
# CS 410/510

## Languages & Low-Level Programming

Mark P Jones
Portland State University

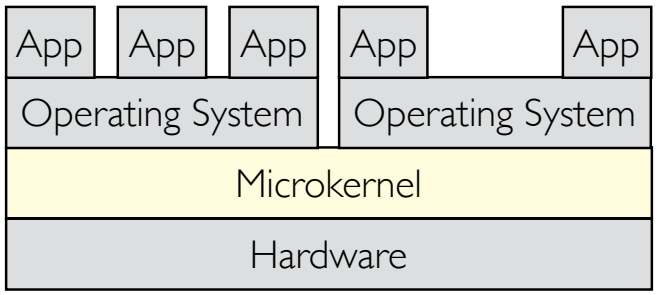
Spring 2016

Week 3: Segmentation, Protected Mode, Interrupts, and Exceptions



# General theme for the next two weeks

- In a complex system …




- Question: how can we protect individual programs from interference with themselves, or with one another, either directly or by subverting lower layers?
- General approach: leverage programmable hardware features!

# Diagrams and Code

- There are a lot of diagrams on these slides
  - Many of these are taken directly from the "Intel® 64 and IA-32 Architectures Software Developer's Manual", particularly Volume 3
  - There is a link to the full pdf file in the Reference section on D2L
- There is also a lot of code on these slides
- Remember that you can study these more carefully later if you need to!



# Taking stock: Code on D2L

| | | |
|---|---|---|
| vram | video RAM simulation | vram.tar.gz |
| hello | boot and say hello on bare metal, via GRUB | hello.tar.gz |
| simpleio | a simple library for video RAM I/O | baremetal.tar.gz |
| bootinfo | display basic boot information from GRUB | |
| mimg | memory image bootloader & make tool | |
| example-mimg | display basic boot information from mimgload | |
| example-gdt | basic demo using protected mode segments (via a Global Descriptor Table) | prot.tar.gz |
| example-idt | context switching to user mode (via an Interrupt Descriptor Table) | |

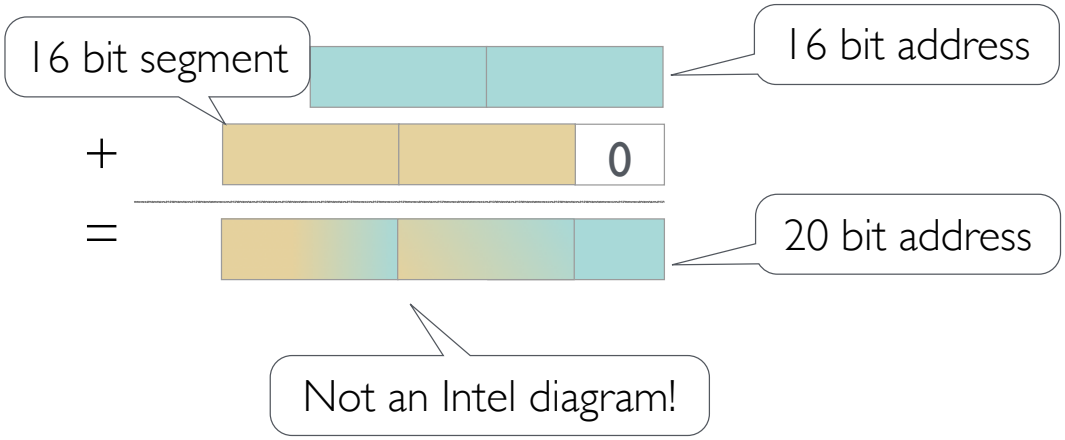
# Segmentation

(or: where do "seg faults" come from?)



## Breaking the 64KB barrier …

- The 8086 and 8088 CPUs in the original IBM PCs were 16 bit processors: in principle, they could only address 64KB
- Intel used **segmentation** to increase the amount of addressable memory from 64KB to 1MB:



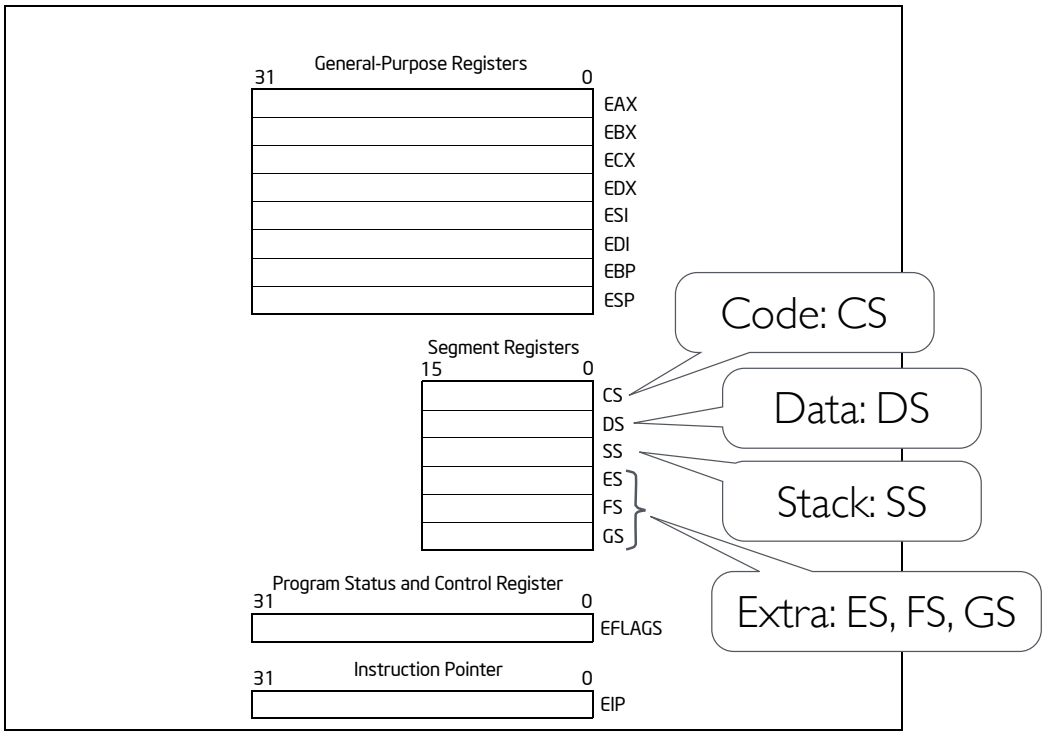



**Figure 3-4. General System and Application Programming Registers**

An Intel diagram!



# How are segments chosen

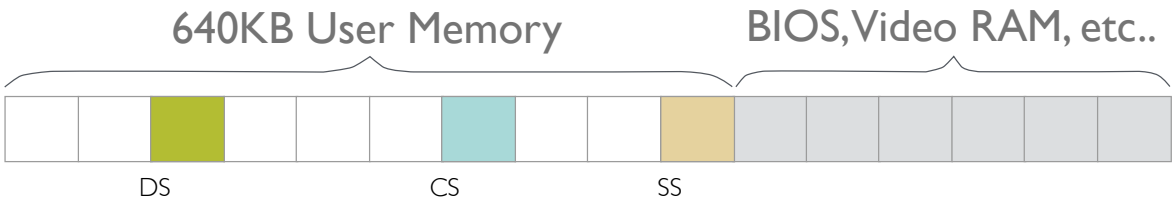
- The default choice of segment register is determined by the specific kind of address that is being used:

**Table 3-5. Default Segment Selection Rules**

| Reference Type | Register Used | Segment Used | Default Selection Rule |
|---|---|---|---|
| Instructions | CS | Code Segment | All instruction fetches. |
| Stack | SS | Stack Segment | All stack pushes and pops.<br>Any memory reference which uses the ESP or EBP register as a base register. |
| Local Data | DS | Data Segment | All data references, except when relative to stack or string destination. |
| Destination Strings | ES | Data Segment pointed to with the ES register | Destination of string instructions. |

- If a different segment register is required, a single byte "segment prefix" can be attached to the start of the instruction

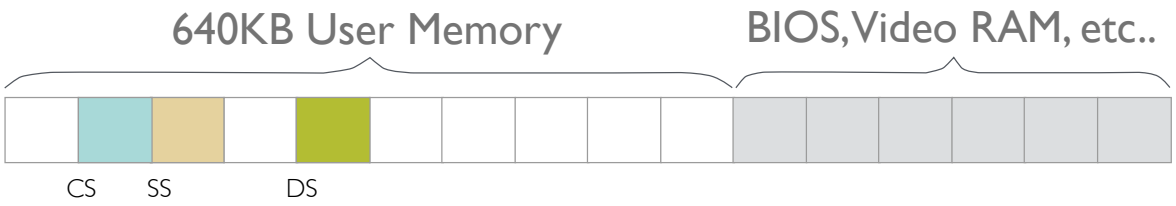
# Back to breaking the 64KB barrier …




- Programs can be organized to use multiple segments:
- For example:
  - One segment for the stack
  - One segment for code
  - One segment for data
- We can relocate these segments to different physical addresses, just by adjusting the segment registers



