

Intermediate IntIX 86 Processor Architecture & Assembly.

Part 2:

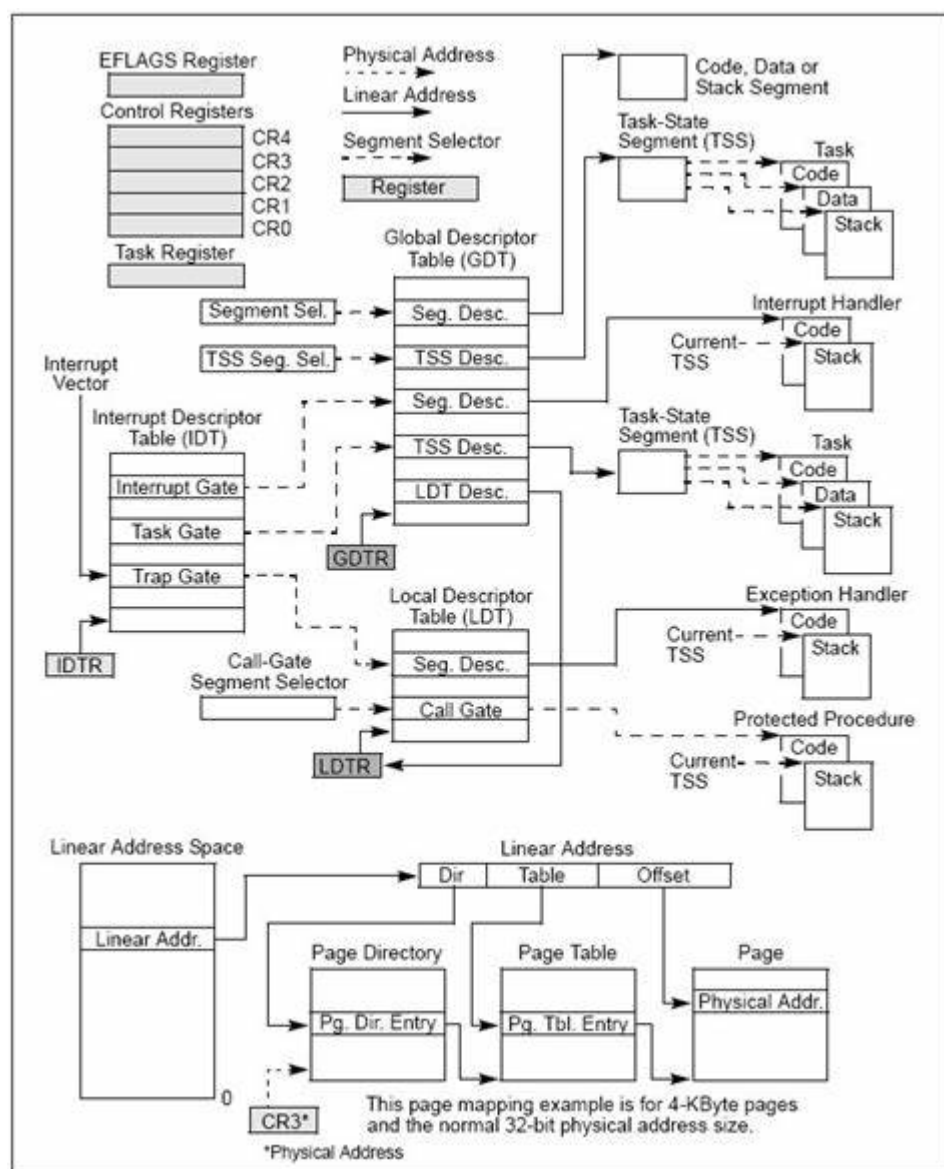
This course will cover the following :

Memory Segmentation Paging Interrupts Debugging, I/O

In this lecture, we will dwell on IA-32, not 64 bit architecture yet. (that is for another one, wait for it 😊)

Also there wont be floating point assembly or registers since in reverse engineering or in malware analysis there is not much of a use for it.

This is what we are goint to learn 😊



CPUID- CPU Feature Identification.

Different processors support different features. **CPUID** is how we know if the chip we are running on supports newer features, such as hardware virtualization, 64 bitmode(asdasdafa), Hyperthreading, thermal monitors etc.

CPUID does not have operands. Rather it **takes input** as value preloaded into **eax** (and possible ecx). After it finishes , the outputs are stored to **eax, ebx, ecx and edx**

so if you want your code to be compatible, you need to check some features before implementing.

ID flag in EFLAGS(bit 21), this is the CPUID flag. If it set to 0, you set to 1 and if it stays at 1, then it has CPUID, if it returned back to 0, then you dont have CPUID. But how do we read and write EFLAGS?

In order to manipulate the flags, we have 2 instructions.

PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	Valid	N.E.	Push RFLAGS.

there are FLAGS which are 16 bits and also there are E(xtended)FLAGS, which are 32 bits. make sure which one to push and pull. the problem occurs because all of them have the same opcode, which is **9C**. the instunction makes the difference.

PUSHFD: If you need to read the entire EFLAGS register, make usre you use PUSHFD not PUSHF. The difference is, PUSHFD uses Dword size flags so its not 16 bits but 32 bits.

