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 my reflection.

[Thom Yorke - Climbing Up the Walls]

Contents

Acknowledgement 3
Introduction 3
Praise to volume 1 3
About the author[s] 3
Introduction to volume 2 4
Chapter 0x01 - Quick Overview and Introduction 6
Modes of Operation 6
Privilege Rings 7
Chapter 0x02 - Segmentation 8
Introduction 8
Segmentation 8
Chapter 0x03 - Paging 19
Introduction 19
Probing CPU features using CPUID instruction 22
Virtual Address Translation Overview – IA-32e Paging 23
Structure of a Page Entry 26
Linear Address Translation for Larger Pages 27
Chapter 0x04 – Caching 29
Why? 29
Translation Lookaside Buffer (TLB) 29
Paging-Structure Cache 29
Cache Invalidation 29
Cache Control 30
Memory Types 30
Write-back 31
Write Combining 32
Strong Uncacheable 32
Memory Type Range Registers (MTRR) 32
Page Attribute Table (PAT) 32
Chapter 0x05 - Interrupts and Exceptions 36

Privilege Transitions 36
Stack Switching 36
What's an Interrupt? 37
Handling an Interrupt & IDT 37
Hardware Interrupts and Software Interrupts 39
Exceptions 39
Faults 39
Traps 40
Aborts 40
Chapter 0x06 - Exploring PE Files 42
Definition of a PE File 42
Exploring a PE file using CFF Explorer 42
DOS Header 43
NT Header 45
Section Headers 49
Imports 52
Chapter 0x07 - Introduction to WINAPI 54

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 volume 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.

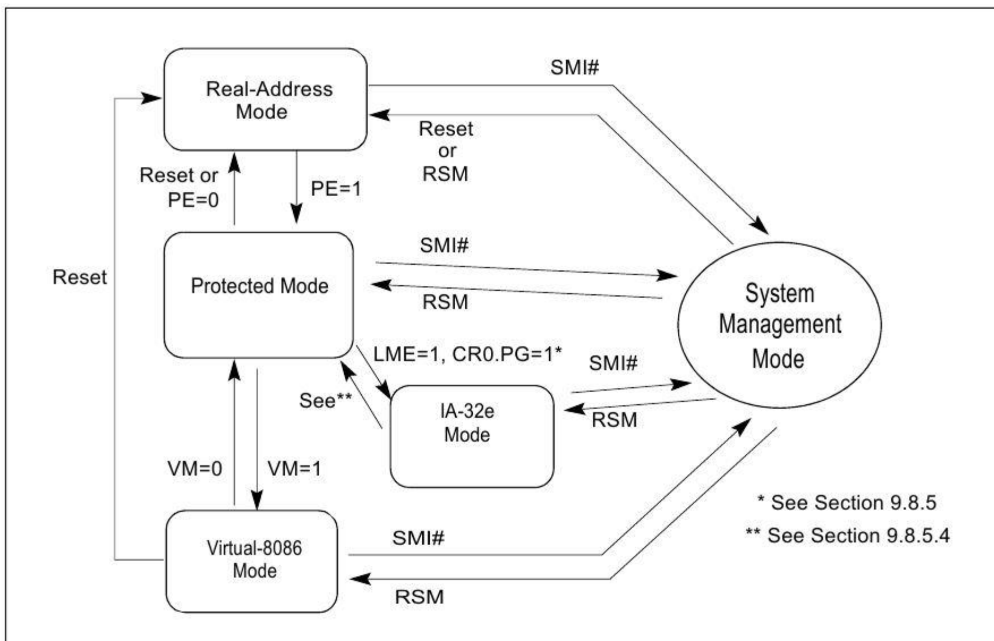


Figure 1- 1

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:

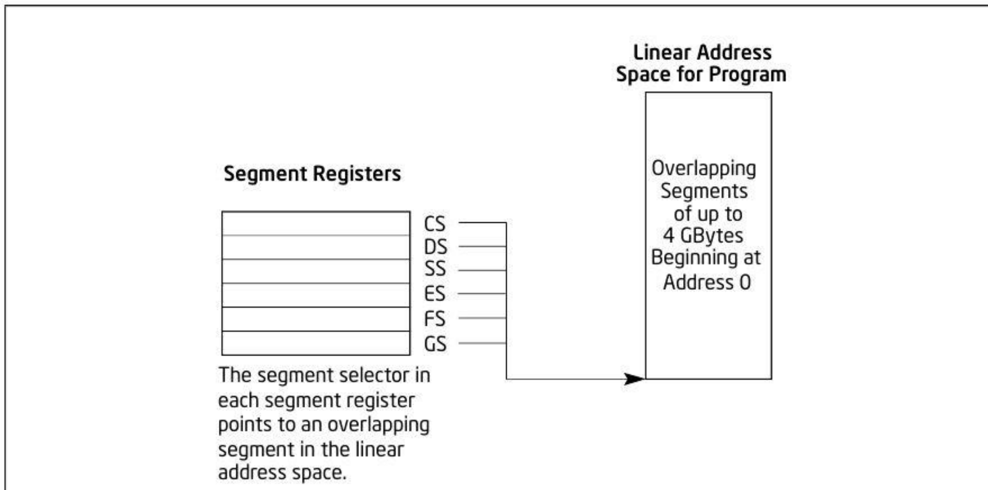


Figure 2- 1

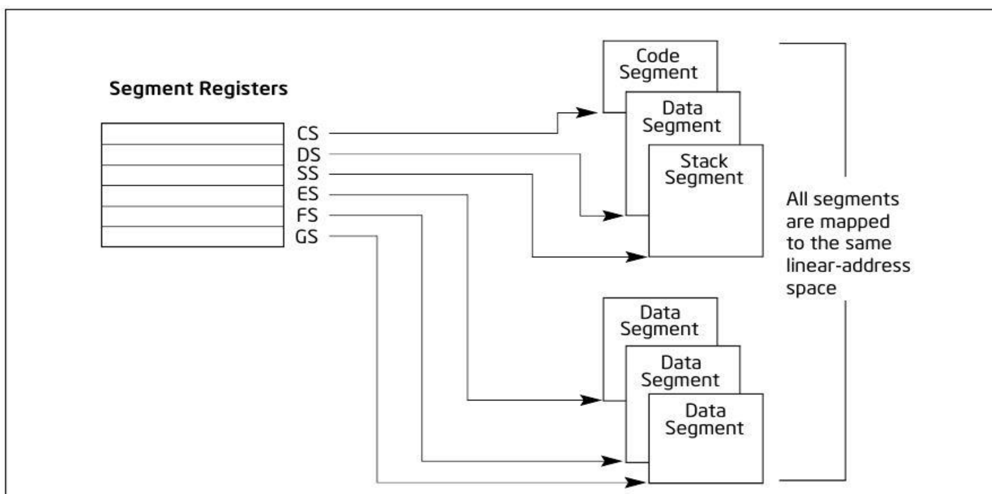


Figure 2- 2

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.

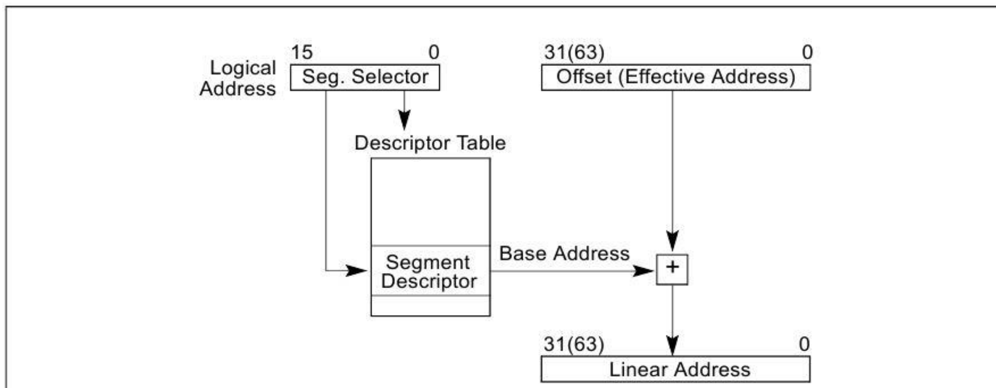


Figure 2- 3

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:

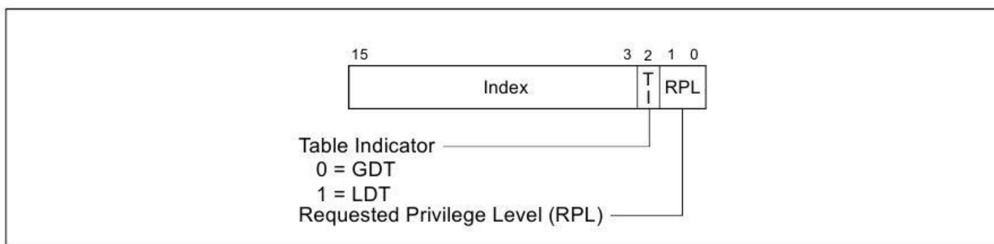


Figure 2- 4

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.

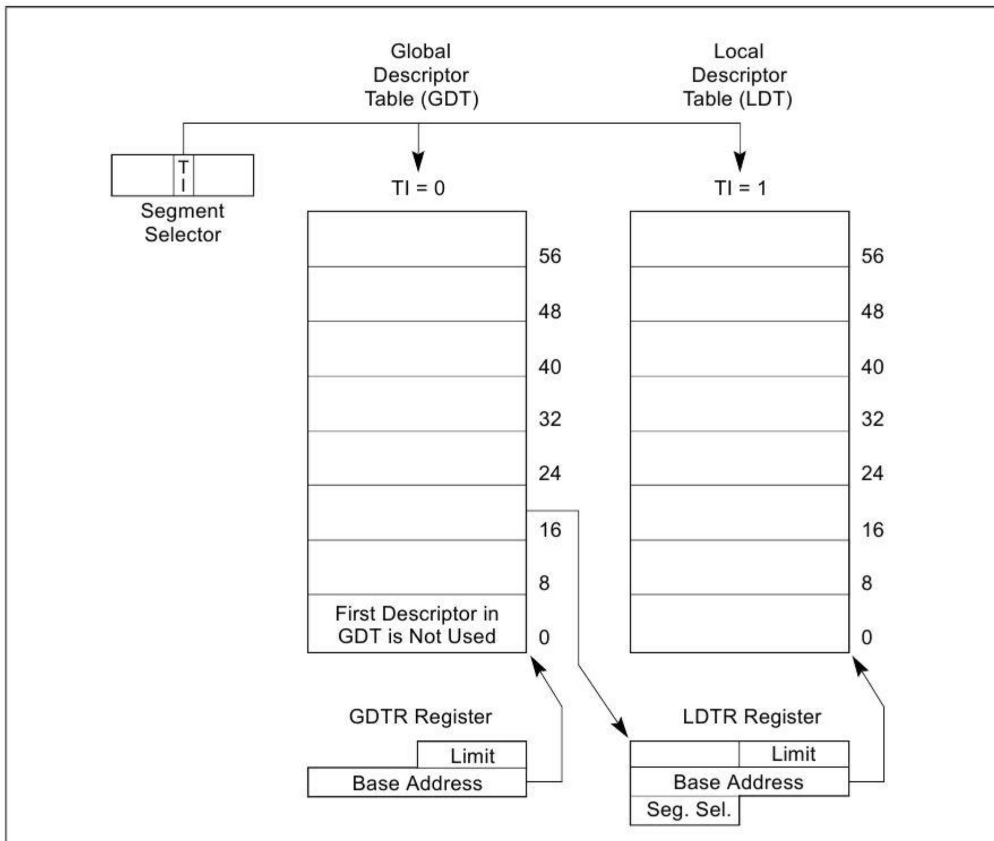


Figure 2- 5

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.

