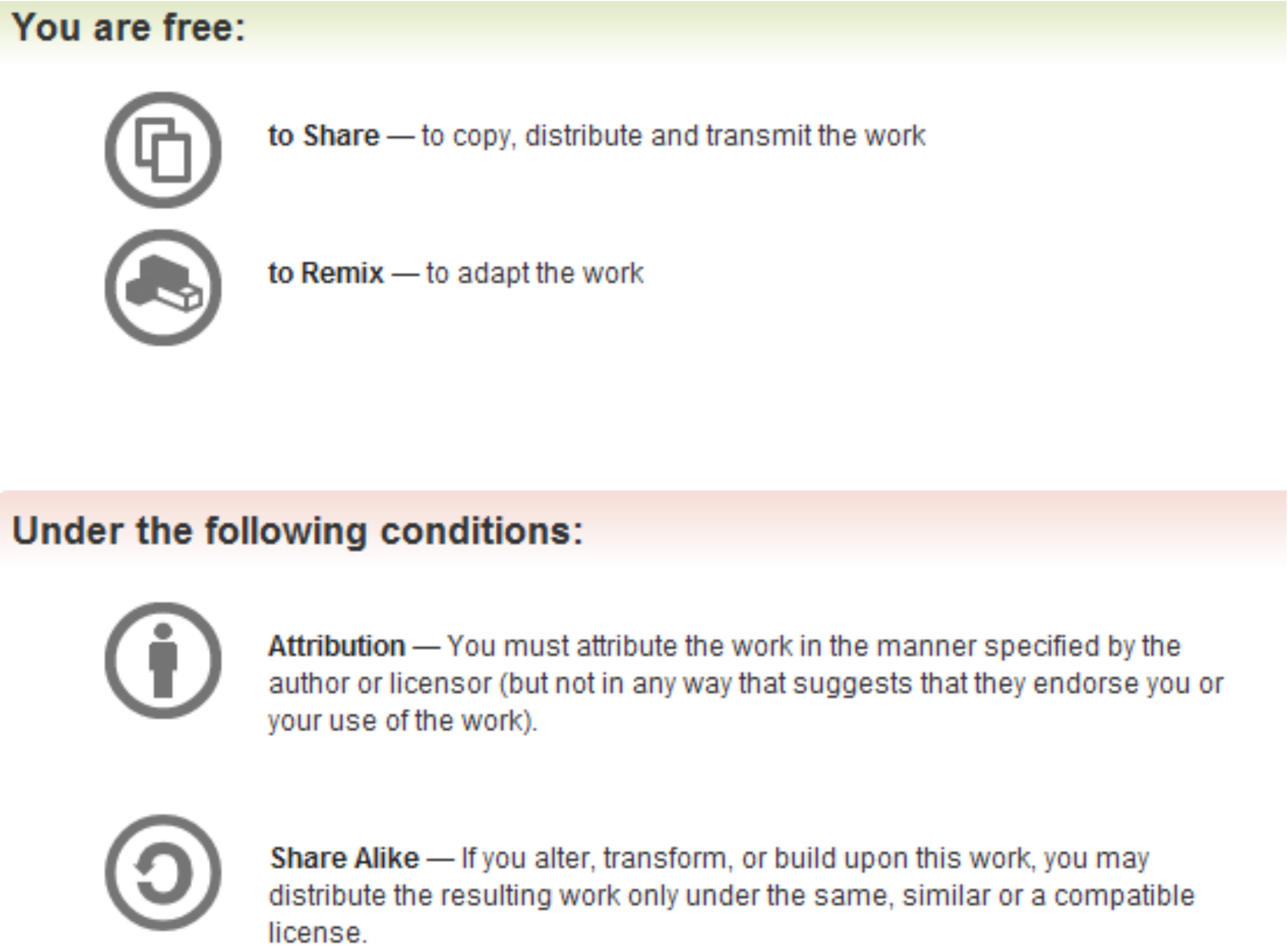
Intermediate Intel x86: Assembly, Architecture, Applications, and Alliteration

Xeno Kovah – 2010

xkovah at gmail

All materials are licensed under a Creative Commons "Share Alike" license.

- http://creativecommons.org/licenses/by-sa/3.0/



You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Credits Page

- Your name here! Just tell me something I didn’t know which I incorporate into the slides!
- Veronica Kovah for reviewing slides and suggesting examples & questions to answer
- Murad Kahn for Google NaCl stuff

Answer me these questions three

- What, is your name?
- What, is your quest?
- What…is your department?

Agenda

- Part 1 - Segmentation
- Part 2 - Paging
- Part 3 - Interrupts
- Part 4 – Debugging, I/O, Misc fun on a bun

Miss Alaineous

- Questions: Ask ‘em if you got ‘em
  - If you fall behind and get lost and try to tough it out until you understand, it’s more likely that you will stay lost, so ask questions ASAP.
- Browsing the web and/or checking email during class is a great way to get lost ;)
- 2 hours, 10 min break, 2 hours, 1 hour lunch, 1 hour at a time with 5 minute breaks after lunch
- Adjusted depending on whether I’m running fast or slow (or whether people are napping after lunch :P)

Class Scope

- We' re going to only be talking about 32bit architecture (also called IA-32). No 16bit or 64bit.
- Also not really going to deal with floating point assembly or registers since you don' t run into them much unless you' re analyzing math/3d code.
- This class focuses on diving deeper into architectural features
- Some new instructions will be covered, but mostly in support of understanding the architectural topics, only two are learning new instruction' s sake. As shown in last class, we have already covered the majority of instructions one will see when examining most programs.

What I hope you get out of the class

- A better understanding of Intel architecture and how it's leveraged by OSes
  - Which can in turn translate to understanding how OSes are virtualized
- Knowledge of where hardware support for security exists, and how it is or isn't used.
- A base for understanding even more advanced features. The curiosity to independently explore advanced features.
- The satisfaction that comes with knowing how something works at a very fundamental level.

Instructions Quiz

- Learned around 26 instructions and variations
- About half are just math or logic operations
- NOP
- PUSH/POP
- CALL/RET
- MOV/LEA
- ADD/SUB
- JMP/Jcc
- CMP/TEST
- AND/OR/XOR/NOT
- SHR/SHL/SAR/SAL
- IMUL/DIV
- REP STOS, REP MOVS
- LEAVE

Stack Quiz: Example1.c

```
//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
    return 0xbeef;
}
int main(){
    sub();
    return 0xf00d;
}
```

```
sub:
00401000  push        ebp
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  call        sub (401000h)
00401018  mov         eax,0F00Dh
0040101D  pop         ebp
0040101E  ret
```

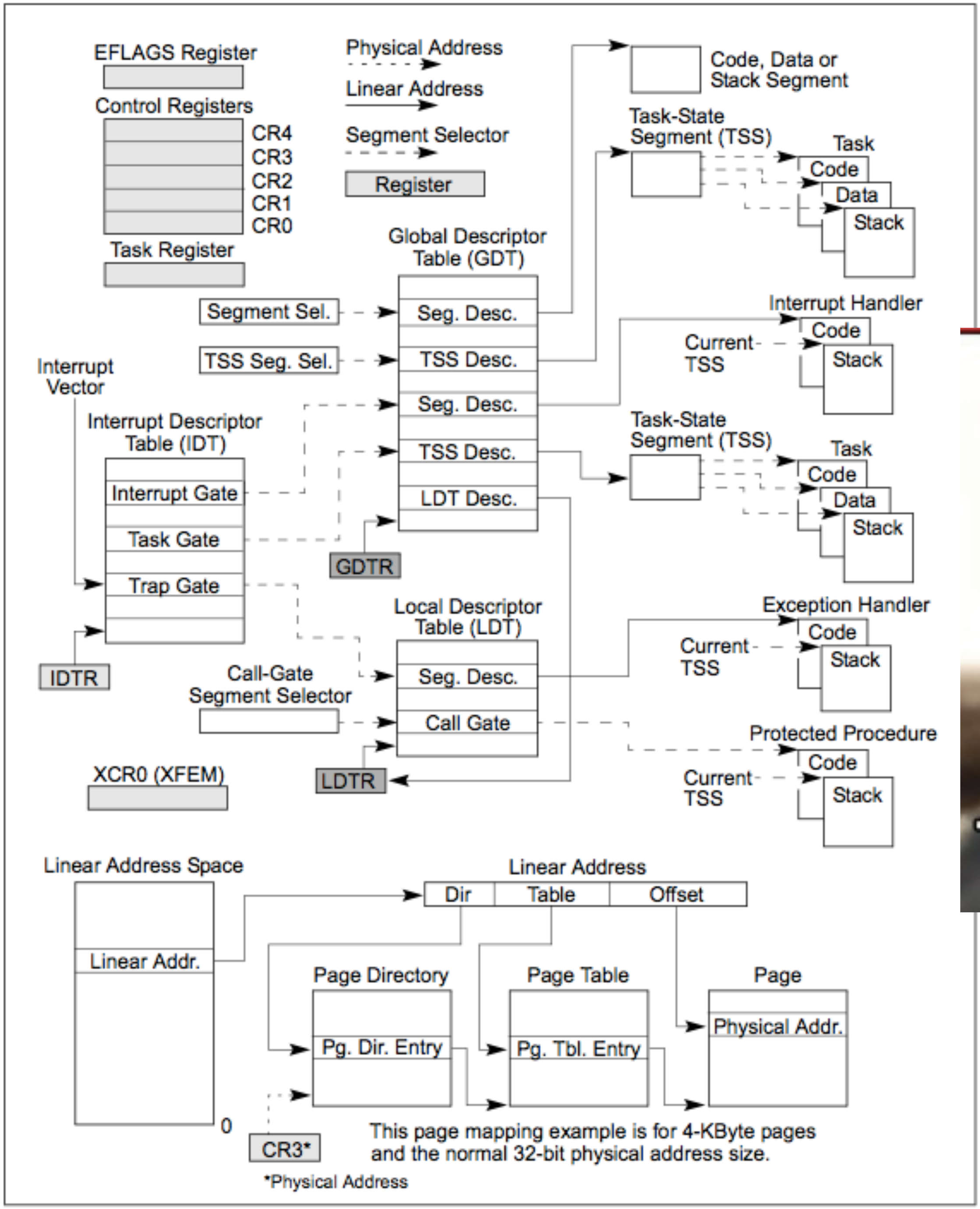


Figure 2-1. IA-32 System-Level Registers and Data Structures

That's what you're going to learn! :D

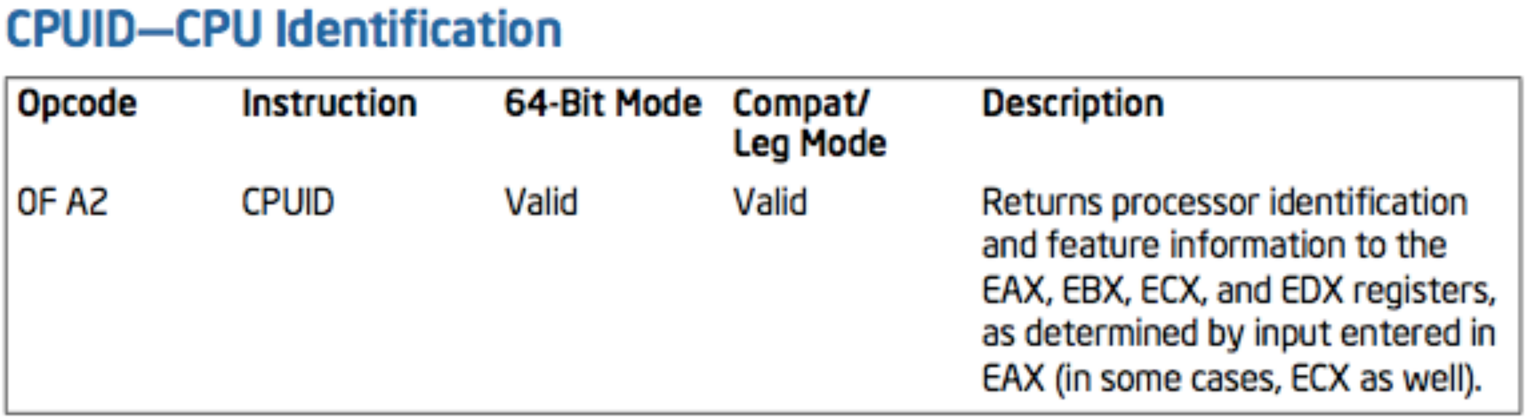
ME

You
by the
end

As a nerd I actually find this graphic to be disappointingly inaccurate because this is *not* where he said the line…I couldn’t find a good picture of that moment.
If I took any pride in my work I would have rented the video and taken a screen shot…

Morning Warm Up
CPUID - CPU (feature) Identification

- Different processors support different features
- CPUID is how we know if the chip we’re running on supports newer features, such as hardware virtualization, 64 bit mode, HyperThreading, thermal monitors, etc.
- CPUID doesn’t have operands. Rather it “takes input” as value preloaded into eax (and possibly ecx). After it finished the outputs are stored to eax, ebx, ecx, and edx.



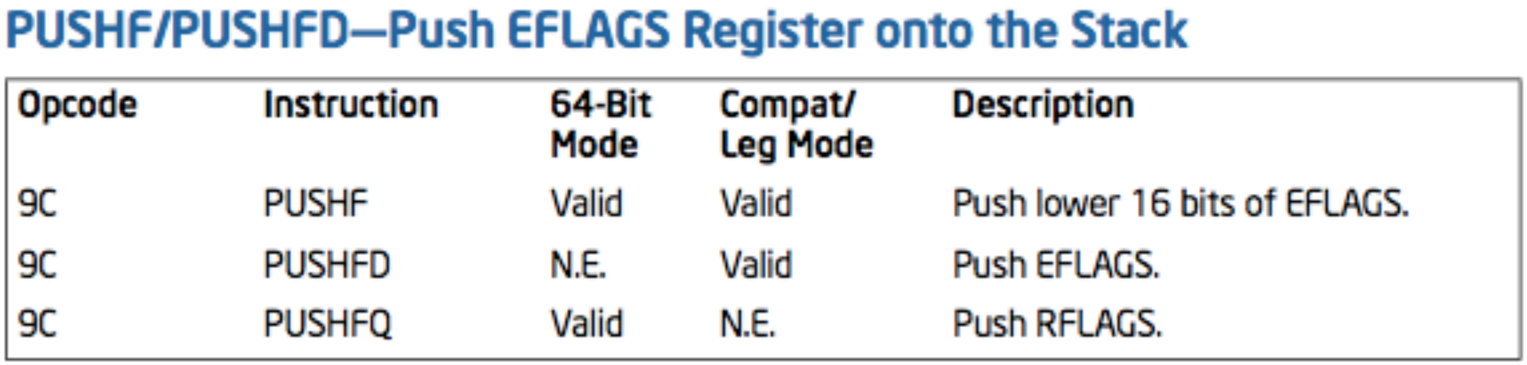
CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

How do we even know if we can use CPUID?

- CPUID not added until late model 486s
- ID Flag in EFLAGS (bit 21)
- "The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction." (Vol. 3a, Sect. 2.3)
- How do we read/write to EFLAGS?
- PUSHFD/POPFD

PUSHFD - Push EFLAGS onto Stack

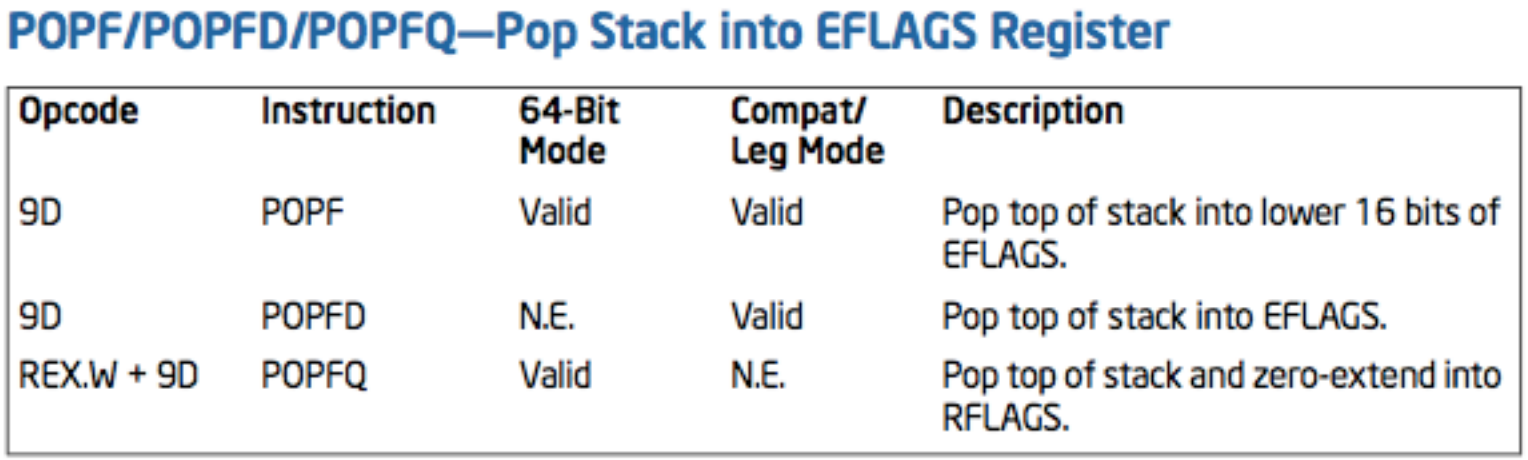


PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

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.

- If you need to read the entire EFLAGS register, make sure you use PUSHFD, not just PUSHF. (I found Visual Studio forces the 16 bit form if you don’t have the D!)