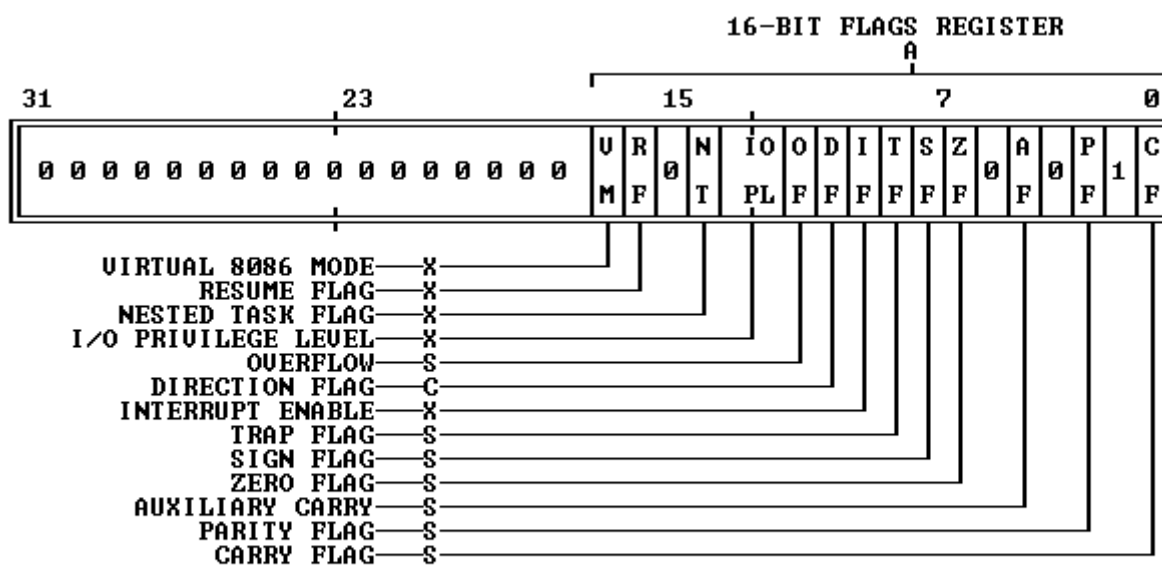
so PUSHFD takes the flag and puses it to the stack, just like anything else.

opcode for PUSHFD is **9C**

POPFD: There are some flags which will not be transferred from the stack to EFLAGS unless you are in ring 0. these are security purpose flags. we are generally operating on ring 3. If you need to set the entiure EFLAGS register, make sure you use **POPFD** not just **POPF**, same 16 bit issue.

opcode for POPFD is **9D**

Figure 2-8. EFLAGS Register



S = STATUS FLAG, C = CONTROL FLAG, X = SYSTEM FLAG

NOTE: 0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE

Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 3-7). "Genu" "ntel" "inel"
1H	EAX EBX ECX EDX	Version Information (Type, Family, Model, and Stepping ID) Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size. (Value * 8 = cache line size in bytes) Bits 23-16: Number of logical processors per physical processor. Bits 31-24: Local APIC ID Reserved Feature Information (see Figure 3-4 and Table 3-9)
2H	EAX EBX ECX EDX	Cache and TLB Information Cache and TLB Information Cache and TLB Information Cache and TLB Information
3H	EAX EBX ECX EDX	Reserved. Reserved. Bits 00-31 of 96 bit processor serial number. (Available in Pentium® III processor only; otherwise, the value in this register is reserved.) Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	Extended Function CPUID Information	
80000000H	EAX EBX ECX EDX	Maximum Input Value for Extended Function CPUID Information (see Table 3-7). Reserved. Reserved. Reserved.
80000001H	EAX EBX ECX EDX	Extended Processor Signature and Extended Feature Bits. (Currently Reserved.) Reserved. Reserved. Reserved.
80000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
Initial EAX Value	Information Provided about the Processor	
80000004H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.

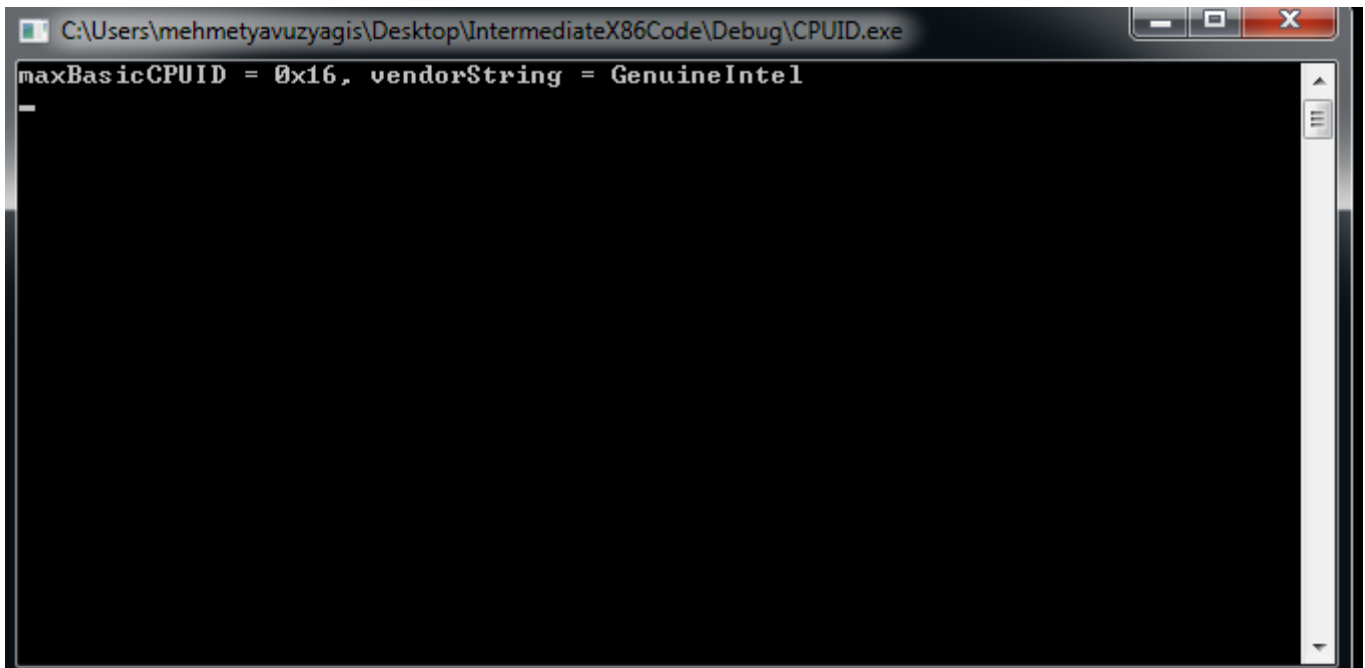
CPUIDs are also returning processor manufacturer ID strings. for example,

```

AMDisbetter
CyrixInstead
GenuineIntel
sis sis sis