# Back to breaking the 64KB barrier …



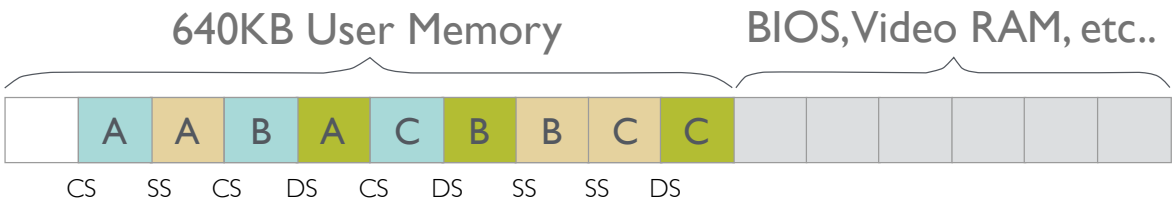

- Programs can be organized to use multiple segments:
- For example:
  - One segment for the stack
  - One segment for code
  - One segment for data
- We can relocate these segments to different physical addresses, just by adjusting the segment registers

# Variations on the theme

- Programs can have multiple code and data segments
  - Programmers could use a general "memory model"
  - Or use custom approaches to suit a specific application
- The machine provides special "far call" and "far jump" instructions that change CS and EIP simultaneously, allowing control transfers between distinct code segments
- There are six segment registers, so programs can have up to 6 active segments at a time (and more by loading new values in to the segment registers)
- Segments do not have to be exactly 64KB
- If segments do not overlap, then a stack overflow will not corrupt the contents of other segments - protection!



# Accommodating multiple programs




- Now we can have multiple programs in memory at the same time, each with distinct code, data, and stack segments
- But what is to stop the code for one program from accessing and/or changing the data for another?
- Nothing!
- We would like to "protect" programs for interfering with one another, either by accident or design …

# Protection!

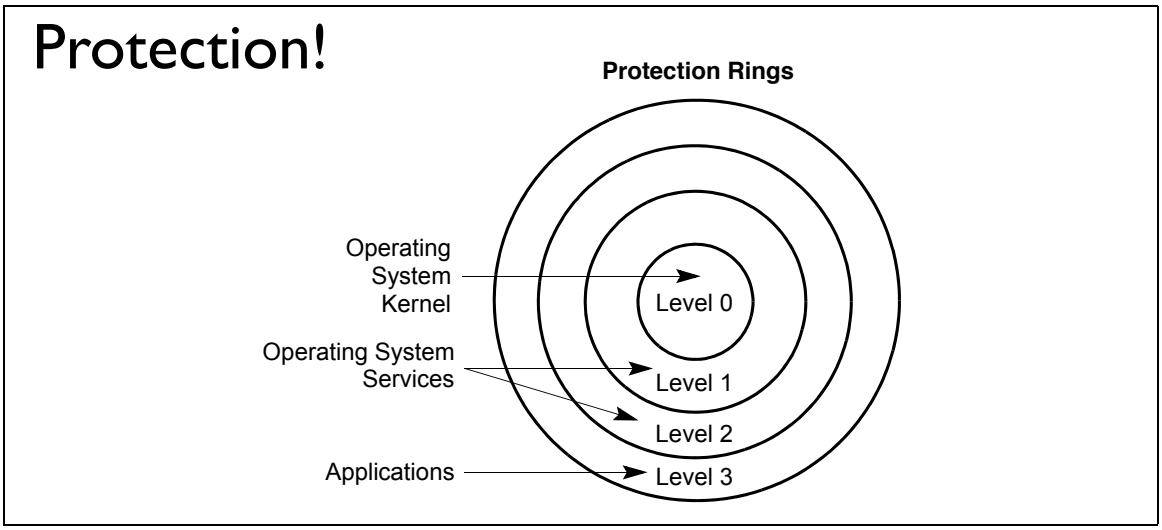



Figure 5-3. Protection Rings

- Ring 0 is sometimes called "supervisor" or "kernel mode"
- Ring 3 is often called "user mode"



# Control registers

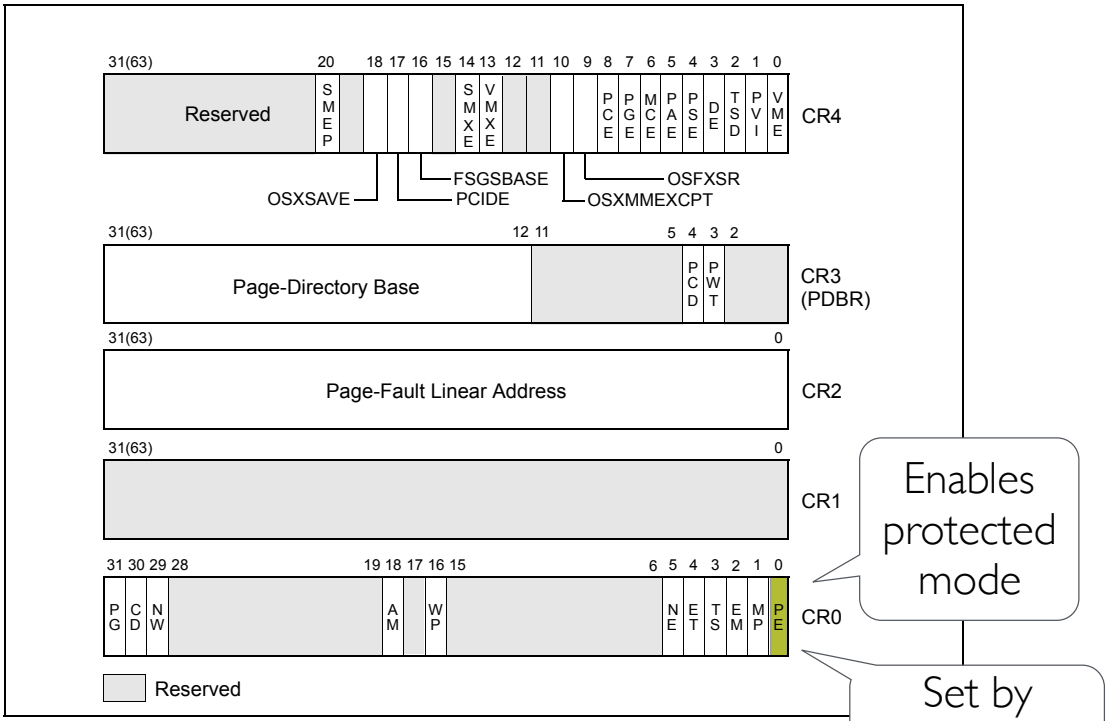



Figure 2-7. Control Registers

# The current mode

- The current mode is saved in the two least significant bits of the CS register
- The value in CS can only be changed by a limited set of instructions (e.g., it cannot be the target of a `movw`), each of which performs a privilege check, if necessary, triggering a CPU exception if a violation occurs
- End result: user mode code cannot change its own privilege level to move out of Ring 3!



