related values of memory addresses are stored temporarily on the registers of the processor, due to that, the length of a memory address value is bounded to the size of the processor's registers, so, in `32-bit` environments, where the size of the registers is usually `32-bit`, the length of the memory address value is **always** `32 bits`, Why am I stressing "always" here? because even if less than `32 bits` it is enough to represent the memory address value, it will be represented in `32 bits` though,

[11] The architecture which each memory address points to `1 byte` is known as *byte-addressable architecture* or *byte machines*. It is the most common architecture, of course, other architectures are possible, such as *word-addressable architecture* or *word machines*.

[12] As we know, `1MB` = `1024 bytes` and since the range starts from `0` and not `1`, then the last memory address in this case is `1023` and not `1024`.

for example, assume the memory address value `1`, in binary, the value `1` can be represented by only `1 bit` and no more, by in reality, when it is stored (and handled) by the `32-bit` processor, it will be stored as the following sequence of bits

```
00000000 00000000 00000000 00000001
```

As you can see, the value `1` has been represented in exactly `32 bits`, appending zeros to the left doesn't change the value itself, it is similar to writing a number as `0000539` which is exactly `539`. The processor works with all values, beside the memory addresses values, as a sequence of *binary number*. It is natural for us as human beings to deal with numbers as *decimal numbers*.

A number by itself is an abstract concept, it is something in our mind, but to communicate with each others, we represent the numbers by using symbols which is named *numerals*. For example, the conceptual number one can be represented by different *numeral systems*. In Arabic numeral system the number one is expressed as `1`, while Roman numeral system it is expressed as `I`. A numeral system is *writing system*, that is, it gives us rules to write a number down as a symbol, it focuses on the way of writing the numbers. On the other hand, the numbers can be dealt with by a *numbering system*, we use the *decimal numbering system* to deal with numbers, think about them and perform arithmetic operations upon them, the processor use the *binary numbering system* to do the same with numbers. There are numbering systems other that the decimal and binary numbering system, and any number can be represented by any numbering system.

A number system is defined by its *base* which is also called *radix*, this base defines the list of available *digits* in the numbering system starting from `0` to `base - 1`, and the total of available digits equals the base. Consider the decimal numbering system, its base is `10` which means the available digits are: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`, a total of `10` digits. These digits can be used to create larger numbers, for example, `539` which consists of the digits `5`, `3` and `9`.

On the other hand, the base of binary numbering system is `2`, therefore, the available digits are only `0` and `1`, and as in the decimal numbering system they can be used to compose larger numbers, for example, the number `two` in binary numbering system is `10` [13], be careful, this numeral does not represent the number `ten`, it represents the number `two` but in binary numbering system. When we discuss numbers in different numbering systems, we put the initial letter of the numbering system name in the last of the number, for example, `10d` and `10b` are two different numbers, the first one is `ten` in **d**ecimal which the second one is `two` in **b**inary.

Furthermore, basic arithmetic operations such as addition and subtraction can be performed on the numbering system, for example, in binary `1 + 1 = 10`

[13] And from here came the well-known joke: "There are 10 types of people in this world, those who understand binary and those who don't".

and it can be performed systematically, also, a representation of any number in any numbering system can be converted to any other numbering system systematically [14], while this is not a good place to show how to perform the operations and conversions for different numbering system, I have dedicated Appendix A for this sake.

By now it should be obvious for you that changing the base (radix) gives us a new numbering system and the base can be any number [15], one of useful and well-known numbering system is *hexadecimal* which its base is `16` and its digits are `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F` where `A` means `ten`, `B` means `eleven` and so on. So, why hexadecimal is useful in our situation? We know binary is used in the processor and it is easier for us to discuss some entities such as the memory addresses in binary instead of decimal due to that, but have you seen the previous example of memory address value in binary?

```
00000000 00000000 00000000 00000001b
```

It is too long and it will be tedious to work with, and for that the hexadecimal numbering system come be useful. Each digit in hexadecimal represents **four** bits [16], that is, the number `0h` in **h**exadecimal is equivalent to `0000b` in binary. As the `8 bits` known as a byte, the `4 bits` is known as a *nibble*, that is, a nibble is a half byte. And, as we have said, but in other words, one digit of hexadecimal represents a nibble. So, we can use hexadecimal to represent the same memory address value in more elegant way.

```
00 00 00 01h
```

From our previous discussions you may glanced that the register size that stores the values of memory address in the processor in order to deal with memory contents affects the available size of main memory for the system. Take for example the instruction pointer register, if its size say `16 bits` then the maximum available memory will be `64KB` [17], and if its size is `32 bits`, then the maximum available memory will be `4GB`. Why is that? To answer this example let's work with decimal numbers first. If I tell you that you have five blanks, what is the largest decimal number you can represent in these five blanks? the answer is `99999d`. In the same manner, if you have 5 blanks, what is the largest binary number you can represent in these 5 blanks? it is `11111b` which is equivalent to `31d`, the same holds for the registers that store the value of memory addresses, given the size of such register is `16 bits`, then there is 16 blanks, and the largest binary number that can be represented in those 16 blanks is `11111111 11111111b` or in hexadecimal `FF FFh`, which is equivalent to `65535d`, that means the last bytes a register of size `16 bits` can refer to is the byte number `65535d` because it is the largest value this register can store and no more, which leads to

[14] I think It's too brave to state this claim, however, it holds at least for the well-known numbering system.

[15] Which implies that the total of numbering systems is infinite!

[16] Can you tell why? [Hint: How the maximum hexadecimal number `F` is represented in binary?]

[17] 64 KB = 65536 Bytes / 1024

the maximum size of main memory this register can handle, it is `65535 bytes` which is equivalent to `64KB`.

x86 Segmentation

The aforementioned view of memory, that is the *addressable array of bytes* can be considered as the *physical* view of the main memory which specifies the mechanism of accessing the data. On top of this physical view a *logical* view can be created and one example of logical views is *x86 segmentation*.

In x86 segmentation the main memory is viewed as separated parts called *segments* and each segment stores a bunch of related data. To access data inside a segment, each byte can be referred to by its *offset*. The running program can be separated into three possible types of segments in x86. The types of x86 segments are: *code segment* which stores the code of the program under execution, *data segments* which store the data of the program and the *stack segment* which stores the data of program's stack.

Segmentation in Real Mode

For the sake of clarity, let's discuss segmentation under real mode first. We have said that logical views (of anything) should be mapped to the physical view either by software or hardware, in this case, the segmentation view is realized and mapped to the architecture of the physical main memory by the x86 processor itself, that is, by the hardware. So, for now we have a logical view, which is the concept of segmentation and dividing a program into separated segments, and the actual physical main memory view which is supported by the real RAM hardware and sees the data as a big array of bytes. Therefore, we need some tools to implement, that is mapping, the logical view of segmentation on top the actual hardware.

