The Holy Book of x86

Volume 2

Intel Architecture - Windows Internals - Linux Internals

Based on 64-bit Mode

Arash TC

I AM THE KEY TO THE LOCK IN YOUR HOUSE THAT KEEPS YOUR TOYS IN THE BASEMENT,
AND IF YOU GET TOO FAR INSIDE,
YOU'LL ONLY SEE YOUR REFLECTION.

[THOM YORKE - CLIMBING UP THE WALLS]

Contents

Acknowledgement

I owe everything I know about x86 architecture to **Xeno Kovah**. A man who shared his class videos and slides freely available to everyone which is a noble act. In return to his great efforts, I decided to write this tutorial on x86 architecture and assembly and publish it for free so everyone who is interested can learn and contribute.

Introduction

This book/guide/tutorial/wiki is about assembly and x86 architecture. It's written by a low-level security dude for low-level security dudes.
If you want to learn Assembly and its structure, reversing basics, Segmentation, Paging, etc. Keep on reading. I highly recommend you check opensecuritytraining.info website and watch Intermediate x86 videos as you read volume 2. This book will teach you x86 architecture with the perspective of Information Security and Trusted Computing. To follow latest updates and added content to this book, please visit the author's web site at this link:
http://www.kernelfarm.com/tutorials/the_holy_book_of_x86.html
or view the project on Github:
https://github.com/Captainarash/The_Holy_Book_of_X86

Praise to volume 1

As I received great feedback from the readers, that pushed me to start writing the volume 2. I hope you all get the most out of the hours of research spent on each paragraph of the 2nd volume.

About the author[s]

Arash TC is the main author and maintainer of this book. He is currently studying IT in Finland and works as an Information Security Engineer at his university. He will appreciate readers' comments, criticisms and contributions. His main interest is low level security and kernel internals.

Other contributors are very appreciated as they help me to complete this project and present you a book which hopefully will be flawless.

You can contact the author[s] by visiting http://www.kernelfarm.com/

Introduction to volume 2

This book/guide/tutorial/wiki will explain and dig into the specifics of the x86 architecture. In order to make sure of the integrity of the content of this book, hours were spent on each and every paragraph. Presenting such a content and spending this amount of time didn't discourage the author[s]. Opposed to the common sense that provides such detailed content in an expensive book with shiny hard covers, the author decided to release it freely although he is almost broke. So please keep that in mind and move towards sharing your knowledge freely because you are entitled to nothing.

This book is divided into 3 sections. Section 1 explains x86 architecture as the Intel manual suggests. Section 2 we will dig into windows internals and review section 1's content in respect to Windows and Section 3 in respect to Linux. Focus will be on 64-bit mode. If you're interested in 32-bit, go buy some book published 10 years ago.

You need Intel Developer's Manual as a quick reference throughout this book. You can download it from the link below:
https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

Section 1

Raw Intel 64 Architecture

Chapter 0x01 - Quick Overview and Introduction

Modes of Operation

Intel's IA-32 architecture supports 3 different modes:

Real Mode: Real mode or real-address mode is a 16-bit mode only and you see it shortly when your reset or turn on your PC. There are no privilege rings and no virtual memory in real mode. It implements the programming environment of Intel 8086 processor with extensions (such as the ability to switch to protected mode or system management mode). DOS runs in Real-Mode.

Protected Mode: This mode is the native state of the processor. It offers privilege rings, virtual memory, paging, segmentation, multi-tasking, etc. Among these capabilities, it also supports running DOS programs which run in Real Mode (16-bit) as a backwards compatibility and Intel named it as Virtual-8086 mode. This name tells the story behind it and confirms that it's not a separate mode and it's only a backwards compatibility within Protected Mode. All modern OSes operate in Protected Mode.

System Management Mode: This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC). This is all you need to know about SMM for now but to just hype you up, SMM is a popular target for advanced rootkits since when it starts executing, It allocates its own isolated locked down memory so neither ring 0 nor a hyper-visor (ring -1 oh yeah we have negative rings too! You go deeper, you may discover God down there) can access its memory. That's why SMM is sometimes referred to as ring -2 because even a hyper-visor can't read its memory. SMM can access all memory and there is hardware support to lock down SMM so when you put some code into SMM, BIOS will lock it down and nothing can ever access it.

Intel's IA-32e architecture adds IA-32e mode which has 2 sub-modes as described below:

Compatibility Mode: Compatibility mode permits most legacy 16-bit and 32-bit applications to run without re-compilation under a 64-bit operating system. On a 64-bit Operating system, this mode will replace protected mode and they are mostly identical. Their execution environment are the same. It also supports all of the privilege levels that are supported in 64-bit and protected modes. Legacy applications that run in Virtual 8086 mode or use hardware task management will not work in this mode.

64-bit Mode: This mode enables a 64-bit operating system to run applications written to access 64-bit linear address space. 64-bit mode extends the number of general purpose registers and SIMD extension registers from 8 to 16. General purpose registers are widened to 64 bits.