# Segments in protected mode

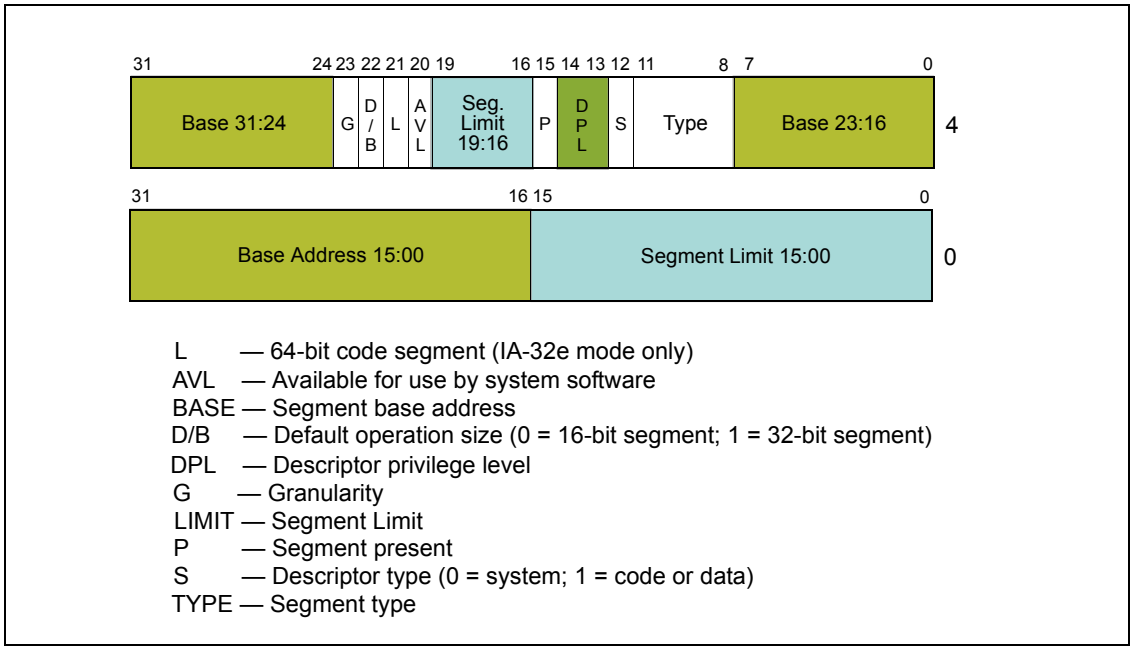



Figure 3-8. Segment Descriptor

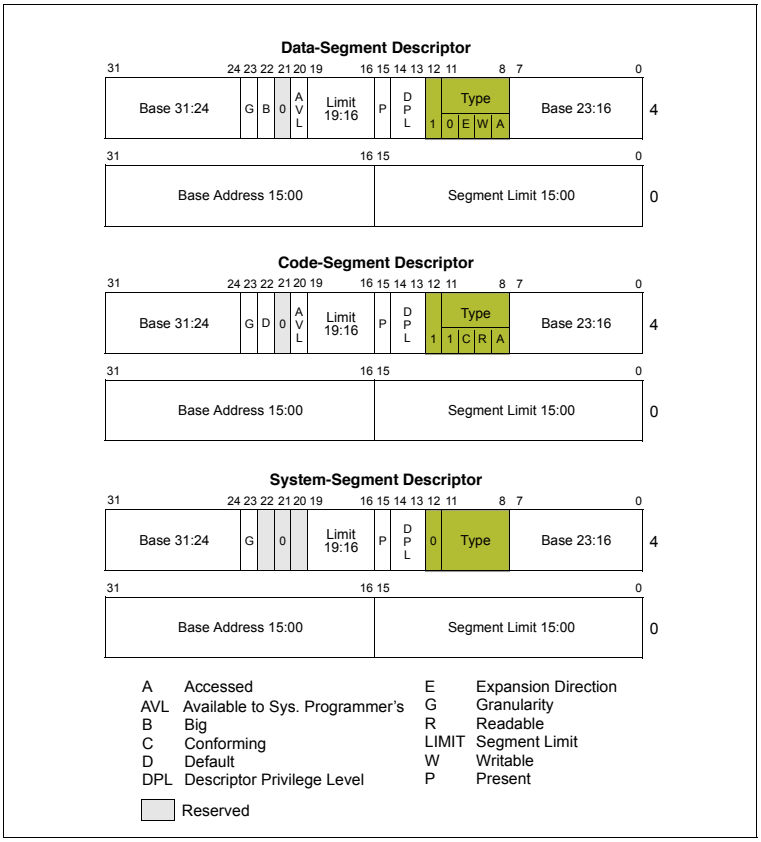



Figure 5-1. Descriptor Fields Used for Protection



# Segment registers hold segment selectors

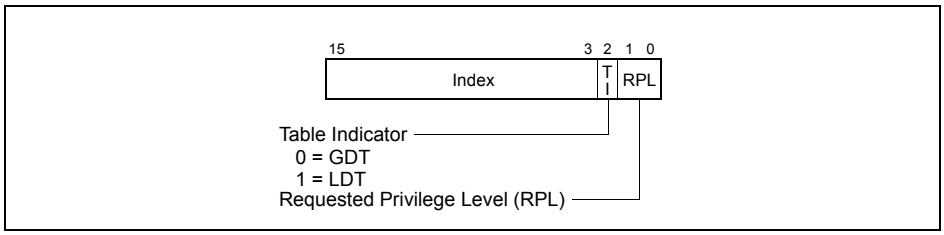



Figure 3-6. Segment Selector

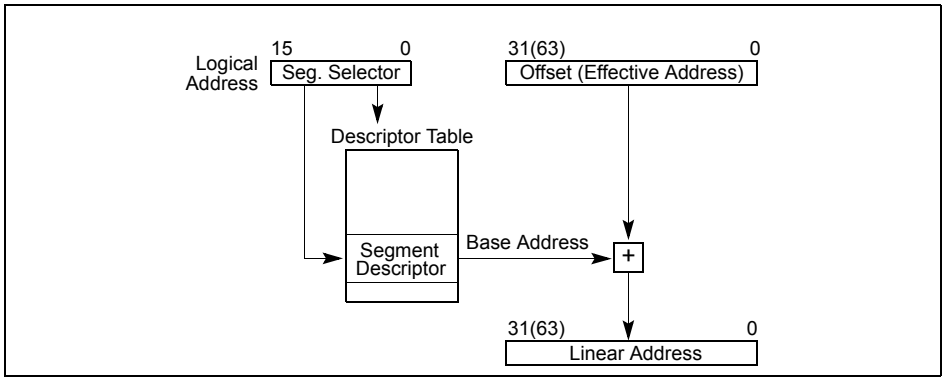



Figure 3-5. Logical Address to Linear Address Translation

# The descriptor cache

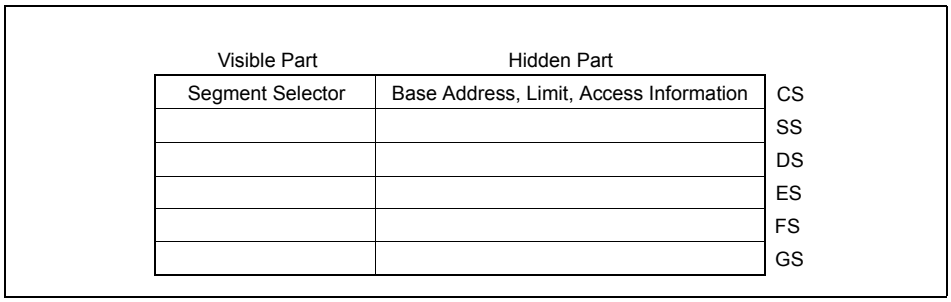



Figure 3-7. Segment Registers



# Global and local descriptor tables

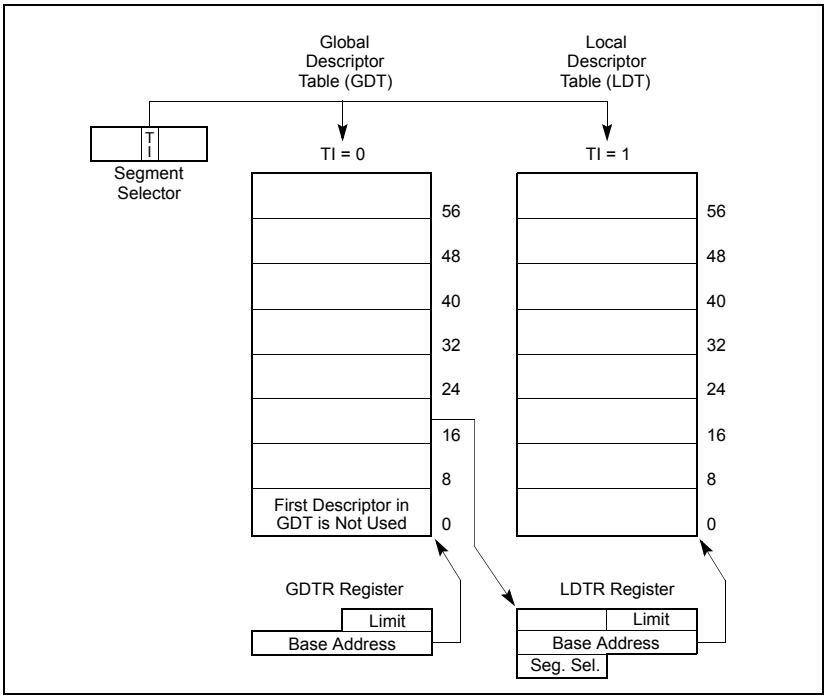



Figure 3-10. Global and Local Descriptor Tables

# Achieving protection

- The global and local descriptor tables are created by the kernel and cannot be changed by user mode programs
- The CPU raises an exception if a user mode program attempts to access:
  - a segment index outside the bounds of the GDT or LDT
  - a segment that is not marked for user mode access
  - an address beyond the limit of the associated segment
- The kernel can associate a different LDT with each process, providing each process with a distinct set of segments