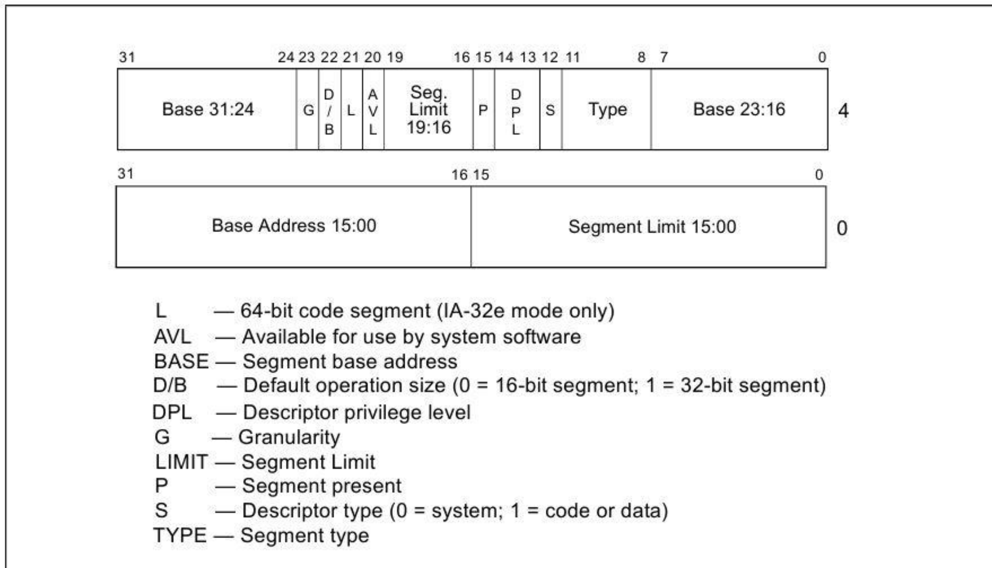


Figure 2- 6

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 user space 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:

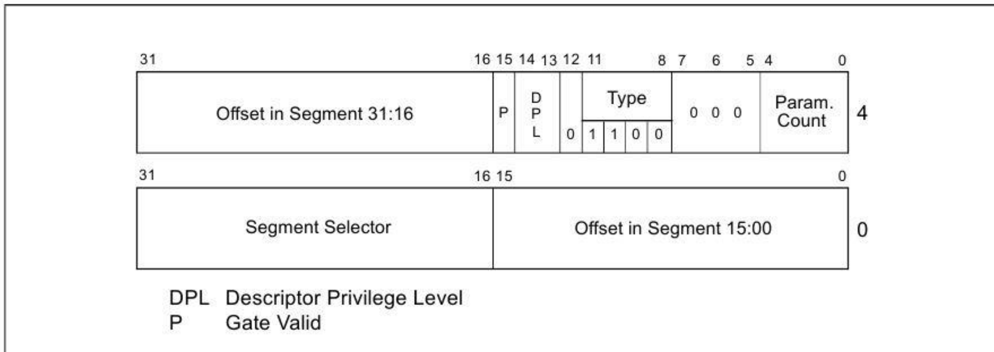


Figure 2- 8

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:

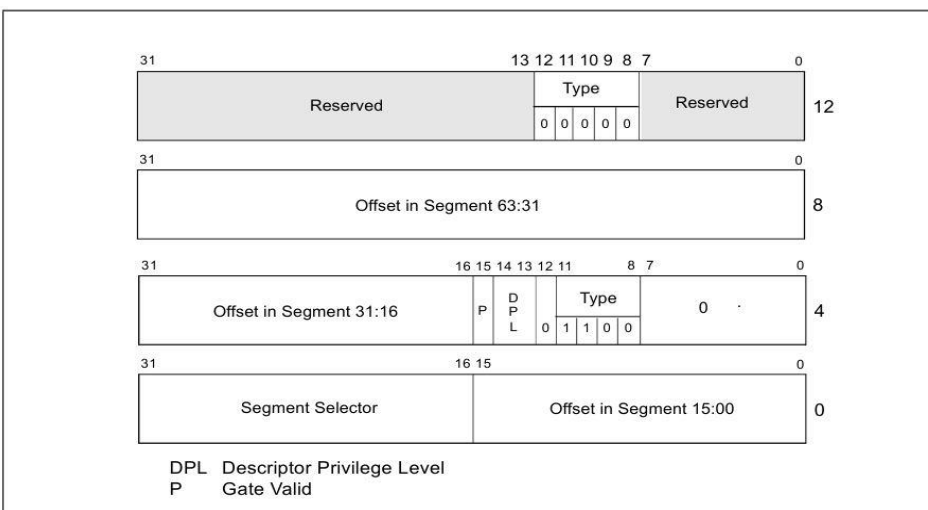


Figure 2- 9

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.

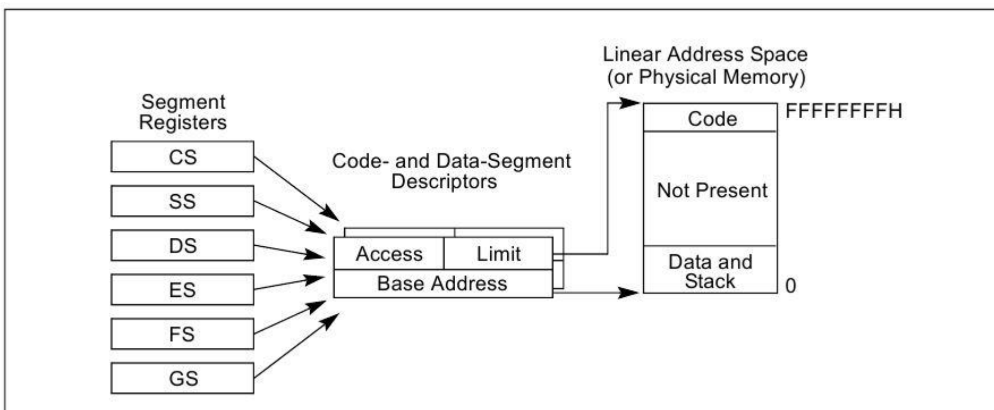


Figure 2- 10

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].

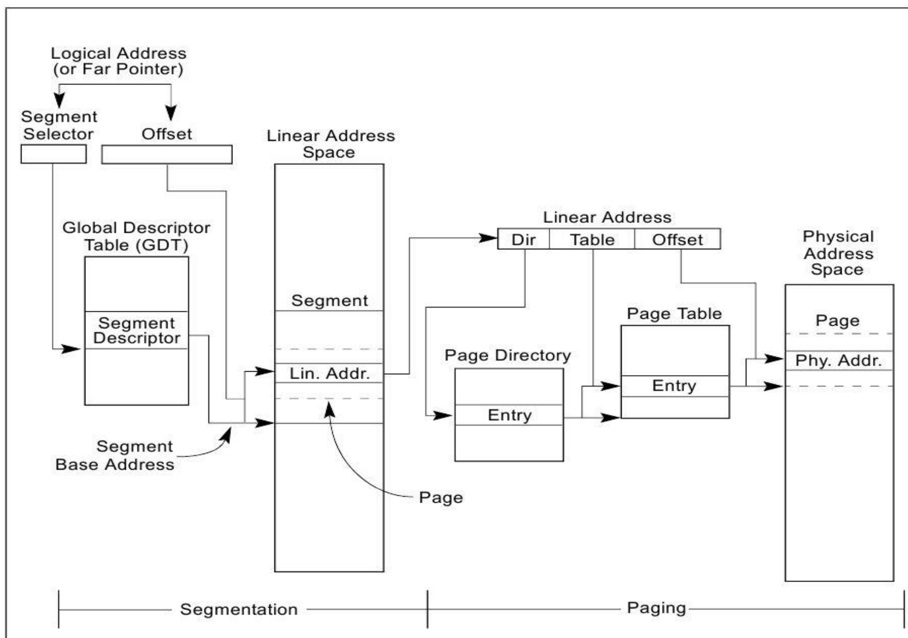


Figure 3- 1

[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:
0000000000000000011111111111111111111111111111111111111111111111

⇨ From 0 to 0x7FFFFFFFFFF

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

⇨ From 0xFFFF80000000000 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

Different CPU models have different features. In order to get that information, for example seeing if hardware virtualization is available, we use CPUID instruction. CPUID doesn't accept any operands, rather it takes its input from RAX/EAX and sometimes RCX/ECX. You can check all its functionality in Intel's Developer Manual[13]. As an example, you can use this code to check for the vendor of the CPU:

File: Header.h

```
#pragma once
#define CPUID_H

#ifdef _WIN32
#include <limits.h>
#include <intrin.h>
typedef unsigned __int32  uint32_t;

#else
#include <stdint.h>
#endif

class CPUID {
	uint32_t regs[4];

public:
	explicit CPUID(unsigned i) {
#ifdef _WIN32
		__cpuid((int *)regs, (int)i);

#else
		asm volatile
			("cpuid" : "=a" (regs[0]), "=b" (regs[1]), "=c"
(regs[2]), "=d" (regs[3])
```

[13] Vol 2A, 3-190

```
                            : "a" (i), "c" (0));
              // ECX is set to zero for CPUID function 4
#endif
       }

       const uint32_t &EAX() const { return regs[0]; }
       const uint32_t &EBX() const { return regs[1]; }
       const uint32_t &ECX() const { return regs[2]; }
       const uint32_t &EDX() const { return regs[3]; }
};

//#endif // CPUID_H
```

File: cupid.cpp

```
#include "Header.h"

#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
       CPUID cpuID(0); // Get CPU vendor

       string vendor;
       vendor += string((const char *)&cpuID.EBX(), 4);
       vendor += string((const char *)&cpuID.EDX(), 4);
       vendor += string((const char *)&cpuID.ECX(), 4);
       cout << "CPU Vendor: " << vendor << endl;
       system("pause");
       return 0;
}
```

Running above code in Visual Studio gives you the vendor of your CPU. Don't skip this part because it will be very useful in the future projects of this book. For now, you just read through, later, you must do.