```

This is what I got running the cpuid.c file in the source files



here **0x16** is actually what is stored in my **eax**. so nothing big.

according to CPUID return value, 0x16 gives this kind of information:

```
Skylake-based processors (proc base & max freq; Bus ref. freq) 0x16
0x8000 0008
```

Information provided about the processor :

EAX: VirtualPhysical Address size bits 7-0: Physical Address bits 8-15: bVirtual address bits 31-16: reserved

EBX: Reserved=0 ECX: Reserved=0 EDC: Reserved=0

So, rings and modes:

Real mode is , when you restart the processor it enter the mode called **real mode** and basically its like compatibility mode. For example when you run DOS now, it runs on this real mode. No virtual memory, no privilege rings, 16 bit mode.

thats it actually.

Most of the OSs run in protected mode.

Protected Mode

this mode is the native state of the processor. Among the capabilities of protected mode is the ability to directly execute **Real-address mode**. 8086 software in a protected, multi taskin environment. This feature is called **virtual-8086 mode** although it is not actually a processor mode.

Virtual-8086 is just for backwards compatibility,.

Protected mode adds support for virtual memory and privile rings.

But when cpu is restarting it restarts in real mode and goes into protected mode. so there is a bootstrapping around 16 bit real mode.

System Management Mode(SMM)

This mode provides an operating system or executive with a transferred mechanism for implementing platform-specific functions such as power management and system security. in this mode, you have reach all of the memory, hardware support. OS and hypervisor cannot reach to this level. The processor enters SMM when the external SMM interrupts pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC)

THUS, SMM has become a popular target for advanced rootkit discussions since access to SMM is locked by BIOS, so that neither **ring 0** nor **VMX** hypervisors can access it! Thus, if VMX is more privileged than ring 0 ('ring -1'), SMM is more privileged than VMX('ring -2') because a hypervisor can't even read SMM memory.!

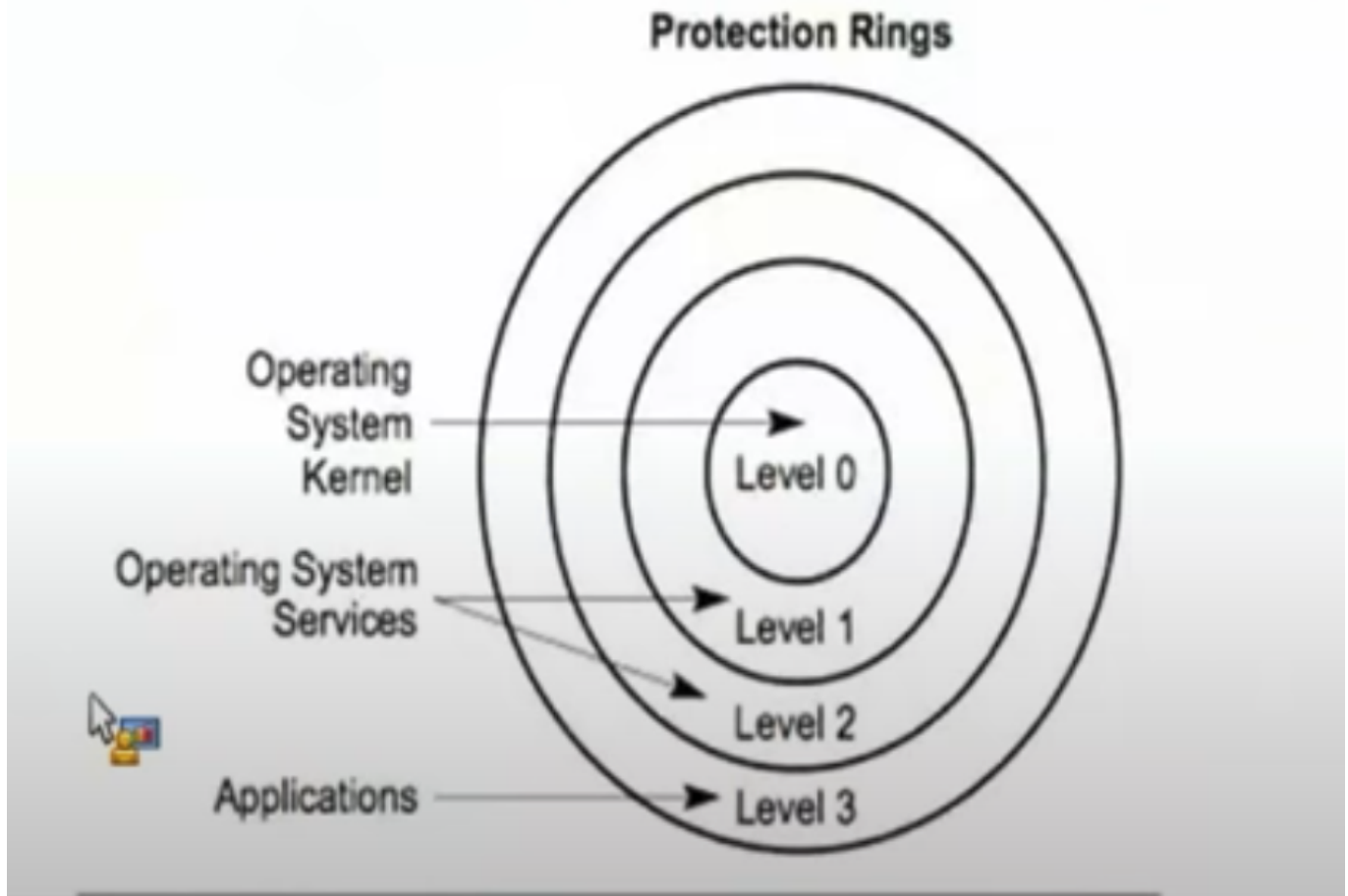
this is sick! once a kit is hooked to this level, it blocks that memory part and is hidden inside!