For this purpose, special registers named *segment registers* are presented in x86, the size of each segment register is `16-bit` and they are: `CS` which is used to define the code segment. `SS` which is used to define the stack segment. `DS`, `ES`, `FS` and `GS` which can be used to define data segments, that means each program can have up to four data segments. Each segment register stores the *starting memory address* of the segment [18]. In real mode, the size of each segment is `64KB` and as we have said we can reach any byte inside a segment by using the *offset* of the desired byte, you can see the resemblance between a memory address of the basic view of memory and an offset of the segmentation view of memory [19]. Segmentation in x86 is unavoidable, the processor always runs with the mind that the running program is divided into segments.

[18] And here you can start to observe the mapping between the logical and physical view.

[19] The concept and term of offset is not exclusive on segmentation, it is used on other topics related to the memory.

Let's take an example to make the matter clear, assume that we have a code of some program loaded to the memory and its starting physical memory address is `100d`, that is, the first instruction of this program is stored in this address and the next instructions are stored right after this memory address one after another. To reach the first byte of this code we use the offset `0`, so, the whole physical address of the first byte will be `100:0d`, as you can see, the part before the colon is the starting memory address of the code and the part after the colon is the offset that we would like to reach and read the bytes inside it. In the same way, let's assume we would like to reach the offset `33`, which means the byte `33` inside the loaded code, them the physical address that we are trying to reach is actually `100:33d`. To make the processor handle this piece of code as the *current* code segment then its starting memory address should be loaded into the register `CS`.

As we have said, the x86 processor always runs with the mind that the segmentation is in use. So, let's say it is executing the following assembly instruction `jmp 150d` which jumps to the address `150d`. What really happens here is that the processor consider the value `150d` as an offset instead of a memory address, so, what the instruction requests from the processor here is to jump to the offset `150` which is inside the *current* code segment, therefore, the processor is going to retrieve the value of the register `CS` to know what is the currently active code segment and append the value `150` to it. Say, the value of `CS` is `100`, then the memory address that the processor is going to jump to is `100:150d`.

This is also applicable on the internal work for the processor, do you remember the register `IP` which is the instruction pointer? It actually stores the offset of the next instruction inside the code segment which is pointed to in the `CS` register instead of the whole memory address of the instruction. Any call (or jump) to a code inside the same code segment of the caller is known as *near call (or jump)*, otherwise is it a *far call (or jump)*. Again, let's assume the current value of `CS` is `100d` and you want to call a label which is on the memory location `9001d`, in this situation you are calling a code that reside in a different code segment, therefore, the processor is going to take the first part of the address which is`900d`, loads it to `CS` then loads the offset `1d` in `IP`. Because this call caused the changed of `CS` value to another value, it is a far call.

The same is exactly applicable to the other two types of segments and of course the instructions deal with different segments based on their functionality, for example, you have seen that `jmp` and `call` both the with the code segment in `CS`, while the instruction `lodsb` deals with the data segment `DS`, the instruction `push` deals with the stack segment `SS` and so on.

In the previous chapter, when we wrote the bootloader, we have dealt with the segments. Let's go back to the source code of the bootloader, you remember that the firmware loads the bootloader on the memory location `07C0h` and because of that we started our bootloader with the following lines.

```
mov ax, 07C0h
mov ds, ax
```

Here, we told the processor that the data segment of our program (the bootloader) starts in the memory address `07C0h` [20], so, if we refer to the memory to read or write and *data*, start with the memory address `07C0h` and then append the offset that we are referring to. The second use of the segments in the bootloader is when we tried to load the kernel from the disk by using the BIOS service `02h:13h` in the following code.

```
mov ax, 1000h
mov es, ax

mov ah, 02h
mov al, 01h
mov ch, 0h
mov cl, 02h
mov dh, 0h
mov dl, 80h
mov bx, 0h
int 13h
```

You can see here, we have used the other data segment `ES` here and defined a new segment that starts from the memory address `1000h`, we did that because the BIOS service `02h:13h` loads the desired content (in our case the kernel) to the memory address `ES:BX`, for that, we have defined the new data segment and set the value of `bx` to `0h`. That means the code of the kernel will be loaded on `1000:0000h` by `02h:13h` and because of that, after loading the kernel successfully we have performed a far jump.

```
jmp 1000h:0000
```

Once this instruction is executed, the value of `CS` will be changed from the value `07C0h` where the bootloader reside to the value `1000h` where the kernel is reside and the execution of the kernel is going to start.

Segmentation in Protected Mode

The fundamentals of segmentation in protected mode is exactly same as the ones explained in real mode, but it has been extended to provide more features such as *memory protection*. In protected mode, a table named *global descriptor table* (GDT) is presented, this table is stored in the main memory and its starting memory address is stored in the special purpose register `GDTR`, each entry in this table called a *segment descriptor* which has the size `8 bytes` and they can be referred to by an index number called *segment selector* which is the offset of

[20]Yes, all segments can be on the same memory location, that is, there is a `64KB` segment of memory which is considered as the currently active code segment, data segment and stack segment. This type of design is known as *flat-memory model* and we will discuss it later on.

the entry from the starting memory address of GDT, that is, the content of the register `GDTR`, the first entry of GDT, which is entry #0, should not be used. An entry of GDT, that is, a segment descriptor, defines a segment (of any type) and has the information that is required by the processor to deal with that segment, the starting memory address of the segment is stored in its descriptor [21], also, the size (or limit) of the segment. The segment selector of the currently active segment should be stored in the corresponding segment register.

To clarify the matter, consider the following example. Let's assume we are currently running two programs and the code of each one of them is stored in the main memory and we would like to use each one of them as a separated code segment. Let's call them `A` which its starting memory address is `800` and `B` which is starting address is `900` and assume that the starting memory address of GDT is `500` and is already loaded in `GDTR`, to be able to use `A` and `B` as segments we should define a segment descriptor for each one of them. We already know that the size of a segment descriptor is `8 bytes`, so, if we define a segment descriptor for the segment `A` as entry #1 [22] then its offset in GDT will be `8`, the segment descriptor of `A` should have the starting address of `A` which is `800`, and we will define the segment descriptor of `B` as entry #2 which has the offset `16` since the previous entry took `8 bytes` from the memory Let's assume now that we want the processor to execute the code of segment `A`, we already know that the processor consults the register `CS` to know which code segment is currently active and should be executed next, for that, the **segment selector** of code segment `A` should be loaded in `CS`, so the processor can start executing it. In real mode, the value of `CS` and all other segment registers was a memory address, on the other hand, in protected mode, the value of `CS` and all other segment registers is a segment selector. In our situation, the processor takes the segment selector of `A` from `CS` which is `8` and the from the starting memory address of `GDTR` walks `8` bytes, so, if `GDTR = 500`, the processor will find the segment descriptor of `A` in the memory address `508`. The starting address of `A` will be found in the segment descriptor and the processor can use it with the value of register `EIP` to execute `A`'s code. Let's assume a far jump is occurred from `A` to `B`, then the value of `CS` will be changed to the segment selector of `B` which is `16`.