Virtual Address Translation Overview – IA-32e Paging

Let's start explaining translation of a valid virtual address in 64-bit mode. This is going to be a high-level summary without going into deep specifics. Translating a 48-bit virtual address to a physical address goes like as shown in figure 3-2. First CR3 is used to find the PLM4 base address. Keep in mind that the address in CR3 is a physical address. In fact, during address translation, all the addresses found in CR3, tables and table entries are physical addresses. Keep an eye on figure 3-2 as you read the rest of this explanation.

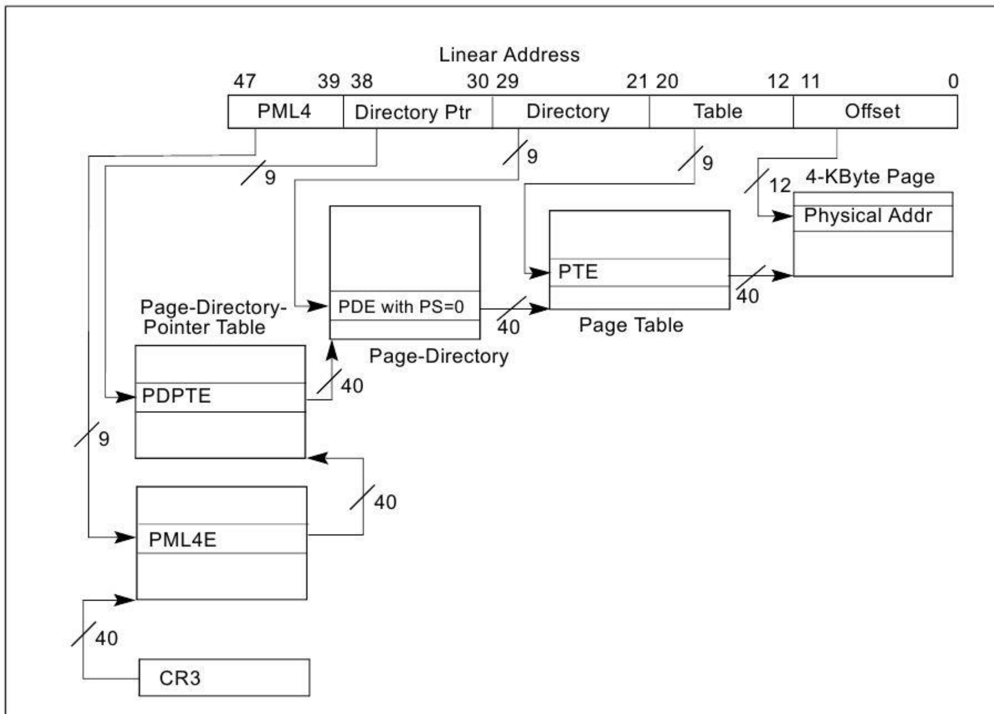


Figure 3- 2

Bits 39-47 of the virtual address is an index into PLM4. Each entry in PLM4 is 8 bytes. So, the offset which bits 39-47 of the virtual address is asking for is found by multiplying that index by 8.

index = bits 39-47 of the virtual address
PLM4E = PLM4 Entry
Offset into PLM4 = index * 8
PLM4E = Base Address (found in CR3) + Offset (which is index * 8)

PLM4E points[14] to the start of another table called Page Directory Pointer Table (PDPT). So, bits 39-47 of the virtual address eventually get the base address[15] of PDPT.

PDPT has the same structure as PLM4. Bits 30-38 of the virtual address is used as an index into the PDPT. The offset into PDPT again is found by multiplying this index by 8 which gives us PDPTE (PDPT Entry).

---

[14] Reminder: a physical address
[15] Physical address

PDPTE stores the physical address of the next table, called Page Directory (PD). Bits 21-29 will select an entry from Page Directory. This entry is again found at an offset which is found by index (bits 21-29) multiplied by 8. Page Directory Entry (PDE) has the physical address to the start of last page before reaching physical memory. This last page is called Page Table.

Bits 12-20 is used to select an offset into Page Table (PT). The selected entry holds a physical address which will be used as the base address to compute the requested physical address. How? By adding the remaining bits of the virtual address (Bits 0-11) to the physical address found in Page Table Entry (PTE). This means that bits 0-11 act as an offset to the final physical base address. That was a brief explanation how a valid virtual address gets translated to a physical address.

Each of the tables mentioned so far is 4 Kbytes in size and each entry in them is 8 bytes. That gives us 512 entries in each table.

So far, we saw that bits 12-47 are used to find the final physical page[16] and we saw bits 0-11 are used as an offset to that physical page. So, we can conclude all virtual addresses with the same value for bits 12-47 will be translated to the same physical page.

All entries in the tables mentioned so far contain **4Kbyte-aligned** addresses. If we get the last physical address found during translation process, which is the physical address found in PTE, which points to the beginning of a physical page; and divide that address by 4Kbyte, what we'll get is an integer. This integer is called **Physical Page Number (PFN)** and it simply represents our physical pages by indexing them as an array. Like, physical page 1, physical page 2, etc. Now, if we do the same for virtual addresses, meaning that we first zero out bits 0-11 and divide the resulting address by 4Kbyte, we will get an integer which gives us a **Virtual Page Number (VPN).**

Here's a rule that summarizes the paragraph above using PFN and VPN terms:

All the addresses with the same VPN, will be translated into the same PFN.[17]

---

[16] A physical page is a 4Kbyte chunk of physical memory.
[17] This whole translation process is referred to as "Mapping".

Structure of a Page Entry

Each page entry has some flags which control different aspects of the paging translation process. A page entry for 4Kbyte-aligned paging looks like this, no matter to which table it belongs:

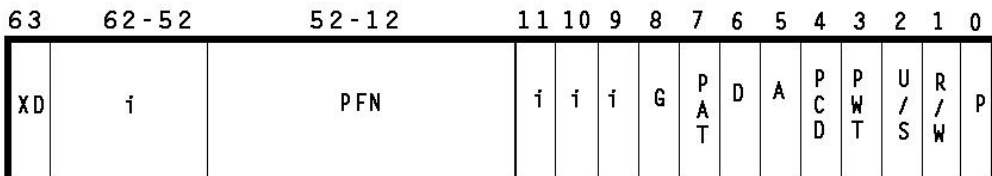


Figure 3- 3

Let's explain the flags:

P: Present flag (P flag) specifies the next table in the hierarchy or the actual physical page is present if it is set to 1. Otherwise, it's not present and all other bits (flags) are ignored. [18]

R/W: Read/Write flag indicates that writing to the requested physical page (or next tables) is allowed when it is set to 1. If 0, then the requested physical page (or next tables) are read-only. Although there is a control flag in CR0 register called "Write Protect"[19] that allows ring 0 to write to a read-only page but Windows sets this bit to 1 in order to enforce read-only pages to remain intact. That way even rings 0 can't write to a page with a cleared R/w flag.

U/S: User/Supervisor flag tells whether the page is accessible by ring 0 or ring 3. If it is set to 1, it is accessible to ring 3 (literally all rings). If it's set to 0, it's only accessible to ring 0. Trying to access with CPL=3 while U/S flag is cleared, causes a General Protection (#GP) exception.

We'll explain the other flags in later chapters. Just for a quick note, **i flags** are all ignored and they're available to software.

---

[18] The other flags are actually available to software.
[19] CR0.WP=1 in Windows by default.

Linear Address Translation for Larger Pages

So far, we introduced address translation for 4Kbyte pages which is the most common paging structure for most modern operating systems. But we can also use 2Mbyte and 1 Gbyte pages.

Paging for 2Mbyte Pages

While paging is set for 2Mbyte pages, there is no Page Table any more. The process is the same as translation to a 4-Kbyte page until it reaches the Page Directory. This means that the Page Directory Entry (PDE) no longer points to the bottom of a Page Table. Instead in points the start of the final physical page and uses bits 0-20 as an offset to that page. Figure 3-4 illustrates this process very clear. If you do the calculation, $2Mbye = 2^{21}$.

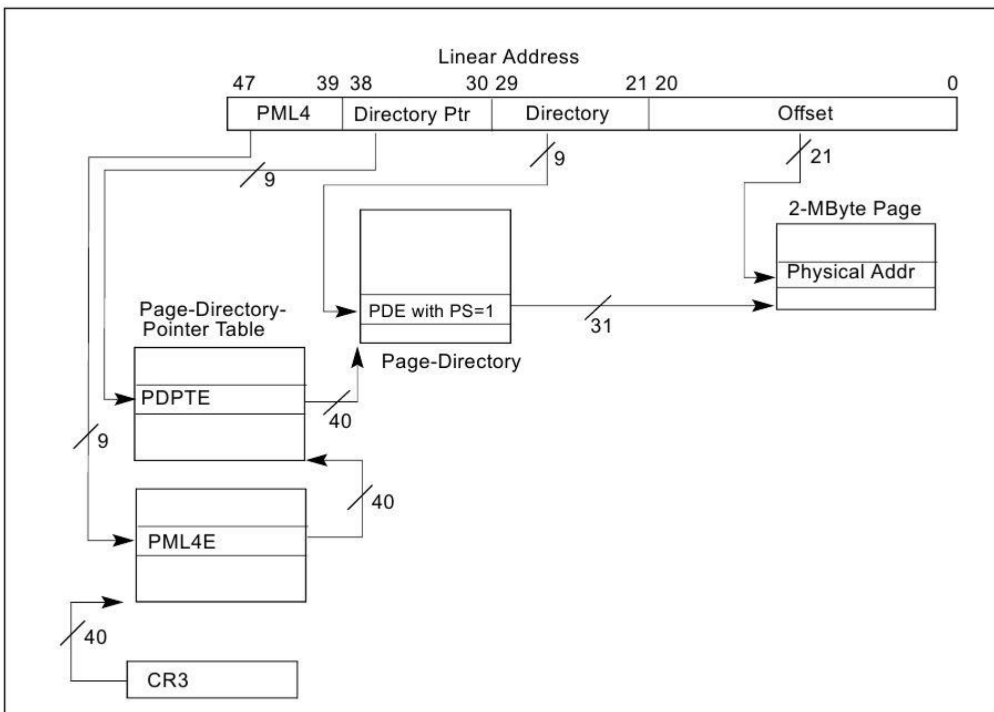


Figure 3- 4

Paging for 1Gbyte Pages

Paging for 1 Gbyte pages eliminates Page Directory on top of Page Table, meaning that PDPTE points to the start of a 1Gbyte chunk of physical memory and bits 0-29 of the linear address is used as an offset to that page[20].