<https://github.com/jussihi/SMM-Rootkit> check this for SMM rootkit example

Privilege Rings

- x86 rings are enforced by hardware.
- you often hear that normal programs execute in ring 3(user space) and the privileged code executes in ring 0(kernel space).
- in order to understand rings, we need to understand a capability called segmentation

Rings on x86



Paravirtualized XEN!

Requires a modified guest os. what is paravirtualization?

Paravirtualization (PV) is an efficient and lightweight virtualization technique introduced by the Xen Project team, later adopted by other virtualization solutions. PV does not require virtualization extensions from the host CPU and thus enables virtualization on hardware architectures that do not support Hardware-assisted virtualization. However, PV guests and control domains require kernel support and drivers that in the past required special kernel builds, but are now part of the Linux kernel as well as other operating systems.

Paravirtualization implements the following functionality

Disk and Network drivers Interrupts and timers Emulated Motherboard and Legacy Boot Privileged Instructions

so instad of guest OS is touching to the hardware, it touches an API and that API controls these requests.

SEGMENTATION

NOTE THAT INX64 ARCHITECTURE, SEGMENTATION IS NOT USED. THIS IS MERELY FOR 32 BIT

Segmentation provides a mechanism dividing the processor's addressable memory space (called linear address space) into smaller protected address spaces called (segments)

When we talk about segmentation and segment registers, we are talking about their interactions with the **Linear Address Space** which is actually on the virtual memory **but maps to physical memory 1 to 1 so it is safe to say that it is physical memory at least until we start to talk about paging.**

so segmentation therefore taking the code from chunk of memory or instructions and saving them into ring 3, ring 0 etc.

During this interaction, segment registers take the instruction or data, pipes it through the **segment addressing** process and then undertakes the operation per requested.

But what is segment addressing?

To locate a byte, (finding an address in the memory for example), a **logical address** also called a far pointer must be provided. A logical address consists of a **segment selector** and an **offset**. Offset belongs to the address that is being selected.

So you select the segment first. and within this segment, you select an address, this address selection is made by providing offset of the address

The physical address space is defined as the range of addresses that the processor can generate on its address bus.

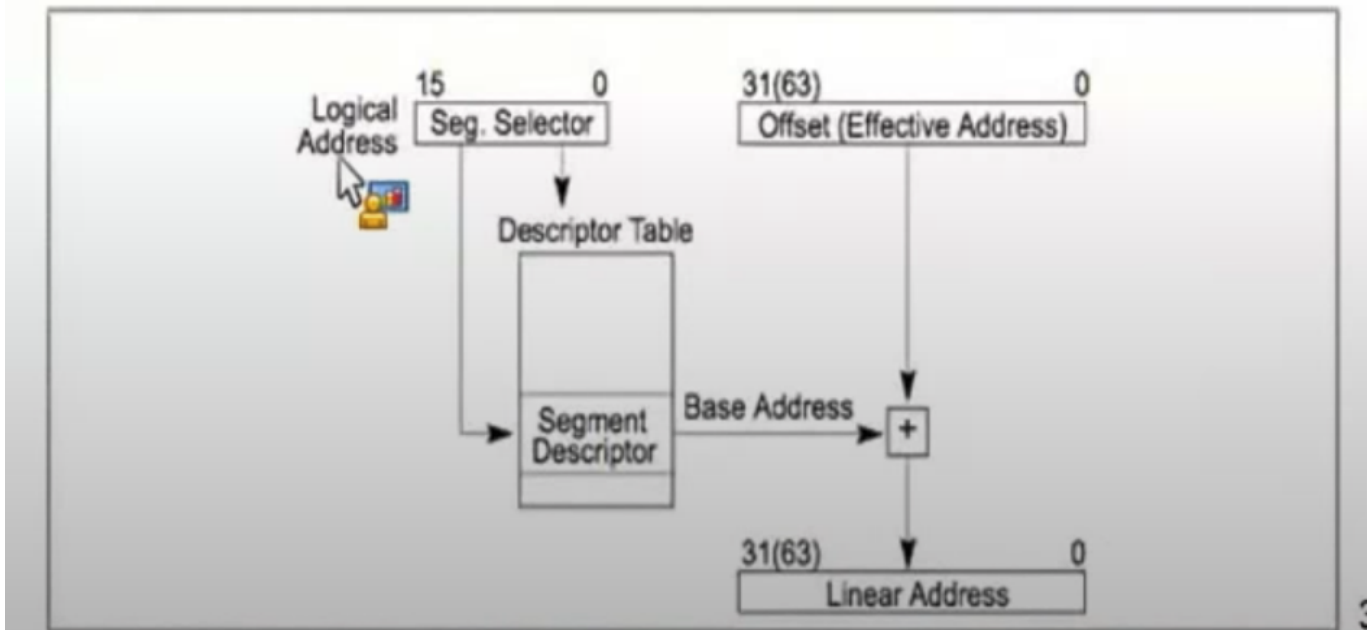
Physical address space is based on how much RAM you have installed basically. in 32 bit, it is up to a maximum of 4GB since $2^{32} = 4\text{GB}$. But there is a mechanism (physical address extensions-PAE) which allows systems to access a space up to 64 GB.

Linear Address space, on the other hand, in 32 bit systems, is a flat 32 bit space.

Segmentation is not optional. Segmentation translates logical addresses to linear addresses automatically in hardware by using table lookups.

logical address(far pointer yani) == 16 bit segment selector + 32 bit offset.

if paging is disabled, linear addresses map directly to physical addresses.



when using a selector of logical address, it goes to segment selector. This segment selector goes to the descriptor table (this is just a big array that holds access - limit and base addresses which we pull) and selects what is needed based on the offset because each segment descriptor in the descriptor table holds many addresses, offset is used to specify which one to pull exactly. Once the base address pulled up from the descriptor table, it maps this address to linear address.

think of it like cargo companies.

there is an address parcel needs to be delivered to. this is the base address. We first specify which city and neighborhood it is in. this is segment selector.

