

236366 Operating Systems Engineering
Recitation #3 (part 2):
**Interrupt and Exception
Handling on the x86**

(heavily) based on MIT 6.828 (2005, lec8)

x86 Interrupt Nomenclature

- Hardware Interrupt (external)
 - IRQ - **i**nterrupt **r**equ**e**st from device
 - NMI - **n**on-**m**askable **i**nterrupt (HW error)
 - IPI - **i**nter-**p**rocessor **i**nterrupt (in SMP)
- Exception (internal, CPU-generated)
 - Fault - (usually) correctable error
 - Trap - programmer-initiated (not a error)
 - Abort - severe error (CPU almost crashed)

x86 Interrupt Vectors

- Single hardware mechanism to handle all interrupt types
- Every Exception/Interrupt type is assigned a number:
 - **interrupt vector**
- When an interrupt occurs, the vector determines what code is invoked to handle the interrupt.
- JOS example: vector 14 → page fault handler
vector 32 → clock handler → scheduler

0	Divide Error
2	Non-Maskable Interrupt
3	Breakpoint Exception
6	Invalid Opcode
11	Segment Not Present
12	Stack-Segment Fault
13	General Protection Fault
14	Page Fault
18	Machine Check
32-255	User Defined Interrupts

Sources: Hardware Interrupts

- Hardware Interrupt Types:
Non-Maskable Interrupt
 - Never ignored

INTR Maskable

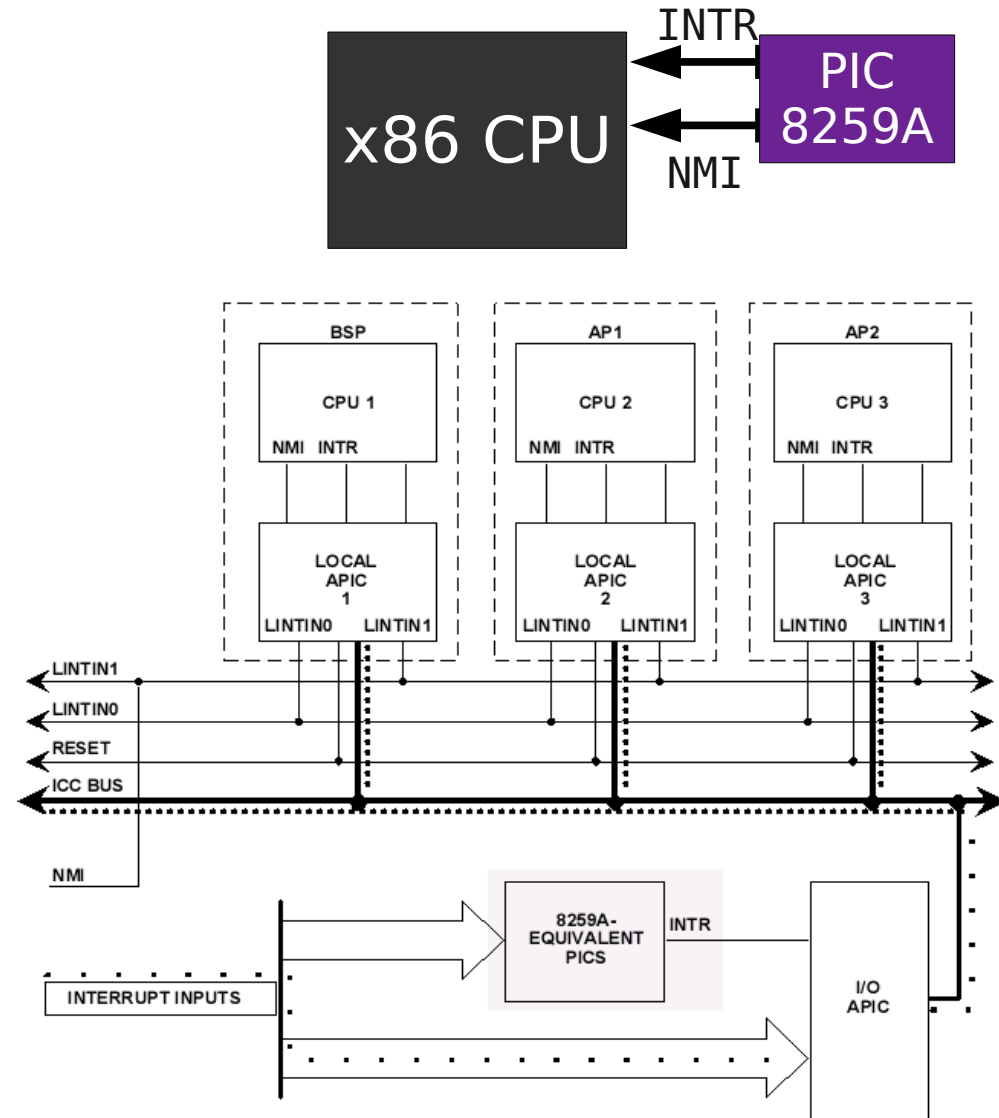
- Ignored when IF is 0

- PIC: *Programmable* Interrupt Controller (8259A)

- IRQ→IV mapping