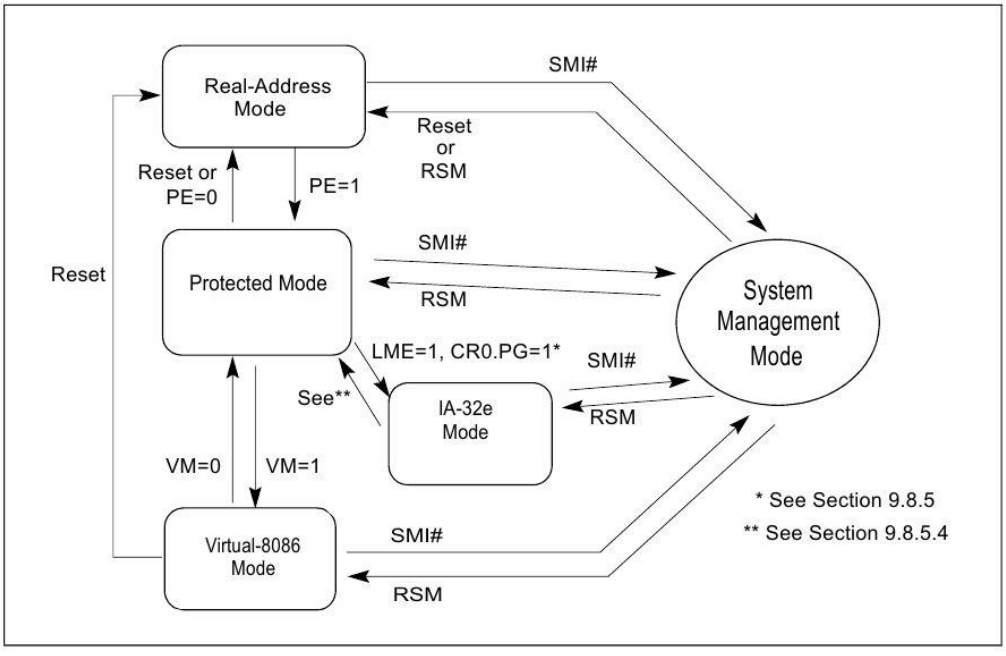

Figure 1- 1

Privilege Rings

As discuss earlier in Vol 1, there are privilege rings which define levels of access for an object, process, thread, memory page (you name it) in a system. We have 4 different rings. Ring 0 has the highest privilege, ring 3 has the least. Although Modern Operating Systems never use ring 1 and 2 and all the operations are divided into ring 0 (kernel mode) and ring 3 (user mode). x86 privilege rings are enforced by hardware.

Chapter 0x02 - Segmentation

Introduction

As mentioned earlier in the Introduction section, this book will focus on 64-bit version mostly. Segmentation is generally disabled in 64-bit. Anyways, I decided to explain it fully because it is fully enabled when running in IA-32e Compatibility Mode. You'll see why Segmentation is generally (but not completely) disabled in 64-bit mode after explaining segmentation fully. Keep in mind that we're explaining segmentation based on 32-bit. There are a few terms we need to know before explaining segmentation.

Linear Address Space: Linear address space is the processor's addressable memory and it's a flat 32-bit space. Until we introduce paging, we refer to physical memory as linear address space.

Physical Address Space: Physical address space is a rage of address that the processor can generate. If you think about it, it basically depends on how much RAM you got up to a limit of 2^32 which is 4 GB. So that means you can have more than 4 GB of RAM on a 32-bit OS, right? Well, there is something called PAE (Physical Address Extension) allows a 32-bit OS to access up to 64 GB of RAM. We'll talk about PAE more later.

Logical Address: A logical address (also referred to as far pointer) consists of a 16-bit segment selector and a 32-bit offset into that segment.

Segmentation

Segmentation is basically the way that the processor divides addressable memory into different segments which can be protected by assigning read, write or execute flags and a way to translate a logical address into a linear address.

To access (or select) a segment you should access something called a segment selector. A segment selector is a 16-bit value held in a segment register. In Intel IA-32 architecture we have 6 segment register:

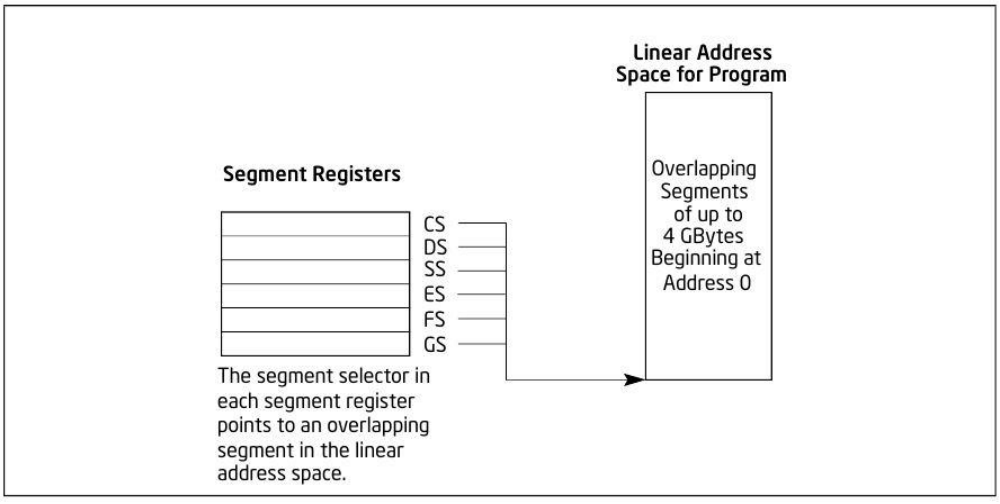

Figure 2- 1

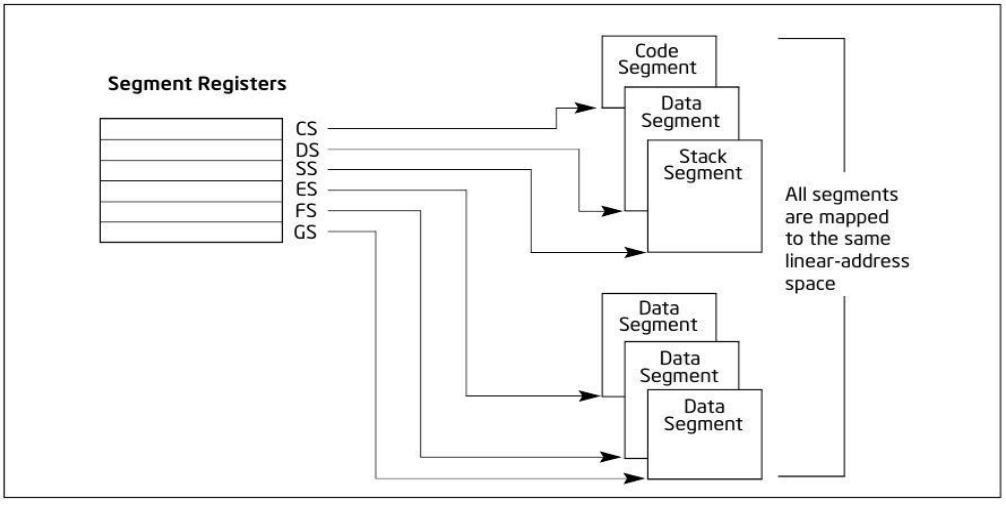

Figure 2- 2

CS or code section in the most basic form is where all the code of some process (or thread to b specific) resides and in its most basic form, it's readable, executable but not writable. DS or data section has read access but no write or execute (in its most basic form).

An important note worth mentioning here is that CS and DS (and SS which is basically a DS with read/write access) are the mostly used segments. ES, FS and GS are there for you. You can use them however you want. We will explore more about these segment registers when we do some debugging sessions in windows in the next section of this book.

As mentioned earlier logical address is a 16-bit segment selector plus a 32-bit offset into that segment and that translates to a linear address. So

when we want to access a byte in physical memory[1], we say I want to access this segment and I want the byte at this offset into that segment.

The 16-bit value of the segment selector is an offset into a table called Descriptor Table. Each index in the descriptor table defines a base address and a limit (think of it as a chunk of memory) and you take the base address from the entry pointed at by your segment selector, and add the 32-bit offset to that base address to get to the place you wanted. Look at figure 1-4.

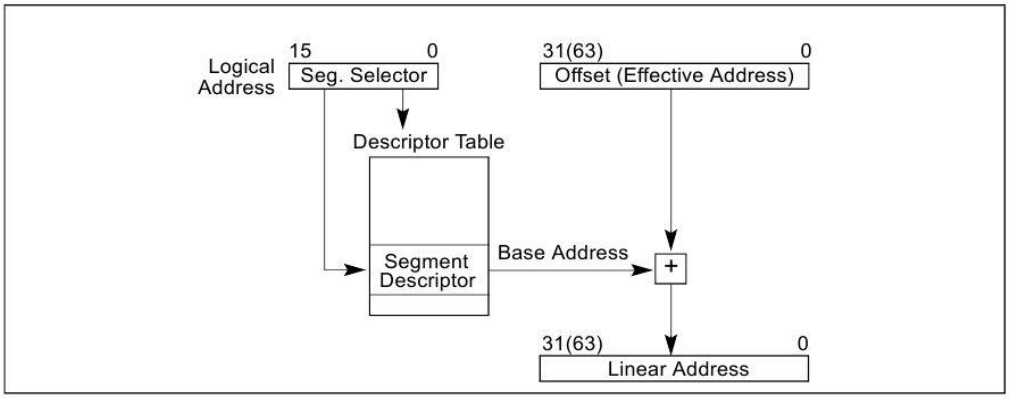

Figure 2- 3

The 16-bit value of a segment selector is divided into 3 section. A 13-bit offset (so the actual offset is not 16-bit), one bit which determines you want an offset from GDT or LDT (which we talk about in a minute) and a 2-bit section which determines Requested Privilege Level (RPL). Here's the actual placement of the bits in a segment selector:

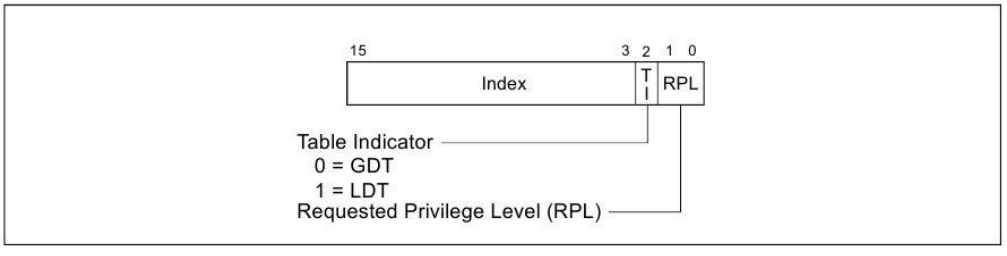

Figure 2- 4

Requested Privilege Level may give you some notion of security implementation and privilege rings but for now just keep that in your mind. In figure 1-4, we can see bit 3 to 15 is the actual offset to the descriptor table

[1] Until we explain paging, a linear address is a physical address.

and bit 2, is a table indicator which specifies which table we want to go into; either GTD or LDT but what are they exactly?

GDT and LDT

A segment descriptor table is an array of segment descriptors and it can have up to 8192 (2^13) entries and each of these entries are 8 bytes long. We have 2 types of segment descriptor tables, GDT and LDT.

Each system must have one GDT (Global Descriptor Table) which is visible to all running threads and task. Each entry of GDT is basically a 32-bit base address which later the offset will be added to it (figure 1-4) to get to the intended linear address.[2]

Same goes for LDT but LDT is actually found via GDT (Explained later). LDT or Local Descriptor Table is a per process descriptor table. There can be one or more LDTs present in an OS for each process.

Still there is one more question unanswered: Where or how does the hardware or the OS find the GDT and LDT? Via 2 registers called GDTR (GDT register) and LDTR (LDT register).

[2] GDT can contain LDT, TSS or Call Gate. You'll read about them in the later chapters.

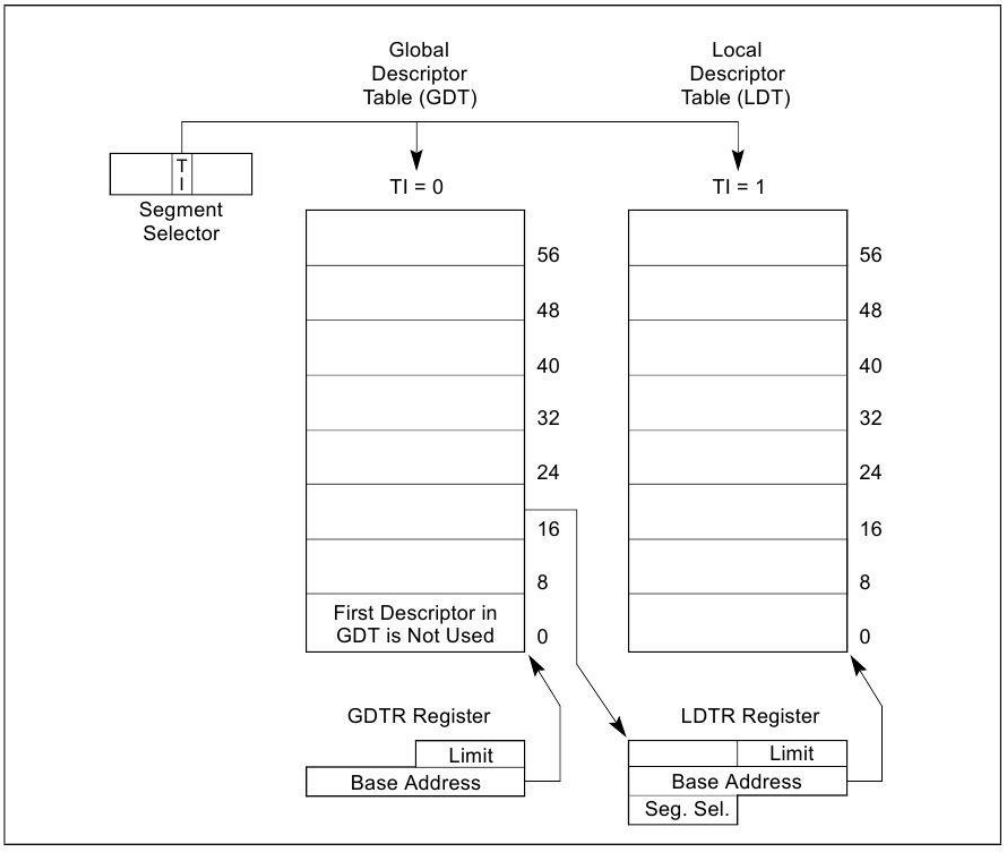

Figure 2- 5

GDTR is a 48-bit register that consists of a 32-bit base address and a 16-bit table limit which indicates the size of the table.[3]

LDTR on the other hand in just a 16-bit segment selector which goes through GDT, finds the entry it wants from the GDT and through that, finds the LDT. So as we mentioned before LDT is found via GDT and the entry in GDT which is pointing to the LDT has a flag set which indicates "I'm an LDT"! You'll see shortly what are these GDT entries made of. Of course they are not just a 32-bit base address.