A nice description below

Segment selector says this is in Istanbul_Uskudar. Okay but there are lots of addresses in istanbul uskiudar. then the next piece of information comes. to this selector, we add offset. offset gives the full address and parcel is reached.

so offset is added to the segment selector after segmentation is reached.

after that, hardware maps it to the linear addresses, so that real memory can be reached.

Segment Selectors

A segment selector is a 16 bit value held in a segment register which is also 16 bit.

Segment registers are up to 6. namely they are **CS SS DS ES FS GS**

It is used to select an index for a segment descriptor from one of two tables:

Global Descriptor Table (GDT) this is for system-wide use

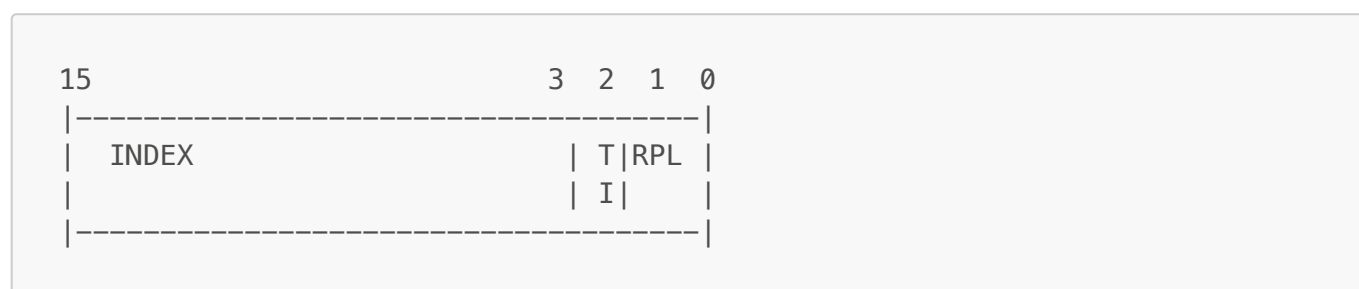
Local Descriptor Table (LPT)

Intended to be a table per-process and switched when the kernel switches between process contexts.

Note that the table index is actually 13 bits not 16, so the tables can each hold $2^{13}=8292$ descriptors

index is supposed to match the required offset. based on its position on the array, segment selectors get indexed and offset finds them on index numbers.

Anatomy of a segment selector



RPL == Requested Privilege level

2 bit value means it can hold values from 0 to 3 so think of it as ring0, ring3

TI = Table indicator

this is one bit. binary. either Global table or Local table where:

0 =GDT

1=LDT

the Six Segment Registers

- **CS** ==> Code Segment

Hardware always uses the code segment. during the code execution, hardware implicitly goes to logical address and uses CS register. So in the code you are jumping from 1 to 3, the hardware reads the code and goes to CS and it looks the table and index and then executes the code. So CS is ALWAYS used implicitly by the hardware for code access!

Processor differentiates between code access and data access. When we observe a change in **EIP** register that is a code access. hardware knows where to go into the memory to pull the EIP and address for the next instruction. but when we move data from here to there, **MOV** its the job of the stack segment

- **SS** ==> Stack Segment

Stack segments are data segments which must be read/write segments. Loading the SS register with a segment selector for a nonwritable data segment generates a general-protection exception(#GP)

hardware uses the SS for data access. All the data and all the frame is sort if within the SS.

- **DS** ==> Data Segment

Data register is the register that holds and acts on data. rep movs or rep stos for example moves data back and forth between DS and ES.

This is like a general purpose register, you can put whatever you want and use it freely.

- **ES/FS/GS** ==> Extra(usually data) segment registers

Segments can overlap. code can be data and data can be code. this is very normal.

One big intake from here is **Hardware uses CS for all these nice instructions!**

SS for data moving, CS is for all these jumps, compares, adds etc.

Visible Part		Hidden Part	
Segment Selector	Base Address, Limit, Access Information		
			CS
			SS
			DS
			ES
			FS
			GS

There is a visible part and there is a hidden part.

visible part is where segment selector, which is 16 bits. but since the hardware does not want to walk through the table all the time. upon the call, hardware walks the table once, and the caches information into hidden part. So takes the description, takes what is asked from it, and caches the rest into hidden part.

so once CS reaches one part of the table, rest is also reachable via hidden part.

- the "hidden part" is like a cache so that segment descriptor info doesnt have to be looked ip each time.

Implicit use of segment registers:

When you are **accessing** the stack, you are implicitly using a logical address that is using the SS(stack register) as the segment selector so **ESP ==SS:ESP** pushing and popping esp for example.

when you are **modifying** EIP,(with jumps, calls, rets) you are implicitly using the CS(code segment) register as the segment register. **EIP = CS:EIP**

Even if a disassembhler does not show it, the use of segment registers is built into some of the instructions.

Explicit use of segment registers:

You can write assembly which explicitly specifies which segment registers to use. Just prefix the memory address with a segment register and a colon.

`mov eax,[ebx]` vs `mov eax, fs:[ebx]`

the assembly just puts a prefix on the instruction to say when this inst is asking for memory, its actually asking for memory in this segment.