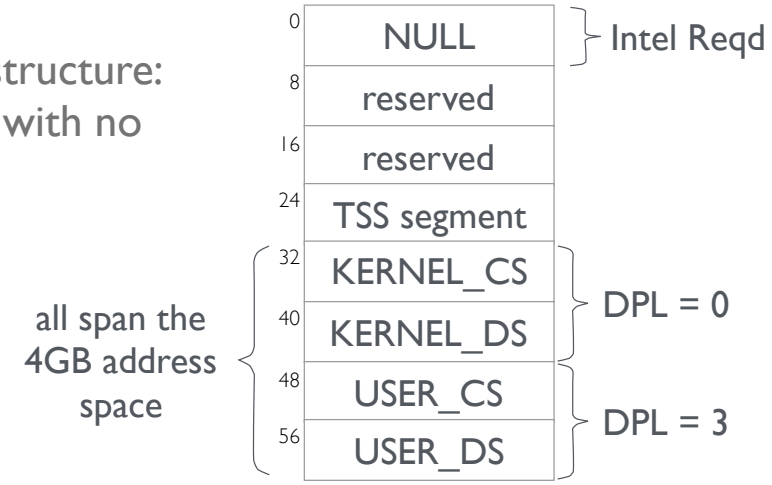


# Segments and capabilities

- The GDT and LDT for a given user mode program determine precisely which regions of memory that program can access
- As such, these entries are our first example of a **capability** mechanism
- The user mode program refers to segments by their index in one of these tables, but it has no access to the table itself:
  - It cannot, in general, determine which regions of physical memory they are accessing
  - It cannot "fake" access to other regions of memory
- **The principle of least privilege**: limit access to the minimal set of resources that are required to accomplish a task

# What if we don't want to use segments?

- Segmentation cannot be disabled in protected mode
- But we can come pretty close by using segments with:
  - base address 0
  - length = 4GB
- A common GDT structure: (e.g., in Linux, etc., with no LDT)






# Storage for the GDT

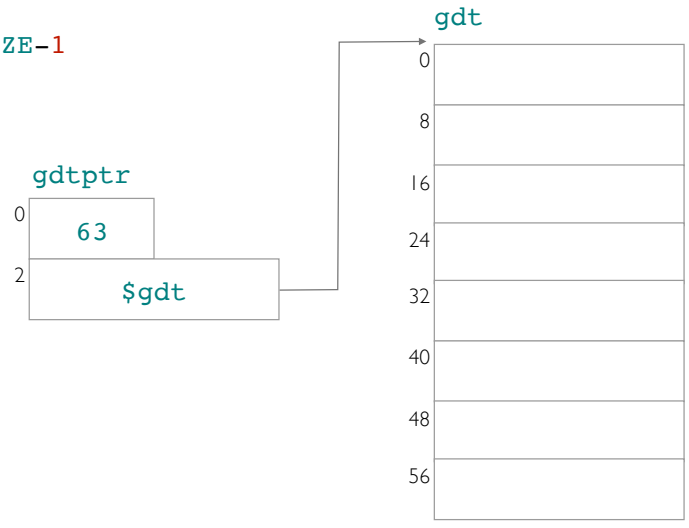
```
        .set    GDT_ENTRIES, 8
        .set    GDT_SIZE, 8*GDT_ENTRIES # 8 bytes for each descriptor

        .data
        .align  128
gdt:    .space  GDT_SIZE, 0

        .align  8
gdtptr: .short  GDT_SIZE-1
        .long   gdt
```

ready to begin?

```
lgdt gdtptr
```



# Calculating GDT descriptors

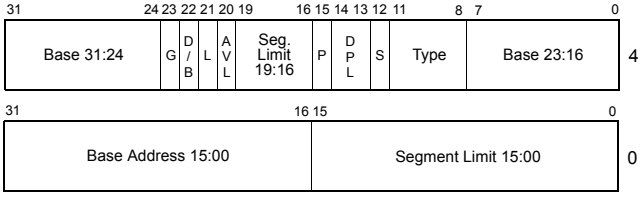
```
.macro  gdtset name, slot, base, limit, gran, dpl, type

.set    \name, (\slot<<3)|\dpl
.globl  \name
movl    $\base, %eax    # eax = bhi # bmd # blo
movl    $\limit, %ebx   # ebx = ~ # lhi # llo

mov     %eax, %edx      # edx = base
shl     $16, %eax       # eax = blo # 0
mov     %bx, %ax        # eax = blo # llo
movl    %eax, gdt+(8*\slot)

shr     $16, %edx       # edx = 0 # bhi # bmd
mov     %edx, %ecx      # ecx = 0 # bhi # bmd
andl    $0xff, %ecx     # ecx = 0 # 0   # bmd
xorl    %ecx, %edx      # edx = 0 # bhi # bmd
shl     $16,%edx        # edx = bhi # 0
orl     %ecx, %edx      # edx = bhi # 0 # bmd
andl    $0xf0000, %ebx  # ebx = 0 # lhi # 0
orl     %ebx, %edx      # edx = bhi # 0 # lhi # 0 # bmd
#
# The constant 0x4080 used below is a combination of:
#  0x4000     sets the D/B bit to indicate a 32-bit segment
#  0x0080     sets the P bit to indicate that descriptor is present
# (\gran<<15) puts the granularity bit into place
# (\dpl<<5)   puts the protection level into place
# \type       is the 5 bit type, including the S bit as its MSB
#
orl     $(((\gran<<15) | 0x4080 | (\dpl<<5) | \type)<<8), %edx
movl    %edx, gdt + (4 + 8*\slot)
.endm
```

macro assembler



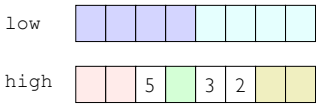


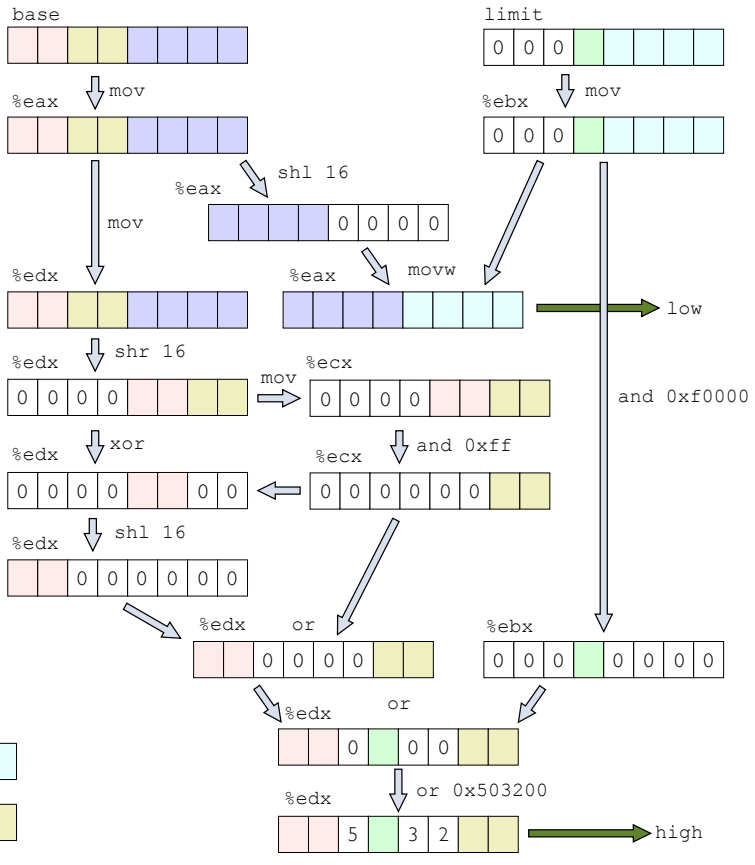

# Deja vu?

```
movl    base, %eax
movl    limit, %ebx

mov     %eax, %edx
shl     $16, %eax
mov     %bx, %ax
movl    %eax, low

shr     $16, %edx
mov     %edx, %ecx
andl    $0xff, %ecx
xorl    %ecx, %edx
shl     $16,%edx
orl     %ecx, %edx
andl    $0xf0000, %ebx
orl     %ebx, %edx
orl     $0x503200, %edx
movl    %edx, high
```