POPFD - Pop Stack Into EFLAGS



POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9D	POPF	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	N.E.	Valid	Pop top of stack into EFLAGS.
REX.W + 9D	POPFQ	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

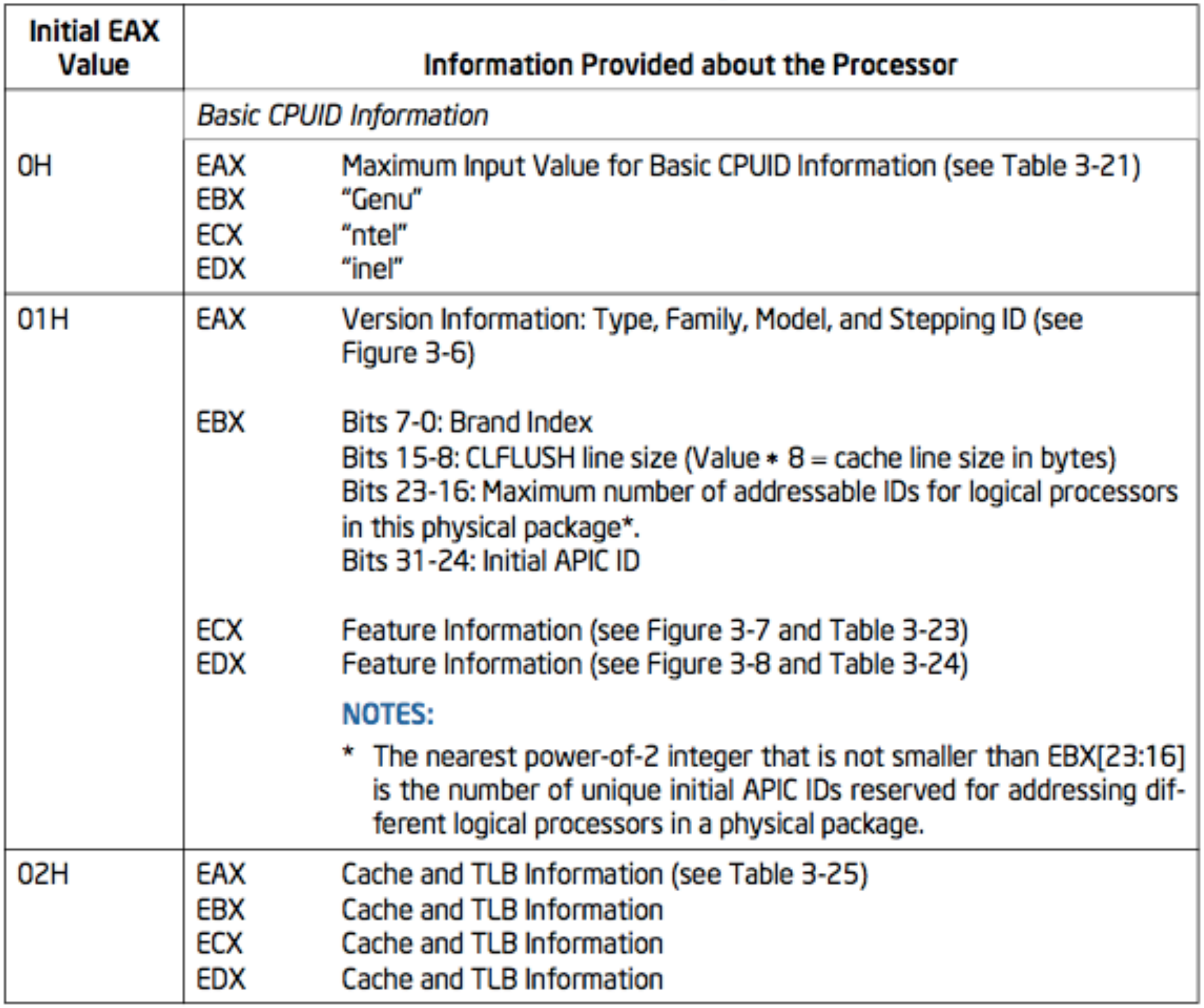
- There are some flags which will not be transferred from the stack to EFLAGS unless you’re in ring 0.
- If you need to set the entire EFLAGS register, make sure you use POPFD, not just POPF. (I found Visual Studio forces the 16 bit form if you don’t have the D!)

Some Example CPUID Inputs and Outputs

REGISTER BEFORE CPUID EXECUTES

REGISTERS AFTER CPUID EXECUTES

Table 3-20. 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 3-21) "Genu" "ntel" "inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6)
	EBX	Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID
	ECX EDX	Feature Information (see Figure 3-7 and Table 3-23) Feature Information (see Figure 3-8 and Table 3-24)
		NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package.
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 3-25) Cache and TLB Information Cache and TLB Information Cache and TLB Information

Lab: CPUID.c

```
int main(){
    unsigned int maxBasicCPUID;
    char vendorString[13];
    char * vendorStringPtr = (char *)vendorString; //Move the address into its own register
                                               //because it makes the asm syntax easier

    //First we will check whether we can even use CPUID
    //Such a check is actually more complicated than it seems (OMITED FROM SLIDES)
    if(CheckIfWeCanUseCPUID() == 1){
        __asm{
            mov edi, vendorStringPtr; //Get the base address of the char[] into a register
            mov eax, 0; //We're going to do CPUID with input of 0
            cpuid;      //As stated, the instruction doesn't have any operands
            //Get back the results  which are now stored in eax, ebx, ecx, edx
            //and will have values as specified by the manual
            mov maxBasicCPUID, eax;
            mov [edi], ebx; //We order which register we put into which address
            mov [edi+4], edx; //so that they all end up forming a human readable string
            mov [edi+8], ecx;
        }
        vendorString[12] = 0;
        printf("maxBasicCPUID = %#x, vendorString = %s\n", maxBasicCPUID, vendorString);
    }
    else{
        printf("Utter failure\n");
    }
    return 0xb45eba11;
}
```

CPUID Misc

- I highly recommend Amit Singh's CPUID info dumping program for *nix systems
  - http://www.osxbook.com/blog/2009/03/02/retrieving-x86-processor-information/
- Also see "Intel Processor Identification and the CPUID Instruction - Application Note 485" for a lot more info about CPUID
  - http://www.intel.com/Assets/PDF/appnote/241618.pdf

In the beginning, there was real mode. And it was teh suck.

- **Real-address Mode** - (I call it Real Mode or maybe “For Reals Mode…Seriously. For Reals. Mode.”) “This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.”
- DOS runs in Real Mode.
- No virtual memory, no privilege rings, 16 bit mode

Vol. 1, Sect. 3.1

Processor Modes 2

- **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-tasking environment. This feature is called **virtual-8086 mode**, although it is not actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.”
- Virtual-8086 is just for backwards compatibility, and I point it out only to say that Intel says it’s not really its own mode.
- Protected mode adds support for virtual memory and privilege rings.
- Modern OSes operate in protected mode

Vol. 1, Sect. 3.1

Processor Modes 3

- **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).”
- SMM has become a popular target for advanced rootkit discussions recently because access to SMM memory is locked 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.
- Reserving discussion of VMX and SMM for Advanced x86 class

Vol. 1, Sect. 3.1

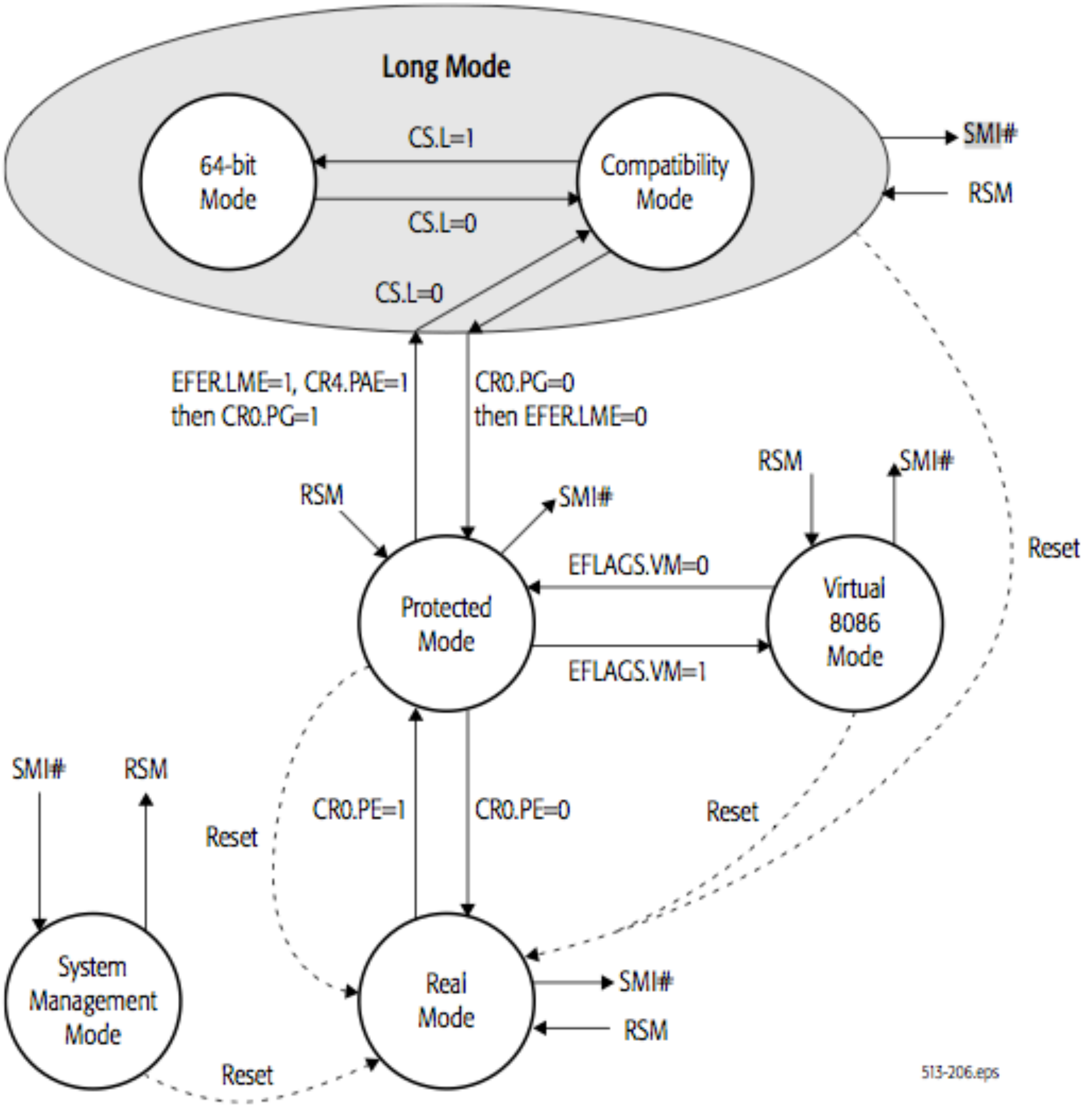


Figure 1-6. Operating Modes of the AMD64 Architecture

From http://support.amd.com/us/Processor_TechDocs/24593.pdf

Privilege Rings

- MULTICS was the first OS with support for hardware-enforced privilege rings
- x86's rings are also enforced by hardware
- You often hear that normal programs execute in "ring 3" (userspace/usermode) and the privileged code executes in "ring 0" (kernelspace/kernelmode)
- The lower the ring number, the more privileged the code is
- In order to find the rings, we need to understand a capability called segmentation

Rings on x86

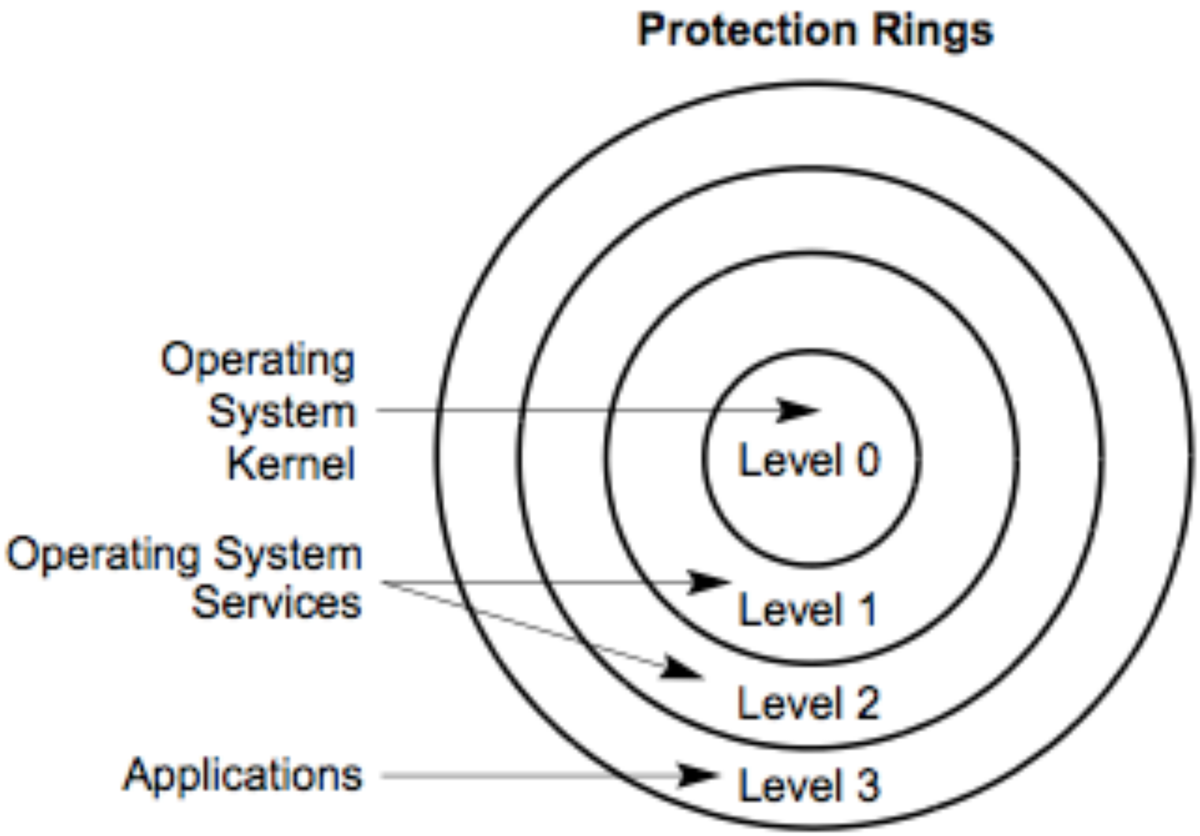


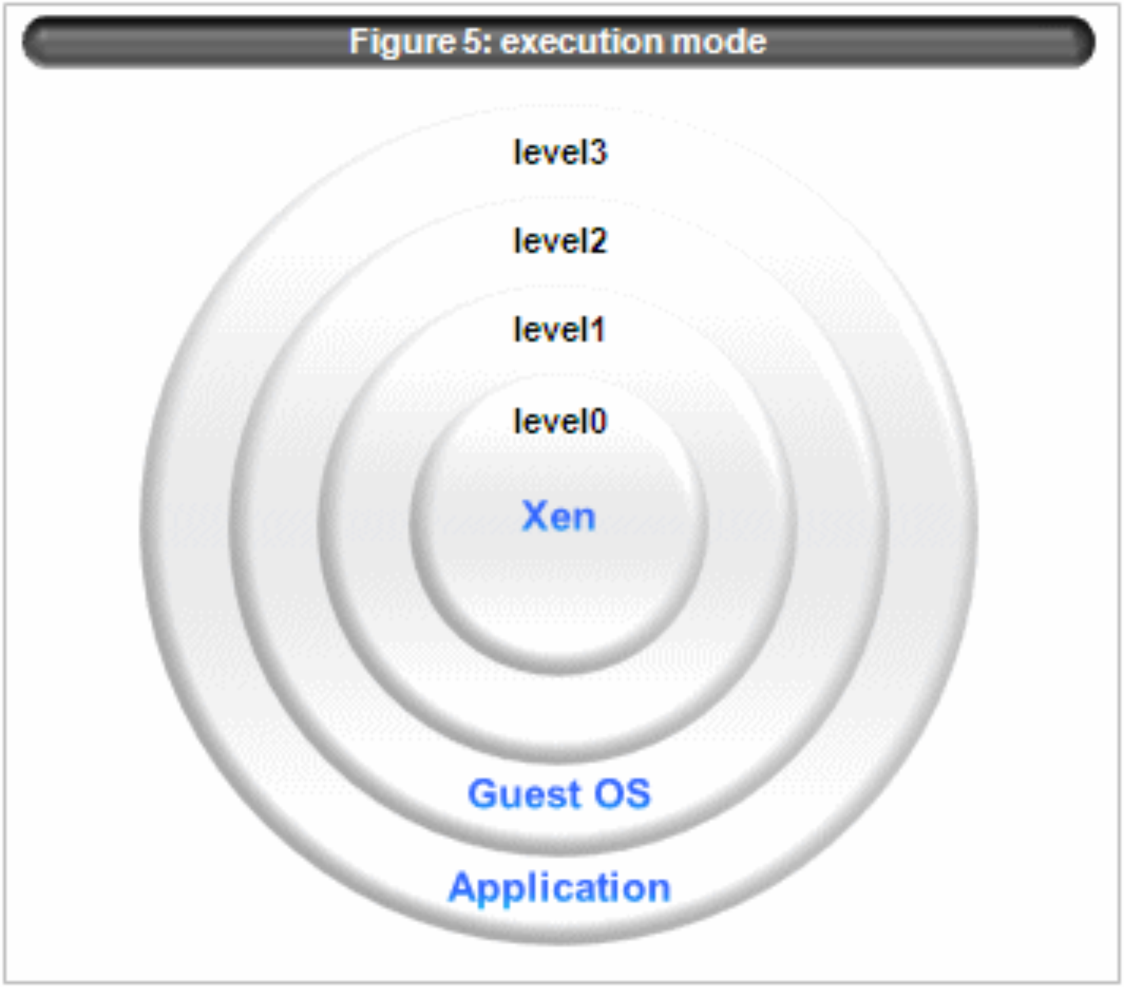
Figure 5-3. Protection Rings

Vol. 3a
Sect. 5.5

Paravirtualized Xen

(requires a modified Guest OS)