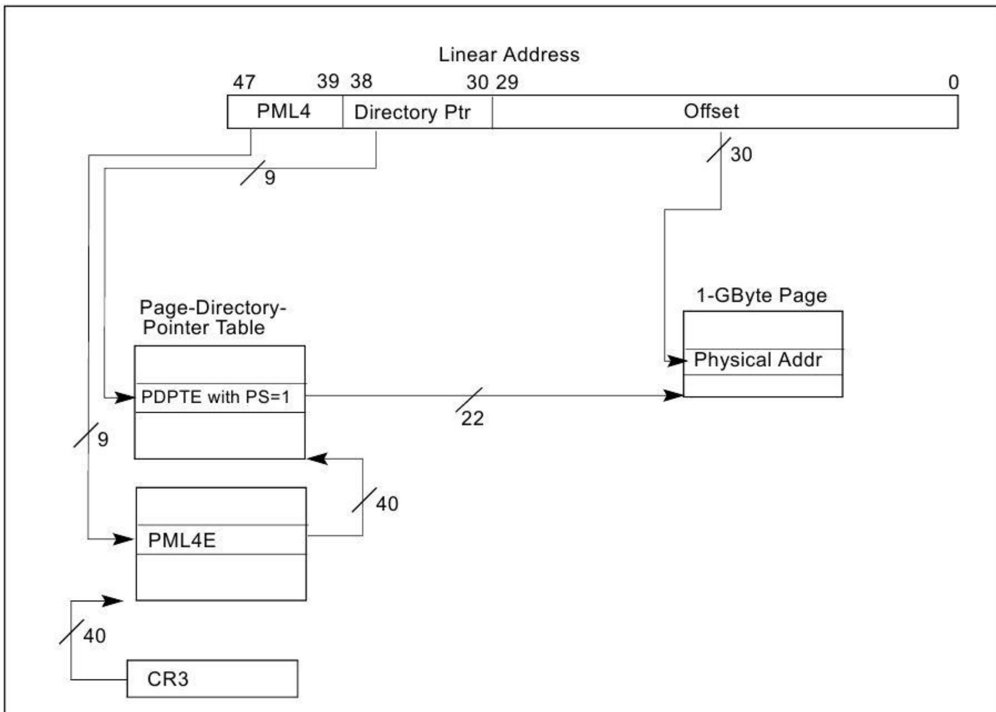


Figure 3- 5

[20] $1Gbyte = 2^{30}$

Chapter 0x04 – Caching

Why?

Caching is done based on the simple fact that every time a virtual address needs to be translated to a physical address, it must go through 4 memory pages. Accessing memory pages for each paging structure besides the execution time, is slow. To speed things up, the processor attempts to cache these virtual-to-physical translations. We have 2 different caching mechanisms. The first one is may ring a bell.

Translation Lookaside Buffer (TLB)

TLB works in such a simple efficient way. Each TLB entry, translates a logical address to its corresponding physical address along with all of it control flags, i.e. R/W flag, U/S flag, etc. If a virtual address matches with an entry in TLB, the translation will be completed. If the virtual address isn't found in TLB, then processor tries to find it through another caching mechanism called Paging-Structure Cache.

Paging-Structure Cache

The other option is Paging-Structure Caches which stores translations from one page entry to the next one. For example, the entries in PML4 get cached. The advantage point in this method caching compared to not using caches at all, is the point that the processor doesn't have to find each table anymore. So, there will be no more base-address plus offset here. The entries are directly cached and they are selected the same way as before, using their corresponding bits in the virtual address. For example, to select a PDE, bit 21-29 directly select the proper entry from Paging-Structure Caches without the need for finding PD's base address. All the control flags are also found and enforced using these caches just like TLB. It is obvious that TLB is much faster than Paging-Structure Caches.

Cache Invalidation

Imagine an entry in one of the paging tables gets updated or changed for any reason and the caches are still the same. That would fatal errors in the operating system's life cycle. So, the processor must make sure to invalidate caching tables when some entry gets updated but that's a heavy

task and slows down the performance. So, there must be a better solution with less overhead. As mentioned before in chapter 2, caches get flushed when value in CR3 is changed. By loading a new value in CR3, a new address space is loaded and it completely makes sense to invalidate the old caches. On top of that, **software must explicitly invalidate caching tables when it updates a cache entry.** This is necessary since the processor only takes care of cache invalidation when a value is loaded into CR3 register. So, OS has the most responsibility.

There's one more thing to mention for the second time. What happens to the memory regions which are used very often?[21] It doesn't make sense at all to flush caches for those memory regions such as shared libraries. PGE flag in CR4 control register allows the OS to set the paging tables of shared libraries as global and later OS sets G flag to 1 in the PTE of a shared library.[22] [23] It's very important to know that this rule is **only applied to TLB**. **Paging-Structure Cache is always flushed** and invalidated since G bit is ignored in intermediate page entries.[24]

Cache Control

Cache control has a very specific structure. Not every memory type is cached, not every cache can be written back to memory if something is changed in them and not every caching mechanism gets synchronized with other parts of the system. There are some control registers and flags which define the which caching mechanism must be applied to which memory range. We explain those registers and bit after we explain memory types.

Memory Types

Intel architecture defines 6 different memory types. We list all of them here but only 3 of them interests us. [25]

- Strong Uncacheable (UC)
- Uncacheable (UC-)
- Write Combining (WC)

---

[21] Refer to page 20 and 21 - PGE flag in CR4.
[22] For 4Kbyte pages.
[23] For large pages, this is done by the last Page Entry before the final physical page is found. For 2Mbyte pages, it is set in PDE and for 1Gbyte pages, it's set in PDPTE.
[24] Remember, G bit is only effective in the last page before the final physical page.
[25] For more info on all 6 memory types, check Intel Developer Manuals, 11-6, Vol. 3A

- Write-through (WT)
- Write-back Memory (WB)
- Write Protected

Write-back

During read, first the processor checks it cache, if there is a match, it reads from the cache, if not, it grabs a copy from the memory and puts it in its cache. The next time, when it wants to access the same data, it finds it through cache. This is called a "cache hit". Now what happens if a cache gets updated? When executing a write on some WB memory range that is already cached, writes will only update the cache itself. When some object tries to access a memory range, the check the cache and updated or not-updated, it one cache for all. But what happens in multi-core CPUs?

Snooping

To make sure every access made to a memory range gets the latest updated one, a mechanism called snooping comes into play. Whenever a processor wants to access some chunk of RAM and in that time, it notices that another processor is writing to that location, it invalidates its current cache and gets the updated copy from memory.

All processors always keep track of each other memory accesses. So, whenever processor one[26] wants to access a chunk of memory which it's not getting written to in that current time, because of the snooping mechanism, the other processor that has modified that piece of memory before, will notify processor one that the memory range was updated. Then, processor one will wipes its cache for that memory range and gets the updated version for the other processor.

Quick Note: If you're interested to know where cached data are stored, they are stored at L1, L2 and L3 caches. You can read more about them in intel developer manuals, 11-2 Vol. 3A.

Summarizing the paragraphs above, in WB memory type, all read and write operation are done on cache line, not the physical memory. WB memory type is the most frequently used memory type in modern operating

[26] Processor one is just a naming convention to simplify the explanation.

systems. It is also the memory type which is used for shared libraries most of the times.

Write Combining

With a WC memory type, memory reads and writes are not cached. However, when a write operation happens, the processor puts the data in its internal buffer without caring about the order of the data and attempts to write to memory. This mechanism is mostly useful for video buffers where the write operation must happen out of the CPU cache and the order of the data stream being written in real-time is not an issue. Snooping is not present in WC memory types so there will be no guarantee for memory coherency. For a video buffer, using this type of memory makes absolute sense. Using it in shared libraries and code sections reduces the speed greatly. If there is a need for memory coherency, there are ways to deliberately flush the internal buffer of the processor.

Strong Uncacheable

System memory locations are not cached. Reads and Writes are not cached and every R/W operation must happen in program order. This type of memory is used on memory-mapped I/O devices and it shouldn't be used with normal RAM which cause the performance to fall greatly.

Now let's explain those control registers and flags which control and manage the caching mechanisms.

Memory Type Range Registers (MTRR)

Memory type range registers are a set of registers that can define a memory type for a memory range. MTRR is programmable by OS and set the memory types for the physical memory.

Page Attribute Table (PAT)

PAT is a **Model Specific Register (MSR)** which defines memory types for virtual address ranges. PAT register is 64-bits long and has 8 sections, 8-bits each. As you can see in figure 4-1, the lower 3 bits of each section is the effective part and the higher 5 bits are reserved.

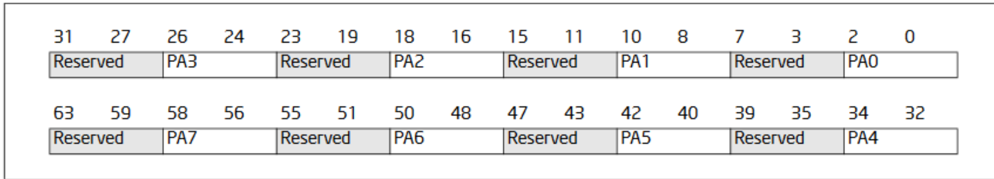


Figure 4- 1

Depending on the value inside the PAT fields (i.e. PA0, PA1, PA2), a virtual address range can be set to WB, WC, UC, etc. Here are the values that can be set for the effective 3 bits in a PAT field.

Encoding	Mnemonic
00H	Uncacheable (UC)
01H	Write Combining (WC)
02H	Reserved*
03H	Reserved*
04H	Write Through (WT)
05H	Write Protected (WP)
06H	Write Back (WB)
07H	Uncached (UC-)
08H - FFH	Reserved*

Figure 4- 2

The last paging structure before reaching the final physical page (PTE for 4KByte pages and PDE for 2MByte pages) select a memory type from PAT. It's important to remember that PAT is programmable by the OS. So, each OS may set its PAT differently than another.

In the previous chapter, we skipped some flags in page structure entries because it wasn't so convenient to introduce them back then. Here they are:

PCD flag: The 5th bit in PTE (or PDE) indirectly determines the memory type of the next page.

PWT flag: The 4th bit in PTE (or PDE) indirectly determines the memory type of the next page.

PAT flag: The 13th bit in PTE (or PDE) indirectly determines the memory type of the next page.