# Initializing the GDT entries

```
initGDT:# Kernel code segment:
        gdtset  name=KERN_CS, slot=4, dpl=0, type=GDT_CODE, \
                base=0, limit=0xffffff, gran=1

        # Kernel data segment:
        gdtset  name=KERN_DS, slot=5, dpl=0, type=GDT_DATA, \
                base=0, limit=0xffffff, gran=1

        # User code segment
        gdtset  name=USER_CS, slot=6, dpl=3, type=GDT_CODE, \
                base=0, limit=0xffffff, gran=1

        # User data segment
        gdtset  name=USER_DS, slot=7, dpl=3, type=GDT_DATA, \
                base=0, limit=0xffffff, gran=1

        # TSS
        gdtset  name=TSS, slot=3, dpl=0, type=GDT_TSS32, \
                base=tss, limit=tss_len-1, gran=0
```



# Activating the GDT

```
        lgdt    gdtptr
        ljmp    $KERN_CS, $1f           # load code segment
1:
        mov     $KERN_DS, %ax           # load data segments
        mov     %ax, %ds
        mov     %ax, %es
        mov     %ax, %ss
        mov     %ax, %gs
        mov     %ax, %fs

        mov     $TSS, %ax               # load task register
        ltr     %ax

        ret
```

# The Task State Segment

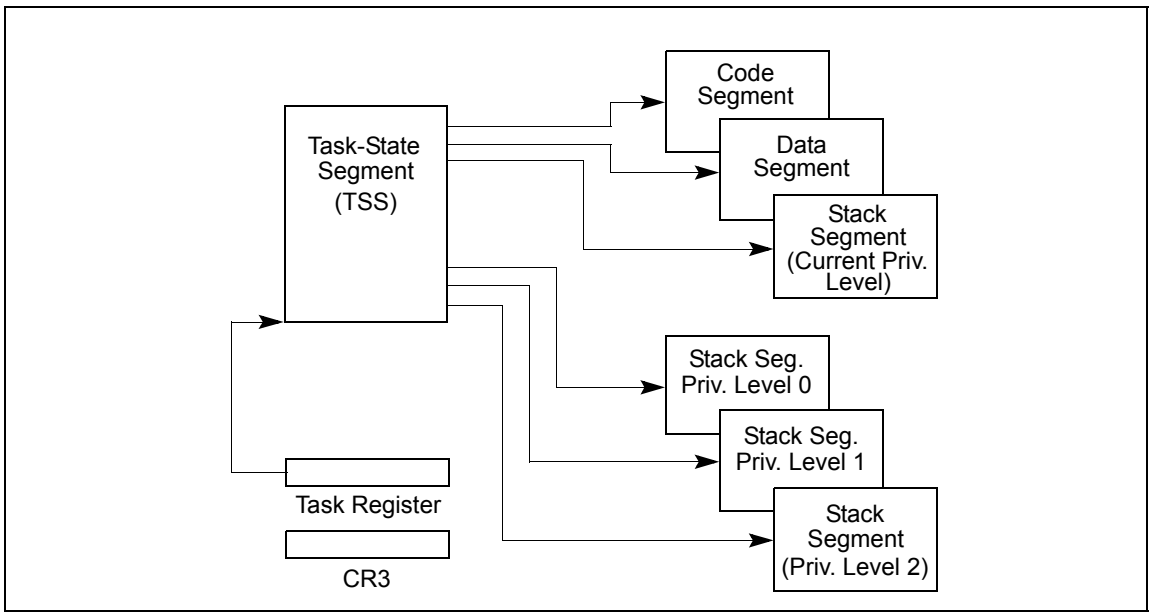



Figure 7-1. Structure of a Task



# The Task State Segment

| 31 | 15 | 0 | |
|---|---|---|---|
| I/O Map Base Address | Reserved | T | 100 |
| Reserved | LDT Segment Selector | | 96 |
| Reserved | GS | | 92 |
| Reserved | FS | | 88 |
| Reserved | DS | | 84 |
| Reserved | SS | | 80 |
| Reserved | CS | | 76 |
| Reserved | ES | | 72 |
| EDI | | | 68 |
| ESI | | | 64 |
| EBP | | | 60 |
| ESP | | | 56 |
| EBX | | | 52 |
| EDX | | | 48 |
| ECX | | | 44 |
| EAX | | | 40 |
| EFLAGS | | | 36 |
| EIP | | | 32 |
| CR3 (PDBR) | | | 28 |
| Reserved | SS2 | | 24 |
| ESP2 | | | 20 |
| Reserved | SS1 | | 16 |
| ESP1 | | | 12 |
| Reserved | SS0 | | 8 |
| ESP0 | | | 4 |
| Reserved | Previous Task Link | | 0 |

Reserved bits. Set to 0.

Figure 7-2. 32-Bit Task-State Segment (TSS)

# Implementing the TSS

```
        .data
tss:    .short  0, RESERVED             # previous task link
esp0:   .long   0                       # esp0
        .short  KERN_DS, RESERVED       # ss0
        .long   0                       # esp1
        .short  0, RESERVED             # ss1
        .long   0                       # esp2
        .short  0, RESERVED             # ss2
        .long   0, 0, 0                 # cr3 (pdbr), eip, eflags
        .long   0, 0, 0, 0, 0           # eax, ecx, edx, ebx, esp
        .long   0, 0, 0                 # ebp, esi, edi
        .short  0, RESERVED             # es
        .short  0, RESERVED             # cs
        .short  0, RESERVED             # ss
        .short  0, RESERVED             # ds
        .short  0, RESERVED             # fs
        .short  0, RESERVED             # gs
        .short  0, RESERVED             # ldt segment selector
        .short  0                       # T bit
        .short  1000                    # I/O bit map base address
        .set    tss_len, .-tss
```



# Interrupts and exceptions

# Exceptions

- What happens if the program you run on a conventional desktop computer attempts:
  - division by zero?
  - to use an invalid segment selector?
  - to reference memory beyond the limits of a segment?
  - etc…
- What happens when there is no operating system to catch you?



Table 6-1. Protected-Mode Exceptions and Interrupts

| Vector No. | Mnemonic | Description | Type | Error Code | Source |
|---|---|---|---|---|---|
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions. |
| 1 | #DB | RESERVED | Fault/ Trap | No | For Intel use only. |
| 2 | — | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt. |
| 3 | #BP | Breakpoint | Trap | No | INT 3 instruction. |
| 4 | #OF | Overflow | Trap | No | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction. |
| 6 | #UD | Invalid Opcode (Undefined Opcode) | Fault | No | UD2 instruction or reserved opcode.[1] |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Abort | Yes (zero) | Any instruction that can generate exception, an NMI, or an INTR. |
| 9 | | Coprocessor Segment Overrun (reserved) | Fault | No | Floating-point instruction.[2] |
| 10 | #TS | Invalid TSS | Fault | Yes | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Fault | Yes | Loading segment registers or acc system segments. |
| 12 | #SS | Stack-Segment Fault | Fault | Yes | Stack operations and SS register l |
| 13 | #GP | General Protection | Fault | Yes | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Fault | Yes | Any memory reference. |
| 15 | — | (Intel reserved. Do not use.) | | No | |
| 16 | #MF | x87 FPU Floating-Point Error (Math Fault) | Fault | No | x87 FPU floating-point or WAIT/FWAIT instruction. |
| 17 | #AC | Alignment Check | Fault | Yes (Zero) | Any data reference in memory.[3] |
| 18 | #MC | Machine Check | Abort | No | Error codes (if any) and source are model dependent.[4] |
| 19 | #XM | SIMD Floating-Point Exception | Fault | No | SSE/SSE2/SSE3 floating-point instructions[5] |
| 20 | #VE | Virtualization Exception | Fault | No | EPT violations[6] |
| 21-31 | — | Intel reserved. Do not use. | | | |
| 32-255 | — | User Defined (Non-reserved) Interrupts | Interrupt | | External interrupt or INT *n* instruction. |

- **Faults** can generally be corrected, restarting the program *at* the faulting instruction
- **Traps** allow execution to be restarted *after* the trapping instruction
- **Aborts** do not allow a restart

# Hardware and software interrupts

- **Hardware**: devices often generate interrupt signals to inform the kernel that a certain even has occurred:
  - a timer has fired
  - a key has been pressed
  - a buffer of data has been transferred
  - …
- **Software**: User programs often request services from an underlying operating system:
  - read data from a file
  - terminate this program
  - send a message
  - …
- These can all be handled in the same way …