Actually this way you are specifying a full logical address/far pointer.

using `UserspaceSegmentRegisters.c` file in the source folder, we will go little bit forward and apply what was on theory now

here is the C code :

```
#include <stdio.h>

//prototype for helper function
void SelectorPrint(char * segRegName, unsigned short segReg);

int main(){
    unsigned short myCS, myDS, myES, myFS, myGS, mySS;

    //Move the segment registers into the C variables
    __asm{
        mov myCS, cs;
        mov myDS, ds;
        mov myES, es;
        mov myFS, fs;
        mov myGS, gs;
        mov mySS, ss;
    }

    //Each segment register holds a 16 bit segment selector which is
    defined as
    //15      3 2 1 0
    //-----
    //| Index |T|RPL| ... ASCII "ART" FTW!
    //-----
    //RPL = Requested Privilege Level (2 bits)
    //T = Table Indicator, 0 = GDT, 1 = LDT (1 bit)
    //Index = This selector selects the Index-th entry in the GDT or LDT
    (13 bits)

    //Parse the meaning of the selectors stored in the registers
    SelectorPrint("cs", myCS);
    SelectorPrint("ss", mySS);
    SelectorPrint("ds", myDS);
    SelectorPrint("es", myES);
    SelectorPrint("fs", myFS);
    SelectorPrint("gs", myGS);

    return 0x0ddba11;
```

```

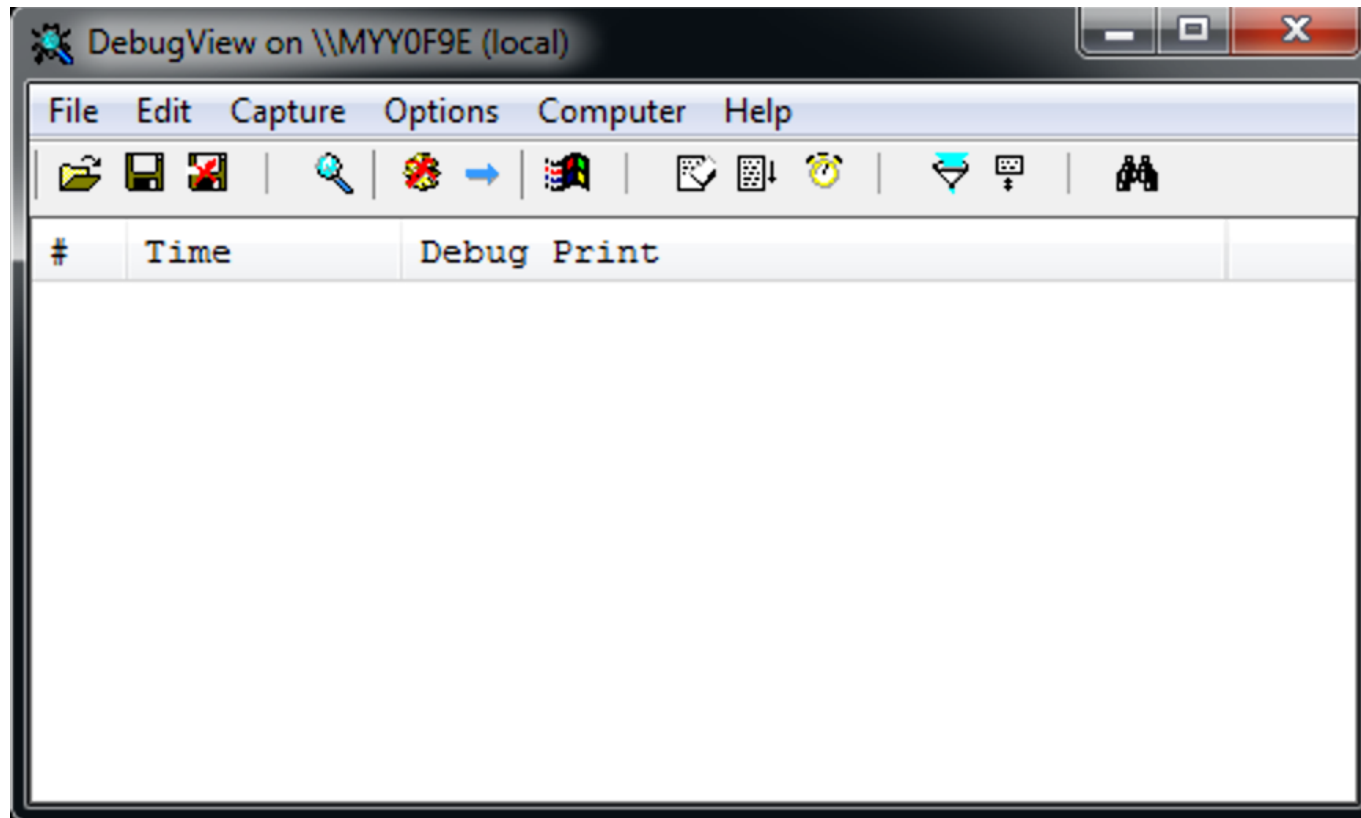
}

void SelectorPrint(char * segRegName, unsigned short segReg){
    printf("The segment selector stored in the %s register = %#x\n",
segRegName, segReg);
    if(segReg == 0){
        printf("A segment selector of 0 is invalid because the 0th entry
of the GDT is never used\n\n");
        return;
    }
    printf("\tIts Requested Privilege Level is %u\n", (segReg & 0x3));
    printf("\tIt selects a segment in the ");
    if((segReg & 0x4) == 0){
        printf("GDT ");
    }
    else{
        printf("LDT");
    }
    printf(" of index %#x\n\n", segReg >> 3);
}

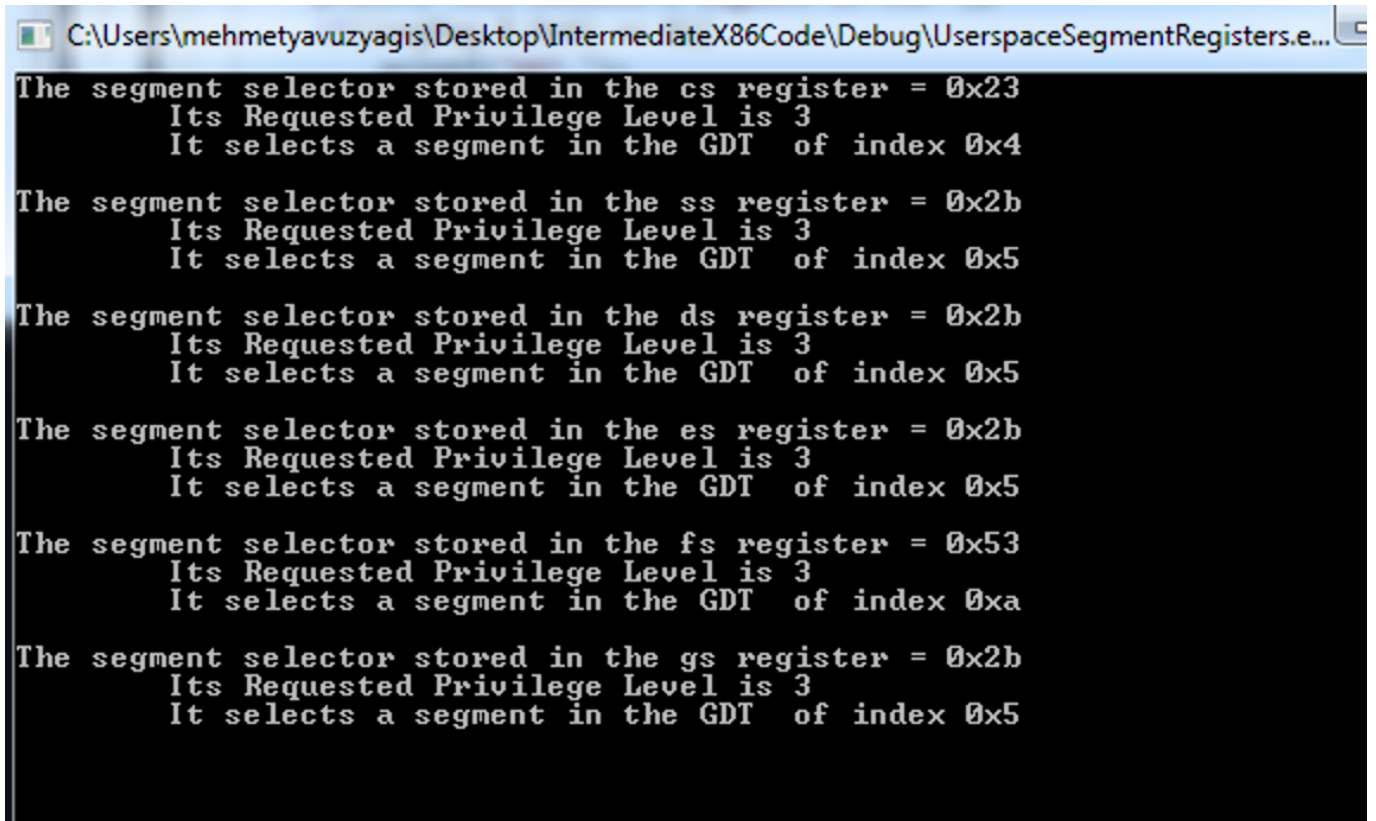
```