(Newest Xen instead uses hw VMX to be more privileged than OS kernel)



http://www.valinux.co.jp/imgs/pict/shot/tech/techlib/eos/xen_ia64_memory/figure5.gif

Segmentation

- “Segmentation provides a mechanism for dividing the processor’s addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**.” (emphasis theirs)

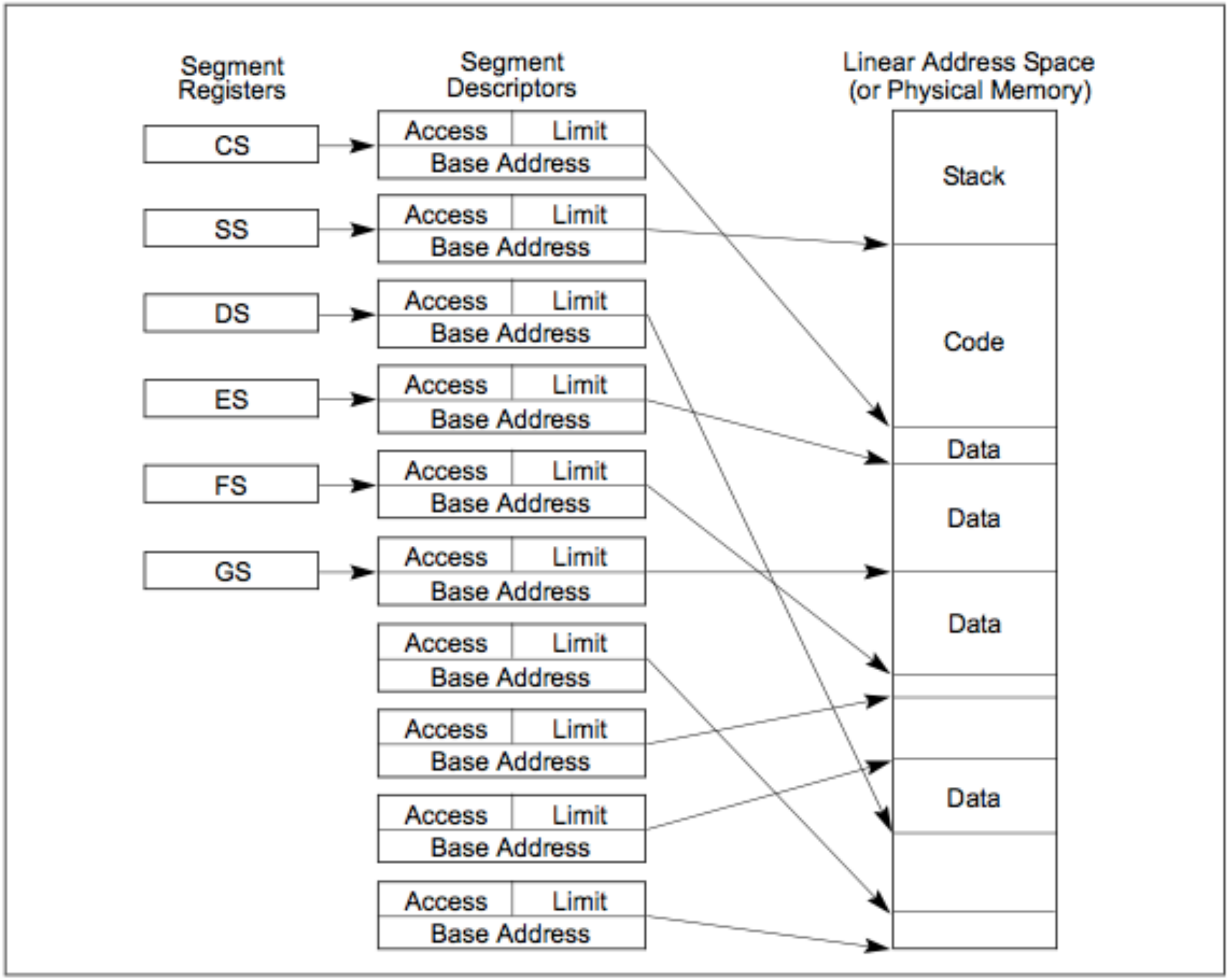


Figure 3-4. Multi-Segment Model

Vol.3a, Sect. 3.1 & 3.2.3

Segment Addressing

- “To locate a byte in a particular segment, a **logical address** (also called a far pointer) must be provided. A logical address consists of a segment selector and an offset.”
- “The physical address space is defined as the range of addresses that the processor can generate on its address bus”
  - Normally the physical address space is based on how much RAM you have installed, up to a maximum of 2^32 (4GB). But there is a mechanism (physical address extentions - PAE) which we will talk about later which allows systems to access a space up to 2^36 (64GB).
  - Basically a hack for people with more than 4GB of RAM but who aren’t using a 64 bit OS.
- Linear address space is a flat 32 bit space
- If paging (talked about later) is disabled, linear address space is mapped 1:1 to physical address space

Vol.3a, Sect. 3.1

Segmentation Restated

- Segmentation is not optional
- Segmentation translates logical addresses to linear addresses automatically in hardware by using table lookups
- Logical address (also called a far pointer) = 16 bit segment selector + 32 bit offset
- If paging (which is talked about later) is disabled, linear addresses map directly to physical addresses

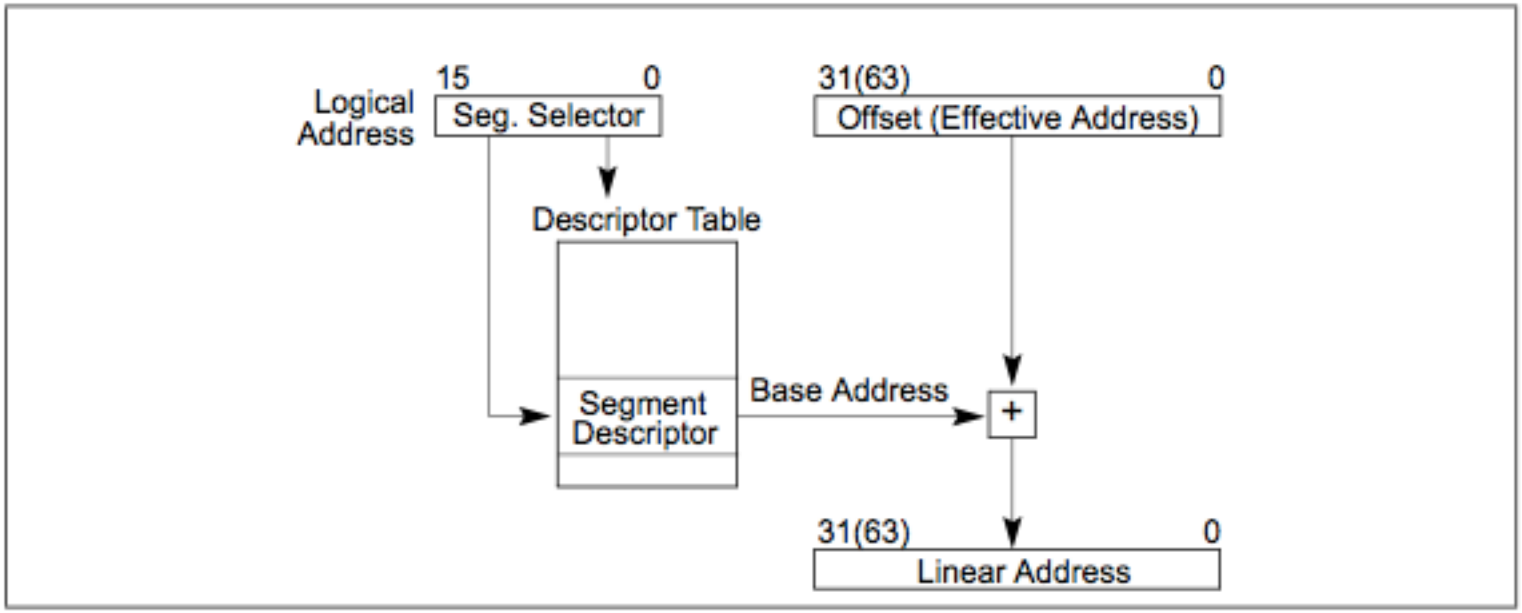
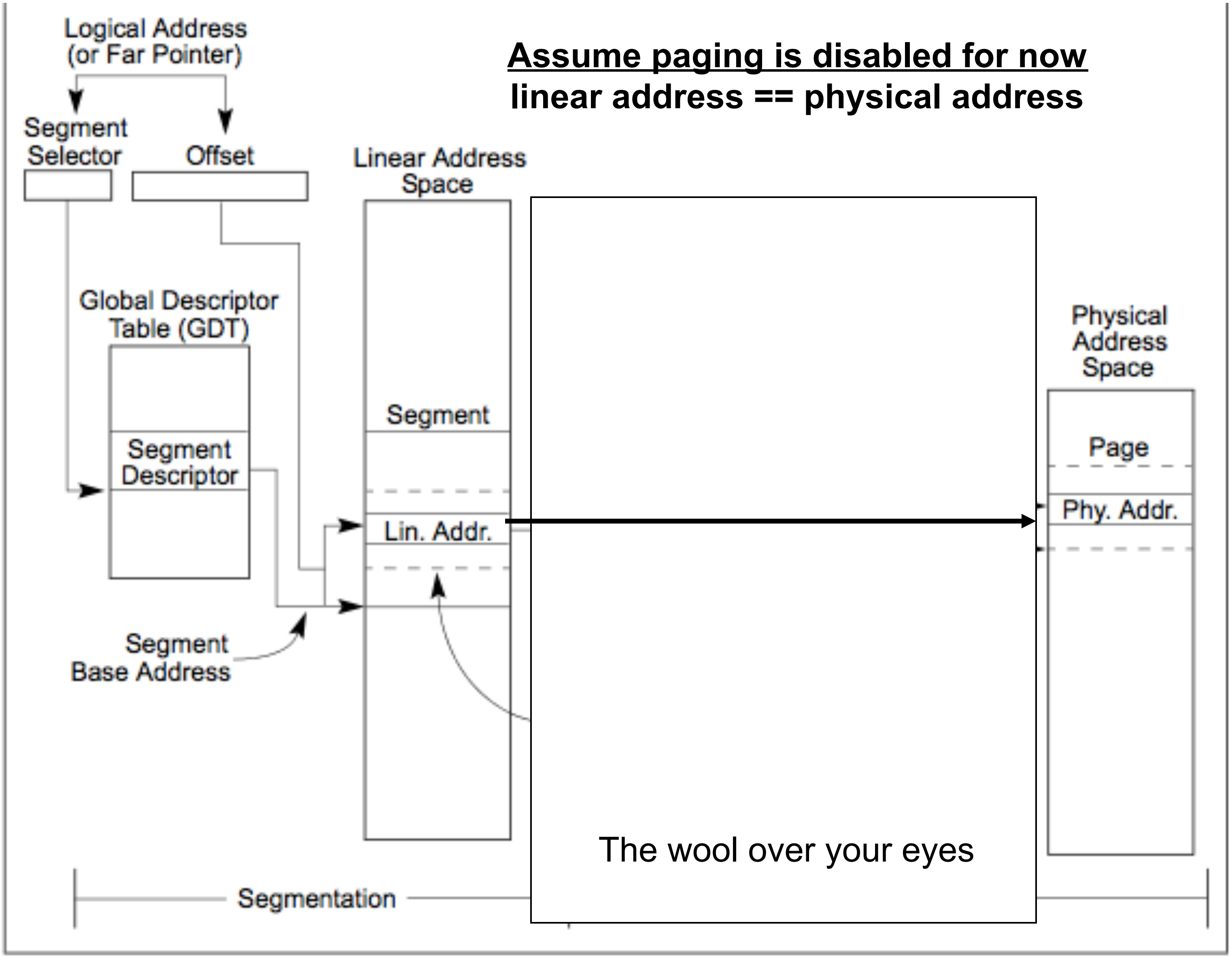


Figure 3-5. Logical Address to Linear Address Translation

The Big Picture



Assume paging is disabled for now

linear address == physical address

The wool over your eyes

Figure 3-1. Segmentation and Paging

Vol.3a, Sect. 3.1

Segment Selectors

- A segment selector is a 16 bit value held in a segment register. It is used to select an index for a segment descriptor from one of two tables.
  - GDT - Global Descriptor Table - for use system-wide
  - LDT - Local Descriptor Table - 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 = 8192 descriptors

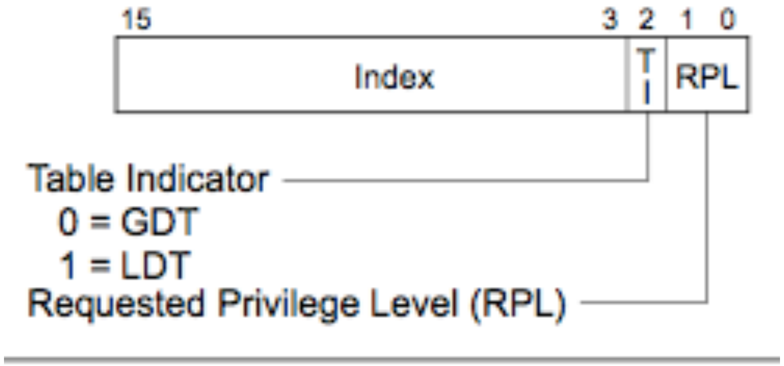


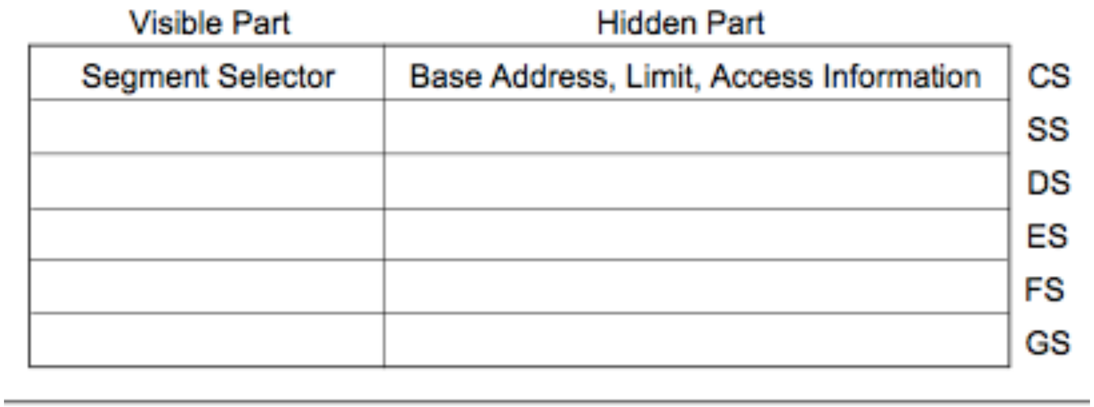
Figure 3-6. Segment Selector

2 bit "privilege level"? Hmmm…getting warm push that onto a mental stack

The Six Segment Registers

(Harbingers of DOOOOOM!!!)

- CS - Code 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)”
- DS - Data Segment
- ES/FS/GS - Extra (usually data) segment registers
- The “hidden part” is like a cache so that segment descriptor info doesn’t have to be looked up each time.

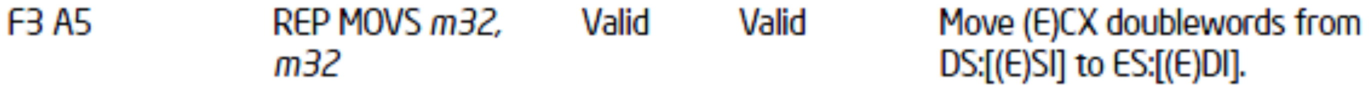


Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

Figure 3-7. Segment Registers

Implicit use of segment registers

- When you’re accessing the stack, you’re implicitly using a logical address that is using the SS (stack segment) register as the segment selector. (I.e. “ESP” == “SS:ESP”)
- When you’re modifying EIP (with jumps, calls, or rets) you’re implicitly using the CS (code segment) register as the segment selector. (“EIP” == “CS:EIP”)
- Even if a disassembler doesn’t show it, the use of segment registers is built into some of your favorite



F3 A5	REP MOVS *m32*, *m32*	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].

Explicit use of segment registers

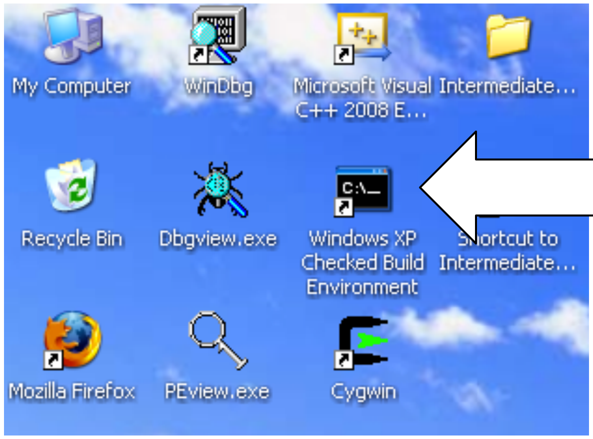
- You can write assembly which explicitly specifies which segment register it wants to use. Just prefix the memory address with a segment register and a colon
- “mov eax, [ebx]” vs “mov eax, fs:[ebx]”
- The assembly just puts a prefix on the instruction to say “When this instruction is asking for memory, it’s actually asking for memory in this segment”. We will talk about segment prefixes along with other instruction prefixes at the end of the class if we have time.
- In this way you’re actually specifying a full logical address/far pointer.