The Structure of Segment Descriptor

A segment descriptor is an `8 bytes` entry of global descriptor table which stores multiple *fields* and *flags* that describe the properties of a specific segment in the memory. With each memory reference by the running code to a segment, the processor is going to consult the descriptor that describes the segment in question to obtain basic information like starting memory address of this segment, also, segmentation in x86 is considered as a way for *memory protection*, a descriptor stores the properties of memory protection. By using those stored properties

[21] In real mode, the starting address of the segment is stored directly on the corresponding segment register (eg, `CS` for code segment).

[22] Remember that the entries on GDT starts from zero.

the processor will be able to protect the different segments on the system from each other and not letting some less privileged to call a code or manipulate data which belong to more privileged area of the system, a concrete example of that is when a userspace software (e.g. Web Browser) tries to modify an internal data structure in the kernel . In the following, the explanation of each field and flag of segment descriptor, but before getting started we need to note that here and in Intel's official x86 manual the term *field* is used when the information the field occupies **more than** `1 bit` from the descriptor, for example the segment's starting memory address is stored in `4 bytes`, then the place where this address is stored in the descriptor is called a field, otherwise the term *flag* is used, which means that the information which is stored in the flag occupies only `1 bit`.

Segment's Base Address and Limit

The most important information about a segment is its starting memory address, which is called the *base address* of a segment. In real mode, the base address was stored in the corresponding segment register directly, but in protected mode, where we have more information about a segment than mere base address, then this information will be stored in the descriptor of the segment [23].

When the currently running code refers to a memory address to read from it, write to it (in the case of data segments) or call it (in the case of code segments) it is actually referring to a specific segment in the system [24]. For the simplicity of next discussions, we call this memory address (which is referenced by the currently running code) the *generated memory address* because, you know, it is generated by the code.

Any generated memory address in x86 architecture is not an actual physical memory address [25], that means, if you hypothetically get a generated memory address and try to get the content of its physical memory location, the obtained data will not be same as the data which is required by the code. Instead, a generated memory address is called by Intel's official manual a *logical memory address* because, you know, it is not real memory address, **it is** logical. Every logical memory address refers to some byte in a specific segment in the system, and to be able to obtain the data from the actual physical memory, this logical memory address should be *translated* to a *physical memory address* [26].

The logical memory address may pass **two** translation processes to obtain the physical memory address instead of one. The first address translation is

[23]Reminder: In protected mode, the corresponding segment register stores the selector of the currently active segment.

[24]And it **should** since segmentation is enabled by default in x86 and cannot be disabled.

[25]Remember our discussion of the difference between our logical view of the memory (e.g. segmentation) and the actual physical hardware

[26]We can see here how obvious the mapping between the logical view of the memory and the real-world memory.

performed on a logical memory address to obtain a *linear memory address* [27], if paging [28] is enabled [29] in the system, a second translation process takes place om this linear memory address to obtain the real physical memory address. If paging is disabled, the linear memory address which is generated by the first translation process is same as the physical memory address. We can say that the first translation is there in x86 due to the segmentation view of memory, while the second translation is there in x86 due to the paging view of memory.

Hence, for now, our focus will be on the translation from a logical memory address to a linear memory address and this memory address, which is the output of the translation process, is same as the physical memory address since paging feature is disabled by default in x86. As we have said before, any reference to a memory address by the running code means that some specific segment is referred and the running code need to obtain (or call) some data inside this specific segment, due to that, the logical memory address and its translation is in fact all about segmentation.

Each logical memory address consists of two parts, a `16-bit` segment selector and a `32-bit` offset. When the currently running code generate a logical memory address, for instance, to read some data from memory, the processor need to perform the translation process to obtain the physical memory address. First, it reads the value of the register `GDTR` which contains the starting physical memory address of GDT, then it uses the `16-bit` segment selector in the generated logical address to locate the descriptor of the segment that the code would like to read the data from, inside segment's descriptor, the physical base address (the starting physical address) of the requested segment can be found, the processor obtains this base address and adds the `32-bit` offset from the logical memory address to the base address to obtain the last result, which is the linear memory address.

During this operation, the processor uses the other information in the segment descriptor to enforce the policies of memory protection. One of these policies is defined by the *limit* of a segment which specifies its size, if the generated code refers to an offset which exceeds the limit of the segment, the processor should stop this operation. For example, assume hypothetically that the running code has the privilege to read data from data segment `A` and in the physical memory another data segment `B` is defined right after the limit of `A`, which means if we can exceed the limit of `A` we will able to access the data inside `B` which is a critical data segment that stores kernel's internal data structures and we don't want any code to read from it or write to it if it is not privileged. This can be achieved by specifying the limit of `A` correctly, and when the unprivileged code tries maliciously to read from `B` be generating a logical memory address that

[27]Which is another not-real and not physical memory address which is there in x86 architecture because of paging feature.

[28]Don't worry about paging right now. It will be discussed later in this book. All you need to know now is that paging is another logical view of the memory.

[29]It is disabled by default which makes paging an optional feature in x86, unlike segmentation.

has an offset which exceeds the limit of `A` the process is going to prevent the operation and protect the content of segment `B`. The limit, or in other words, the size of a given segment is stored in the `20`-bit *segment limit field* of that segment descriptor and how the processor interpret the value of segment limit field depends on the *granularity flag* (G flag) which is also stored in the segment's descriptor, when the value of this flag is `0` then the value of the limit field is interpreted as bytes, let's assume that the limit of a given segment is `10` and the value of granularity flag is `0`, that means the size of this segment is `10` **bytes**. On the other hand, when the value of granularity flag is `1`, the value of segment limit field will be interpreted as of `4KB` units, for example, assume in this case that the value of limit field is also `10` (but G flag = `1`), that means the size of the segment is `10` of `4KB` units, that is, `10 * 4KB` which gives us `40KB` which equals `40960` bytes. Because the size of segment level field is `20` bits, that means the maximum numeric value it can represents is `2^20 = 1,048,576`, which means if G flag equals `0` then the maximum size of a specific segment can be `1,048,576` **bytes** which equals `1MB`, and if G flag equals `0` then the maximum size of a specific segment can be `1,048,576` of `4KB` units which equals `4` **GB**.