# The interrupt vector

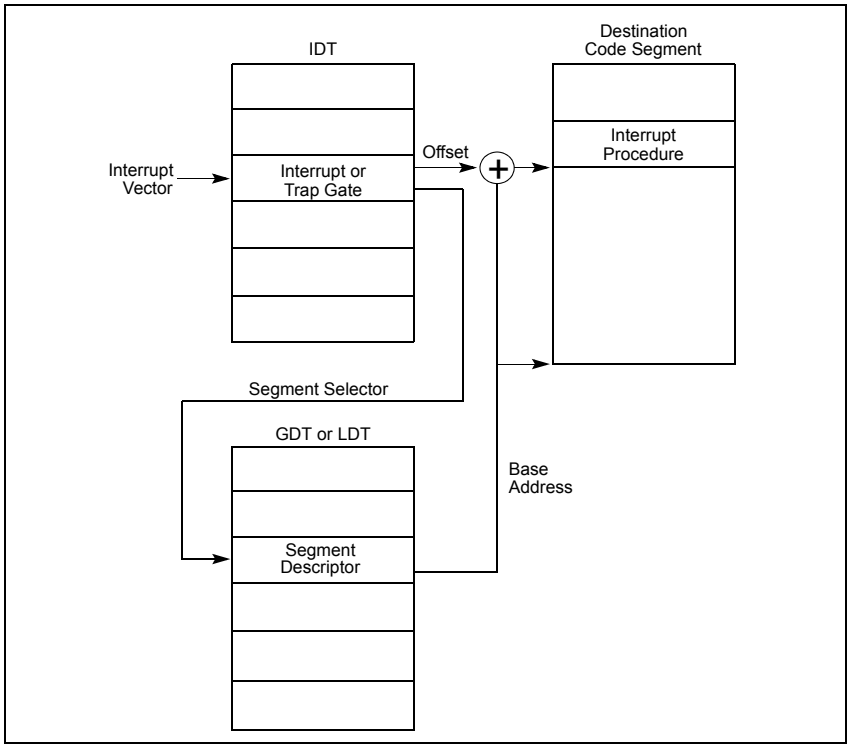



Figure 6-3. Interrupt Procedure Call

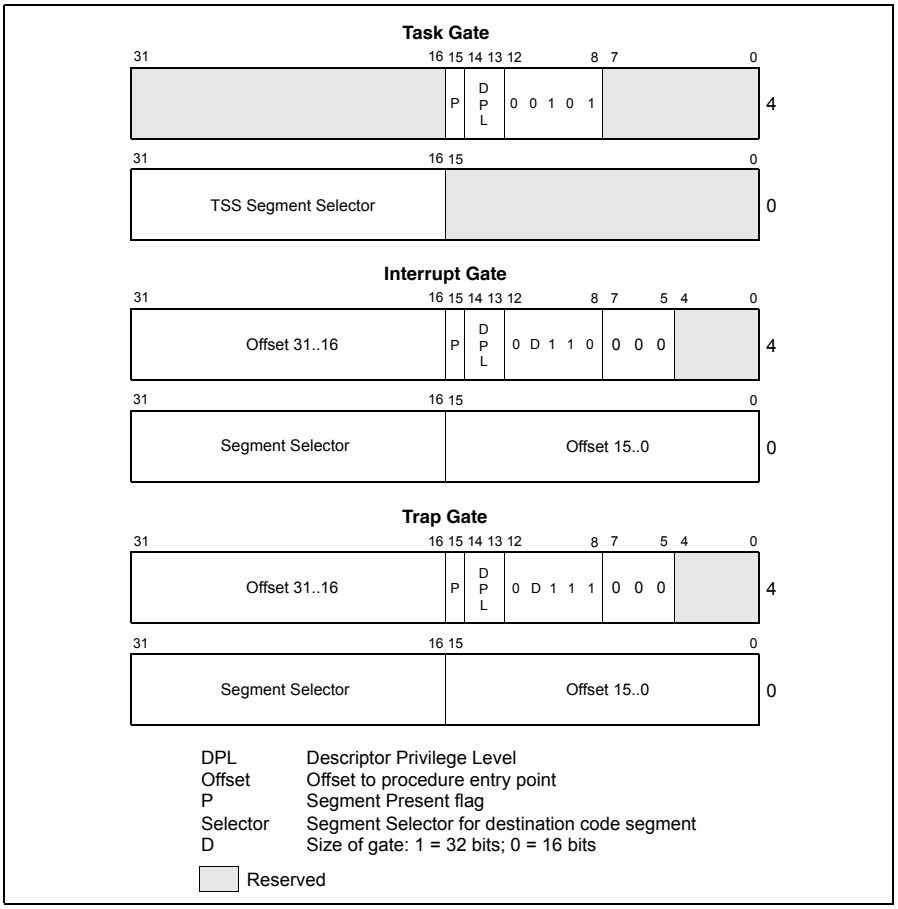



Figure 6-2. IDT Gate Descriptors



# Storage for the IDT

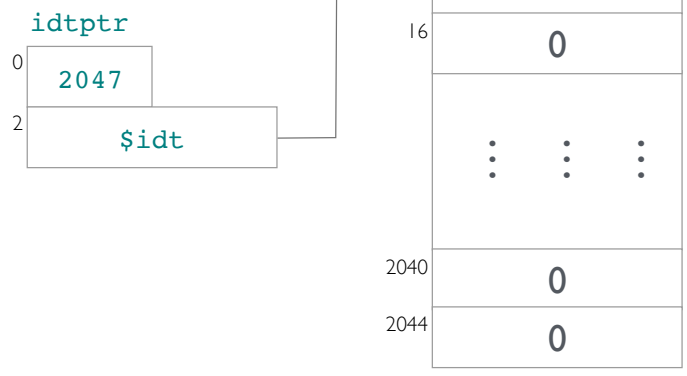
```
        .set    IDT_ENTRIES, 256        # Allow for all poss. interrupts
        .set    IDT_SIZE, 8*IDT_ENTRIES # Eight bytes for each idt desc.
        .set    IDT_INTR, 0x000         # Type for interrupt gate
        .set    IDT_TRAP, 0x100         # Type for trap gate

        .data
        .align  8
idt:    .space  IDT_SIZE, 0

idtptr: .short  IDT_SIZE-1
        .long   idt
```

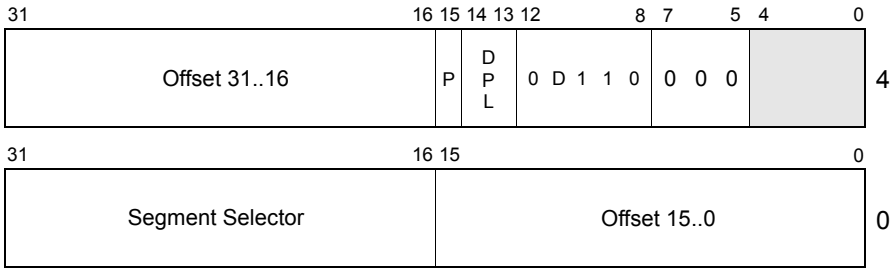
ready to begin?

```
lidt idtptr
```



# Calculating IDT descriptors

```
.macro  idtcalc handler, slot, dpl=0, type=IDT_INTR, seg=KERN_CS
# type = 0x000 (IDT_INTR)  => interrupt gate
# type = 0x100 (IDT_TRAP)  => trap gate
#
# The following comments use # for concatenation of bitdata
#
mov     $\seg, %ax              # eax =   ? # seg
shl     $16, %eax               # eax = seg #   0
movl    $\handler, %edx         # edx = hhi # hlo
mov     %dx, %ax                # eax = seg # hlo
mov     $(0x8e00 | (\dpl<<13) | \type), %dx
movl    %eax, idt + (    8*\slot)
movl    %edx, idt + (4 + 8*\slot)
.endm
```