- IRQ priorities
- selective masking

- APIC: Advanced PIC:

- LAPIC + IOAPIC
- MMIO, not PIO
- SMP aware



Sources: Software-generated Interrupts

Traps - Programmed Interrupts

- x86 provides INT instruction.
- Invokes the interrupt handler for vector N (0-255)
- JOS: we use 'INT 0x30' for system calls

Faults - Software Exceptions

- Processor detects an error condition while executing an instruction.
- Ex: **divl %eax, %eax**
 - Divide by zero if EAX = 0
- Ex: **movl %ebx, (%eax)**
 - Page fault or seg violation if EAX is un-mapped virtual address.
- Ex: **jmp \$BAD_JMP**
 - General Protection Fault (jmp'd out of CS)

Enabling / Disabling Interrupts

Maskable Hardware Interrupts

- IF (interrupt enabled flag) is a part of EFLAGS
- Clearing the IF flag inhibits processing hardware interrupts delivered on the INTR line.
- Use the **STI** (set IF) and **CLI** (clear IF) instructions.
- IF affected by: interrupt/task gates, POPF, and IRET.

Non-Maskable Interrupt

- Invoked by NMI line from PIC.
- Always Handled immediately.
- Handler for interrupt vector 2 invoked.
- No other interrupts can execute until NMI is done.

IDT: Interrupt Descriptor Table

IDT:

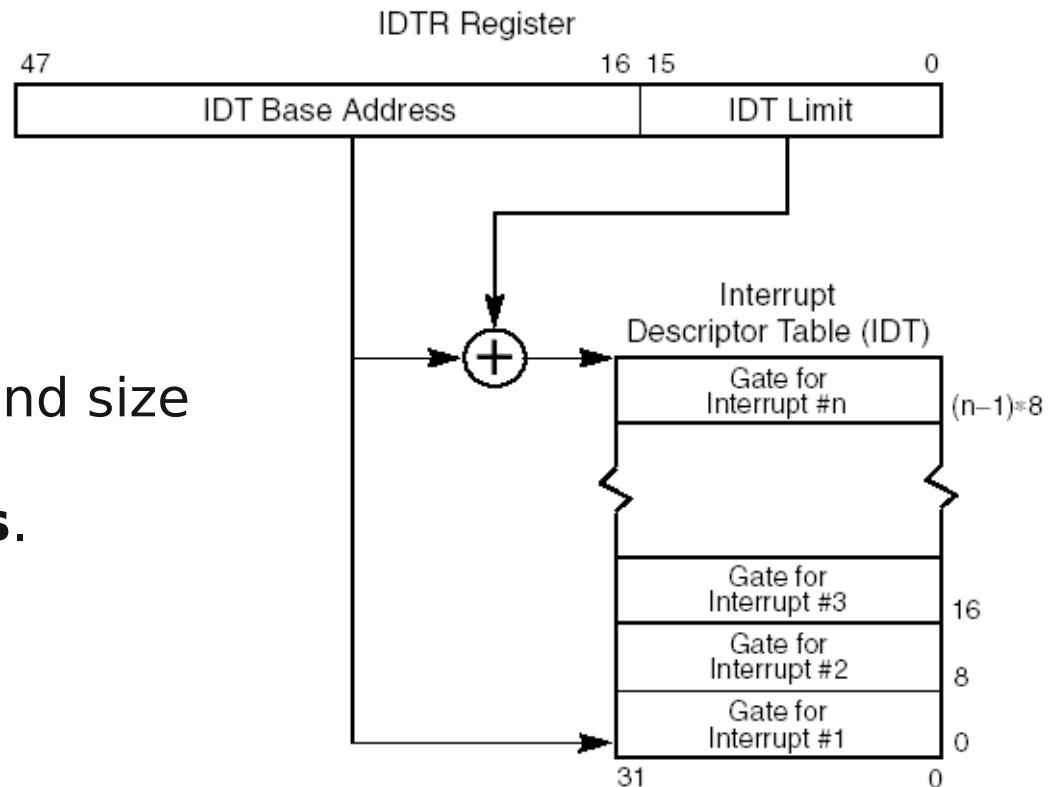
- Table of 256 8-byte entries (similar to the GDT).
- In JOS: Each specifies a protected entry-point into the kernel.
- Located anywhere in memory.