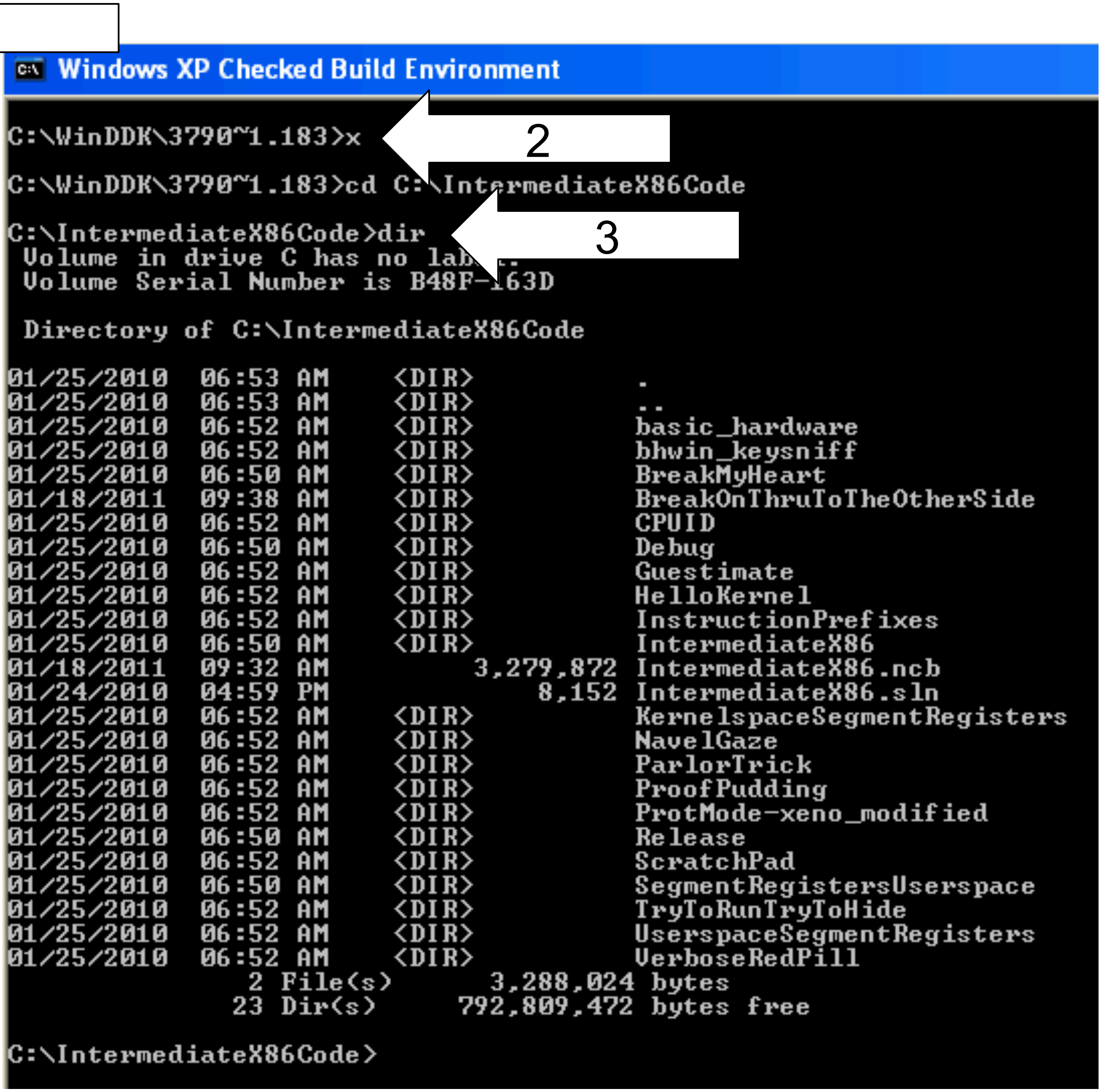
Lab: UserspaceSegmentRegisters.c

- Userspace version of code which reads the segment
- Moves from segment registers to memory, but the manual considers that the same as the other types of moves (but it does describe the special constraints which exist when moving to/ from segment registers)

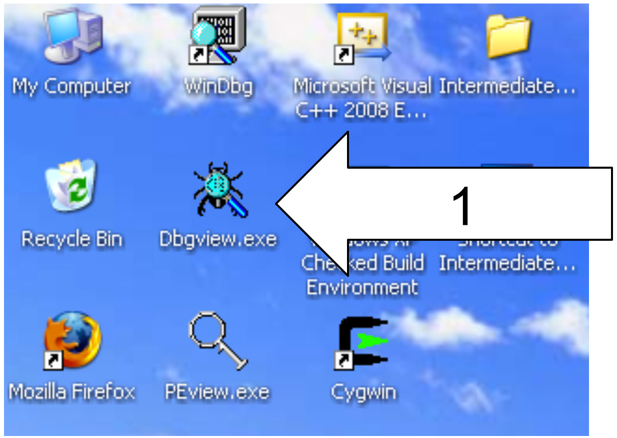
Lab: KernelspaceSegmentRegisters.c

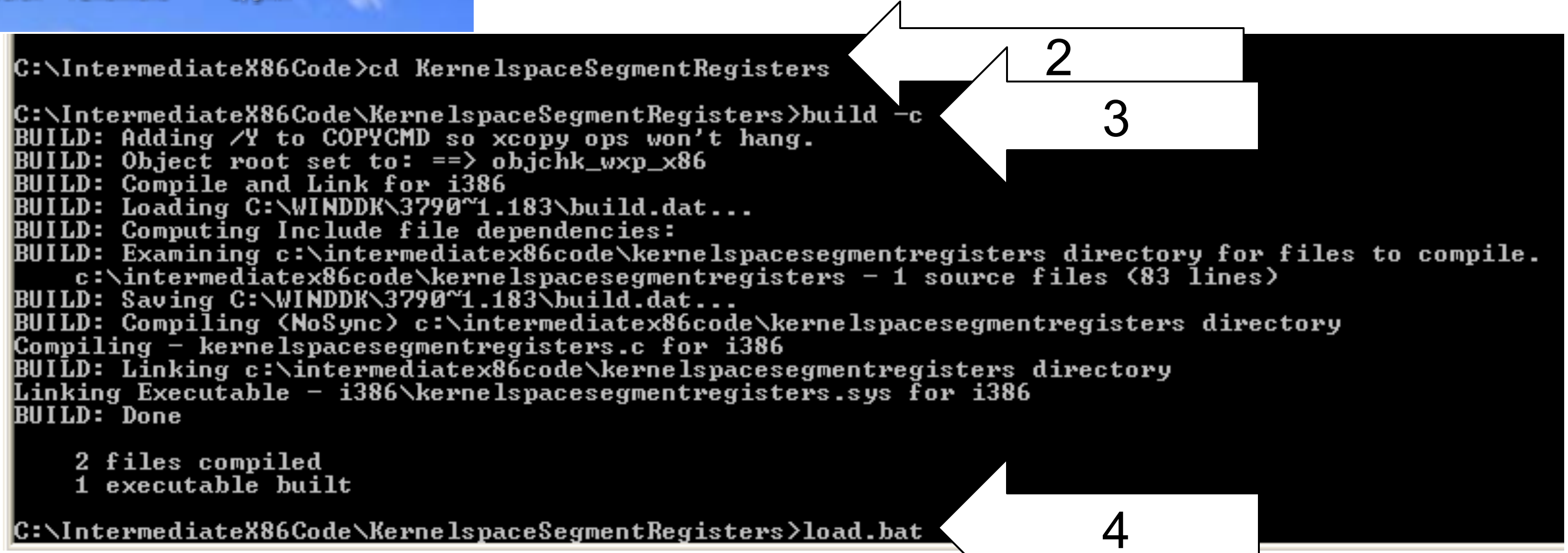
- Kernel version of the same code, implemented as a kernel driver.
- This isn’t a class on windows drivers, so
  - open the “Windows XP Checked Build Environment” link on your desktop, navigate to IntermediateX86Code\KernelspaceSegmentRegisters
  - type “build –c”
  - run the magic “load.bat”
- We use Sysinternals’ DebugView to see the output of the kernel space DbgPrint() statements. (They would also show up if attached to a kernel debugger.)


My Computer
WinDbg
Microsoft Visual C++ 2008 E...
Intermediate...
Recycle Bin
Dbgview.exe
Windows XP Checked Build Environment
Shortcut to Intermediate...
Mozilla Firefox
PEview.exe
Cygwin
1

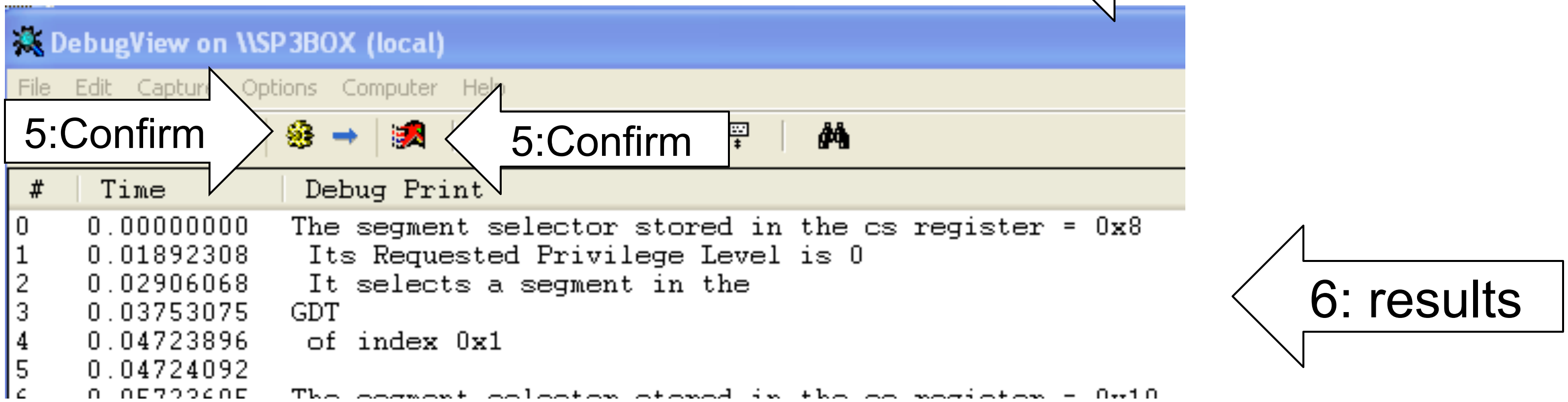


```
C:\WinDDK\3790~1.183>x

C:\WinDDK\3790~1.183>cd C:\IntermediateX86Code

C:\IntermediateX86Code>dir
 Volume in drive C has no label.
 Volume Serial Number is B48F-163D

 Directory of C:\IntermediateX86Code

01/25/2010  06:53 AM    <DIR>          .
01/25/2010  06:53 AM    <DIR>          ..
01/25/2010  06:52 AM    <DIR>          basic_hardware
01/25/2010  06:52 AM    <DIR>          bhwin_keysniff
01/25/2010  06:50 AM    <DIR>          BreakMyHeart
01/18/2011  09:38 AM    <DIR>          BreakOnThruToTheOtherSide
01/25/2010  06:52 AM    <DIR>          CPUID
01/25/2010  06:50 AM    <DIR>          Debug
01/25/2010  06:52 AM    <DIR>          Guestimate
01/25/2010  06:52 AM    <DIR>          HelloKernel
01/25/2010  06:52 AM    <DIR>          InstructionPrefixes
01/25/2010  06:50 AM    <DIR>          IntermediateX86
01/18/2011  09:32 AM         3,279,872 IntermediateX86.ncb
01/24/2010  04:59 PM             8,152 IntermediateX86.sln
01/25/2010  06:52 AM    <DIR>          KernelspaceSegmentRegisters
01/25/2010  06:52 AM    <DIR>          NavelGaze
01/25/2010  06:52 AM    <DIR>          ParlorTrick
01/25/2010  06:52 AM    <DIR>          ProofPudding
01/25/2010  06:52 AM    <DIR>          ProtMode-xeno_modified
01/25/2010  06:50 AM    <DIR>          Release
01/25/2010  06:52 AM    <DIR>          ScratchPad
01/25/2010  06:50 AM    <DIR>          SegmentRegistersUserspace
01/25/2010  06:52 AM    <DIR>          TryToRunTryToHide
01/25/2010  06:52 AM    <DIR>          UserspaceSegmentRegisters
01/25/2010  06:52 AM    <DIR>          VerboseRedPill
               2 File(s)      3,288,024 bytes
              23 Dir(s)     792,809,472 bytes free

C:\IntermediateX86Code>
```





```
C:\IntermediateX86Code>cd KernelspaceSegmentRegisters

C:\IntermediateX86Code\KernelspaceSegmentRegisters>build -c
BUILD: Adding /Y to COPYCMD so xcopy ops won't hang.
BUILD: Object root set to: ==> objchk_wxp_x86
BUILD: Compile and Link for i386
BUILD: Loading C:\WINDDK\3790~1.183\build.dat...
BUILD: Computing Include file dependencies:
BUILD: Examining c:\intermediatex86code\kernelspacesegmentregisters directory for files to compile.
    c:\intermediatex86code\kernelspacesegmentregisters - 1 source files (83 lines)
BUILD: Saving C:\WINDDK\3790~1.183\build.dat...
BUILD: Compiling (NoSync) c:\intermediatex86code\kernelspacesegmentregisters directory
Compiling - kernelspacesegmentregisters.c for i386
BUILD: Linking c:\intermediatex86code\kernelspacesegmentregisters directory
Linking Executable - i386\kernelspacesegmentregisters.sys for i386
BUILD: Done

    2 files compiled
    1 executable built

C:\IntermediateX86Code\KernelspaceSegmentRegisters>load.bat
```



```
#    Time          Debug Print
0    0.00000000    The segment selector stored in the cs register = 0x8
1    0.01892308     Its Requested Privilege Level is 0
2    0.02906068     It selects a segment in the
3    0.03753075    GDT
4    0.04723896     of index 0x1
5    0.04724092
6    0.05723605    The segment selector stored in the cs register = 0x10
```

Results for our WinXP systems

(subject to change on other versions, service pack levels, etc)

UserspaceSegmentRegisters.c

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

KernelspaceSegmentRegisters.c

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

Inferences

- 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 doesn’t change DS or ES when moving between userspace and kernel (they were the exact same values)
- Windows doesn’t use GS

One more time

One of the segment registers (SS/CS/DS/ES/FS/GS)