Back to the structure of descriptor, the bytes `2`, `3` and `4` of the descriptor store the *least significant bytes* of segments base address and the byte `7` of the descriptor store the *most significant byte* of the base address, that's `32-bit` for the base address. While the bytes `0` and `1` of the descriptor store the *least significant bytes* of segment's limit and byte `6` stores the *most significant byte* of the limit. The granularity flag is stored in the most significant **bit** of the the byte `6` of the descriptor.

Before finishing this subsection, we need to define the meaning of *least significant* and *most significant* byte or bit. Take for example the following binary sequence which may represent anything, from a memory address value to a UTF-32 character.

0111 0101 0000 0000 0000 0000 0100 110*1*

You can see the first bit from left is on bold format and its value is `0`, based on its position in the sequence we call this bit the *most significant bit* or *high-order bit*, while the last bit on the right which is on italic format and its value is `1` is known as *least significant bit* or *low-order bit.* The same terms can be used on byte level, given the same sequence with different formatting.

0111 0101 0000 0000 0000 0000 *0100 1101*

The first byte (`8-bits`) on the left which is on bold format and its value is `0111 0101` is known as *most significant byte* or *high-order byte* while the last byte on the right which is on italic format and its value is `0100 1101` is known as *least significant byte* or *low-order byte.*

Now, imagine that this binary sequence is the base address of a segment, then the least significant `3` bytes of it will be stored in bytes `2`, `3` and `4` of the descriptor, that is, the following binary sequence.

```
0000 0000 0000 0000 0100 1101
```

While the most significant byte of the binary sequence will be stored in the `7th` byte of the descriptor, that is, the following binary sequence.

```
0111 0101
```

Segment's Type

Given any binary sequence, it doesn't have any meaning until some context is added. For example, what does the binary sequence `1100 1111 0000 1010` represents? It could represent anything, a number, characters, pixels on image or even all of them based on how its user interprets it. When an agent (e.g. a bunch of code in running software or the processor) works with a binary sequence, it should know what does this binary sequence represent to be able to perform useful tasks. In the same manner, when a segment is defined, the processor (the agent) should be told how to interpret the content inside this segment, that is, the type of the segment should be known by the processor.

Till this point, you probably noticed that there is at least two types of segments, code segment and data segment. The content of the former should be machine code that can be executed by the processor to perform some tasks, while the content of the latter should be data (e.g. values of constants) that can be used by a running code. These two types of segment (code and data) belong to the category of *application segments*, there is another category of segment types which is the category of *system segments* and it has many different segment types belong to it.

Whether a specific segment is an application or system segment, this should be mentioned in the descriptor of the segment in a flag called *S flag* or *descriptor type flag* which is the fifth **bit** in **byte** number `5` of the segment descriptor. When the value of S flag is `0`, then the segment which is described by the descriptor is considered as a system segment, while it is considered as an application segment when the value of S flag is `1`. Our current focus is on the latter case.

As we have mentioned before, an application segment can be either code or data segment. Let's assume some application segment has been referenced by a currently running code, the processor is going to consult the descriptor of this segment, and by reading the value of S flag (which should be `1`) it will know that the segment in question is an application segment, but which of the two types? Is it a code segment or data segment? To answer this question for the processor, this information should be stored in a field called *type field* in the segment's descriptor.

Type field in segment descriptor is the first `4-bits` (nibble) of the fifth byte of the descriptor and the most significant bit specifies if the application segment is a code segment (when the value of the bit is `1`) or a data segment (when the value of the bit is `0`). Whether the segment is a code or data segment, the least significant bit indicates if the segment is *accessed* or not, when the value of this flag is `1`, that means the segment has been written to or read from (AKA: accessed), but if the value of this flag is `0`, that means the segment has not been accessed. The value of this flag is manipulated by the processor in one situation only, and that's happen when the selector of the segment in question is loaded into a segment register. In any other situation, it is up to the operating system to decide the value of accessed flag. According to Intel's manual, this flag can be used for virtual memory management and for debugging.

Code Segment Flags

When the segment is a code segment, the second most significant bit (tenth bit) called *conforming flag* (also called `C` flag) while the third most significant bit (ninth bit) called *read-enabled flag* (also called `R` flag.). Let's start our discussion with the simplest among those two flags which is the read-enabled flag, which its value indicates how the code inside the segment in question can be used, when the value of read-enabled flag is `1` [30], that means the content of the code segment can be executed **and** read from, but when the value of this flag is `0` [31] that means the content of the code segment can be **only** executed and cannot read from. The former option can be useful when the code contains data inside it (e.g. constants) and we would like to provide the ability of reading this data. When read is enabled for the segment in question, the selector of this segment can also be loaded into one of data segment register [32].

The conforming flag is related to the privilege levels that we had an overview about them previously in this chapter. When a segment is conforming [33] that means a code which runs in a less-privileged level can call this segment which runs in a higher privileged level while keeping the current privilege level of the environment same as the one of the caller instead of the callee, for example, let's assume for some reason a kernel's designer decided to provide simple arithmetic operations (such as addition and subtraction) for user applications in the kernel code, that is, there is no other way to perform these operations in that system but this code which is provided by the kernel. As we know, kernel's code should run in privilege level `0` which is the most-privileged level, and let's assume a user application which runs in privilege level `3`, a less-privileged level, needs to perform an addition operation, in this case a kernel code, which should be protected by default from being called by less-privileged code, should be called to perform the task, this can only realized if the code of addition operation is provided as a conforming segment, otherwise the processor is going to stop this action where a less-privileged code calls a more-privileged code. But also you should note the code of addition operation is going to run in privilege level `3` although it is a part of the kernel which runs in privilege level `0` and that's because of the original caller which runs in the privilege level `3` and how the

[30] Which means **do** enable read since `1` is equivalent to `true` in the context of flags.
[31] Which means **don't** enable read.
[32] Which makes sense, enabling reads from a code segment means it contains data also.
[33] Which means the value of conforming flag is `1`

conforming segments rules work in x86. Furthermore, although conforming segment can be called by a less-privilege code (e.g. user application calls the kernel), the opposite cannot be done (e.g. the kernel calls a user application's code) and the processor is going to stop the operation.

Data Segment Flags

When the segment is data segment, the second most significant bit (tenth bit) called expansion-direction flag (also called `E` flag) while the third most significant bit (ninth bit) called write-enabled flag (also called `W` flag). The latter one gives us the ability to make some data segment a read-only, which means the value of write-enabled flag should be `0`, or we can make a data segment both **writable** and readable by setting the value of write-enabled flag to `1`.

While the expansion-direction flag and its need will be examined in details when we discuss x86 run-time stack in this chapter, what we need to know right now is that when the value of this flag is `0`, the data segment is going to expand **up**, but when the value of this flag is `1`, the data segment is going to expand **down**.