IDTR register:

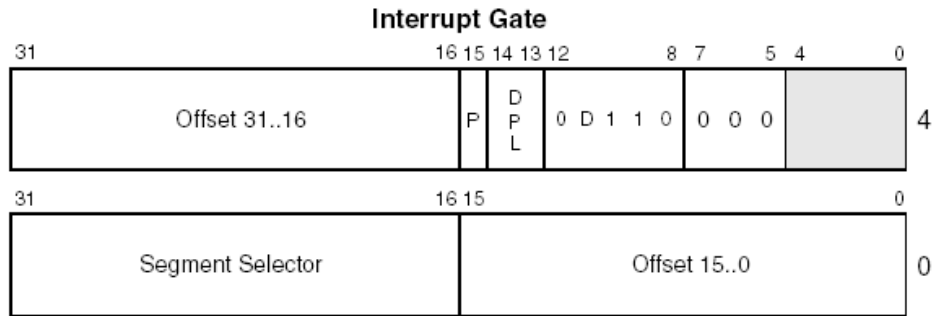
- Stores current IDT.

lidt instruction:

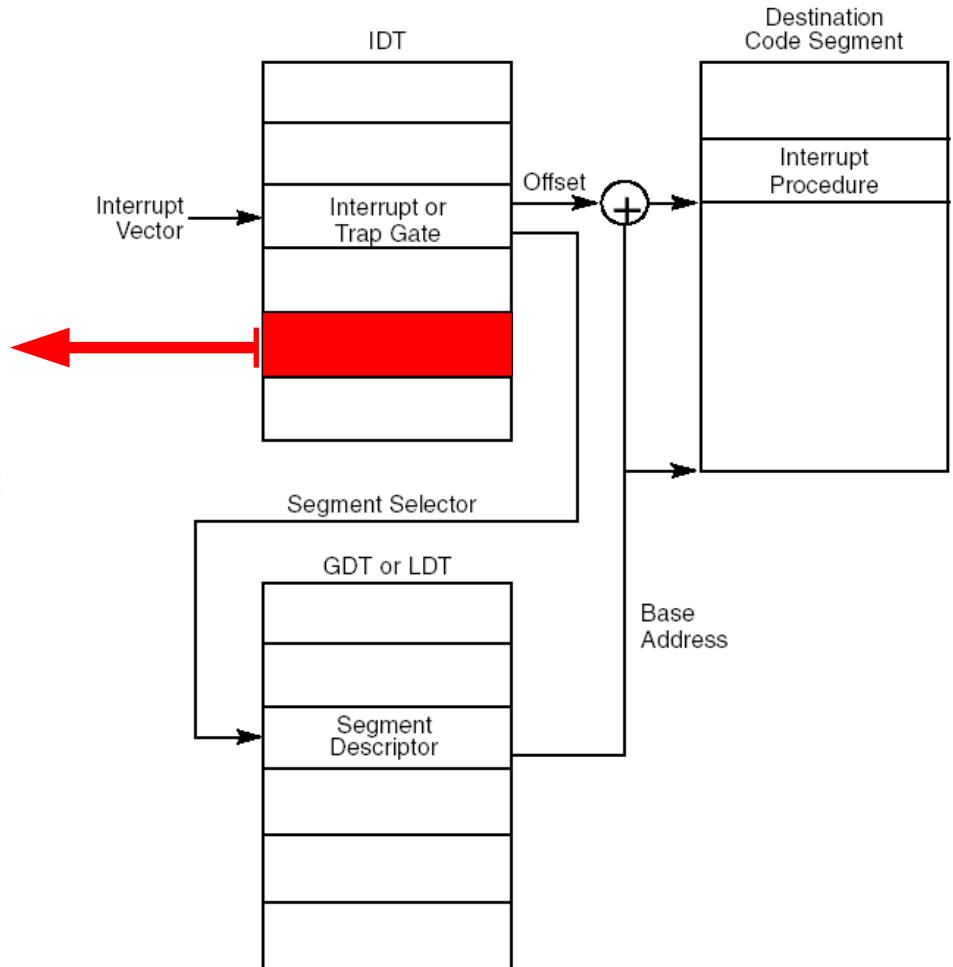
- Loads IDTR with address and size of the IDT.
- Takes in a **linear address**.