Address used in some assembly instruction

GDT or LDT

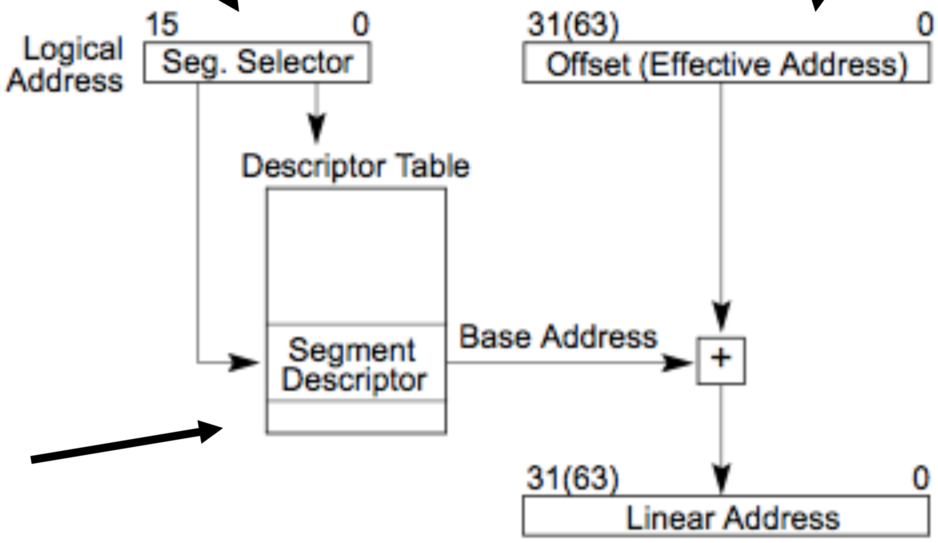


Figure 3-5. Logical Address to Linear Address Translation

GDT & LDT

All entries in these tables are "Segment Descriptor" structures

Special registers point to the base of the tables & specify their size

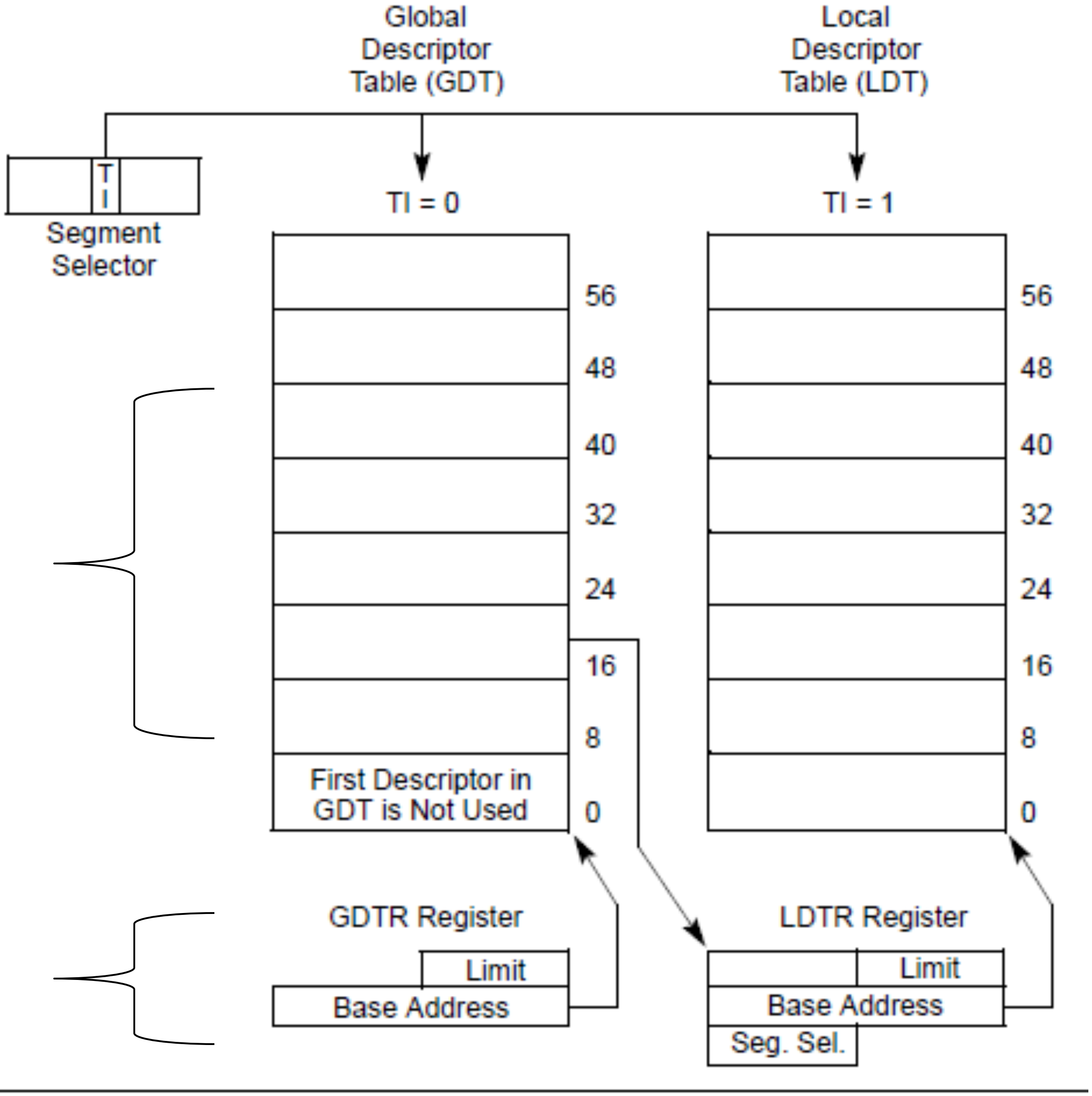
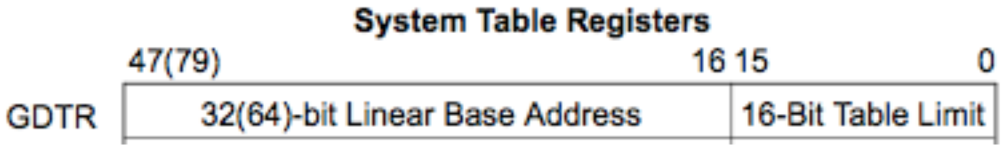


Figure 3-10. Global and Local Descriptor Tables

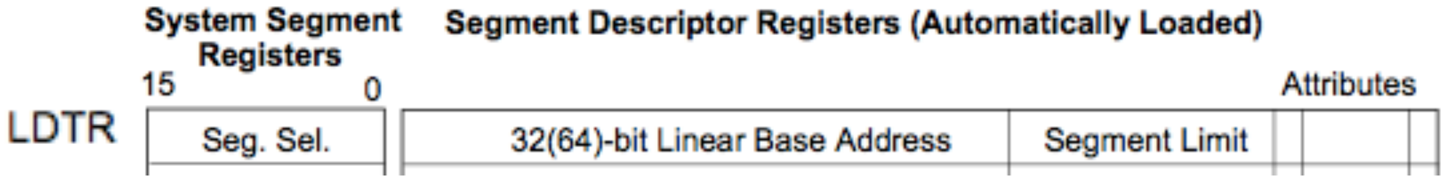
Global Descriptor Table Register (GDTR)



From Vol 3a.
Figure 2-5

- 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
  - LGDT - Load 6 bytes from memory into GDTR
  - SGDT - Store 6 bytes of GDTR to memory

Local Descriptor Table Register (LDTR)

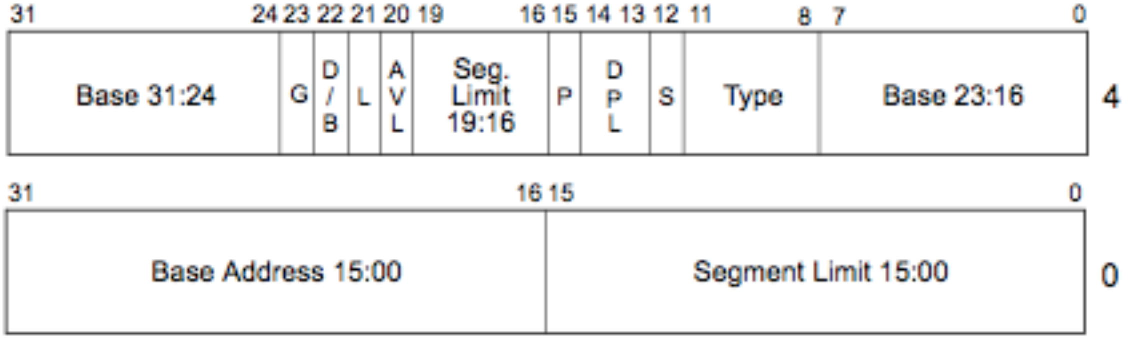

System Segment Registers
Segment Descriptor Registers (Automatically Loaded)
15
0
Attributes
LDTR
Seg. Sel.
32(64)-bit Linear Base Address
Segment Limit

From Vol 3a. Figure 2-5

- 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 LDT.
  – The selector’s Table Indicator (T) bit must be set to 0 to specify that it’s selecting from the GDT, not from itself ;)
- Special instructions used to load a value into the register or store the value out to memory
  – LLDT - Load 16 bit segment selector into LDTR
  – SLDT - Store 16 bit segment selector of LDTR to memory

Segment Descriptors

- "Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level for the, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment)."



31 … 24	23	22	21	20	19 … 16	15	14 13	12	11 … 8	7 … 0	
Base 31:24	G	D / B	L	A V L	Seg. Limit 19:16	P	D P L	S	Type	Base 23:16	4

31 … 16	15 … 0	
Base Address 15:00	Segment Limit 15:00	0

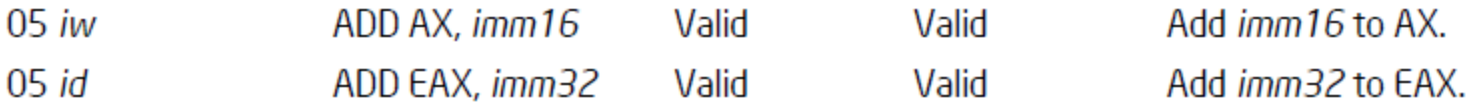
L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

Descriptor Description

- Base (32 bits) - linear address where the segment starts
- Limit (20 bits) - Size of segment (either in bytes or 4kb blocks). End address of segment = base + limit.
- G (Granularity) flag - if 0, interpret limit as size in bytes. If 1, interpret as size in 4kb blocks.
- D/B - Default operation size flag. 0 = 16 bit default, 1 = 32 bit default. ***This*** is what actually controls whether an overloaded opcode is interpreted as dealing with 16 or 32 bit register/memory sizes
- DPL (Descriptor Privilege Level - 2 bits) - Hmm… another interesting field which can range from 0 to 3 “with 0 being the most privileged level”. Push that onto your mental stack with the RPL.

Segmentation and Opcodes



05 *iw*	ADD AX, *imm16*	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	Valid	Valid	Add *imm32* to EAX.

- We can now dig into the simplification I told you in the intro class time about operands being treated as 32 bits just because you’re in protected mode.
- Instead the processor (using the D/B bit in segment descriptors) interprets instructions as referring to address and operand sizes which are the same as the type of code or data segment you’re currently using. So if CS points to a 32 bit segment it uses 32 bit forms, and if it points at a 16 bit segment 16 bit forms.
- A normal OS like Win/Mac/Linux is going to be using 32 bit segments for all normal code.
- But what I said was good enough for before insofar as real mode doesn’t have the ability to use 32 bit segments, and therefore being in protected mode is a prereq to using 32 bit instructions.

Descriptor Description 2

- L Flag - 64 bit segment - ignore
- S (System) Flag - 0 for System segment. 1 for Code or Data segment.
- Type (4 bits) - Whether a segment is code or data, what the permissions are, whether it's been accessed, and some other stuff. See next slide
- P (Present) Flag - 0 for not present. 1 for present. "If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register."

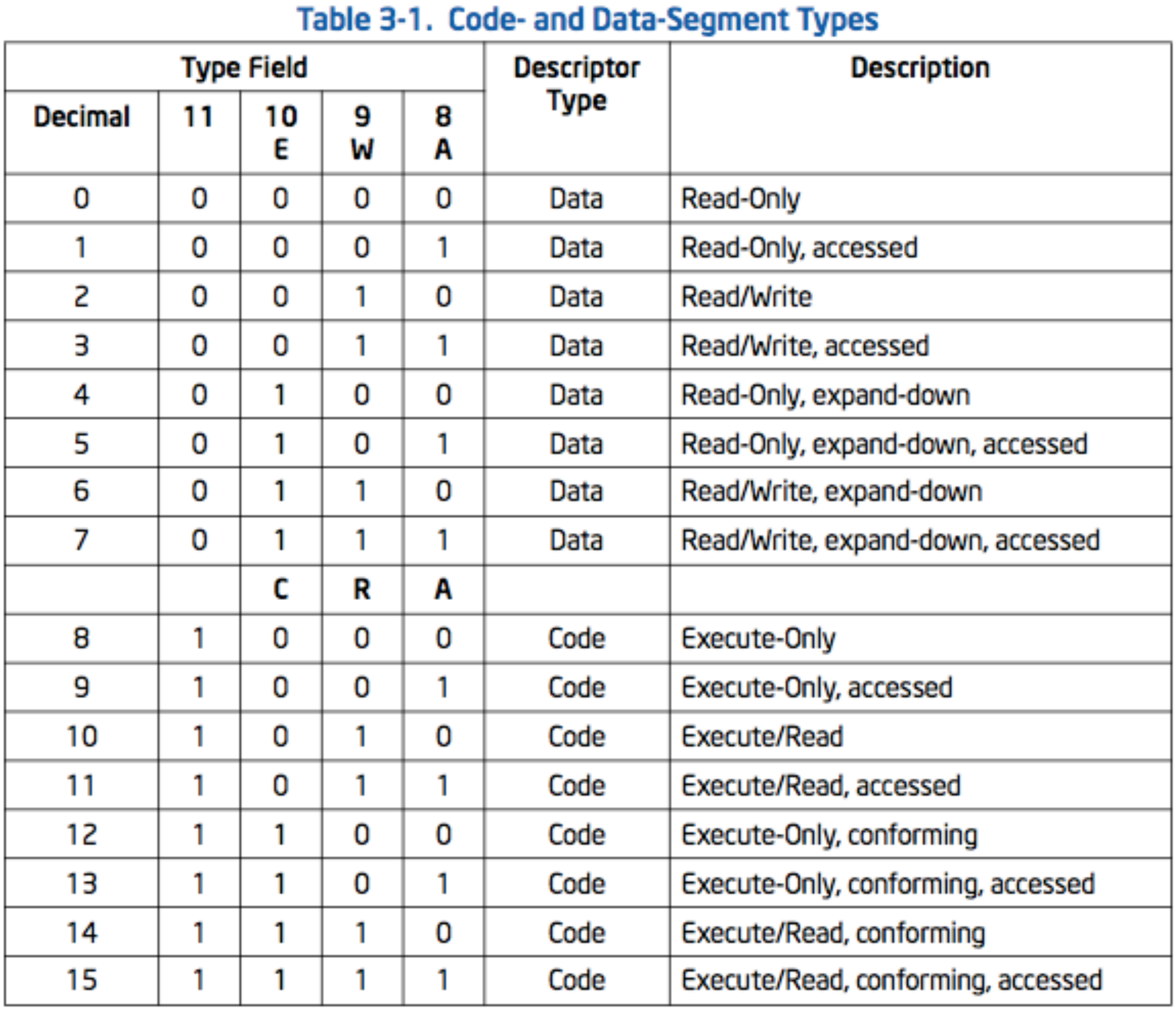


Table 3-1. Code- and Data-Segment Types

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

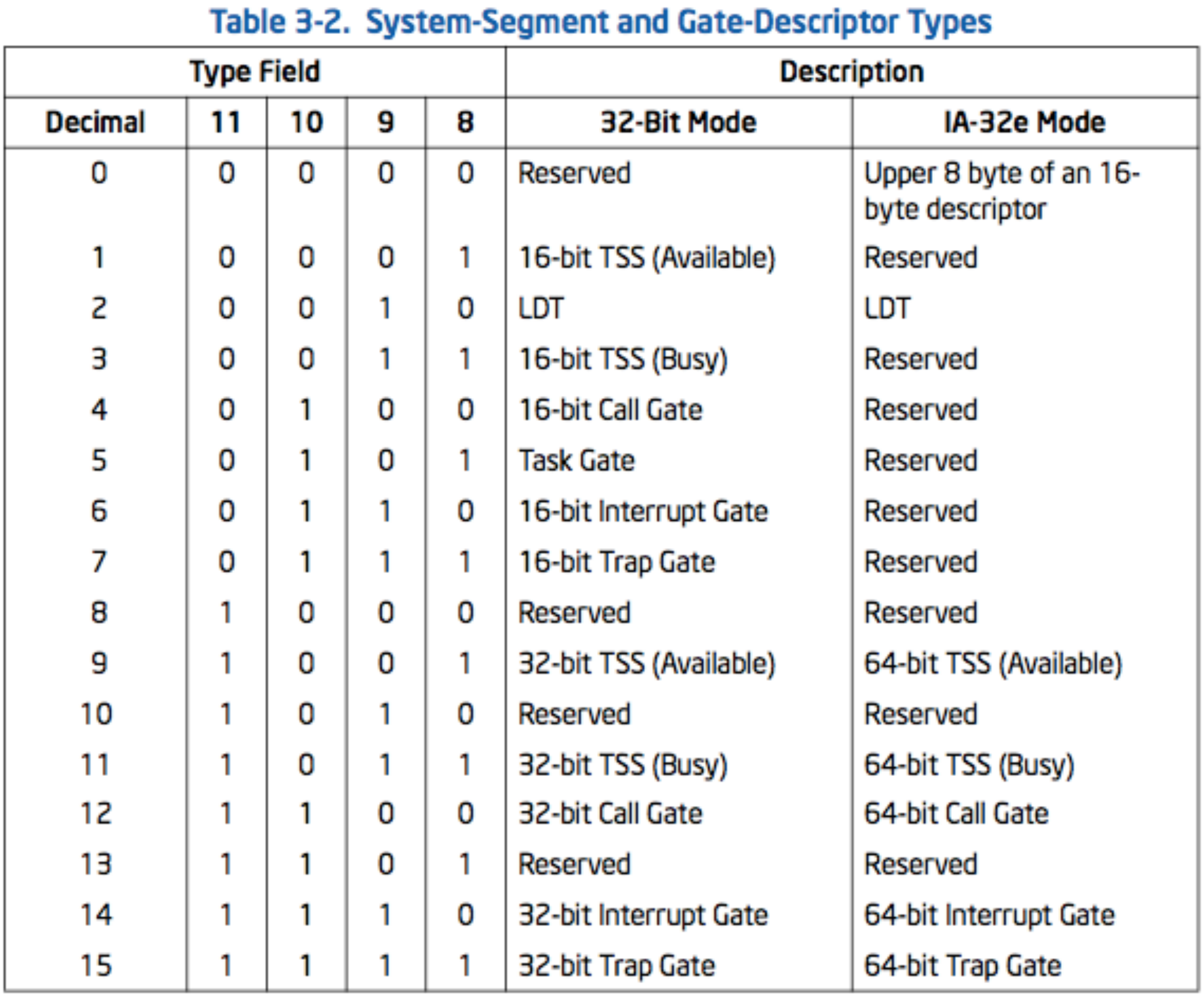


Table 3-2. System-Segment and Gate-Descriptor Types

Type Field					Description	
Decimal	11	10	9	8	32-Bit Mode	IA-32e Mode
0	0	0	0	0	Reserved	Upper 8 byte of an 16-byte descriptor
1	0	0	0	1	16-bit TSS (Available)	Reserved
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-bit TSS (Busy)	Reserved
4	0	1	0	0	16-bit Call Gate	Reserved
5	0	1	0	1	Task Gate	Reserved
6	0	1	1	0	16-bit Interrupt Gate	Reserved
7	0	1	1	1	16-bit Trap Gate	Reserved
8	1	0	0	0	Reserved	Reserved
9	1	0	0	1	32-bit TSS (Available)	64-bit TSS (Available)
10	1	0	1	0	Reserved	Reserved
11	1	0	1	1	32-bit TSS (Busy)	64-bit TSS (Busy)
12	1	1	0	0	32-bit Call Gate	64-bit Call Gate
13	1	1	0	1	Reserved	Reserved
14	1	1	1	0	32-bit Interrupt Gate	64-bit Interrupt Gate
15	1	1	1	1	32-bit Trap Gate	64-bit Trap Gate