I know we repeated the same definition for all 3 flags above, but here's a picture that shows you what we mean by "indirectly":

PAT	PCD	PWT	PAT Entry
0	0	0	PAT0
0	0	1	PAT1
0	1	0	PAT2
0	1	1	PAT3
1	0	0	PAT4
1	0	1	PAT5
1	1	0	PAT6
1	1	1	PAT7

Figure 4- 3

Depending on the settings of those 3 flags in a PTE (or PDE), the PAT entry which defines the memory type gets selected.
There is one question left unanswered here. We said MTRR sets memory types for physical memory. PAT does the same for virtual memory. What happens if a virtual memory maps to a physical page with different memory types? Intel manual define the result of combining 2 different memory types as shown in below table:

MTRR Memory Type	PAT Entry Value	Effective Memory Type
UC	UC	UC[1]
	UC-	UC[1]
	WC	WC
	WT	UC[1]
	WB	UC[1]
	WP	UC[1]
WC	UC	UC[2]
	UC-	WC
	WC	WC
	WT	UC[2,3]
	WB	WC
	WP	UC[2,3]
WT	UC	UC[2]
	UC-	UC[2]
	WC	WC
	WT	WT
	WB	WT
	WP	WP[3]
WB	UC	UC[2]
	UC-	UC[2]
	WC	WC
	WT	WT
	WB	WB
	WP	WP

MTRR Memory Type	PAT Entry Value	Effective Memory Type
WP	UC	UC[2]
	UC-	WC[3]
	WC	WC
	WT	WT[3]
	WB	WP
	WP	WP

Figure 4- 4

Chapter 0x05 - Interrupts and Exceptions

Privilege Transitions

It's time to stop for a second and remember what we learned from volume 1 when we explained system calls. Imagine a code is being executed at ring 3 and wants to do something that is not allowed to be executed in ring 3, in fact, it can only be execute at ring 0. To achieve this, code must call the kernel and ask for a function in the kernel, hand all the necessary information to the kernel and wait until the execution is completed in the kernel side (if it's a legitimate request in the eyes of kernel) and receive the result from the kernel and continue execution at ring 3.

What happens here is called a **privilege transition** which changes the CPL from 3 to 0, execute the ring 0 code and change CPL back to 3 from 0 and get back to the ring 3 code. There are certain ways to cause a privilege transition. We learned on of them in volume 1, using *syscall* and *sysret* instructions for example. We are not going to discuss them here again. The other ways to cause a privilege transitions are **Interrupts** and **Exceptions** but before we begin explaining those, we need to have a clear understanding of the stack and by that, I mean ring 0 stack and ring 3 stack and when a stack switch happens.

Stack Switching

We have 2 different stacks for ring 0 and ring 3. When a privilege transition happens, a call to a piece of kernel code is done. As a subsequence of a *call* instruction, the return address must be saved on the stack. This address will be used when kernel wants to get back to the caller and continue execution. If a privilege transition happens, the processor doesn't want that return address be accessible to ring 3 code. In other words, the processor wants to protect itself from ring 3 code so, it defines a totally isolated stack only for ring 0. When a privilege transition happens, first, the address of the ring 0 stack gets loaded into *rsp* so all the necessary information[27]are saved on ring 0 stack which is untouchable by ring 3. This happens before anything being saved on the stack or any handler comes in play. The address of the ring 0 stack is found via **Task State Segment** which provides addresses of the ring 0 SS and *rsp* register.

---

[27] There is more info saved on the stack than just the return address when a privilege transition is about to happen. We introduce them in bit.

It is very important to understand that a *sysret* (or any return instruction) doesn't need a corresponding *syscall*, meaning that the OS (and processor) doesn't keep that track of calls and return instructions. So, a *sysret* can be executed regardless of its corresponding *call* instruction.

What's an Interrupt?

An interrupt regardless if it’s from hardware or software, as its name suggests is a signal sent to processor which pauses the execution of the currently running thread and asks the processor for immediate attention. An example of an interrupt is a hardware interrupt when the user interacts with a hardware (i.e. pressing a button).[28]

When the processor receives an interrupt, it suspends the execution of the current thread and deals with the interrupt. Depending what is the source of interrupt, the processor executes its proper interrupt handler. After serving the interrupt, it resumes the previously suspended thread.

Handling an Interrupt & IDT

When an interrupt happens while in ring 3, first *rsp* gets loaded with the address of ring 0 stack. Then a stack alignment happens which is basically doing “*and rsp 0xFFFFFFFFFFFFFFF0”.* This happens for performance as explained in volume 1. Second, these registers must be saved on the stack as follows:

1. *SS* Segment register
2. *rsp* value while interrupt happened so stack can be reloaded back to its original address to continue programs execution
3. *rflags* register which has all the information of the program execution at the time of the interrupt
4. *CS* segment register which has the CPL value
5. *rip* to resume the code after executing the interrupt handler

If the interrupt is happening in ring 0, then there is no privilege change and there is no stack switch. So, SS and *rsp* registers don’t get pushed onto the stack. Depending on the type of the interrupt, an error code may be push onto the stack at last.

---

[28] USB devices (i.e. USB mouse) are not interrupt based!

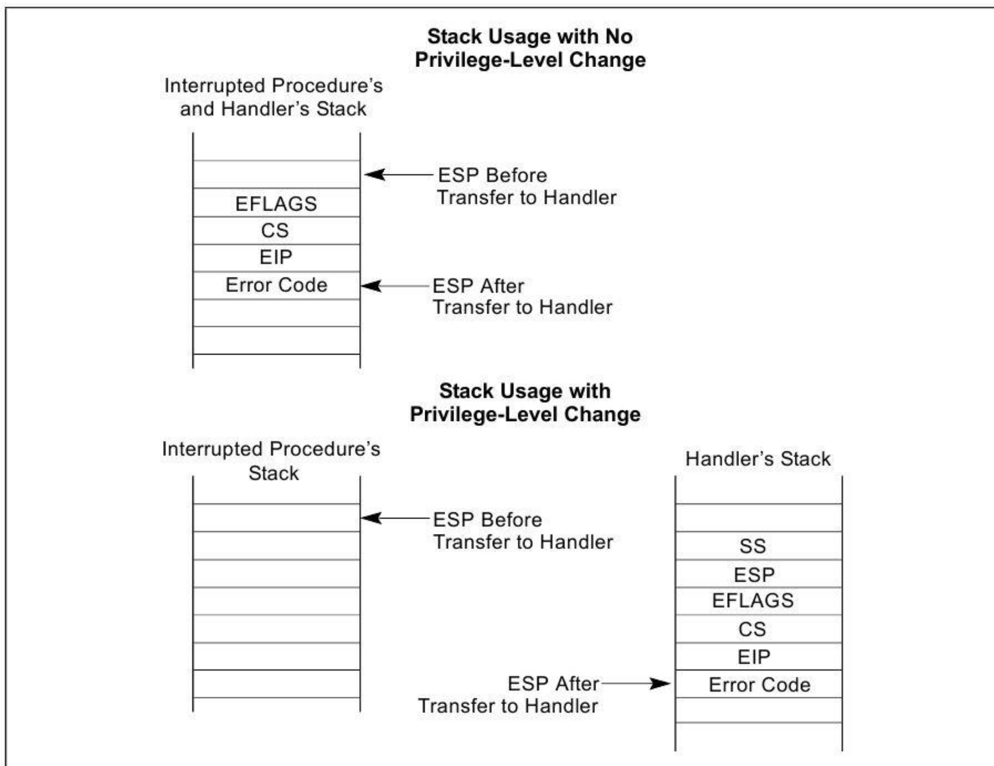


Figure 5- 1

Every interrupt is associated with a number called the interrupt vector which is an offset into a table called **Interrupt Descriptor Table (IDT)**. Every IDT entry is called a gate which is an address to some part of the OS (or a device driver) code which is built specifically to handle a specific interrupt.[29] During an interrupt, this number (interrupt vector) is supplied to the processor which causes the processor to pause the execution of the current process and jump to whatever address is inside the requested IDT entry and execute the interrupt handler. IDT and its entries are editable only by ring 0. So, ring 3 cannot determine or change the entry point to OS code.

Using *iret* instruction, an interrupt handler can resume the previously interrupted code. *iret* instruction restores all the registers which saved on the stack before handling the interrupt back to their original state. If the CPL in CS register was 3, the *iret* changes the CPL from 0 to 3. If it was 0, then no CPL change will occur. It is important to note that *iret* can go from ring 0 to

[29] Now call gates make more sense?

ring 3 but cannot go from ring 3 to ring 0. Executing an *iret* instruction is regardless of any previous interruption just as any other return instruction.

Hardware Interrupts and Software Interrupts

Intel architecture defines software interrupts and hardware interrupts. Hardware interrupts are the ones that are cause by interaction with hardware. Software interrupts are the ones that are intended by the software, meaning that the software itself caused it by using *"int n"* instruction[30]. This does not let ring 3 code to call whatever interrupt it want as most IDT vectors are only accessible by ring 0. Attempting to call to a ring 0 protected IDT vector cause the processor to generate a general protection exception (#GP) which terminates the program.

Exceptions

Exceptions are usually generated when something goes wrong (i.e. divide by zero). Exceptions occur when the processor detects an error such as a page fault, general protection or other types of violations. In this case, the exception must be handled and it works mostly the same way when an interrupt happens but an additional error code will be push onto the stack. Exceptions may have 3 different sources:

- Processor-detected program-error exceptions
- Software-generated exceptions
- Machine-check exceptions

and are categorized in 3 types based on the way they were reported or whether they can be fix or cause the source to crash or terminated.

- Faults
- Traps
- Aborts

Faults

A fault is an exception that can generally be corrected. A fault happens when the next instruction out to be executed have is wrong in some way (i.e. page faults when accessing an invalid page). When a fault happens,

---

[30] Replace n with the interrupt vector you want in hex. i.e. int 0x80, int 0x5 etc.

EIP/RIP holds the address of the faulting instruction and after it's resolved, the processor restores its state to the faulting instruction and resumes execution.

Traps

Explaining traps are a bit weird despite their simplicity in action. Consider the current instruction as "the trapped" instruction. Trap happens after executing the trapped instruction which causes the execution flow to be paused. One genuine example of traps is when a program is attached to a debugger. The debugger sets traps after each instruction to step through the program on instruction after another.

Aborts

When an abort exception happens, EIP/RIP may not precisely point to the instruction which caused the exception. In that sense, there is no guarantee for the suspended program to resume execution. Abort cause the processor to terminate the program which generated the exception.

Section 2

Windows Internals

Chapter 0x06 - Exploring PE Files

In this chapter, we're going to explore PE files top to bottom in order to have a deep understanding of how Windows executables function. This chapter explains the concepts which is the very vital knowledge of a reverse engineer or a malware analyst and it helps a lot in the first step of analyzing malware, triage analysis.[31]

Definition of a PE File

PE stands for **Portable Executable** and by that we're not only referring to .exe files. PE files have much more extensions and here we mention the most common ones:

- .exe (Executable files)
- .dll (Dynamic Link Library)
- .sys/.drv (Device Derivers)
- .osc (ActiveX Control)
- .cpl (Windows Control Panels)
- .src (Screen Savers)

A PE file has several important information about the executable. To learn and remember later, we need some hands-on exploration of a PE file. PE files are full of tiny detail which are very easy to forget. Opposed to Section 1, this section we a have a lot of hands-on learning and labs.