GDT is a descriptor table. These **descriptor tables** are just a big array of **segment descriptors**. Now is the time we dig down more to find out what the segment descriptors (the entries) are made of and what information do they carry. Each segment descriptor tells the address of the first byte in the

[3] On 64-bit, the base address is expanded to 64 which makes the GDTR's size 80 bits.

segment (base address), privilege rings and access rights for that segment, the size of the segment and some other information about the segment.

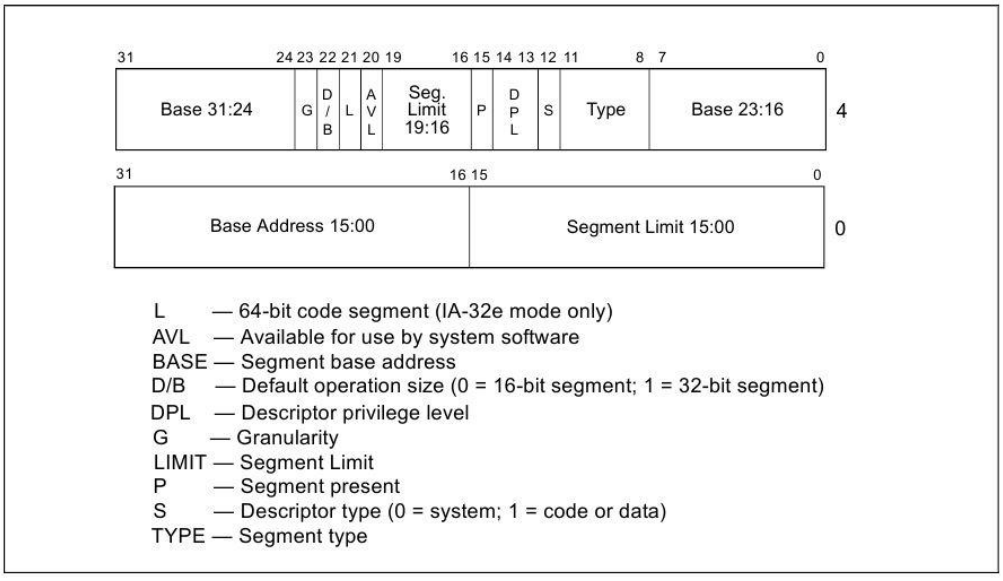

Figure 2- 6

Above picture may sound confusing. Remember every entry in GDT is 8 bytes long. Every segment descriptor is divided into 2 parts, each 32 bits long. The lower 32 bits define a segment limit (bit 0 to 15) and a base address (bit 16 to 31).

The upper 32 bits define 2 base addresses (bit 0 to 8 and bit 24 to 31) and another limit (bit 16 to 19). To understand why is that happening, we need to know that if we want to access memory which is divided into 4-Kbyte chunks, we need a 20-bit value. (2^20 x 2^12 = 2^32 :D). So the OS will stick those 2 parts (the 16-bit limit and the 4-bit limit) and comes up with a 20-bit limit value. Now what happens to the base addresses? Exacts same thing as the limit. A 32-bit linear address will be created by sticking the 16-bit value found in the lower 32-bit part (bit 15 to 31) and the two 8-bit values found in the upper 32-bit part (bit 0 to 8 and bit 24 to 31).

The processor interprets the segment limit value in 1 of the 2 ways below:

- If the **granularity flag** is **clear**, the segment size can range from 1 byte to 1 Mbyte, in **byte** increments.
- If the **granularity flag** is **set**, the segment size can range from 4 Kbytes to 4 Gigabytes, in **4-Kbyte** increments.

The **D/B flag** define how the processor should interpret the opcodes. For example, one instruction can operate in different modes (16, 32 or 64 mode) but its opcodes all the same for all modes; So the processor checks this flag to decide whether to treat the opcode (or the instruction) as executing in 16, 32 or 64-bit mode.

The **DPL flag** is Descriptor Privilege Level. It defines which ring level can access this code. For now, just keep it in mind, we'll discuss the privilege rings and access rights in later chapters.

The **S flag** or system flag defines whether the segment is a system segment (0) or a Code or a Data segment (1).

The **P flag** or present flag declares if the segment is present (1) or not present (0). If some segment register gets loaded with an address which points to a non-present segment, the processor will throw a segment-not-present exception (#NP).

The **L flag** in IA-32e mode declares if the code segment contains native 64-bit code. If it's set to 1, then that code segment must be executed in 64-bit mode. If it's set to 0, then that code segment must be executed in compatibility mode. If the L bit is set, then **D/B flag** must be cleared because there is no need to clarify how the opcodes must be treated meaning that the opcodes must run in 64-bit mode. When running a native 32-bit OS, 64-bit mode is not present and the L flag is reserved and always set to 0.

If the **S flag** is set to **1**, the **type flag** will define the type of the segment meaning, access rights, read/write/execute, etc.

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

Figure 2- 7

An important thing to notice in the picture above is “expand-down”. A segment is in-bound from base address to base-address plus limit value, but not always. When a segment is an expand-down segment, its boundary is from base address to base address minus limit.