Lab:
WinDbg & the !descriptor plugin

- Found a WinDbg plugin for printing out IDT/GDT/LDT entries here
  - http://www.codeguru.com/cpp/w-p/system/devicedriverdevelopment/article.php/c8035/
- Made modifications relevant for the class such as printing out segment types, dumping entire table
- First we need to get cozy with WinDbg

Configuring VMWare for kernel debugging

(tested on VMWare Server 1.x (Windows & Linux), & ESX & vSphere)
(for ESX/vSphere don’t put the \\.\pipe\ in front of names)

This slide is for if you want to test this with your own VMs

Debuggee

*Add virtual serial port

*Use named pipe

- Windows name: \\.\pipe\whatever

- Linux name: /tmp/whatever

*** This end is a server**

(VM Debugger) Other end is a virtual machine

(Host Debugger) Other end is an application

VM Debugger

*Add virtual serial port

*Use named pipe

- Windows name: \\.\pipe\whatever

- Linux name: /tmp/whatever

*** This end is a client**

* Other end is a virtual machine

Host Debugger (Windows only)

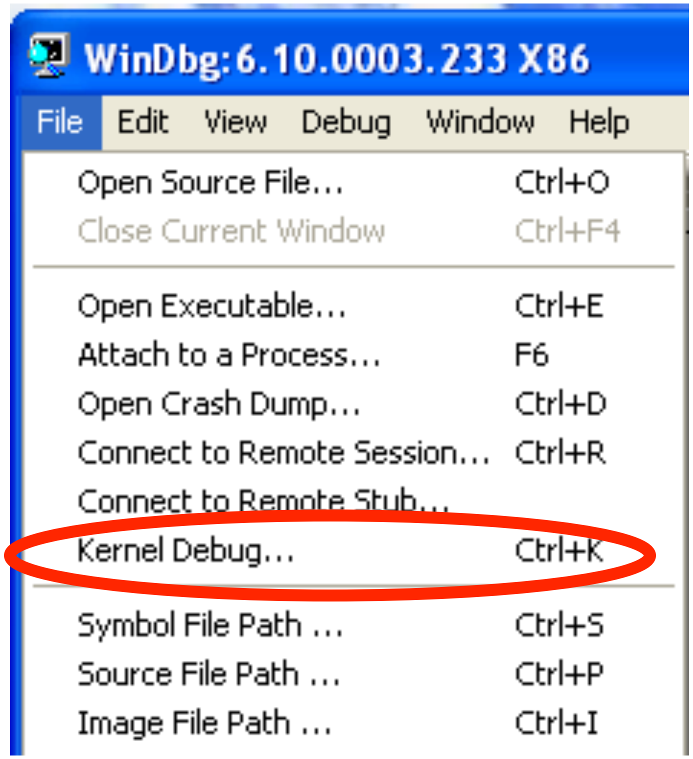
*In WinDbg on the host when you’ve selected kernel debug

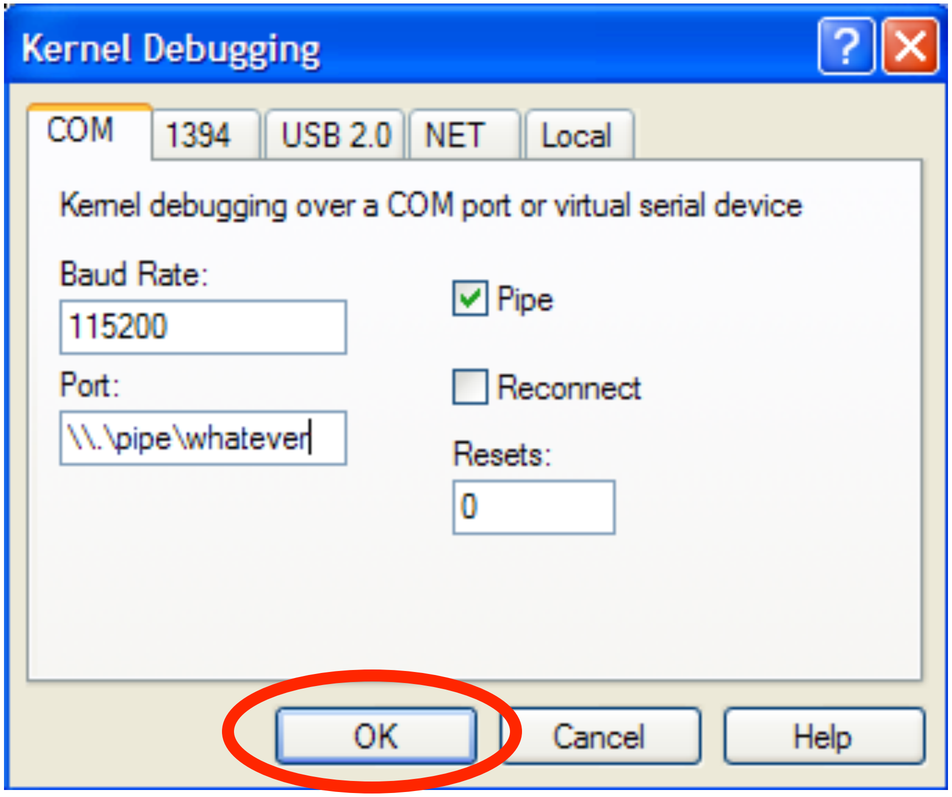
*Under the COM tab

- Port: \\.\pipe\whatever

- Click the “pipe” checkbox

Connecting Debugger


WinDbg:6.10.0003.233 X86
File
Edit
View
Debug
Window
Help
Open Source File... Ctrl+O
Close Current Window Ctrl+F4
Open Executable... Ctrl+E
Attach to a Process... F6
Open Crash Dump... Ctrl+D
Connect to Remote Session... Ctrl+R
Connect to Remote Stub...
Kernel Debug... Ctrl+K
Symbol File Path ... Ctrl+S
Source File Path ... Ctrl+P
Image File Path ... Ctrl+I



Kemel debugging over a COM port or virtual serial device

Baud Rate: 115200

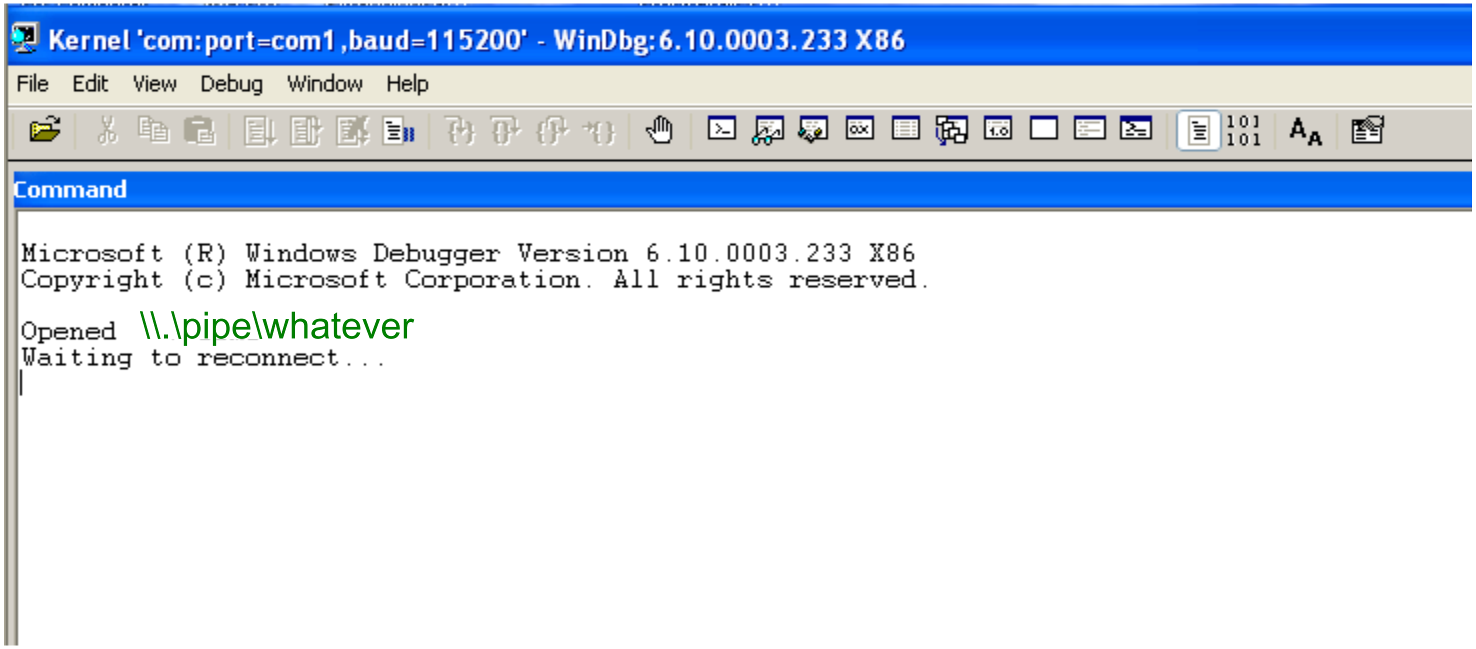
Port: \\.\pipe\whatever

☑ Pipe

□ Reconnect

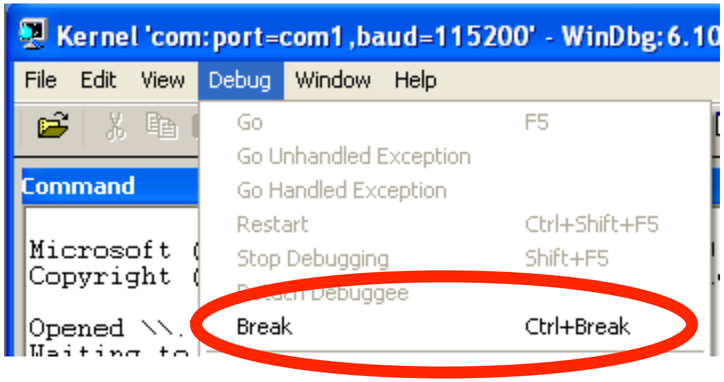
Resets: 0

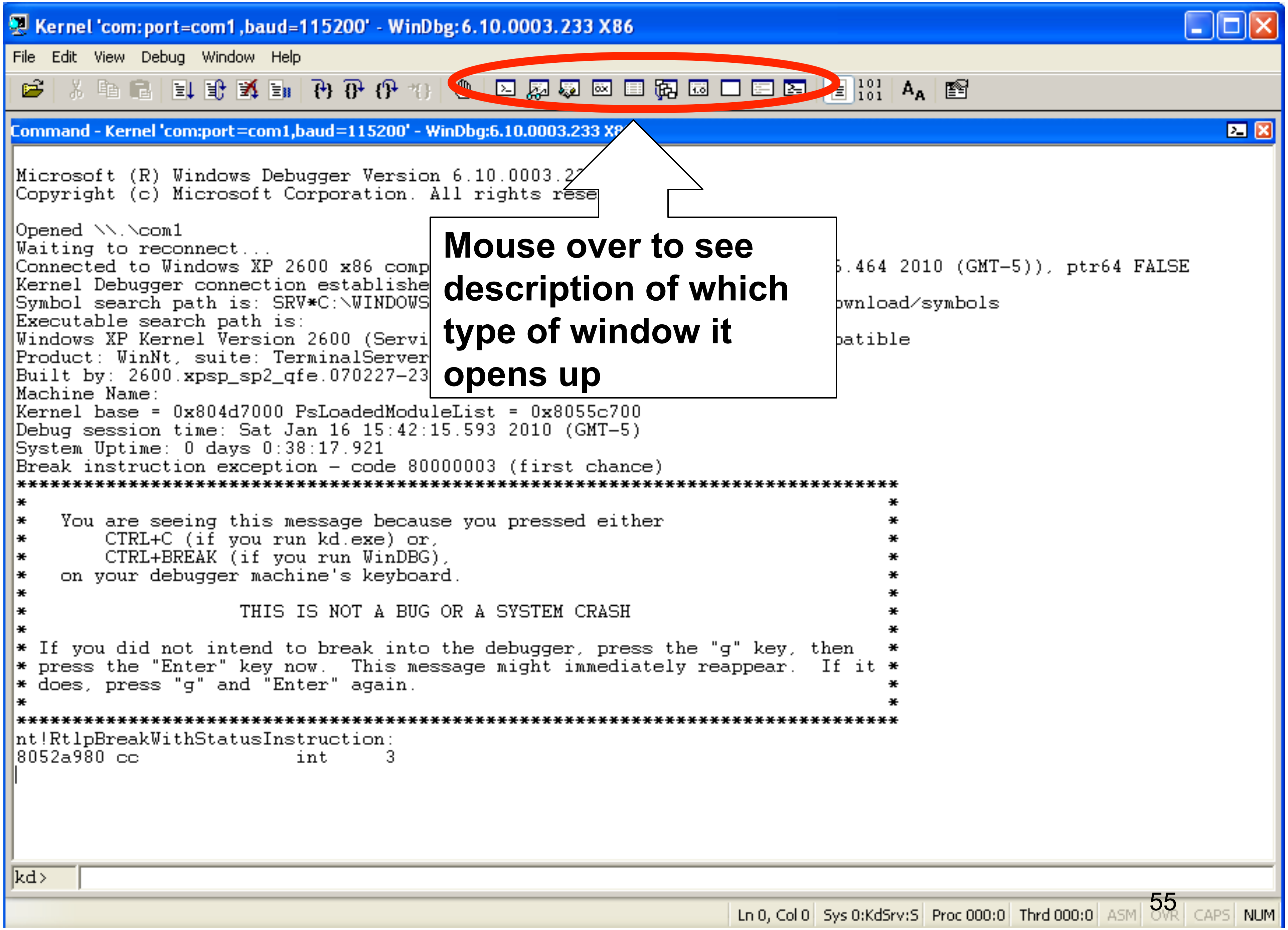
Connecting Debugger 2



Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X86
File
Edit
View
Debug
Window
Help
Command

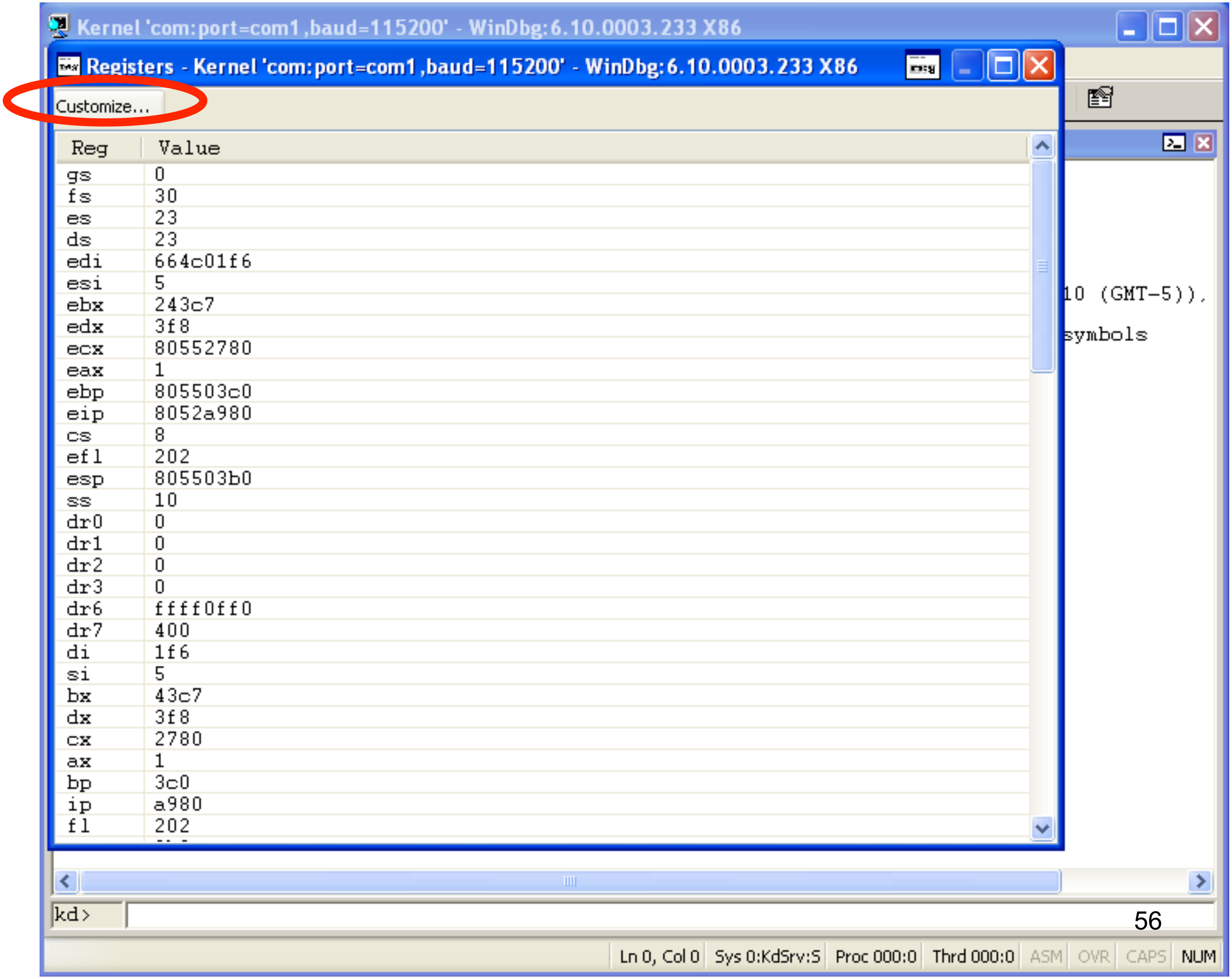
```
Microsoft (R) Windows Debugger Version 6.10.0003.233 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\whatever
Waiting to reconnect...
```




Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X86
File Edit View Debug Window Help
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X8
kd>
Ln 0, Col 0 | Sys 0:KdSrv:S | Proc 000:0 | Thrd 000:0 | ASM | OVR | CAPS | NUM

```
Microsoft (R) Windows Debugger Version 6.10.0003.23
Copyright (c) Microsoft Corporation. All rights rese

Opened \\.\com1
Waiting to reconnect...
Connected to Windows XP 2600 x86 comp                         5.464 2010 (GMT-5)), ptr64 FALSE
Kernel Debugger connection establishe
Symbol search path is: SRV*C:\WINDOWS                         ownload/symbols
Executable search path is:
Windows XP Kernel Version 2600 (Servi                         patible
Product: WinNt, suite: TerminalServer
Built by: 2600.xpsp_sp2_qfe.070227-23
Machine Name:
Kernel base = 0x804d7000 PsLoadedModuleList = 0x8055c700
Debug session time: Sat Jan 16 15:42:15.593 2010 (GMT-5)
System Uptime: 0 days 0:38:17.921
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run kd.exe) or,                                        *
*       CTRL+BREAK (if you run WinDBG),                                       *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!RtlpBreakWithStatusInstruction:
8052a980 cc              int     3
```

Mouse over to see description of which type of window it opens up



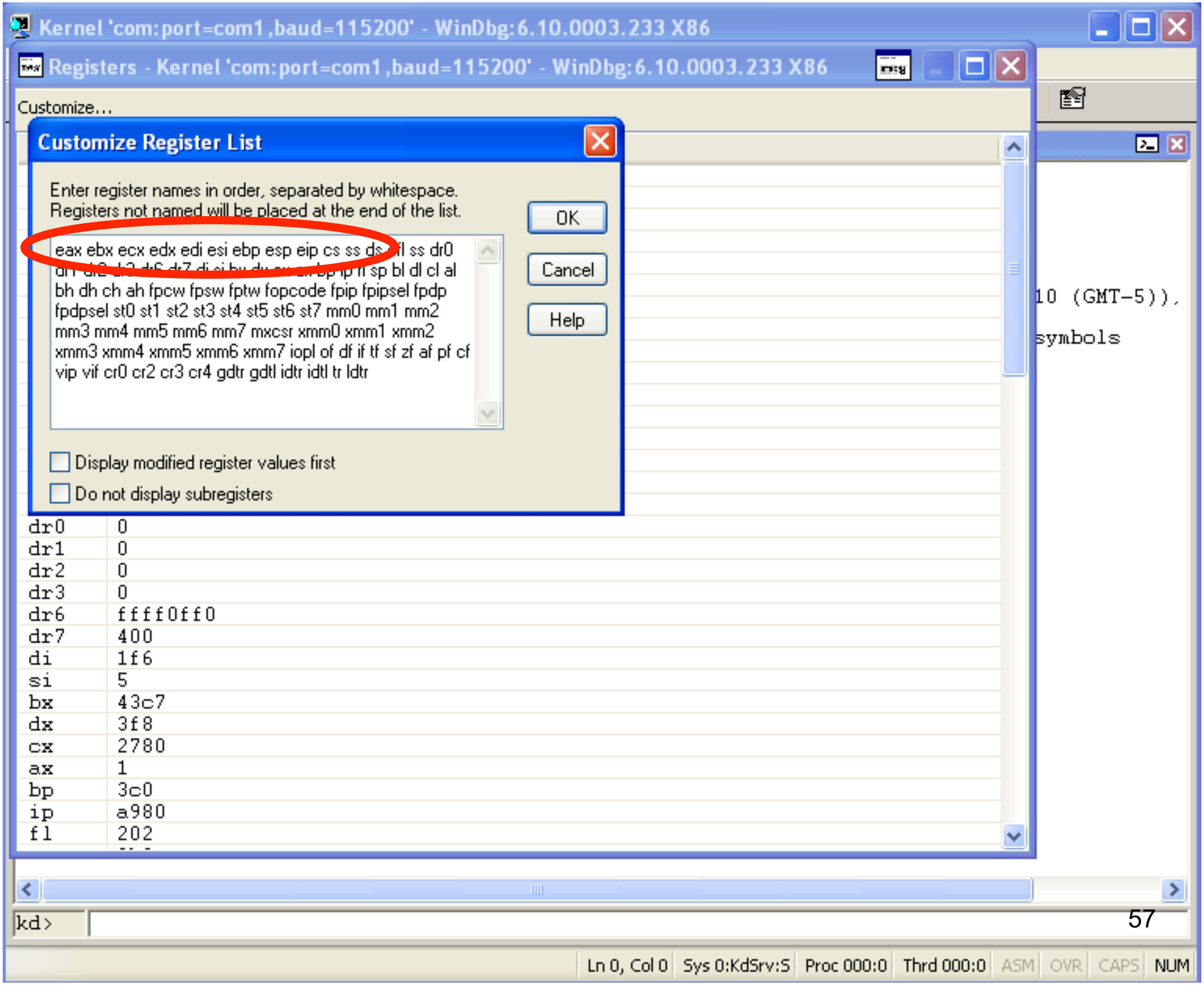
Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X86
Registers - Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X86
Customize...

Reg	Value
gs	0
fs	30
es	23
ds	23
edi	664c01f6
esi	5
ebx	243c7
edx	3f8
ecx	80552780
eax	1
ebp	805503c0
eip	8052a980
cs	8
efl	202
esp	805503b0
ss	10
dr0	0
dr1	0
dr2	0
dr3	0
dr6	ffff0ff0
dr7	400
di	1f6
si	5
bx	43c7
dx	3f8
cx	2780
ax	1
bp	3c0
ip	a980
fl	202

```
10 (GMT-5)),
symbols
```

```
kd>
```

Ln 0, Col 0
Sys 0:KdSrv:S
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM



Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X86
Registers - Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X86
Customize...
Customize Register List
OK
Cancel
Help

Enter register names in order, separated by whitespace.
Registers not named will be placed at the end of the list.

```
eax ebx ecx edx edi esi ebp esp eip cs ss ds efl ss dr0
dr1 dr2 dr3 dr6 dr7 di si bx dx cx ax bp ip fl sp bl dl cl al
bh dh ch ah fpcw fpsw fptw fopcode fpip fpipsel fpdp
fpdpsel st0 st1 st2 st3 st4 st5 st6 st7 mm0 mm1 mm2
mm3 mm4 mm5 mm6 mm7 mxcsr xmm0 xmm1 xmm2
xmm3 xmm4 xmm5 xmm6 xmm7 iopl of df if tf sf zf af pf cf
vip vif cr0 cr2 cr3 cr4 gdtr gdtl idtr idtl tr ldtr
```

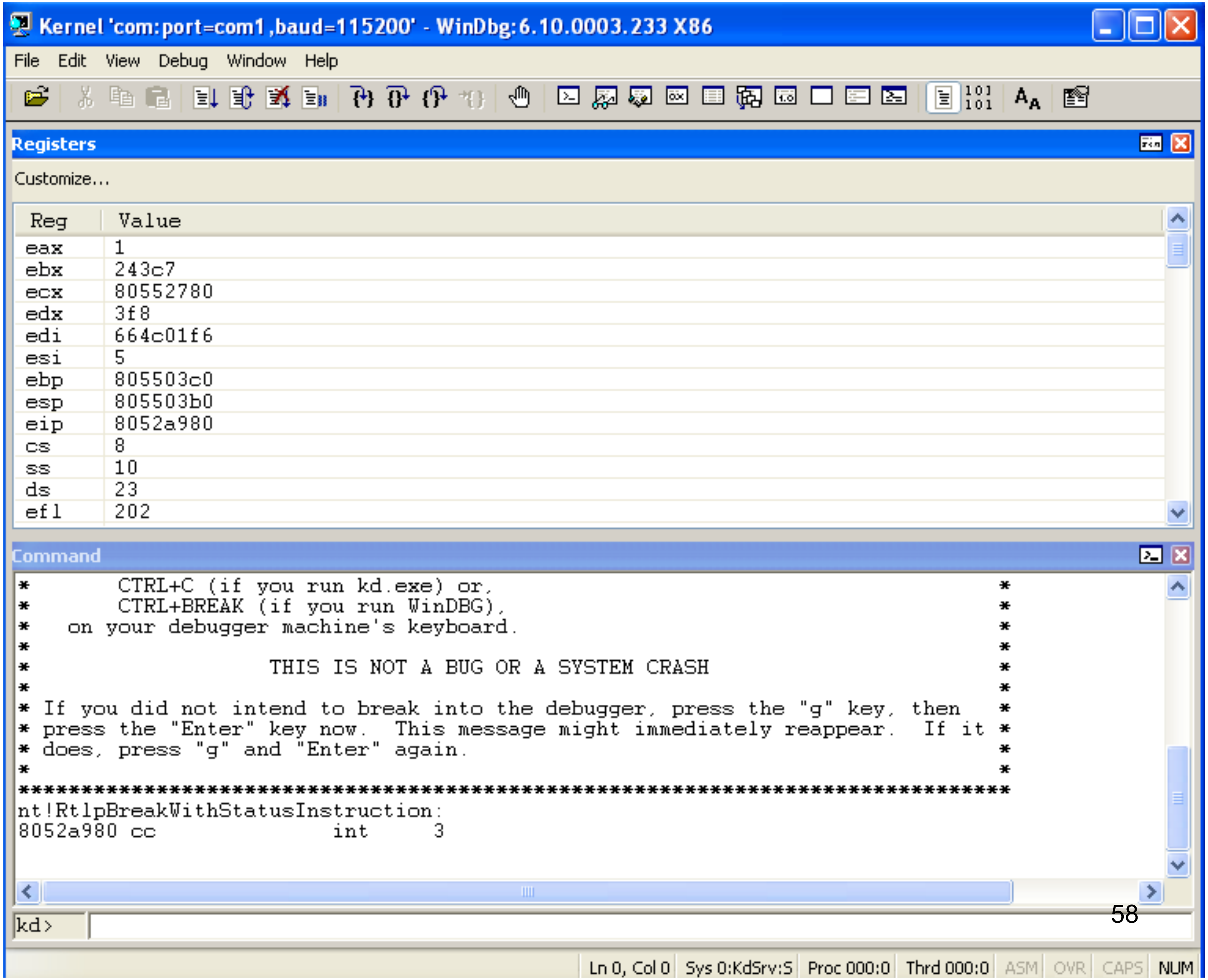
☐ Display modified register values first

☐ Do not display subregisters

```
dr0     0
dr1     0
dr2     0
dr3     0
dr6     ffff0ff0
dr7     400
di      1f6
si      5
bx      43c7
dx      3f8
cx      2780
ax      1
bp      3c0
ip      a980
fl      202
```

```
10 (GMT-5)),
symbols
```

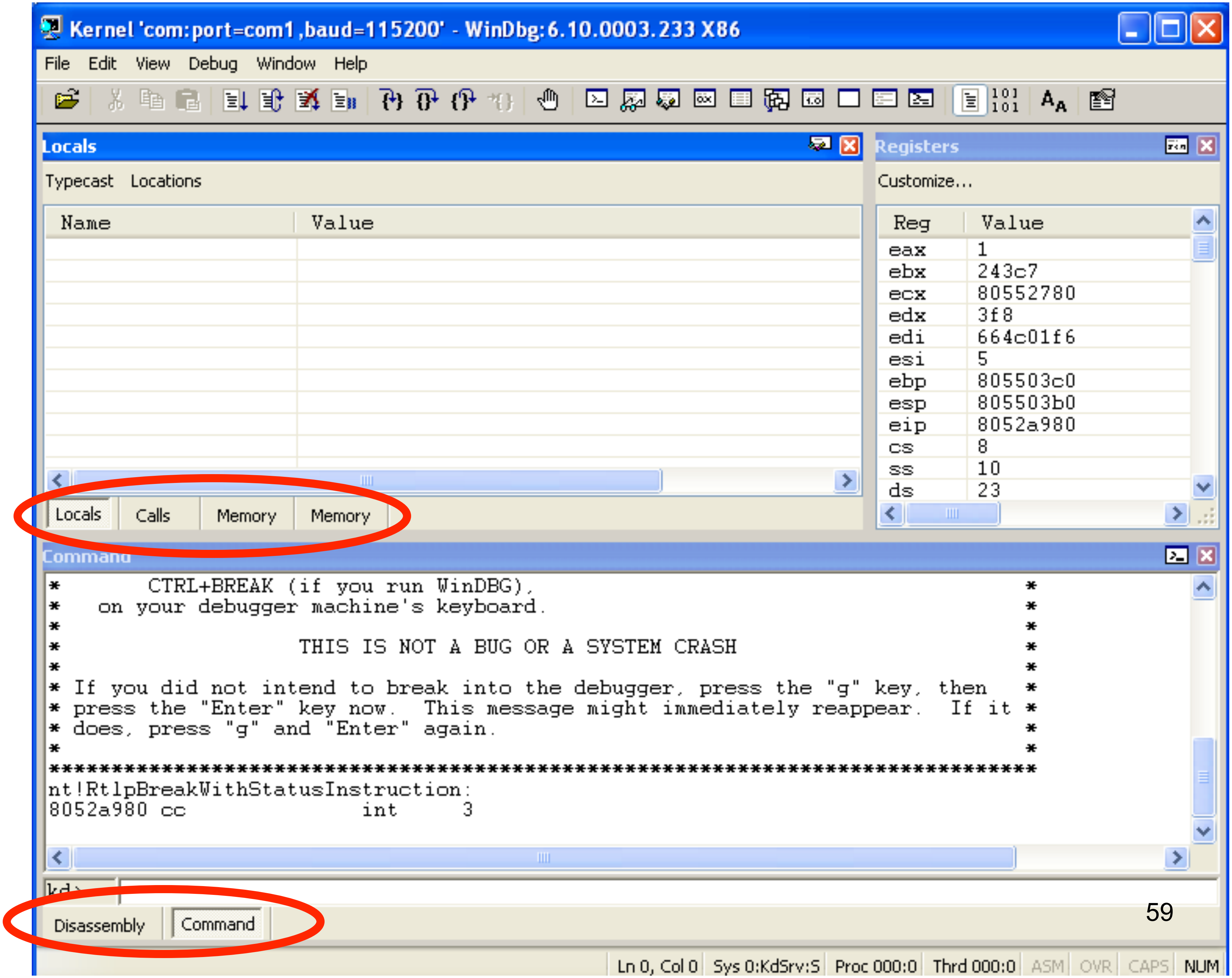
```
kd>
```


Kernel 'com:port=com1,baud=115200' - WinDbg:6.10.0003.233 X86
File Edit View Debug Window Help
Registers
Customize...
Command
Ln 0, Col 0 Sys 0:KdSrv:S Proc 000:0 Thrd 000:0 ASM OVR CAPS NUM

Reg	Value
eax	1
ebx	243c7
ecx	80552780
edx	3f8
edi	664c01f6
esi	5
ebp	805503c0
esp	805503b0
eip	8052a980
cs	8
ss	10
ds	23
efl	202

```
*       CTRL+C (if you run kd.exe) or,                                 *
*       CTRL+BREAK (if you run WinDBG),                                *
*   on your debugger machine's keyboard.                               *
*                                                                      *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                *
*                                                                      *
* If you did not intend to break into the debugger, press the "g" key, then  *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                   *
*                                                                      *
*******************************************************************************
nt!RtlpBreakWithStatusInstruction:
8052a980 cc              int     3
```

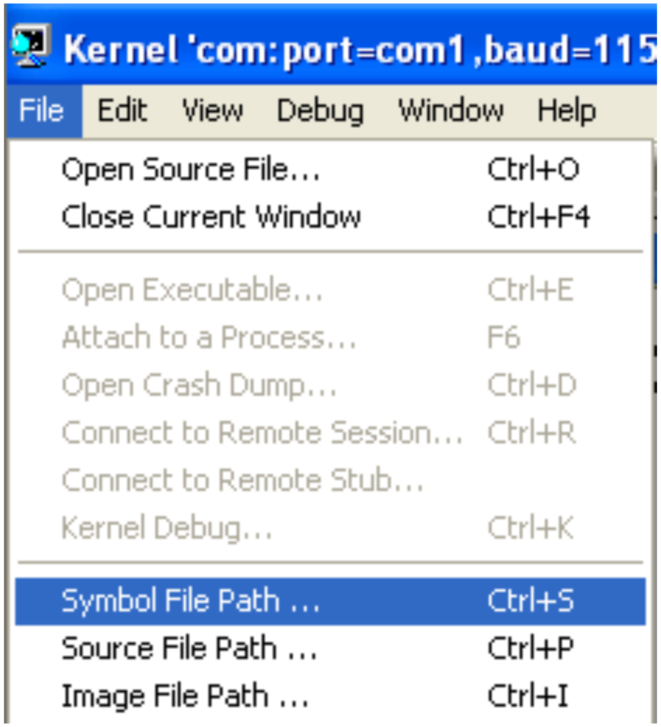
```
kd>
```

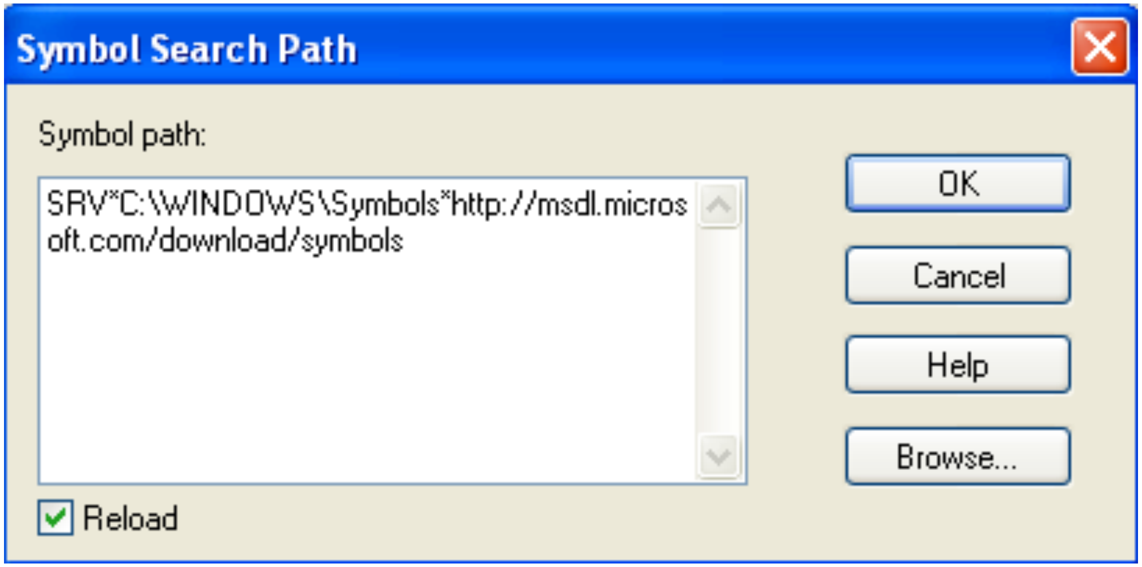


Reg	Value
eax	1
ebx	243c7
ecx	80552780
edx	3f8
edi	664c01f6
esi	5
ebp	805503c0
esp	805503b0
eip	8052a980
cs	8
ss	10
ds	23

```
*       CTRL+BREAK (if you run WinDBG),                                      *
*   on your debugger machine's keyboard.                                     *
*                                                                            *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                      *
*                                                                            *
* If you did not intend to break into the debugger, press the "g" key, then  *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                         *
*                                                                            *
******************************************************************************
nt!RtlpBreakWithStatusInstruction:
8052a980 cc              int     3
```