Exploring a PE file using CFF Explorer

You should download a copy of **CFF Explorer**[32] and install it on your local machine[33]. It's completely free and it has a very nice graphical interface to work with PE files. You can also use PE explorer (or whatever tool you want). We're going to start exploring PE headers of *calc.exe* file.

---

[31] Life of Binaries (2013 Update) by OpenSecurityTranining.info lectured by Xeno Kovah is heavily used as a resource for this chapter. Highly recommended!
[32] http://www.ntcore.com/exsuite.php
[33] I'm using Windows 7 64-bit

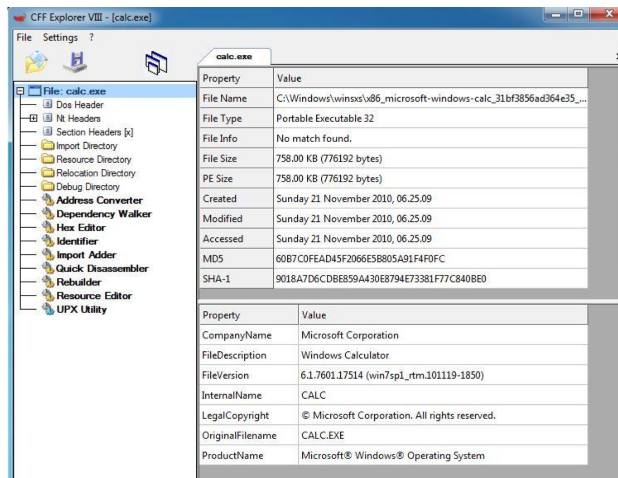


Figure 6- 1

After opening *calc.exe* in CFF explorer, we see some general information about the file created by CFF explorer itself. Looking on the left panel, we can see all the headers of a PE file:

1. **DOS Header**
2. **NT Headers**
3. **Image Section Headers**

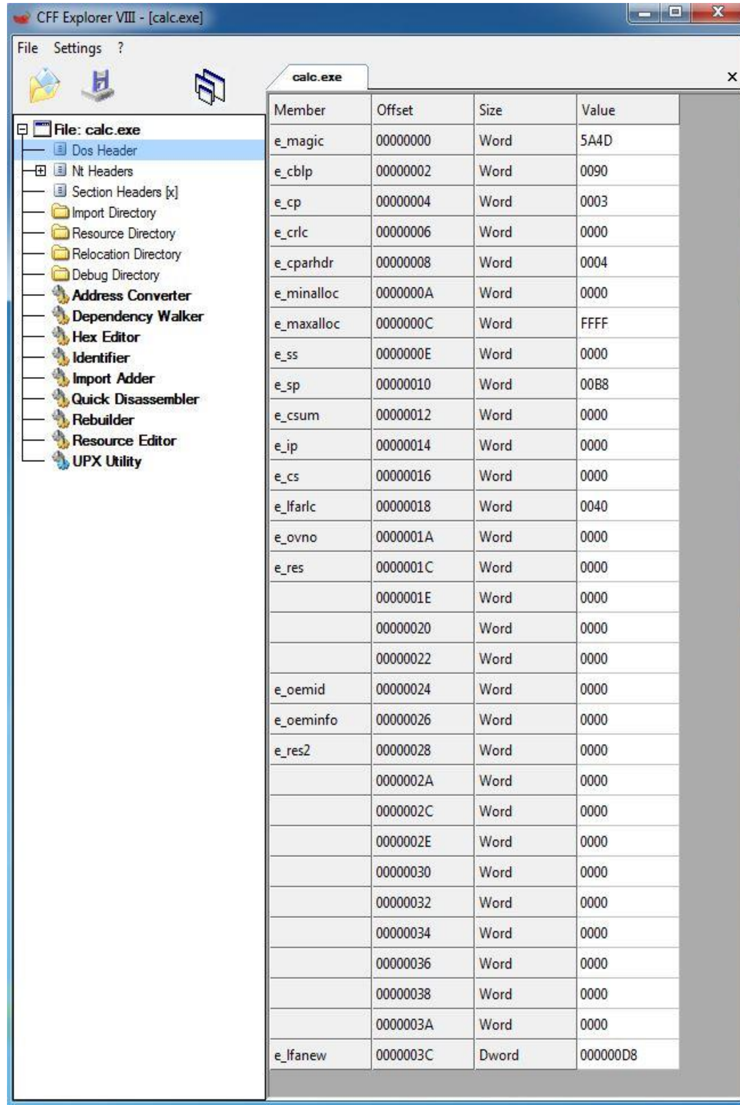
DOS Header

The DOS header is the first structure in PE file starting from offset 0. DOS header doesn't really give us so many information. The only 2 thing we care about in DOS header are *e_magic* and *e_lfanew.*[34]

e_magic is basically a number which specifies the file format so the OS can parse it correctly. e_magic is always set to the hex value 5A4D which on ASCII is MZ which is short for Mark Zbikowski, the developer of MS DOS. e_magic is the first item in the DOS header. All Windows applications (non-DOS applications) have a DOS stub which says: "This program cannot run in DOS mode." This stub is there to inform you when you try to run a Windows executable in DOS.[35]

e_lfanew tells the offset of the next header.

[34] Figure 6-2
[35] Figure 6-3



Member	Offset	Size	Value
e_magic	00000000	Word	5A4D
e_cblp	00000002	Word	0090
e_cp	00000004	Word	0003
e_crlc	00000006	Word	0000
e_cparhdr	00000008	Word	0004
e_minalloc	0000000A	Word	0000
e_maxalloc	0000000C	Word	FFFF
e_ss	0000000E	Word	0000
e_sp	00000010	Word	00B8
e_csum	00000012	Word	0000
e_ip	00000014	Word	0000
e_cs	00000016	Word	0000
e_lfarlc	00000018	Word	0040
e_ovno	0000001A	Word	0000
e_res	0000001C	Word	0000
	0000001E	Word	0000
	00000020	Word	0000
	00000022	Word	0000
e_oemid	00000024	Word	0000
e_oeminfo	00000026	Word	0000
e_res2	00000028	Word	0000
	0000002A	Word	0000
	0000002C	Word	0000
	0000002E	Word	0000
	00000030	Word	0000
	00000032	Word	0000
	00000034	Word	0000
	00000036	Word	0000
	00000038	Word	0000
	0000003A	Word	0000
e_lfanew	0000003C	Dword	000000D8

Figure 6- 2

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ ......ÿÿ..
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.......
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	D8	00	00	00	............Ø...
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..º..´.Í!¸.LÍ!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is.program.canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t.be.run.in.DOS.
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......

Figure 6- 3

NT Header

The NT header contains a signature and 2 embedded structures (headers).

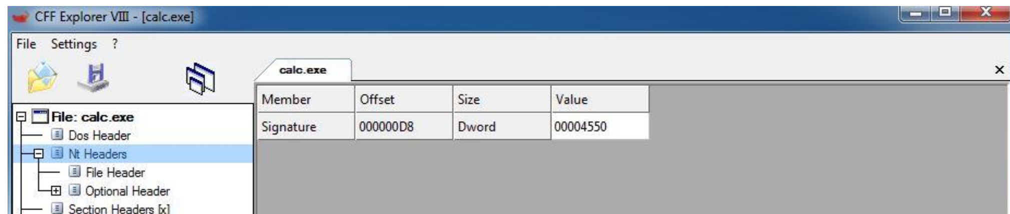


Figure 6- 4

The **signature** in the NT header contains the hex value 0x00004550 which is the ASCII word “**PE**” in little endian.

The **File Header** in the NT header contains 4 pieces that we care about: **Machine, NumberOfSections, TimeDateStamp and Characteristics.**

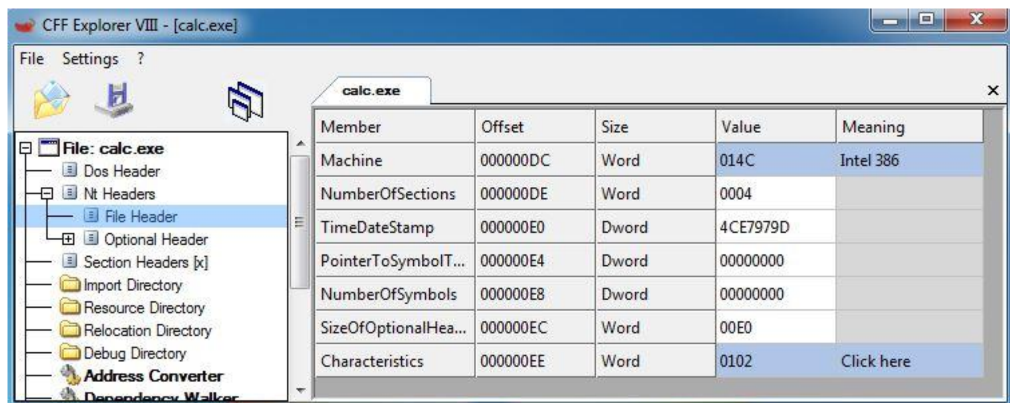


Figure 6- 5

- **Machine:** It basically tells us on which architecture the executable should run. Hex value of **014C** is for **32-bit** applications and **8664** is for **64-bit.** This doesn’t absolutely determine whether the binary is 32-bit or 64-bit.
  Quick note: 32-bit PE files are called **PE32** and **64-bit** ones are called **PE32+.**

- **NumberOfSections:** Tells how many section headers we are going to see later in the file.

- **TimeDateStamp:** It tells us when this executable was compiled. The time is number which indicates Unix time since epoch[36]. This is a very valuable information for a malware analyst but as a quick note, this and some other PE headers can be manipulated to hide the real information and mislead a reverse engineer. Also, all Delphi programs set the TimeDateStamp as June 19, 1992.