A last note about data segments that all of them are **non-conforming**, that is, a less-privileged code cannot access a data segment in a more-privileged level. Furthermore, all data segments can be accessed by a more-privileged code.

Segment's Privilege Level

In our previous discussions, we have stated that a specific segment should belong to a privilege level and based on this privilege level the processors decides the protection properties for the segment in question, for example, whether that segment is a kernel-mode or user-mode segment, and the code that runs in which privilege level can reach this segment.

A field called *descriptor privilege level field* (DPL) in segment descriptor is where the operating system should set the privilege level of a given segment. The possible values of this field, as we know, are `0`, `1`, `2` and `3`, we have already discussed the meanings of these values previously in this chapter . Descriptor privilege level field occupies the second and third most significant bits of byte `5` in a descriptor.

Segment's Present

One of common operations that happen while running a system is to load data from secondary storage (e.g. hard disk) into the memory and one example of that is loading a program code into the memory when the user of the system runs a program, so, creating a new segment descriptor (hence, creating new segment in the memory) for this data can precede the completion of loading the data into the main memory, therefore, there could be some segment descriptors in the system that points to memory locations that don't contain the real data yet. In this case, we should tell the processor that the data in the memory location that a specific descriptor points to is not the real data, and the real segment is not presented in the memory yet, this helps the processor to generate error when some code tries to access [34] the segment's data. To tell the processor whether a segment is presented in the memory or not, *segment-present flag* (P flag) can be used, when its value is `1` that means the segment is present in memory, while the value `0` means the segment is not present in memory, this flag is the most significant bit of byte `5` of a descriptor.

Other Flags

We have covered all segment's descriptor fields and flags but three flags. The first one has multiple names based on the type of the segment, when the segment in question is a code segment, this flag is called *default operation size* (D flag). When the processor executes the code of this segment it uses this flag to decide the length of instructions' operand, depending on the currently executing instruction of the code segment, if the value of D flag is `1` the processor is going to assume the operand has the size of `32`-bit if it is a memory address and `32`-bit or `8`-bit operand if it is not a memory address, if the value of D flag is `0` the processor is going to assume the operand has the size of `16`-bit if it is a memory address and `16`-bit or `8`-bit operand if it is not a memory address. When the segment in question is a stack segment [35], this flag is called *default stack pointer size* (B flag), and it decides the size of the memory address (as a value) which points to the stack (stack pointer) and used implicitly by stack instructions such as `push` and `pop`. When the value of B flag is `1`, then the size of stack pointer will be `32`-bit and its value will be stored in the register `ESP` (rather then `SP`), When the value of B flag is `0`, then the size of stack pointer will be `16`-bit and its value will be stored in the register `SP` (rather then `ESP`). When the segment in question is a data segment that grows upward, this flag is called *upper bound flag* (B flag), when its value is `1` the maximum possible size of the segment will be `4GB`, otherwise, the maximum possible size of the segment will be `64KB` . Anyway, the value of this flag (D/B flag) **should** be `1` for `32`-bit code and data segments (stack segments are included of course) and it should be `0` for `16`-bit code and data segments. D/B flag is the second most significant bit in the byte `6`.

The second flag left is known as *64-bit code segment flag* (L flag) which is the third most significant bit in the byte `6` and from its name we can know that this flag is related to code segments. If the value of this flag is `1` that means the code inside the segment in question is a `64`-bit code while the value `0` means otherwise [36]. When we set the value of L flag to `1` the value of D/B flag should be `0` .

The final flag is the fourth most significant bit in the byte `6`, the value of this flag has no meaning for the processor, hence, it will not use it to make any decisions

[34] We use the term *access* here for all both types of application segments, while this term is valid for data segment, we mean *execute* for code segment.

[35] The processor knows it is a stack segment if the segment selector is loaded into stack segment selector register `SS`

[36] In terms of Intel's manual: *compatibility mode.*

upon the segment in the question as we have seen on all other flags and fields. On the other hand, this flag is available for the operating system to use it in whatever way it needs it, or just ignores it by set it to any of possible values [37] without any semantic.

The Special Register `GDTR`

The special register `GDTR` stores the base physical address [38] of the global descriptor table, that is, the starting point of `GDT` table. Also, the same register stores the limit (or size) of the table. To load a value into the register `GDTR` the x86 instruction `lgdt` [39] should be used, this instruction takes one operand which is the whole value that should be loaded into `GDTR`, the structure of this value should be similar to the structure of `GDTR` itself which is shown in figure which shows that the total size of `GDTR` is `48` bits divided into two parts. The first part starts from bit `0` (the least significant bit) to bit `15`, this part contains the limit of `GDT` table that we would like to load. The size of limit part of `GDTR` register is `16` bits which can represents that value `65,536` at maximum, that means the maximum size of `GDT` table can be `64KB = 65,536 Bytes / 1024`, and as we know, the size of each of descriptor is `8` bytes, that means the `GDT` table can hold `8,192` descriptors maximum. The second part of `GDTR` starts from bit `16` to bit `47` (the most significant bit) and stores the base memory address of `GDT` table that we would like to load.

x86 Run-time Stack

A user application starts its life as a file stored in user's hard disk, right at this stage it does nothing, it is just a bunch binary numbers that represent the machine code of this application, when the user decides to use this application and opens it, the operating systems loads this application into the memory and in this stage this user application becomes a process [40]. Typically, the memory of a process is divided into multiple regions which each one of them stores a different kind of application's data, one of those regions stores the machine code of the application in the memory, there are also two important regions of process' memory, the first one is known as *run-time heap* [41] which provides an area for dynamically allocated objects (e.g. variables), the second

[37] `0` or `1` since it is one bit.

[38] More accurately, the linear address. Refer to discussion of memory translation process in this chapter.

[39] Which stands for *load* global descriptor table

[40] A term used in operating systems literature to describe a running program. Another well-known term is *task* which is used by Linux kernel.

[41] Or just heap for short

one is known as *run-time stack* [42] [43] which is used to store the values of local variables and function parameters, we can say that the run-time stack is a way to implement *function's invocation*, that is, how function `A` can call function `B`, pass to it some parameters and then return back to the same point of code where function `A` called function `B` and finally get the returned value from function `B`, the implementation details of these steps is known as *calling convention* and the run-time stack is one way of realizing these steps. There are multiple known calling convention for x86 and different compilers and operating systems may implement different calling conventions, we are not going to cover those different methods but what we need to know that, as we said, those different calling conventions use the run-time stack as a tool to realize function's invocation. The memory region in x86 which is called run-time stack uses a data structure called *stack* to store the data inside it and to manipulate that data.