we are downloading and using [debugview V 4.76](#) here. I will add it to the repo.

after opening the app with escalated privileges, we will enable the kernel capture (the red button)



running the app above , we have this output:



```
C:\Users\mehmetavuziyagis\Desktop\IntermediateX86Code\Debug\UserspaceSegmentRegisters.e...  
The segment selector stored in the cs register = 0x23  
  Its Requested Privilege Level is 3  
  It selects a segment in the GDT of index 0x4  
The segment selector stored in the ss register = 0x2b  
  Its Requested Privilege Level is 3  
  It selects a segment in the GDT of index 0x5  
The segment selector stored in the ds register = 0x2b  
  Its Requested Privilege Level is 3  
  It selects a segment in the GDT of index 0x5  
The segment selector stored in the es register = 0x2b  
  Its Requested Privilege Level is 3  
  It selects a segment in the GDT of index 0x5  
The segment selector stored in the fs register = 0x53  
  Its Requested Privilege Level is 3  
  It selects a segment in the GDT of index 0xa  
The segment selector stored in the gs register = 0x2b  
  Its Requested Privilege Level is 3  
  It selects a segment in the GDT of index 0x5
```

So segment selector store in the cs selector = 0x23. its requested privilege is 3 means its on ring 3, user space. its in the GDT level, so got the value 0 on T1.

in command prompt we cd into Desktop/intermediatex86/KernelSpaceSegmentRegisters folder, which is added to this repo. Basically it is the code I copied above. and run **load.bat** the same output can be seen on the DebugView app.

Segment Register	RPL	Table	Index
CS = 0x1b	3	GDT	3
SS = 0x23	3	GDT	4
DS = 0x23	3	GDT	4
ES = 0x23	3	GDT	4
FS = 0x3B	3	GDT	7
GS = 0	Invalid	Invalid	Invalid

Segment Register	RPL	Table	Index
CS = 0x8	0	GDT	1
SS = 0x10	0	GDT	2
DS = 0x23	3	GDT	4
ES = 0x23	3	GDT	4
FS = 0x30	0	GDT	6
GS = 0	Invalid	Invalid	Invalid

Userspace is set everything to work on RPL 3, ring 3, which is the user space itself whereas kernelSpace segmentation shows taht RPL is set to 0 and 3. **DS** and **ES** do not change between kernel and userspaces.

so whtt is our intake from this?

windows maintains different CS, SS & FS segment selectors for userspace processes vs kernel ones.