There is also “conforming” segments which is about privilege rings. I try to describe it shortly but there is more to it. We explain privilege rings fully in later chapters. You can only access a non-conforming code segment with the same privilege. However, if a code segment is conforming, it shares its procedure (or code) with the calling program (or thread) so there is no change in privilege ring. A transfer of execution into a more-privileged conforming segment allows execution to continue at the current privilege level. A transfer into a nonconforming segment at a different privilege level results in a general-protection exception (#GP). An example of accessing a conforming code segment are math libraries and exception handlers.

Keep that in mind that execution cannot be transferred by a call or a jump to a less-privileged code segment, regardless of whether the target segment is a conforming or nonconforming code segment. Attempting such an execution transfer will result in a general-protection exception (#GP).

Now let’s pause for a minute. We need to understand what happens when we want to execute a code in ring 0 when we are in ring 3. When we want to execute some code, which resides in ring 0, you cannot directly jump

to a ring 0 code segment and start executing. You must hand execution to a call gate or a task gate to do the job for you and get back to you with the results. In between there will be various security checking and transition which we will explain along the book.

Current Privilege Level

The **CPL** is the privilege level of the currently executing program or task. It is stored in the first 2 bits of the CS and SS segment registers. It defines whether a thread or task is currently at ring 0 or ring 3. You may think: "Sounds interesting! I can easily load a value in CS register which gives me ring 0 access and I become root!", but I have to stop you right there. Intel has the notion of privileged instructions. You must be at ring 0 (CPL=0) to execute a privileged instruction which mostly have something to do with segment and control registers.[4] Plus, MOV instruction cannot be used to load values into CS register.

There's always a privilege ring checking happening, when you select a segment, when you execute an instruction, when you talk to kernel, etc. That privilege checking in its most basic form is:

If CPL <= DPL, then access is granted.

Call Gate

Call gates are basically a way to transfer execution from on segment to another which may be at different ring levels, with different sizes (in term of whether they're 16-bit or 32-bit mode). A call-gate descriptor may reside in the GDT or in an LDT, but not in the interrupt descriptor table (IDT)[5]. The key point of a Call Gate is that when kernel wants to export some of its functionality to userspace in a controlled manner, it uses a Call Gate which only allows the user space to jump (or call)[6] to a specific location in the kernel space. [7]

---

[4] Intel® 64 and IA-32 Architectures Software Developer Manuals - Vol. 3A – Chapter 5 – 5.9 privileged instructions - Page 5-23

[5] Explained Later 😃 Be patient! 😉

[6] CALL/JUMP FAR-POINTER is used when you want to use a call gate in x86.

[7] int 0x80 on Linux and int 0x2E on Windows don't use Call Gate. They use interrupts but it's good to know what a Call Gate is.

As mentioned in previous paragraph, a Call Gate resides in GDT (or LDT) so when you select an entry from GDT which is a Call Gate, that entry looks like this:

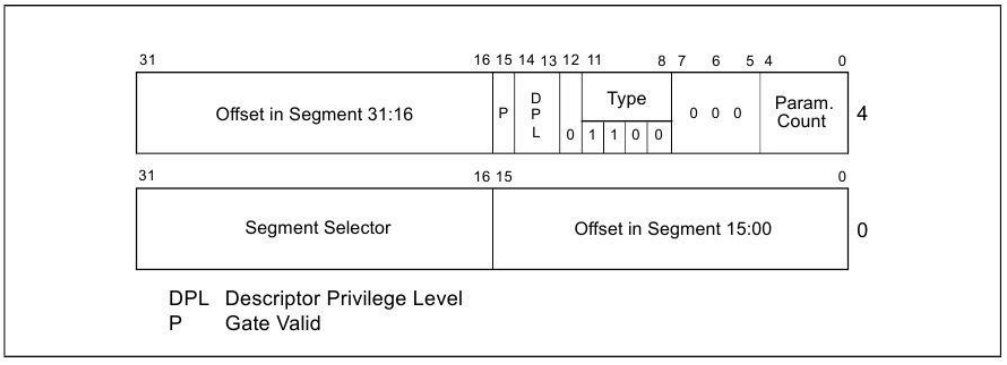

Figure 2- 8

A Call Gate entry in GDT, has a 16-bit segment selector and a 32-bit offset which eventually takes you to the specific predefined location by the kernel to execute whatever function you ask for. On top of that, there is a DPL flag that specifies what ring can access this entry. It also has P (present) flag and Type flag which was explained earlier. There is also a Parameter Count flag which defines how many parameters you should pass to that specific call gate. Later when we introduce Ring 0 Stack vs Ring 3 stack and the concept of stack switching, this part of the puzzle will eventually gets filled in your head.