Typically, a *data structure* as a concept divided into two components, this first one is the way of storing the data in a high-level way [44] and the second one is the available operations to manipulate this data, and of course the reason of the existence of each data structure is to solve some kind of problem. In stack data structure, the data will be stored in first-in-last-out (FILO) [45] manner, that is, the first entry which is stored in the stack can be get out of the stack at last. Also, two operations are used available in stack data structure [46], *push* and *pop*, the first one put some value on the *top of the stack*, which means that the top of stack always contains the last value that have been inserted into a stack. The latter operation `pop` *removes* the value which resides on the top of the stack and returns it to the user, that means the most recent value that has been `push`ed to the stack will be return when we `pop` the stack.

Let's assume that we have the string `ABCD` and we would like to push each character into the stack. First we start with the operation `push A` which puts the value `A` on the top of the stack, then we execute the operation `push B` which

[42]Or stack for short. It's also called *call stack* but we are going to stick to the term run-time stack in our discussions.

[43]Please note that the sort names of run-time stack (that is, stack) and run-time heap (that is, heap) are also names for **data structures**. As we will see shortly, a data structure describes a way of storing data and how to manipulate this data, while in our current context these two terms are used to represent **memory regions** of a running process although the stack (as memory region) uses stack data structure to store the data. Due to that, here we use the more accurate term *run-time stack* to refer the memory region and *stack* to refer the data structure.

[44]A data structure is not concerned about how to store the data in low-level (e.g as bits, or bytes. In the main memory or on the disk, etc.). But it answers the question of storing data as a high-level concept (as we will see in stack example) without specifying the details of implementation and due to that, they are called *abstract data structures*.

[45]On contrary, *queue data structure* stores data in first-in-**first**-out (FIFO) manner.

[46]That doesn't mean no more operations can be defined for a given data structure in the implementation level. It only means that the conceptual perspective for a given data structure defines those basic operations which reflect the spirit of that data structure. Remember that when we start to use x86 run-time stack with more operations than `push` and `pop`, though those other operations are not canonical to the stack data structure, but they can be available if the use case requires that (and yes they may violate the spirit of the given data structure!).

puts the value `B` on top of the value `A` as we can see in the figure , that is, the value `B` is now on the top of the stack and not the value `A`, the same is going to happen if we push the value `C` next as you can see in the figure and the same for the value of `D` and you can see the final stack of these four push operations in figure . Now let's assume that we would like to read the values from this stack, the only way to read data in stack data structure is to use the operation `pop` which as we have mentioned removes the value which resides ion the top of the stack and returns it to the user, that is, the stack data structure in contrary of array data structure [47] doesn't have the property of *random access* to the data, so, if you want to access any data in the stack, you can only use `pop` to do that. That means if you want to read the first pushed value to the stack, then you need to `pop` the stack `n` times, where the value of `n` is the number of pushed elements into the stack, in other words, the size of the stack. In our example stack, to be able to read the first pushed value which is `A` you need to `pop` the stack four times, the first one removes the value `D` from the top of stack and returns it to the user, which makes the values `C` on the top of stack as you can see in figure and if we execute `pop` once again, the value `C` will be removed from the top of the stack and returns it to the user, which makes the value `B` on the top of the stack as you can see in figure , so we need to `pop` the stack two times more the get the first `push`ed value which is `A`. This example makes it obvious for us why the stack data structure is described as first-in-last-out data structure. [48]

The stack data structure is one of most basic data structures in computer science and there are many well known applications that can use stack to solve a specific problem in an easy manner. As an example of applications that can use a stack to solve a specific problem, let's back to our example of `push`ing `ABCD` into a stack character by character and then `pop`ping them back, the input is `ABCD` but the output of `pop` operation is `DCBA` which is the reverse string of the input, so, the stack data structure can be used to solve the problem of getting the reversed string of an input by just pushing it into a stack character by character and then popping this stack, concatenating the returned character with the previously returned character, until the stack becomes empty. Other problems that can be solved easily with stack are palindrome problem and parenthesis matching problem which is an important one for the parsers of programming languages.

Now, with our understanding of the theoretical aspect of stack data structure, let's see how the run-time stack is implemented in x86 architecture to be used for the objectives that we have mentioned in the beginning of the sub-section. As we have said earlier, the reason of x86 run-time stack's existence is to provide a way to implement function's invocation, that is, the lifecycle of functions. Logically,

[47] Which is implemented by default in most major programming languages and know as arrays (in C for example) or lists (as in Python)

[48] As you can see in this brief explanation of stack data structure, we haven't mention any implementation details which means that a specific data structure is an abstract concept that describes a high-level idea where the low-level details are left for the implementer.

we know that a program consists of multiple functions [49], when a program is running (a process) a number of these functions [50] should be called to fulfill the required job. In run-time context, a function `B` starts its life when it's called by another function `A`, so, the function `A` is the *caller*, that is, the function that originated the call, and the function `B` is the *callee*. The caller can pass a bunch of parameters to the callee which can reach the value of these parameters while it's running, the callee can define its own local variables which should not be reached by any other function, which means that these variables can be removed from the memory once the callee finishes its job. When the callee finishes its job, it may *return* some value to the caller [51]. Finally, the run-time environment (the processor in the case of compiled languages) should be able to know, when the callee finishes, where is the place of the code that should be executed next, and logically, this place is the the line in the source code of the caller function which is next to the line that called the callee in the first place.

In x86, each process has its own run-time stack [52], we can imagine this run-time stack as a big [53] memory region that obeys the rules of stack data structure. This run-time stack is divided into multiple mini-stacks, more formally, these mini-stacks are called *stack frames*. Each stack frame is dedicated to **one** function which has been called during the execution of the program, once this function exists, its frame will be removed from the larger process stack, hence, it will be removed from the memory. The x86 register `EBP` (which is called the *stack frame base pointer*) contains the starting memory address of the current stack frame, and the register `ESP` (which is called the *stack pointer*) contains the memory address of the top of the stack. To push a new item into the run-time stack, an x86 instruction named `push` can be used, this instruction decrements the value of `ESP` to get a new *starting* memory location [54] to put the new value on and to keep `ESP` pointing to the top of the stack, decrementing the value of `ESP` means that the newly pushed items are stored in a lower memory location and that the run-time stack in x86 *grows downward*. When we need to read the value on the top of the stack and removes this value from the stack the x86 instruction `pop` can be used which is going to store the value (which resides on the top of stack) on the specified location on its operand [55] and then increments the value of `ESP`, so the top of stack now refers to the previous value. Note that the `pop` instruction only increments `ESP` to get rid of the popped value and didn't clear

[49]Or *routines* which the other term which is used to describe the same thing.

[50]Not necessarily all of them.

[51]Some programming languages, especially those which are derived from Algol differentiate between a *function* which **should** returns a value to the caller, and a *procedure* which **shouldn't** returns a value to the caller.