The RPL field seems to correlate with the ring for kernel or userspace

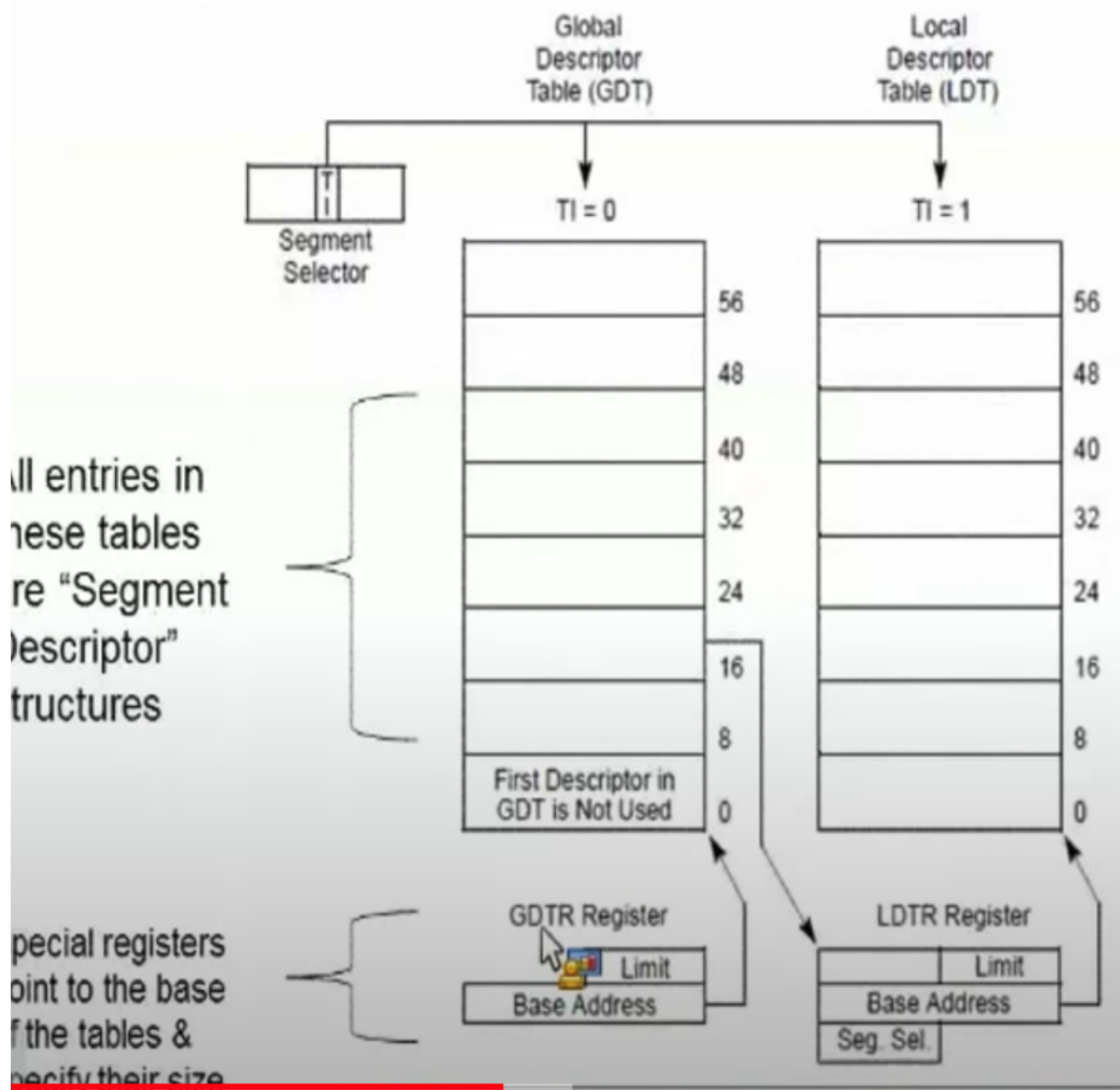
windows does not change DS and ES between kernel and userspaces.

Windows does not use GS.

GDT & LDT

GDT

GDT & LDT



so we have GDT and LDT arrays. the segment selector is loaded with the information to where to look which is a 2 bit place of TI. it is either GDT(0) or LDT(1).

Each entry in this table is 8 bytes.

GDT:

the upper 32 bits ("base address") of the register specify the linear address where the GDT is stored.

the lower 16 bits("table limit) specify the size of the table in bytes

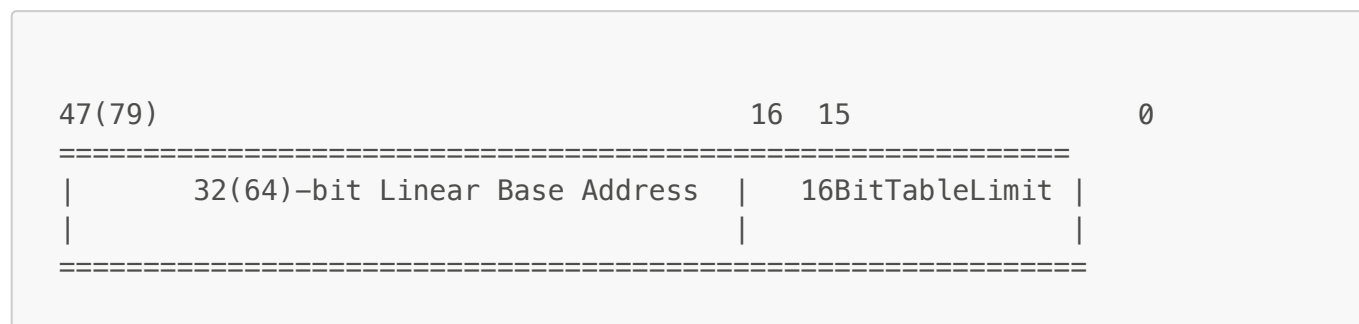
special instructions used to load a value into the register or store the value out to memory:

these special instructions are :

LGDT ==> load 6 bytes from memory to GDTR

GDTR ==> store 6 bytes of GDTR to memory.

A GDT ASCII schema.



so GDTR has 16bit chunk of **limit** and 32 bit size of **base**. upon call, register goes to the table and checks the limit of the gdt. based on this limit, it locates the logical memory address.

this is the playground of the operating system, it does not want third party softwares to mess with it.

LDT

like the segment registers, the LDT has a visible part, the segment selector and a hidden part, the cached segment info which specifies the size of the lds

special instructions:

LLDT == > load 16 bit segment into LDTR

SLDT ==> store 16 bit segment sel3efctor of LDTR to memory

So what are the **Segment Descriptors**?