# Initializing and activating the IDT

```
initIDT:# Fill in IDT entries

        # Add descriptors for protected mode exceptions:
        .irp    num, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,16,17,18,19
        idtcalc exc\num, slot=\num
        .endr

        # Add descriptors for hardware irqs:
        # ... except there aren't any here (yet)

        # Add descriptors for system calls:
        # These are the only idt entries that we will allow to be
        # called from user mode without generating a general
        # protection fault, so they are tagged with dpl=3.
        idtcalc handler=syscall, slot=0x80, dpl=3

        # Install the new IDT:
        lidt    idtptr
        ret
```

macro loop

# Transferring control to a handler

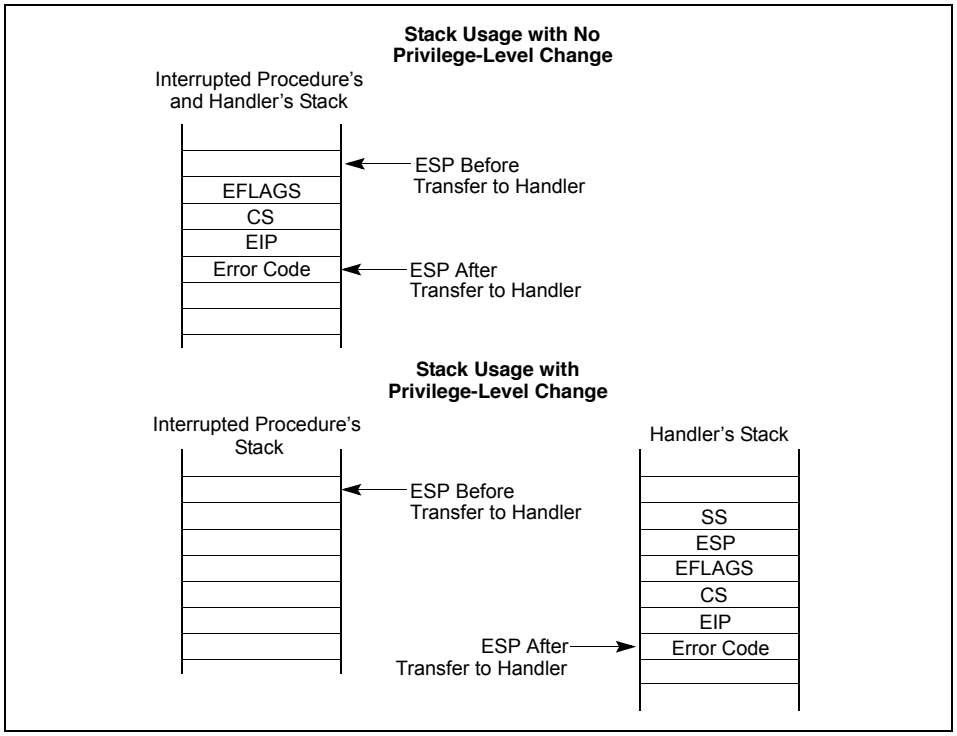
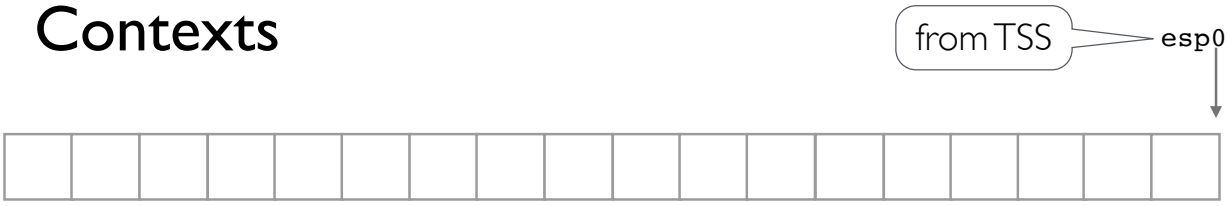


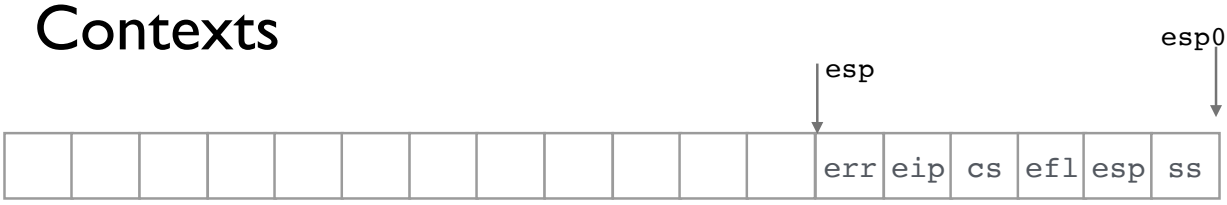

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines



# Contexts



# Contexts

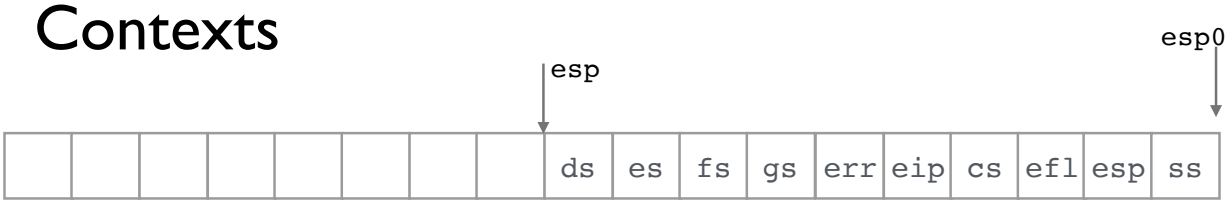



```
struct Iret {
  unsigned error;
  unsigned eip;
  unsigned cs;
  unsigned eflags;
  unsigned esp;
  unsigned ss;
};
```

Automatic (CPU)



# Contexts



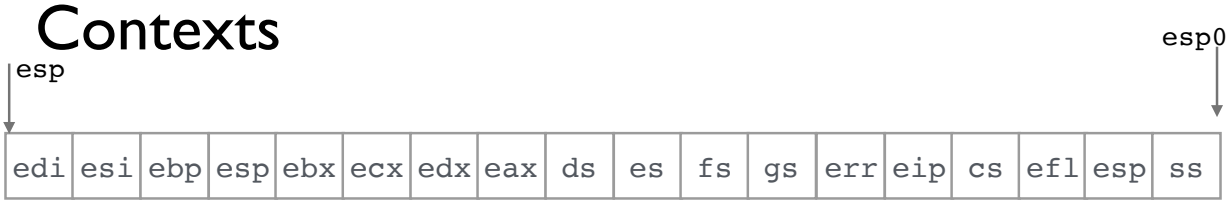

```
struct Segments {
  unsigned ds;
  unsigned es;
  unsigned fs;
  unsigned gs;
};
```

```
struct Iret {
  unsigned error;
  unsigned eip;
  unsigned cs;
  unsigned eflags;
  unsigned esp;
  unsigned ss;
};
```

```
push    %gs
push    %fs
push    %es
push    %ds
```

Automatic (CPU)

# Contexts




```
struct Registers {
  unsigned edi;
  unsigned esi;
  unsigned ebp;
  unsigned esp;
  unsigned ebx;
  unsigned edx;
  unsigned ecx;
  unsigned eax;
};
```

```
struct Segments {
  unsigned ds;
  unsigned es;
  unsigned fs;
  unsigned gs;
};
```

```
struct Iret {
  unsigned error;
  unsigned eip;
  unsigned cs;
  unsigned eflags;
  unsigned esp;
  unsigned ss;
};
```

```
pusha
```

```
push    %gs
push    %fs
push    %es
push    %ds
```

Automatic (CPU)



# Contexts


```
struct Registers {
  unsigned edi;
  unsigned esi;
  unsigned ebp;
  unsigned esp;
  unsigned ebx;
  unsigned edx;
  unsigned ecx;
  unsigned eax;
};
```

```
struct Segments {
  unsigned ds;
  unsigned es;
  unsigned fs;
  unsigned gs;
};
```

```
struct Iret {
  unsigned error;
  unsigned eip;
  unsigned cs;
  unsigned eflags;
  unsigned esp;
  unsigned ss;
};
```

Captures the "context" of the interrupted program

```
struct Context {
  struct Registers regs;
  struct Segments  segs;
  struct Iret      iret;
};
```