On IA-32e mode (64-bit), the Call Gate descriptor looks like this:

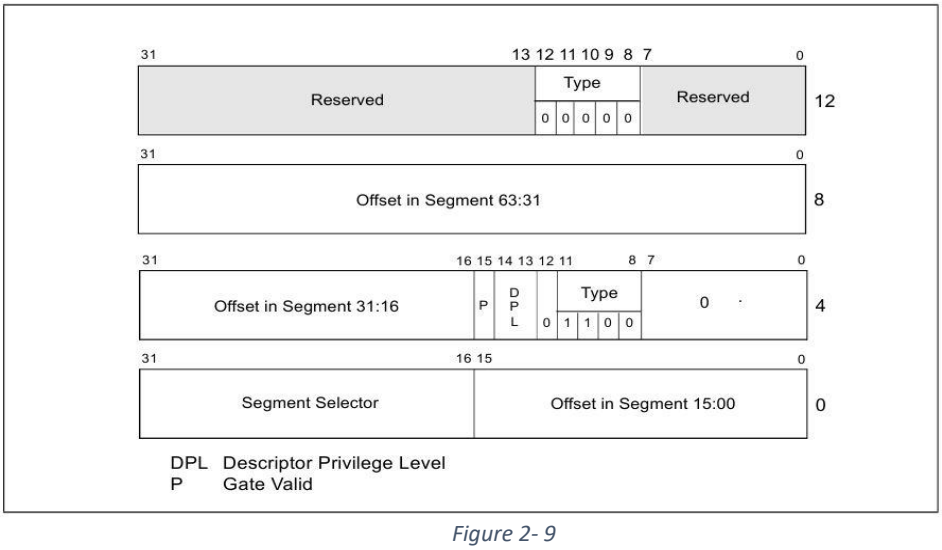

Figure 2- 9

Figure 1- 10

The use of Call Gates has become very rare because of the introduction of new sort of instructions to talk to kernel which are SYSENTER/SYSEXIT and SYSCALL/SYSRET.

In modern operating systems, segmentation is not used for memory protection anymore. Instead, they use a flat memory model which puts the whole linear address space into one segment with read/write/execute permissions and rely completely on paging for security.

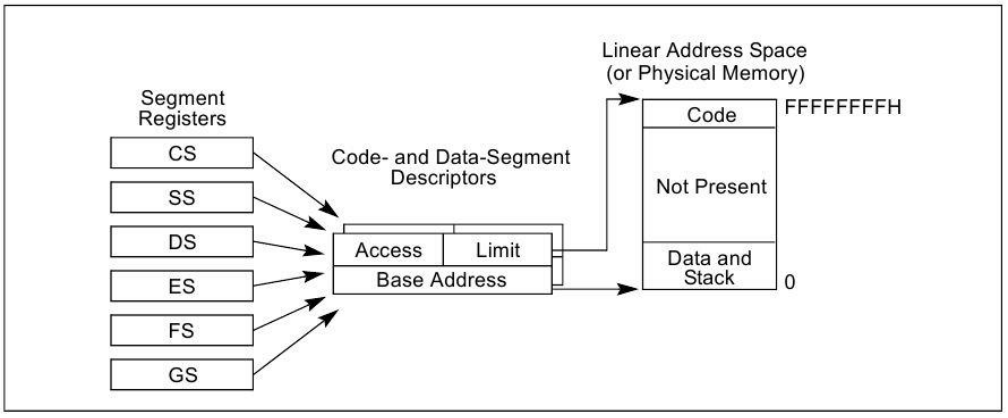

Figure 2- 10

Chapter 0x03 - Paging

Introduction

In previous chapter, we learned how a logical address gets translated to a linear address using segmentation. In modern operating systems, segmentation is generally disabled and a flat memory model is used So all virtual address space[8] (0x0… to 0xff…) is defined inside one segment. When paging is disabled, logical addresses map 1:1 to physical addresses. But when paging is enabled, a linear address must be translated into a physical address.

The name “paging” is chosen because physical memory gets divided into fixed size chunks like the pages in a book and exactly like a page in a book, when you want some information written specifically in a page of a book, you go to library and find the shelf, the you look at the indexes to find the page that you want. Figure 3-1 shows the big picture of the journey of a logical address until it reaches physical memory[9].