[52]We claim that for the purpose of explanation. But actually the matter of separated run-time stack for each process is a design decision that the operating system's kernel programmer/designer is responsible for.

[53]Or even small, that depends on practical factors.

[54]Whether the new memory location is a starting memory location of pushed item or not depends on the size of pushed item as we will explain in a moment.

[55]Can be a register or a memory location.

it from memory by, for example, writing zeros on its place [56], that's better for performance.

To make the matter clear with how `push` and `pop` work, lets take an example. Assume that the current memory address of the top of stack (`ESP`) is `102d` and we executed the instruction `push A` where `A` is a character encoded in UTF-16, which means its size is 2 bytes (16 bits) and it is represented in hexadecimal as `0x0410`, then the processor is going to subtract `2` (because we need to push `2` bytes into the stack) from `ESP` which gives us the new memory location `100d`, then the processor stores the first byte of UTF-16 `A` (`0x04`) in the location `100d` and the second byte in the location `101d` , the value of `ESP` will be changed to `100d` which now represents the top of the stack (and the first byte of UTF-16 `A`), when we need to `pop` the character `A` from the top of the stack, both bytes should be read and `ESP` should be **incremented** by `2`. In this case, the new memory location `100d` can be considered as a *starting* location because it doesn't store the whole value of `A` but a part of it, the case where the new memory location is not considered as starting memory location is when the newly pushed values is pushed as whole in the new memory location, that is, when the size of this value is `1` byte.

When a function `A` needs to call another function `B`, a new stack frame for `B` should be created but before doing that, it's the responsibility of the caller `A` to push the parameters which should be passed to callee `B` onto the stack, that means the parameters of `B` will be stored on the stack frame of `A`, after that the x86 instruction `call` can be used to jump to function `B` code, this instruction pushes the current value of `EIP` (this is, the returning memory address) onto the stack and that's is going to help the processor later to know which instruction of the running code should be executed after the function `B` finishes, at this stage the value of `EIP` is the instruction of `A` which is after the `call` instruction, and the value of `EBP` represents the starting memory address of the function `A` while the value of `ESP` which is the top of stack contains the returning memory address. When `call` instruction is executed, the code of function `B` starts to execute. It's the job of the function to create its own frame, therefore, the first piece of any function code should be responsible for creating a new stack frame, this happens by moving the value of `ESP` which contains the memory address of the top of stack to the register `EBP`, but before that, we should not lose the previous value of `EBP` which is currently the starting memory address of the caller `A` because we are going to need it when we return from the callee `B`, so, the function `B` should push the value of `EBP` onto the stack and only after that it can change `EBP` to the value of `ESP` which creates a new stack frame for function

[56] And this is one of the reasons when you refer to some random memory location, for example in C pointers, and you see some weird value that you probably don't remember that you have stored it in the memory, once upon a time, this value may have been pushed into the run-time stack and its frame has been removed. This same practice is also used in modern filesystems for the sake of performance, when you delete a file the filesystem actually doesn't write zeros instead of the original content of the file, instead, it just refer to its location as a free space in the disk, and maybe some day this location is used to store another file (or part of it), and this is when the content of the deleted file are actually cleared from the disk.

`B`, at this stage, the both `EBP` and `ESP` points to the top of the stack which contains the memory address of the previous `EBP`, that is, the starting memory location of `A`'s stack frame. Now, the currently running code is function `B` with its own stack frame which contains only one item that has been push by `B` itself. Depending of `B`'s code, new items can be pushed onto the stack, and as we have said before, the local variables of the function are pushed onto the stack by the function itself. Also, the function `B` should be able to access the parameters which have been passed by `A`, those parameters, as we know, are stored on the stack frame of `A` instead of `B`, so, to be able to reach to those values `EBP` can be used. The memory location which is stored in `EBP` contains the previous value of `EBP` and the entry before that is the old value of `EIP` which has been pushed by `call` instruction, and before this item the values of the parameters of `B` can be found. We can see here clearly that the operations of the stack `push` and `pop` should not be used, but we should read the memory addresses directly instead. When `B` finishes, it should deallocates its own stack frame, this task can be accomplished easily by popping all values of `B`'s stack frame until we reach to first value pushed value by `B` which is, again, the starting memory address of the caller `A` stack frame then we set this value to the register `EBP` instead of the current value. In this stage, the top of the stack contains the returning memory address which should be loaded to `EIP` so we can resume the execution of the caller `A`, that's can be done by using the x86 instruction `ret` which pops the stack to get the returning address then loads `EIP` with this value. Finally, when `A` gains the control again it can deallocate the parameters of `B` to save some memory by just popping them. The details of calling another function that we have just described are **implementation details** and we mentioned previously that these implementation details of function's invocation are known as calling conventions. The calling convention that we have described is known as `cdecl` [57], there are other conventions, which means the one which we have described is not an strict standard for x86 architecture, instead, the operating systems, compilers and low-level code writers can decide which calling convention that we would like to use [58] according to their objective. However, the reason behind choosing `cdecl` to be explained here is that it is a well-known and widely used calling convention , Also, it serves our purpose of explaining the basics of x86 run-time stack.

When we explained how x86 instructions `push` and `pop` work, we have claimed that the x86 run-time stack *grows downward*, so, what does growing downward or upward exactly means? Simply, when we said that x86 run-time stack grows downward we meant the the older items of stack are pushed on larger memory addresses while the most recent ones are pushed onto smaller memory addresses. For example, starting from the memory address `104d`, let's assume we have pushed the value `A` after that we pushed the value `B`, then `A`'s memory location will be `104d` while `B`'s memory location will be `100d`, so the new values will always be pushed on the bottom of the old ones in the memory. What makes we

[57]Stands for *C Declaration*

[58]Or maybe make up a wholly new one

claim that, for instance, the address `100d` is at the bottom of `104d` (instead of the other way around) is how we visualize the run-time stack inside the main memory. Let's look at the figure which shows a run-time stack that contains three items `M`, `R` and `A` and all of them are of `4` bytes, on the right corner of the figure we can see the starting memory address of each item. As we can see, in this visualization of the run-time stack, the smaller memory addresses are on the bottom and the larger memory addresses are on the top. Let's assume, we would like to run the instruction `push C` on this run-time stack, and from the figure we can see that the value of `ESP` is `8d` [59], as we have mentioned before, the instruction `push` of x86 is going to decrease the value of `ESP` by the size of data which we are pushing to get a new starting memory address for the new item. Let's assume that the size of `C` is `4` bytes, then `push` is going to decrease `8d` (current `ESP` value) by `4d` (The size of pushed item `C` in bytes) which gives us the new starting memory location `4d` for the item `C`. If we visualize the run-time stack after pushing `C` it will be the one as on figure and we can see, depending on the way of `push` instruction works, that the stack grew downwards by pushing the new item `C` on the bottom. So, according to this visualization of run-time stack, which puts larger memory addresses on the top and smaller on the bottom, we can say x86 run-time stack grows downward *by default*. This visualization is just one way to view how the run-time stack grows, which means they may be other visualizations, and the most obvious one is to reverse the one that we have just described by putting the smaller addresses on the top and the larger addresses on the bottom as shown in figure , you can note that in contrast of figure Fig10062021_1 the smallest address `4d` is on top, so, based on this visualization the stack grows upward! Actually this latter visualization of run-time stack is the one which is used in Intel's manual and the term *expand-up* is the term that is used in the manual to describe the direction of stack growth. To sum it up, the direction in which the run-time stack grows (down or up) depends on how do you visualize the run-time stack, as in figure Fig10062021_1 or as in figure Fig10062021_2 . Our discussion in this book we are going to depend on the first visualization[60], so, simply, the run-time stack of x86 grows downward.