IDT Entries



Selector Segment Selector for dest. code segment
Offset Offset to procedure entry point
P Segment Present Flag
DPL Descriptor Privilege Level
D Size of gate: 1 = 32 bits; 0 = 16 bits
[bit 40] 0 = interrupt gate; 1 = trap gate



JOS: Interrupts and Address Spaces

- JOS approach tries to minimize segmentation usage
 - so ignore segmentation issues with interrupts

Priority Level Switch

- CPL is low two bits of CS (11=kernel, 00=user)
- Loading new CS for handler can change CPL.
- JOS interrupt handlers run with kernel CPL.

Addressing Switch

- No address space switch when handler invoked.
- Paging is not changed.
- However in: Kernel VA regions now accessible


Stack Switch (User » Kernel)

- stack switched to a kernel stack before handler is invoked.

TSS: Task State Segment

- Specialized Segment for hardware supported multi-tasking
 - most OS don't use this feature
 - implement SW multitasking
 - JOS & xv6 are among these OS
- TSS Resides in memory
 - TSS descriptor goes into GDT (size and linear address of the TSS)
 - `ltr(GD_TSS)` loads descriptor
- **In JOS's TSS:**
 - SS0:ESP0 kernel stack used by interrupt handlers.
 - All other TSS fields ignored

31	15	0	
I/O Map Base Address		T	100
		LDT Segment Selector	96
		GS	92
		FS	88
		DS	84
		SS	80
		CS	76
		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
		SS2	24
ESP2			20
		SS1	16
ESP1			12
		SS0	8
ESP0			4
		Previous Task Link	0

 Reserved bits. Set to 0.

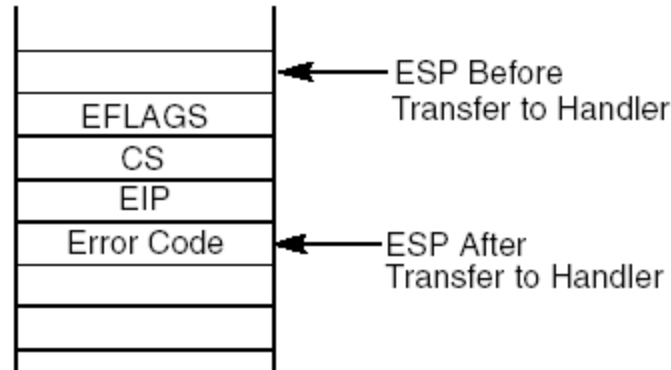
Exception Entry Mechanism

Kernel»Kernel

(New State)