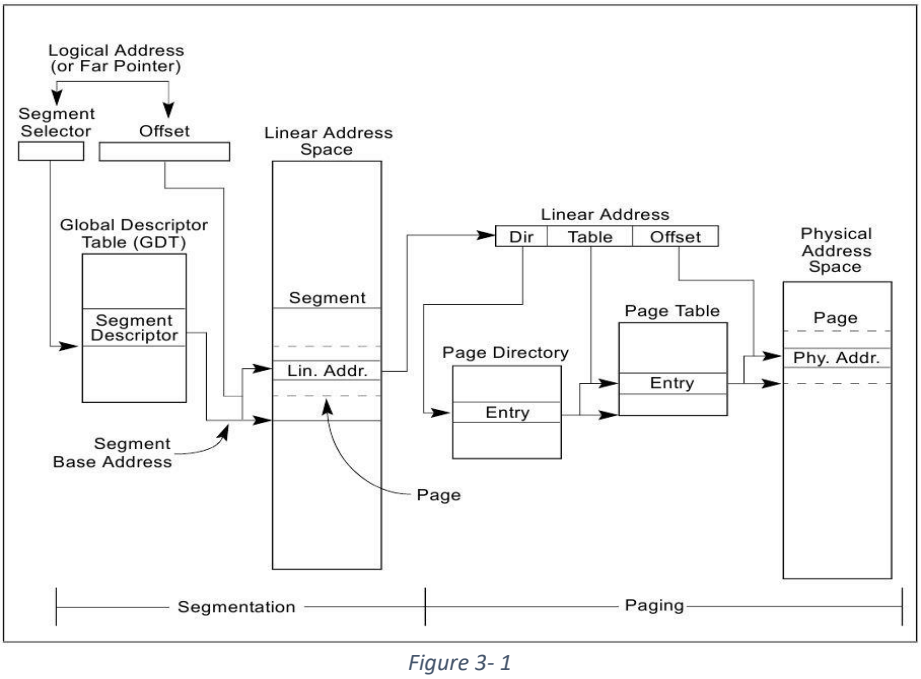

Figure 3- 1

[8] Virtual address space is the same as linear address space. So, a virtual address is a linear address.

[9] Figure 3-1 shows paging in 32-bit non PAE mode. In 64-bit, paging is different. Anyways the picture delivers its purpose.

Alright, let's define some terms and conventions before diving into how paging is done on 64-bit protected mode. First of all, since segmentation uses the flat model on 64-bit, so we may use the terms logical address, linear address and virtual address interchangeably. A linear address in 64-bit only has 48 effective bits. What that means is that bit 0 to 47 define the address. So, what happens to bit 48 to 63?

Canonical Address: When bits 48 to 63 of virtual address are all ones or all zeros, that address is a canonical address. All virtual addresses on 64-bit must be canonical. We can also define a canonical address this way: Bits 48 to 63 must be equal to bit 47. That means we can only have 2 range of address in 64-bit virtual address space:

First valid address for the first range:
0000000000000000000000000000000000000000000000000000000000000000
Last valid address for the first range:
0000000000000000111111111111111111111111111111111111111111111111

⇨ From 0 to 0x7FFFFFFFFFF

First valid address for the second range:
1111111111111111100000000000000000000000000000000000000000000000
Last valid address for the second range:
1111111111111111111111111111111111111111111111111111111111111111

⇨ From 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF

Yeah, I know you all may have the question why not just use all 64 bits? A 64-bit address can address 2^64 bytes which is a very very huge number. So far nobody needs that much of address space and it also brings a lot of complexity if a processor wants to manage the whole 64-bit address space which had no use in the first place. Even now that a 48-bit address is used, it still can access 256 Terabytes of memory. Still, there's a lot of free unused space in such a big address space. On 64-bit protected mode, every virtual address must be canonical. Trying to access a non-canonical address will throw a Page Fault exception.

Control Registers: In x86 architecture, there are some control registers named CR0, CR1, CR2, CR3, CR4, CR8 and EFER.[10]

[10] CR8 register is only available on 64-bit mode. EFER was first introduced in AMD64 and later adopted by Intel x86_64.

CR0 register has various control flags that modify the basic operation of the processor. For example, if bit 0 (AKA PE flag) is set to 1, we're in protected mode, if set to 0, we're in real mode. Bit 31 of CR0 (AKA PG flag) enables Paging if it's set to 1. Note that PG flag, requires PE flag to be set.

CR1 is reserved by Intel for future use.

Whenever a page fault occurs, the **linear address** which caused the page fault gets copied into **CR2** register. This value is called Page Fault Linear Address (PFLA).

CR3 is the most important register when Paging is enabled. CR3 basically holds the **physical address** of a table which is used for paging.[11] On 32-bit mode (or compatibility mode) CR3 point to the base of some table called Page Directory. On 64-bit mode, it points to the base of a table called Page Map Level 4 (PLM4). Remember that CR3 is loaded or changed per process, meaning that each process has its own paging tables and its own view of memory. So, CR3 always point to the Page Directory (or PML4) table of the **current process**.

CR4 contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. We mention some of its flags here, others will be mentioned as they come up.

Physical Address Extension (bit 5 of CR4): When set, enables paging to produce physical addresses with more than 32 bits. When clear, restricts physical addresses to 32 bits. PAE must be set before entering IA-32e mode.

Page Global Enable (bit 7 of CR4): If PGE is flag is set to 1, it allows frequently used or shared pages to be marked as global to all users. Why? Because:

1. Whenever you switch between applications or processes, CR3 must be reloaded.
2. The is a caching mechanism in x86 architecture called **Translation Lookaside Buffer (TLB)**[12] which stores the logical-to-physical mappings for the current process so the processor doesn't go to through all those tables and paging translations to locate physical addresses for each second of execution.

[11] Explained later. Don't want to confuse you now.
[12] Explained in much greater detail in later chapters.

3. TLB gets flushed whenever a task switch happens or whenever CR3 gets reloaded. But if the PGE is set, the OS is allowed to set the tables of frequently used processes as global. Global pages' caches in TLB **don't** get flushed which results in a much better performance. Easy, ha? PGE is 99% of the times enabled by the OS to ensure performance.

CR8 and EFER are related to Interrupts. They are explained in later chapters, we don't care about them for now.

Probing CPU features using CPUID instruction