We have emphasized that x86 run-time stack **by default** grows downward, this default behavior can be changed if we wish to which is going to make the run-time stack to grow upwards instead and the way to do that is to used expansion-direction flag [61] of run-time stack's segment descriptor. When we want the run-time stack to grow downward [62] the value of this flag should be

[59] As a reminder, don't forget that all these memory address are actually **offesets** inside a stack segment and not a whole memory address.

[60] And many other books actually uses the first visualization as I recall and for that I chose it in this book. And according to my best knowledge the only reference that I've seen depends on the second visualization is Intel's manual.

[61] Which we have mentioned when explained the structure of segment descriptor and postponed its details till here.

[62] Or in Intel's term which depends on the second visualization of run-time stack: **expand-up**

0, on the other hand, when we want the run-time stack to grow upward [63] the value of this flag should be 1. Modern operating systems use the default behavior [64], so, the value of expansion-direction flag will be 0, but the other available option is there to solve a potential problem and whether this problem is going to show up in a specific kernel depends on how this kernel's architecture is designed [65]. This problem, which we can solve by making our kernel select to make the run-time stack grows upward instead of downward, is related to the need of growing the size of run-time stack and the fact that the run-time stack stores memory addresses [66] on it. Let's assume that our kernel created a new stack segment for a specific process X and this stack segment has a fixed size which is 50 bytes [67] for example. The process X starts its work and at some point of time its run-time stack of size 50 bytes becomes full which means that we need to resize it so it can hold more items, and let's assume the new size is 60 bytes. Before going any further with our discussion, let's see the figure which represents a snapshot of process X's run-time stack when it became full. We can see from the figure that X's stack segment starts from the **physical** memory address 300d [68] and ends at the **physical memory address** 250d, also, the items of run-time stack are referred to based on its offsets [69] inside the stack segment. We can see that a bunch of values have been pushed onto the stack, some of those values are shown on the figure and some other are omitted from the figure and replaced by dots which mean there are more values here in those locations. Normal values are called "some value" in the figure and the last pushed value in the stack is the value Z, also, a value which represents a **logical memory address** has been pushed onto the stack [70], as we explained earlier, all memory addresses that the processes work with are logical and not physical. This value 26d is a local variable P of the type pointer [71] which points to another local variable R that has the value W and is stored in the memory location 26d. Figure shows X's stack after resize, as you can see we have got our new free space of 10 bytes, also, because the stack grows downward so the new free space should be added at the bottom of the stack to be useful which means the previous offsets should be changed, therefore, the largest offset 50d

[63]Or in Intel's term: **expand-down**

[64]We will see that this design decision is taken due to the choice of *flat memory model* by modern operating systems

[65]That is, which *memory model* is used in this kernel as we will see later

[66]The previous values of EBP and EIP. Also the application programmer may store memory addresses of local variables in the stack (e.g. by using pointers in C)

[67]As you may recall, the size of the segment can be decided by the base address of the segment and its limit as specified in the segment's descriptor

[68]Segment's base address.

[69]As in figures Fig10062021_0, Fig10062021_1 and Fig10062021_2 but we haven't state that explicitly there since it doesn't affect the discussion.

[70]More accurately, this value represents an **offset** within the current stack segment, a **full** logical memory address actually consists of both offset **and** segment selector as we have explained earlier in this chapter when we discussed address translation. But for that sake of simplicity, we are going call this stored value as "memory address" or "memory location" in our current explanation.

[71]As in C

has been updated to `60d` by adding `10d` (which is the newly added free space to the stack in bytes) to it and so on for the rest of offsets, also, `ESP` has been simply updated in the same manner. Now we can see that process of updating the offsets, that we are forced to perform because the stack grows downward, has caused a problem in the offsets which have been pushed onto the stack before resizing it. You can see the pointer `P` which still has the original value `26d` which doesn't point the the variable `R` anymore, instead it is going to point to another memory location now with a value other that `W`, and the same problem holds for all pushed `EBP` values on `X`'s stack. A potential solution for this problem is to update all stack items that contain memory addresses in the range of stack after resizing the it, exactly as we have done with `EBP`, but more simpler solution is to make the stack to grow upwards instead of downwards! [72]

Now let's see what happens in the same scenario but with changing the growth direction if the stack from downward which caused the problem to upwards. In this case, as we have said before, the new items will be stored on the larger memory addresses [73]. Figure shows the same snapshot of `X`'s run-time of stack as in the one of figure Fig10062021_3 but this time it grows upwards instead of downward. You can notice that the older values are now on the bottom of the stack, that is, on smaller memory addresses, what interest us in this stack is the entry which stores the memory address `20d` that points to the memory location which has the value `W` and it is the one which caused the problem in the first place. When the stack was growing downward, the memory location of the value `W` was `26d`, but this time it is `20d`. So, what happens when we need to resize this run-time stack? In the same way of the previous one, the limit of the stack [74] will be increased from `50d` to `60d` as shown in figure , but in contrast to the previous one, we don't need to update the value of `ESP` anymore, because as you can see from the two figures Fig12062021_1 and Fig12062021_2 the memory address `50d` represents the top of the stack on both stacks. The same holds for the stack item which stores the memory address `20d`, we don't need to update it because the value `W` is still on the same memory address and can be pointed to by the memory address `20d`. So, we can say that deciding the direction of run-time stack growth to be upward instead of downward can easily solve the problem of getting wrong stored memory address after resizing the run-time stack [75].

[72] Modern operating systems solves this problem by not dividing the memory into segments, but instead they view the memory as one big segment for all data, code and stacks and that is essentially the flat memory model.

[73] Offsets to be more accurate

[74] Its largest offset.

[75] Actually, the well-know stack overflow vulnerability in x86 is also caused by stack growing downward and can be avoided easily in growing upwards stacks!