- **Characteristics:** There are lots of different attributes you can specify as characteristics, i.e. Is the file executable? is it a dll? Can it be mapped to larger pages? and so on.[37]

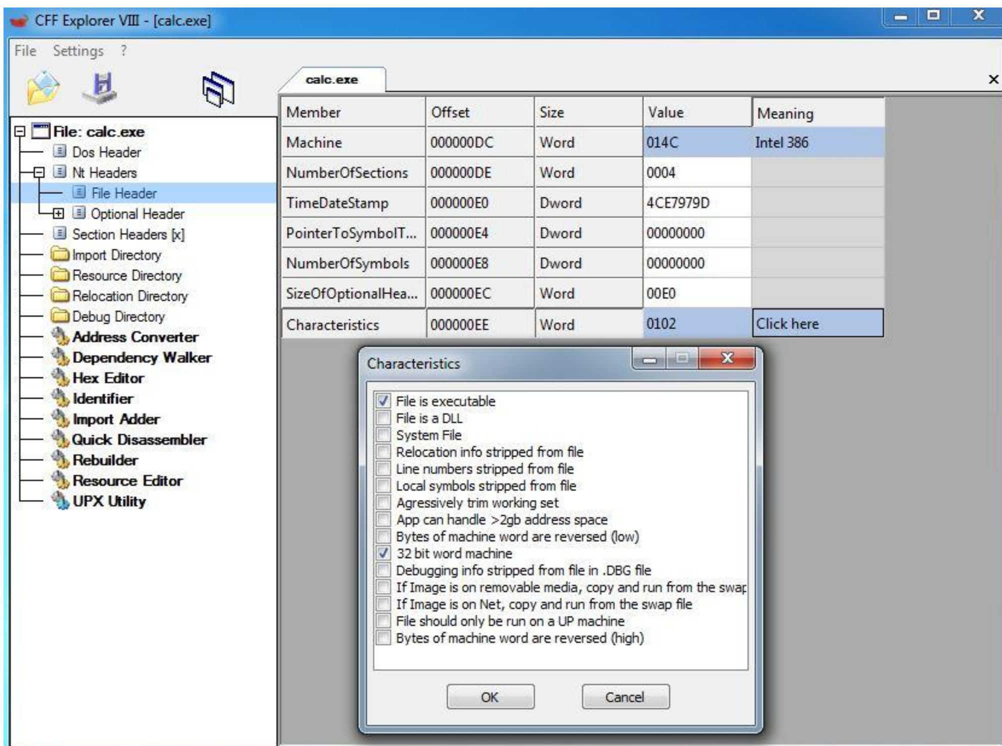


Figure 6- 6

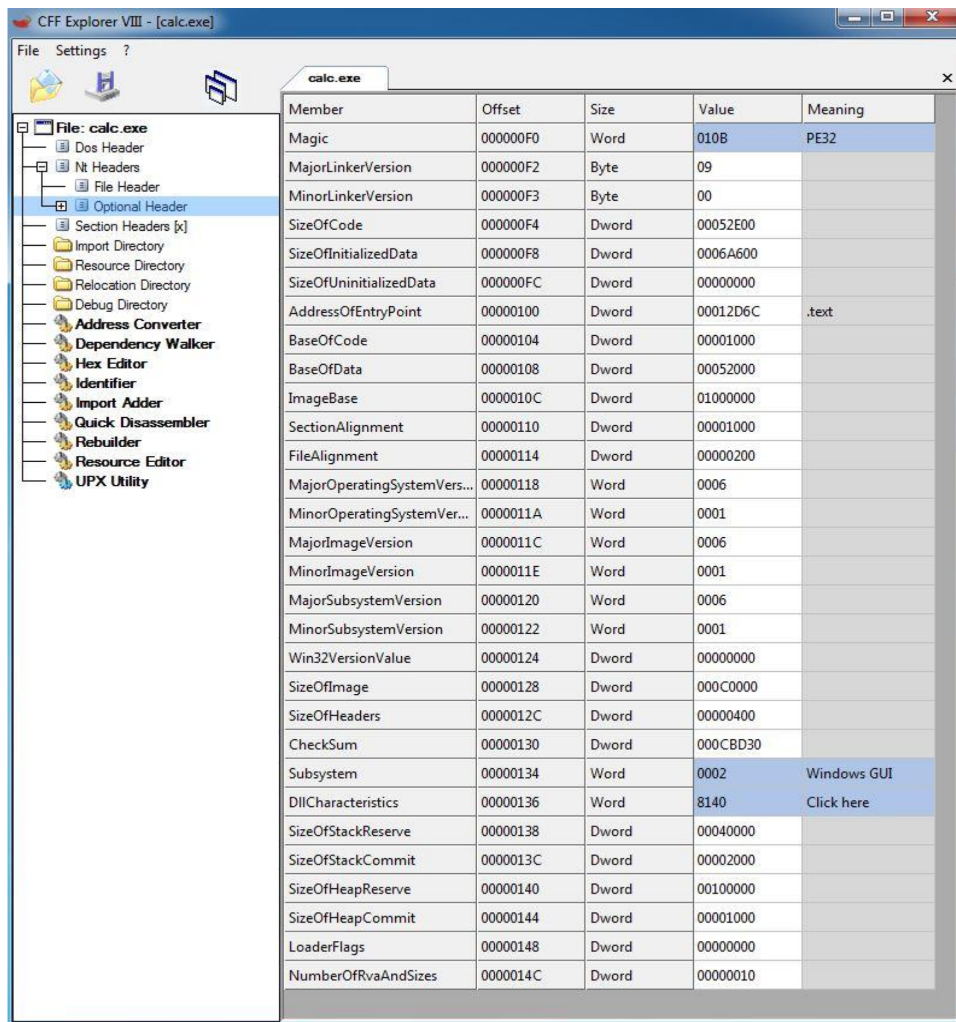
We can see in Figure 6-6 that calc.exe is a 32-bit PE file and should run a 32-bit environment (compatibility mode on 64-bit Windows).

---

[36] Seconds since January 1st 1970 00:00:00 UTC.

[37] You can take a look at all the attributes here: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680313(v=vs.85).aspx

The second embedded data structure inside NT header is the **Optional Header**. The optional header is **not optional** at all and it contains 8 items that are important to us: Magic, AddressOfEntryPoint, SizeOfImage, DllCharacteristics and DataDirectory. Depending on whether the binary is 32-bit or 64-bit, some fields in the Optional Header may be ULONGLONG instead of a DWORD.[38]



Member	Offset	Size	Value	Meaning
Magic	000000F0	Word	010B	PE32
MajorLinkerVersion	000000F2	Byte	09	
MinorLinkerVersion	000000F3	Byte	00	
SizeOfCode	000000F4	Dword	00052E00	
SizeOfInitializedData	000000F8	Dword	0006A600	
SizeOfUninitializedData	000000FC	Dword	00000000	
AddressOfEntryPoint	00000100	Dword	00012D6C	.text
BaseOfCode	00000104	Dword	00001000	
BaseOfData	00000108	Dword	00052000	
ImageBase	0000010C	Dword	01000000	
SectionAlignment	00000110	Dword	00001000	
FileAlignment	00000114	Dword	00000200	
MajorOperatingSystemVers...	00000118	Word	0006	
MinorOperatingSystemVer...	0000011A	Word	0001	
MajorImageVersion	0000011C	Word	0006	
MinorImageVersion	0000011E	Word	0001	
MajorSubsystemVersion	00000120	Word	0006	
MinorSubsystemVersion	00000122	Word	0001	
Win32VersionValue	00000124	Dword	00000000	
SizeOfImage	00000128	Dword	000C0000	
SizeOfHeaders	0000012C	Dword	00000400	
CheckSum	00000130	Dword	000CBD30	
Subsystem	00000134	Word	0002	Windows GUI
DllCharacteristics	00000136	Word	8140	Click here
SizeOfStackReserve	00000138	Dword	00040000	
SizeOfStackCommit	0000013C	Dword	00002000	
SizeOfHeapReserve	00000140	Dword	00100000	
SizeOfHeapCommit	00000144	Dword	00001000	
LoaderFlags	00000148	Dword	00000000	
NumberOfRvaAndSizes	0000014C	Dword	00000010	

Figure 6- 7

- **Magic:** This one is actually the variable which determines whether the binary is 32-bit or a 64-bit. If the magic value is **0x10B then it's a PE32** and **if it's 0x20B, then it's PE32+.**

[38] i.e. ImageBase and SizeOfStackReserve.

- **AddressOfEntryPoint:** This is a relative address (an offset) which points to first byte where code execution must be started. This doesn't necessarily point to *main()* or the first byte in *.text* section. This address tells the OS loader that after mapping the binary into memory, jump to this location and start executing code.

- **SizeOfImage:** This is the total size of the binary once it's mapped into memory. So, the OS loader take this size, allocates it on RAM and start mapping the piece in memory.

- **DllCharacteristics:** There are some security options in DllCharacteristics which can specify whether the binary uses ASLR, DEP, etc.

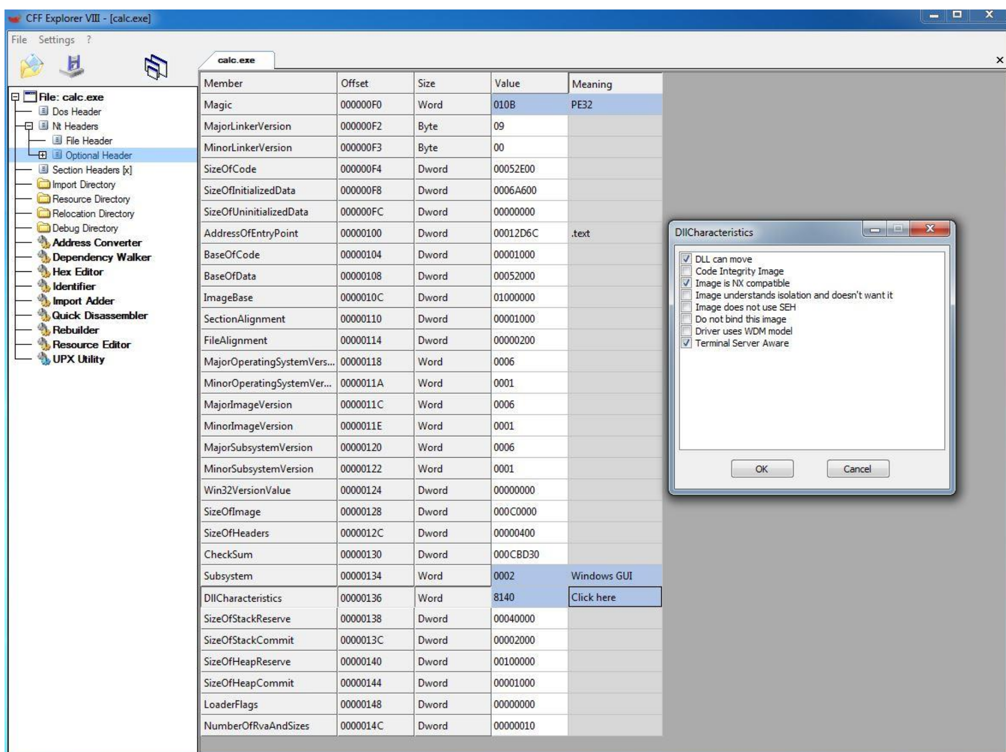


Figure 6- 8

DLL can move tells us that this DLL supports ASLR and relocations in memory.

Code Integrity Image indicates that when OS loader to check the digitally signed hash of the binary to make sure of its integrity and then map it into memory if passed.

Image is NX compatible turns on DEP on memory regions except .text to ensure that no code can be executed in stack, heap and other sections of the file when loaded into memory. This option is very important and can prevent some exploitations (although it's possible to bypass DEP).