Table 6-1. Protected-Mode Exceptions and Interrupts

| Vector No. | Mne-monic | Description | Type | Error Code | Source |
|---|---|---|---|---|---|
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions. |
| 1 | #DB | RESERVED | Fault/ Trap | No | For Intel use only. |
| 2 | — | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt. |
| 3 | #BP | Breakpoint | Trap | No | INT 3 instruction. |
| 4 | #OF | Overflow | Trap | No | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction. |
| 6 | #UD | Invalid Opcode (Undefined Opcode) | Fault | No | UD2 instruction or reserved opcode.[1] |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Abort | Yes (zero) | Any instruction that can generate exception, an NMI, or an INTR. |
| 9 | | Coprocessor Segment Overrun (reserved) | Fault | No | Floating-point instruction.[2] |
| 10 | #TS | Invalid TSS | Fault | Yes | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Fault | Yes | Loading segment registers or acc system segments. |
| 12 | #SS | Stack-Segment Fault | Fault | Yes | Stack operations and SS register l |
| 13 | #GP | General Protection | Fault | Yes | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Fault | Yes | Any memory reference. |
| 15 | — | (Intel reserved. Do not use.) | | No | |
| 16 | #MF | x87 FPU Floating-Point Error (Math Fault) | Fault | No | x87 FPU floating-point or WAIT/FWAIT instruction. |
| 17 | #AC | Alignment Check | Fault | Yes (Zero) | Any data reference in memory.[3] |
| 18 | #MC | Machine Check | Abort | No | Error codes (if any) and source are model dependent.[4] |
| 19 | #XM | SIMD Floating-Point Exception | Fault | No | SSE/SSE2/SSE3 floating-point instructions[5] |
| 20 | #VE | Virtualization Exception | Fault | No | EPT violations[6] |
| 21-31 | — | Intel reserved. Do not use. | | | |
| 32-255 | — | User Defined (Non-reserved) Interrupts | Interrupt | | External interrupt or INT *n* instruction. |

- **Faults** can generally be corrected, restarting the program *at* the faulting instruction
- **Traps** allow execution to be restarted *after* the trapping instruction
- **Aborts** do not allow a restart



# Exception handler

Some exceptions do not generate an error code …

```
        .macro  handler num, func, errorcode=0
exc\num:.if     \errorcode==0
        subl    $4, %esp        # Fake an error code if necessary
        .endif
        push    %gs             # Save segments
        push    %fs
        push    %es
        push    %ds
        pusha                   # Save registers

        push    %esp            # Push pointer to frame for handler
        movl    $\num, %eax
        call    \func
        addl    $4, %esp

        popa                    # Restore registers
        popl    %ds             # Restore segments
        popl    %es
        popl    %fs
        popl    %gs
        addl    $4, %esp        # remove error code
        iret
        .endm
```

call `func(struct Context *esp)` with `num` in `eax`

"return from interrupt"

# Defining a family of (non) handlers

```
# Protected-mode exceptions and interrupts:
#
handler num=0,  func=nohandler                  # divide error
handler num=1,  func=nohandler                  # debug
handler num=2,  func=nohandler                  # NMI
handler num=3,  func=nohandler                  # breakpoint
handler num=4,  func=nohandler                  # overflow
handler num=5,  func=nohandler                  # bound
handler num=6,  func=nohandler                  # undefined opcode
handler num=7,  func=nohandler                  # nomath
handler num=8,  func=nohandler, errorcode=1     # doublefault
handler num=9,  func=nohandler                  # coproc seg overrun
handler num=10, func=nohandler, errorcode=1     # invalid tss
handler num=11, func=nohandler, errorcode=1     # segment not present
handler num=12, func=nohandler, errorcode=1     # stack-segment fault
handler num=13, func=nohandler, errorcode=1     # general protection
handler num=14, func=nohandler, errorcode=1     # page fault
handler num=16, func=nohandler                  # math fault
handler num=17, func=nohandler, errorcode=1     # alignment check
handler num=18, func=nohandler                  # machine check
handler num=19, func=nohandler                  # SIMD fp exception
```



# Defining a family of (non) handlers

```
nohandler:                          # dummy interrupt handler
        movl    4(%esp), %ebx       # get frame pointer
        pushl   %ebx
        pushl   %eax
        pushl   $excepted
        call    printf
        addl    $12, %esp

1:      hlt
        jmp 1b

        ret
excepted:
        .asciz  "Exception 0x%x, frame=0x%x\n"
```

call `printf(excepted, num, ctxt)`

# Initializing a context

```
struct Context user;

…

  initContext(&user, userEntry, 0);

…

void initContext(struct Context* ctxt, unsigned eip, unsigned esp) {
  extern char USER_DS[];
  extern char USER_CS[];
  printf("user data segment is 0x%x\n", (unsigned)USER_DS);
  printf("user code segment is 0x%x\n", (unsigned)USER_CS);
  ctxt->segs.ds     = (unsigned)USER_DS;
  ctxt->segs.es     = (unsigned)USER_DS;
  ctxt->segs.fs     = (unsigned)USER_DS;
  ctxt->segs.gs     = (unsigned)USER_DS;
  ctxt->iret.ss     = (unsigned)USER_DS;
  ctxt->iret.esp    = esp;
  ctxt->iret.cs     = (unsigned)USER_CS;
  ctxt->iret.eip    = eip;
  ctxt->iret.eflags = INIT_USER_FLAGS;
}
```



# Initializing the flags

```
#define INIT_USER_FLAGS (3<<12 | 1<<9 | 0x2)
```

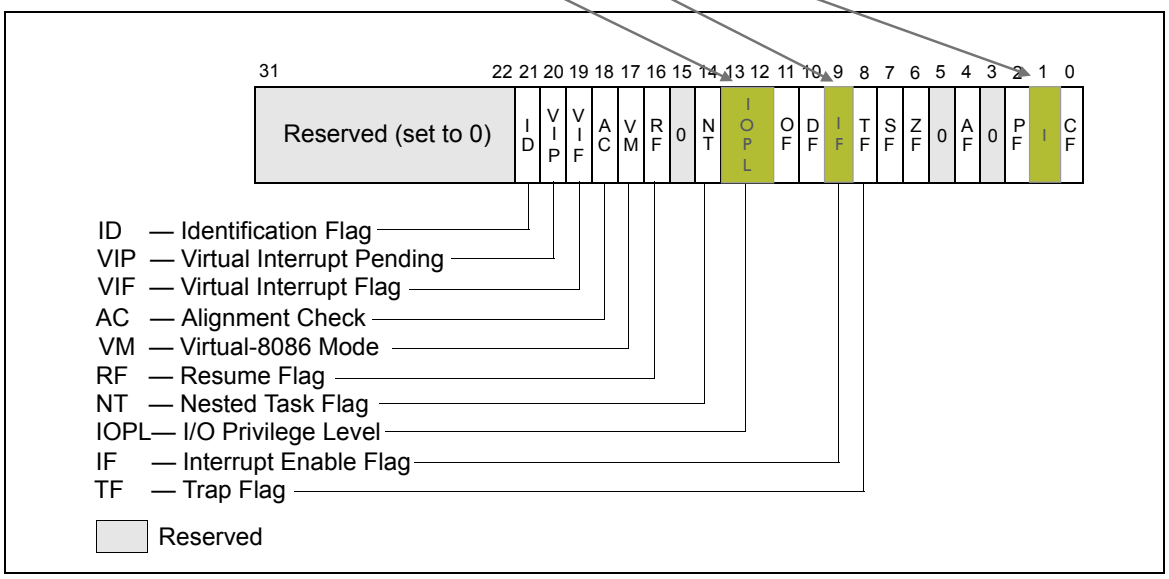



Figure 2-5. System Flags in the EFLAGS Register

# Switching to a user program

From C:

```
extern int switchToUser(struct Context* ctxt);
```

To Assembly:

```
        .set    CONTEXT_SIZE, 72
        .globl  switchToUser
switchToUser:
        movl    4(%esp), %eax   # Load address of the user context
        movl    %eax, %esp      # Reset stack to base of user context
        addl    $CONTEXT_SIZE, %eax
        movl    %eax, esp0      # Set stack address for kernel reentry
        popa                    # Restore registers
        pop     %ds             # Restore segments
        pop     %es
        pop     %fs
        pop     %gs
        addl    $4, %esp        # Skip error code
        iret                    # Return from interrupt
```



# Entering a system call (kernel view)

From Assembly:

```
        .text
syscall:subl    $4, %esp        # Fake an error code
        push    %gs             # Save segments
        push    %fs
        push    %es
        push    %ds
        pusha                   # Save registers
        leal    stack, %esp     # Switch to kernel stack
        jmp     csyscall
```

To C:

```
   void csyscall() {  /* A trivial system call */
      putchar(user.regs.eax);
      switchToUser(&user);
    }
```

# Entering a system call (user view)

From C:

```
extern void myputc(unsigned);
```

To Assembly:

```
        .globl  myputc
myputc: pushl   %eax
        mov     8(%esp), %eax
        int     $128
        popl    %eax
        ret
```



# Reflections

- Bare Metal
  - Segmentation, protection, exceptions and interrupts
- Programming/Languages
  - Representation transparency, facilitates interlanguage interoperability
  - Memory areas
    - Vendor-defined layout: GDT, GDTTR, TSS, IDT, IDTR, IRet, Registers, …
    - Self-defined: Context, …
  - "Bitdata"
    - Segment and interrupt descriptors, eflags, cr0, …