SS unchanged
ESP (new frame pushed)
CS:EIP (from IDT)

Interrupted Procedure's
and Handler's Stack

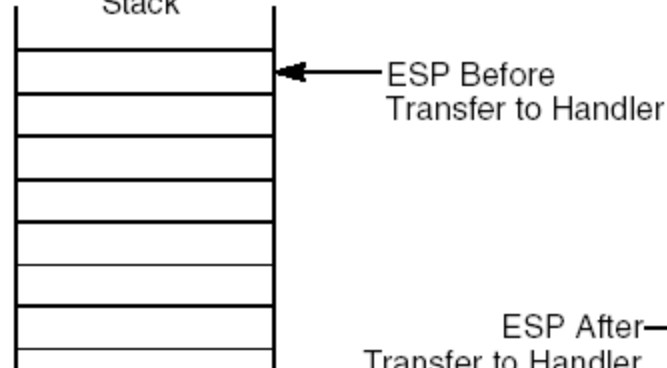


User»Kernel

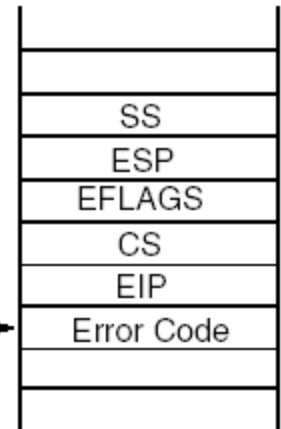
(New State)

SS:ESP TSS ss0:esp0
CS:EIP (from IDT)
EFLAGS:
interrupt gates: clear IF

Interrupted Procedure's
Stack



Handler's Stack



Interrupt Dispatch

- it may be convenient to have a single entry point for all interrupts
- but this is not supported by x86
 - interrupt number isn't passed
- workaround: table with tiny “handlers”
 - store constant interrupt number
 - jump to common handler (alltraps)
- no task switch
 - save all current state manually
 - restore upon iret
 - setup segment regs (why?)
- call C dispatcher ASAP

JOS Trap Frame

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
};

void trap(struct Trapframe *tf) {...}
```

Syscalls

- to perform a syscall user does

```
movl N, %eax
int 0x30
```
- common trap handler calls common syscall handler
- syscall() analyses saved (where?) %eax and calls specific syscall handler
- return value is saved in place of stored %eax to be restored upon return to user
- dereference user pointers with a special care
 - ensure points to user's data
 - ensure the whole data-structure (struct, array, string,...) is accessible by the user

Exception Return Mechanism

iret: interrupt return instruction
(top of stack should point to old EIP)

Where do we return?

- **Hardware Interrupts**
old CS:EIP points past last completed instruction.
- **Traps** (INT 0x30, ...)
old CS:EIP points **past** instruction causing exception
- **Faults** (page fault, GPF, ...)
old CS:EIP points **to** instruction causing exception
- **Aborts** (HW errors, bad system table vals...)
uncertain CS:EIP, serious problems, CPU confused

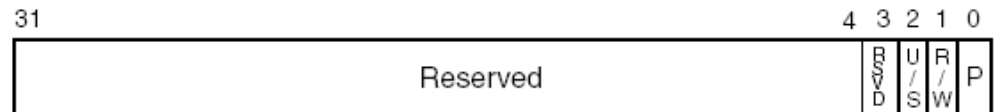
Example: Page Fault Exceptions

Why?

x86 Page Translation Mechanism encountered an error translating a linear address into a physical address.

Error Code

special error code format:



P	0 The fault was caused by a non-present page. 1 The fault was caused by a page-level protection violation.
W/R	0 The access causing the fault was a read. 1 The access causing the fault was a write.
U/S	0 The access causing the fault originated when the processor was executing in supervisor mode. 1 The access causing the fault originated when the processor was executing in user mode.
RSVD	0 The fault was not caused by reserved bit violation. 1 The fault was caused by reserved bits set to 1 in a page directory.

CR2 register

Linear Address that generated the exception.

Saved CS:EIP

Point to the instruction that generated the exception