Image does not use SEH tells the OS that in case of an exception, the binary doesn't use OS exception handling mechanism.

That's it for now about Optional Header. I know I didn't explain DataDirectory but we'll get back to it after explaining some other headers and data structures of PE files.

Section Headers

The raw data in a PE file is grouped in different sections so they can have their own set of permissions and characteristics according to their main functionality (i.e. Read/Write/Execute). This information is store in Section Headers which first defines some sections (i.e. *text*, *rdata*, etc.). You can see some of the most common section names in figure 6-9.

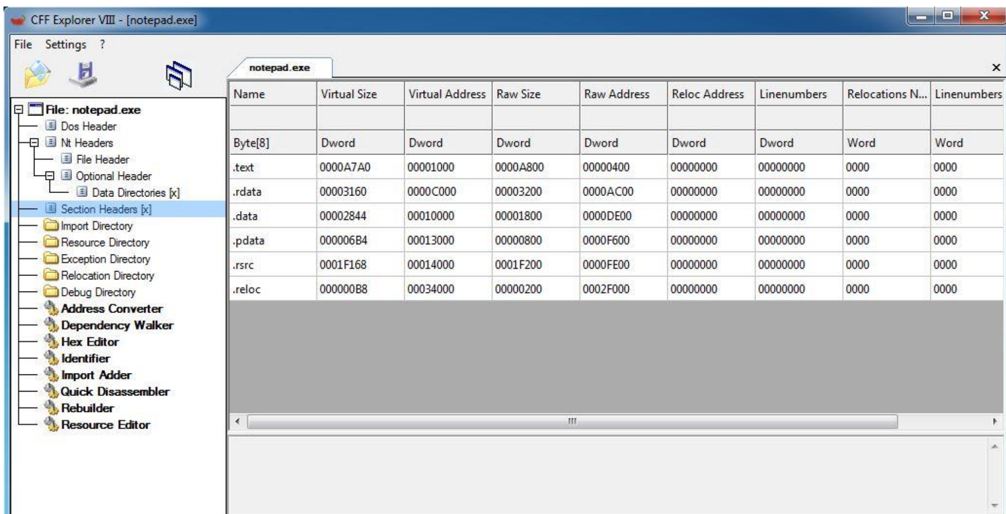


Figure 6- 9

As described in volume 1, ***text*** section is where all the code of the executable resides. It should be set as readable and executable but not writeable. One of the most common characteristics of a malware is that its *text* section is writable which means that the malware has some self-

modifying code or such. ***rdata*** is a section of the binary that has all the read-only data such as strings. *data* section has some data with read/write permission. ***bss*** is mostly for uninitialized global variables. You can notice that *bss* has 0 size on disk but takes some space in memory as specified in its Virtual Size attribute which confirms this. ***idata*** is where the executable holds the addresses needed for imported functionalities which mostly gets merged with *text* or *rdata* section. ***edata*** is the section which holds exported functions.[39] ***reloc*** has the information for relocation in memory. When a PE file wants to get loaded into memory, it specifies a preferred address. If that address is already in use, then it must be relocated, then this section will be used to manage that properly. ***rsrc*** has different resources like icons and embedded binaries and other data.

It is important that you see a different and more original data structure of the Section Headers presented by MSDN which describes the section headers in a more generic way rather than the "nice" way of CFF explorer. You can see this data structure in figure 6-10.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
  AGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

```
01 .text     VirtSize: 00005AFA  VirtAddr:  00001000
   raw data offs:   00000400  raw data size: 00005C00
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00009220  line #'s:      0000020C
   characteristics: 60000020
     CODE  MEM_EXECUTE  MEM_READ

02 .bss      VirtSize: 00001438  VirtAddr:  00007000
   raw data offs:   00000000  raw data size: 00001600
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: C0000080
     UNINITIALIZED_DATA  MEM_READ  MEM_WRITE

03 .rdata    VirtSize: 0000015C  VirtAddr:  00009000
   raw data offs:   00006000  raw data size: 00000200
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: 40000040
     INITIALIZED_DATA  MEM_READ

04 .data     VirtSize: 0000239C  VirtAddr:  0000A000
   raw data offs:   00006200  raw data size: 00002400
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: C0000040
     INITIALIZED_DATA  MEM_READ  MEM_WRITE

05 .idata    VirtSize: 0000033E  VirtAddr:  0000D000
   raw data offs:   00008600  raw data size: 00000400
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: C0000040
     INITIALIZED_DATA  MEM_READ  MEM_WRITE

06 .reloc    VirtSize: 000006CE  VirtAddr:  0000E000
   raw data offs:   00008A00  raw data size: 00000800
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: 42000040
     INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

Figure 6- 10

[39] bss, idata and edata aren't present in notepad.exe but you can check some other binaries to find them.

In the characteristics of every section header some of the very important attributes of the PE sections are defined. Look at figure 6-11. To understand these important characteristics better, this time we used Lord PE.[40]

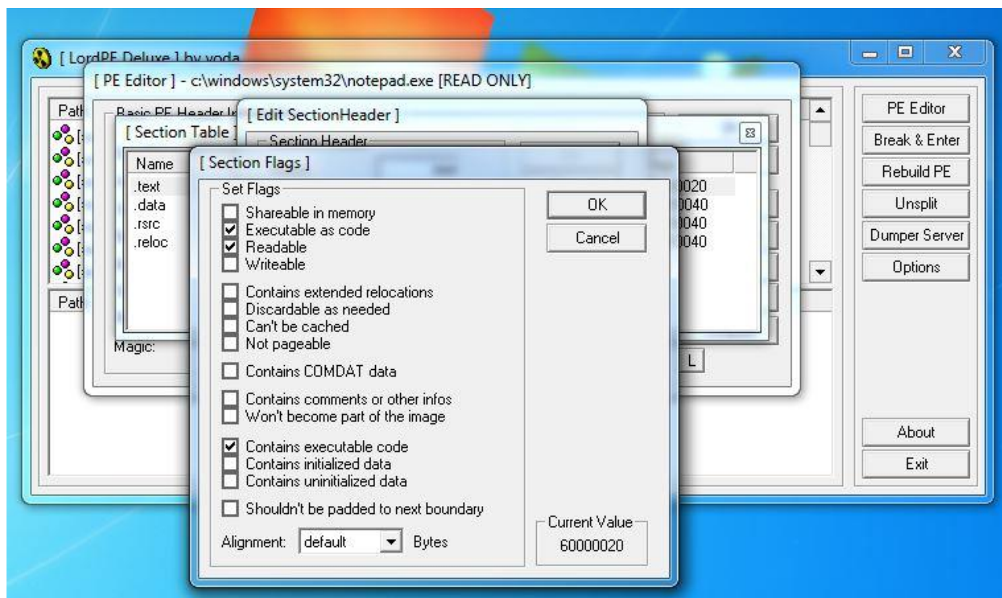


Figure 6- 11

In the picture above, we are looking at the characteristics of the text section of notepad.exe. As shown, the text section of notepad.exe is Readable and Executable but not writable. It can be cached and It is NOT shareable in memory. We will explain more of these characteristics as they come up.

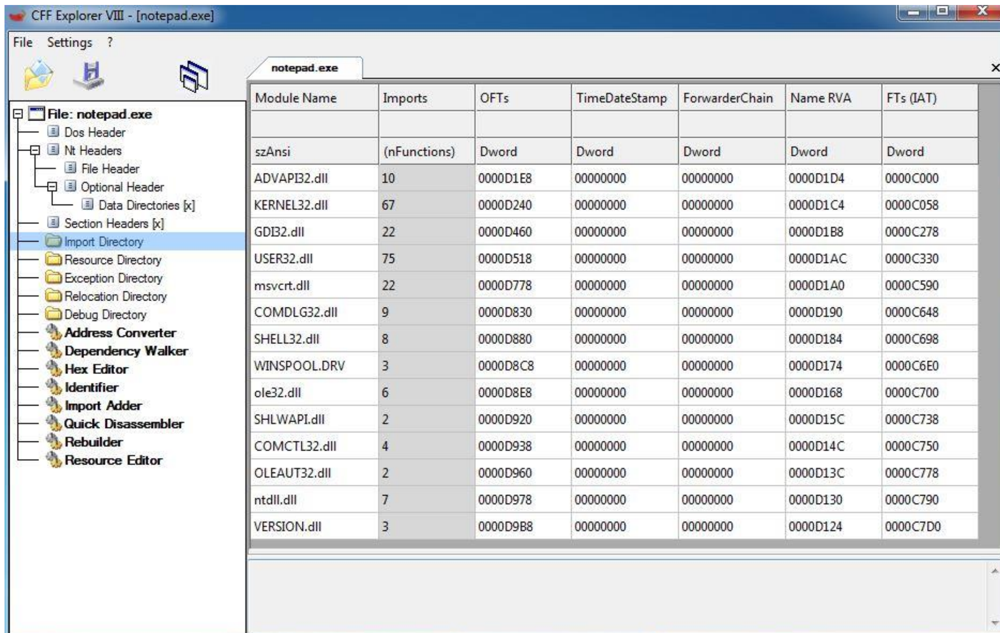
Back to the Section Headers, there is one more thing to mention. Virtual Size and Raw Size are quite important, specially to a malware analyst. In some malware, we may see the raw size to be set to 0, but the virtual size is 1000, 2000 bytes or so. It means that the malware will require that much of space (1000, 2000, etc. for Virtual Size) when it gets loaded (mapped) into memory. Why? Because it may unpack, decrypt or decode some code in memory later when it is running and jump to that section to execute. The reason for this twist is that by using this way, malware can bypass signature based analysis of the anti-virus.

---

[40] You can download Lord PE at http://www.softpedia.com/get/Programming/File-Editors/LordPE.shtml

Imports

Imports section of a PE file includes all imported modules and functions of the program. We can see in figure 6-12 imported modules of notepad.exe.



This table tells a lot about a binary and it's very valuable to a malware analyst to get an idea what functionalities the malware may have. If the malware is packed, this table doesn't tell much.

Static Linking vs Dynamic Linking

We have 2 ways of importing functions and libraries; Static and Dynamic. In static way, the whole imported library will be copied into the program source itself and after linking, there will be made a big binary with all the imported functionalities included.

In Dynamic Linking, a list of pointers to these functions will be saved in the binary after linking so instead of copying the whole library, only a pointer will be saved which results in a much smaller binary. This pointer helps the program to find the requested function in runtime. Another advantage of Dynamic Linking is when a library needs to be patched, only that library gets patched and there will be no need to recompile the binaries

which import those libraries. Imagine if Windows wants to patch kernel32.dll and the binaries in the OS are all statically linked. You should reinstall the whole Windows OS again just to patch 1 DLL!

That’s it for now regarding PE files. There will be updates for this chapter in the future.

Chapter 0x07 - Introduction to WINAPI