Getting kernel debug symbols


Kernel 'com:port=com1,baud=115
File
Edit
View
Debug
Window
Help
Open Source File... Ctrl+O
Close Current Window Ctrl+F4
Open Executable... Ctrl+E
Attach to a Process... F6
Open Crash Dump... Ctrl+D
Connect to Remote Session... Ctrl+R
Connect to Remote Stub...
Kernel Debug... Ctrl+K
Symbol File Path ... Ctrl+S
Source File Path ... Ctrl+P
Image File Path ... Ctrl+I

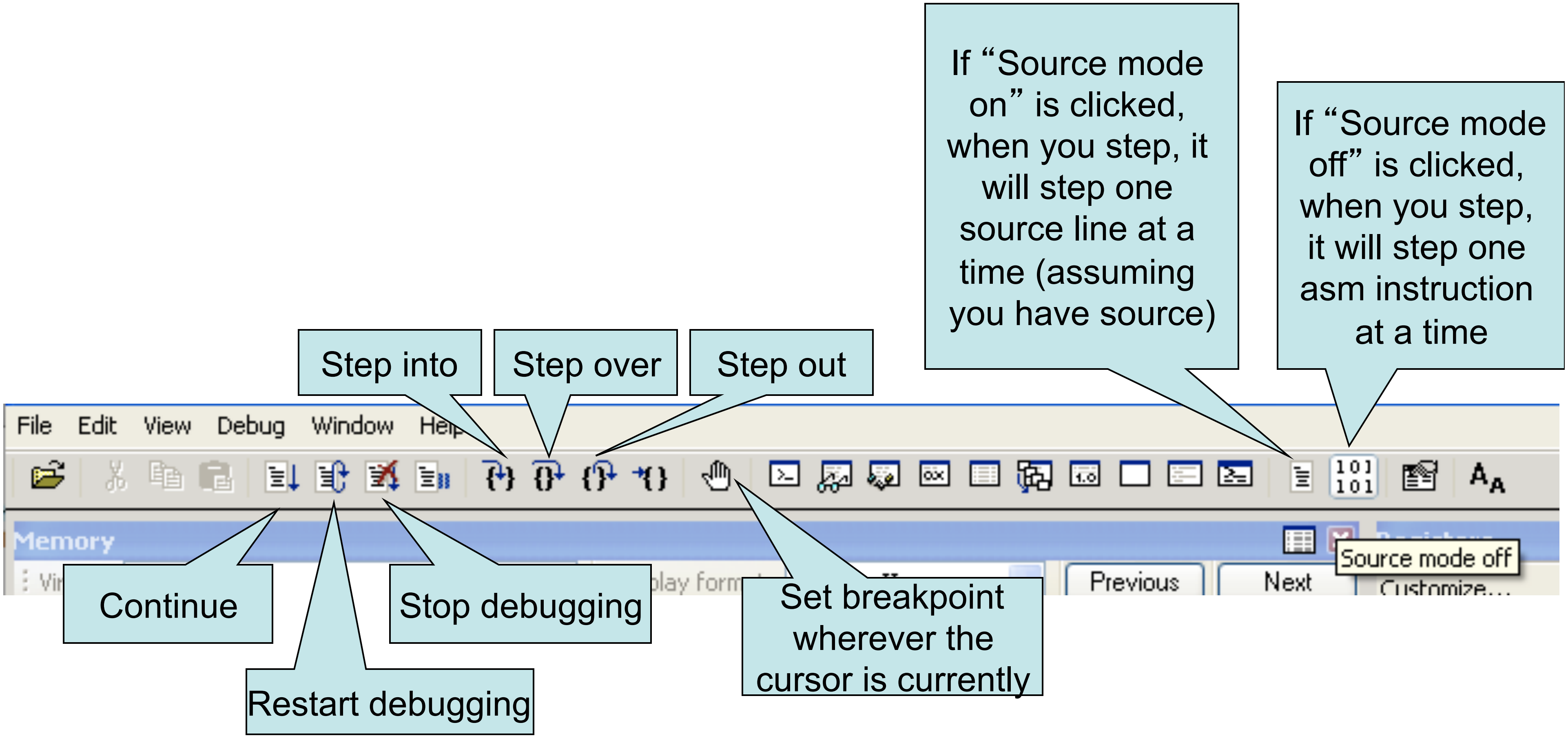


Symbol path:

```
SRV*C:\WINDOWS\Symbols*http://msdl.micros
oft.com/download/symbols
```

☑ Reload

You can also download symbols for offline debugging, in which case you' d just put put the folder you installed them into. Also if you' re working on your own code, you can specify the folder where you have the .pdb files.



If “Source mode on” is clicked, when you step, it will step one source line at a time (assuming you have source)

If “Source mode off” is clicked, when you step, it will step one asm instruction at a time

Step into

Step over

Step out

Continue

Restart debugging

Stop debugging

Set breakpoint wherever the cursor is currently

WinDbg breakpoints

- bp <address> : Set breakpoint
  - Address can be number or human readable input like “main” or “Example1:main”
  - This will be a software (int 3) breakpoint
- bl : Breakpoints list
- bd <bp ID> : Breakpoint disable
  - <bp ID> as given by first column of bl
- be <bp ID> : Breakpoint enable
  - <bp ID> as given by first column of bl
- bc <bp ID> : Breakpoint clear (delete)

WinDbg misc of note

- WinDbg lists the upper 32 bits of the GDTR as “gdtr” but the lower 16 bits as “gdtl”
- Load the plugin in windbg with ".load protmode"
- type "!descriptor" to list the possible commands (also supports tab completion)

Stop!

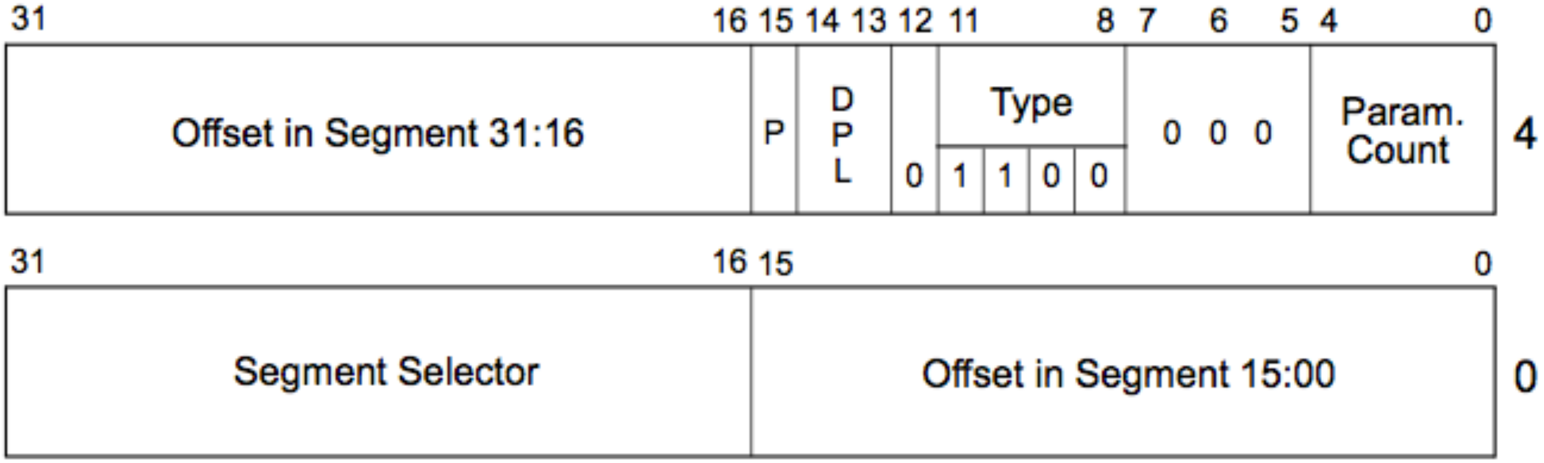
- We’ ve actually overshot what we need to know to discuss the protection rings. They are the interaction between the Requested Privilege Level (RPL), Descriptor Privilege Level (DPL), and introducing the Current Privilege Level (CPL)
- “The CPL is defined as the protection level of the currently executing code segment.” (Sect 2.1.1)
- Privilege rings are automatically enforced by the hardware on certain operations.
  - E.g. if attempting to jump/call/return from one segment into a different segment, the hardware will check the DPL of the target segment and allow the access only if CPL <= DPL
  - E.g. if attempting to use many privileged assembly instructions, the hardware will only allow it if CPL == 0
- Kernel is responsible for setting up userspace in the first place and making sure that when it allows userspace programs to run, their CPL == 3.

You wish

- You may be saying, “Oh, well, if the CPL is just the lower two bits of CS, I’ll just go ahead and load a segment selector which is the same as the current one, but with those bits set to 0, and I will be ring 0!”
- Intel says “Yeah right” - “The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception”

Call Gates

("I'm down with Bill Gates, I call him Money for short. I phone him up at home, and I make him do my tech support!"
- Weird Al, "It's All About the Pentiums")



DPL Descriptor Privilege Level
P Gate Valid

Figure 4-8. Call-Gate Descriptor

- Call gates are basically a way to transfer control from one segment to another segment (possibly at a different privilege ring, possible at a different size in terms of whether it's 16/32 bits.)
- But the key point is you don't want people to be able to call to anywhere in the other segment, you want the interface to be controlled and well-understood. So calling to a call gate brings code to a specific place which the kernel has set up.

Call Gates 2

- The CALL, RET, and JMP x86 instructions have a special form for when they are doing inter-segment control flow transfer (normal call, ret, jmps are intra-segment for reasons which will become clear shortly.)
- Each of them takes a single far pointer as an argument (though in ret's case, it's popping it off the stack).
- A call gate expects as many parameters as specified by the "Param Count" field on the previous slide (max of 16 due to 4 bit field). Parameters are just pushed onto the stack right to left like a normal cdecl/stdcall calling convention.
- Return value from the far call is returned in eax.
- __asm{call fword ptr 0x08:0x12345678};

Surprise! No one uses segmentation directly for memory protection! :D

- On most systems, segmentation is not providing the primary RWX type permissions, they instead rely on paging protections.

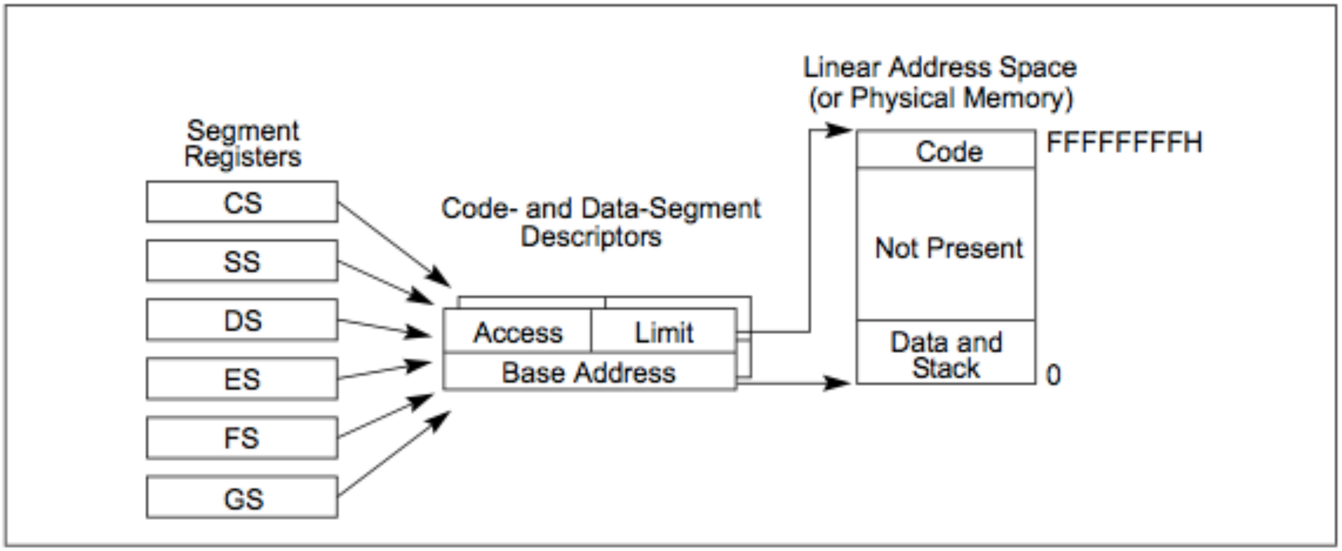


Figure 3-2. Flat Model

Vol.3a, Sect. 3.2.1

Why did we even bother learning it?

- Because it subtly influences aspects of the system.
  - We've already seen that it's the basis for the notions of userspace/kernel separation (which includes the enforcement of limiting access to privileged instructions), but it also influences most of the topics we will be covering in this class
- On 32 bit systems, the GDT is required, and at least flat segmentation must be set up.
  - Segmentation support mostly removed in x86-64, but it's so embedded in the architecture, and chip makers so prize backward compatibility. that it will continue to influence design for a while.
- It's just good to understand how stuff works as accurately as possible :)

Who uses segmentation for memory protection?

- Paravirtualized Xen uses it to protect the hypervisor from the OS. Jives with the notion of putting the OS in ring 2 per the picture we saw early on. (http://www.cs.uiuc.edu/class/sp06/cs523/lectures/05/523-5-xen.pdf)
- Google Native Client (NaCl)!?
  - Thanks to Murad Khan for pointing this out
  - System for sandboxing browser plugins, which aims to allow the plugin to be custom compiled to x86 code, and then it only executes x86 instructions natively if they meet criteria which ensures NaCl can analyze them to ensure safety
  - Segmentation is used to provide a “data sandbox” which the code cannot access outside of
  - Combination of a lot of other academic work, but segmentation is basically just an optimization to prevent having to intercept reads/writes looking for things targeted to the outside of the sandbox (can just check at the analysis stage)
  - http://nativeclient.googlecode.com/svn/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf

Misc usage of segments

- On Windows as a RE you will see access to the FS segment register frequently (e.g. mov eax, fs:[0]). Windows manages the FS register to have it always pointing at the base of the Thread Environment Block (TEB) which is used to store some per-thread information.
- In the Intro x86 class I noticed in a Linux/GCC example which had stack cookies enabled, it seemed to be pulling the random cookie from some structure based at GS. But I don’t know what it was, so if anyone wants to figure that out and LMK, I’d be much obliged.

Why isn’t segmentation widely used?

- Answer: I dunno. Ideas? LMK
- Speculation: It’s one of the standard security tradeoffs, security vs. performance. How much overhead does it add? Probably not much, but since many OS design decisions were made in the unfortunate time after COTS overtook MLS (Multi-level secure) OSes, the designers probably sided with performance.
- Speculation 2: With only 6 segment registers, you can’t have a 1:1 mapping of segments to binary memory sections, because some binaries have > 6 sections, so then the questions becomes, what are the most appropriate places to apply segmentation (other than code vs data), how frequently do you want to switch, or what if there’s no compiler support?
- Speculation 3: Wikipedia says segmentation “make[s] programming and compilers design difficult because the use of near and far pointers affect performance” Citation needed ;), but I can see how it would make compilers more difficult. How does the compiler know what your OS is doing with segmentation?
- Speculation 4: All of the above. Whatever it is, support for

Misc: NoPill & ScoopyNG, using LDT/GDT to detect that we’re in a VM

- NoPill
  - http://www.offensivecomputing.net/files/active/0/vm.pdf
  - Redpill equivalent which profiles the LDTR rather than IDTR (talked about later)
  - Done because RedPill signature can have false positives outside of VMs.
- ScoopyNG
  - Does 7 checks including LDT/GDT as well as other things like the VMWare I/O channel
  - http://www.trapkit.de/research/vmm/scoopyng/index.html
  - Source code is in the zip file

Misc Instructions Picked Up Along The Way

- CPUID – Identify CPU features
- PUSHFD/POPFD – Push/Pop EFLAGS
- SGDT/LGDT – Store/Load GDTR
- SLDT/LLDT – Store/Load LDTR