CS5460/6460: Operating Systems

Lecture 6: Interrupts and Exceptions

Several slides in this lecture use slides developed by Don Porter

Anton Burtsev
January, 2014

Why do we need interrupts?

Remember:
hardware interface is designed to help OS

Why do we need interrupts?

- Fix an abnormal condition
  - Page not mapped in memory
- Notifications from external devices
  - Network packet received
- Preemptive scheduling
  - Timer interrupt
- Secure interface between OS and applications
  - System calls

Two types

Synchronous

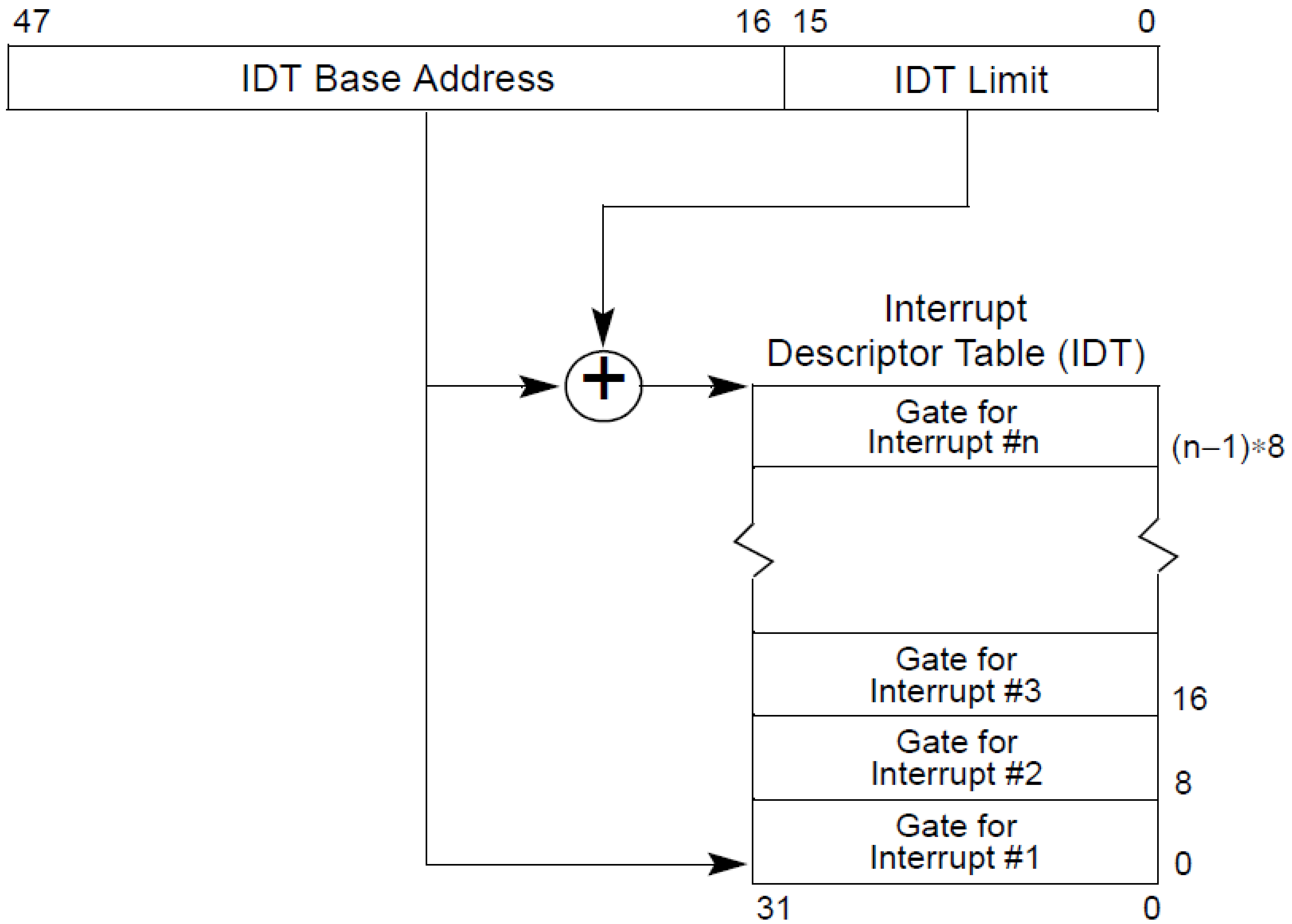
- Exceptions – react to an abnormal condition
  - Map the swapped out page back to memory
  - Invoke a system call
  - Intel distinguishes 3 types: faults, traps, aborts

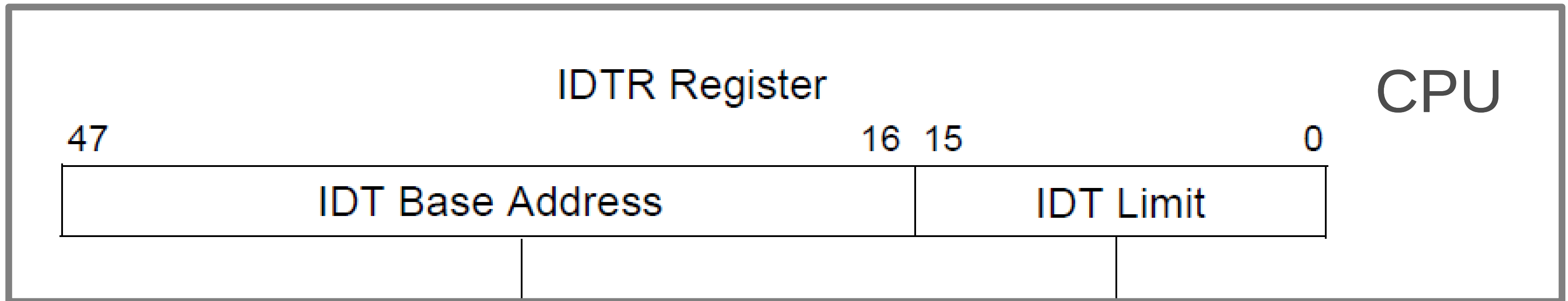
Asynchronous

- Interrupts – preempt normal execution
  - Notify that something has happened (new packet, disk I/O completed, timer tick, notification from another CPU)

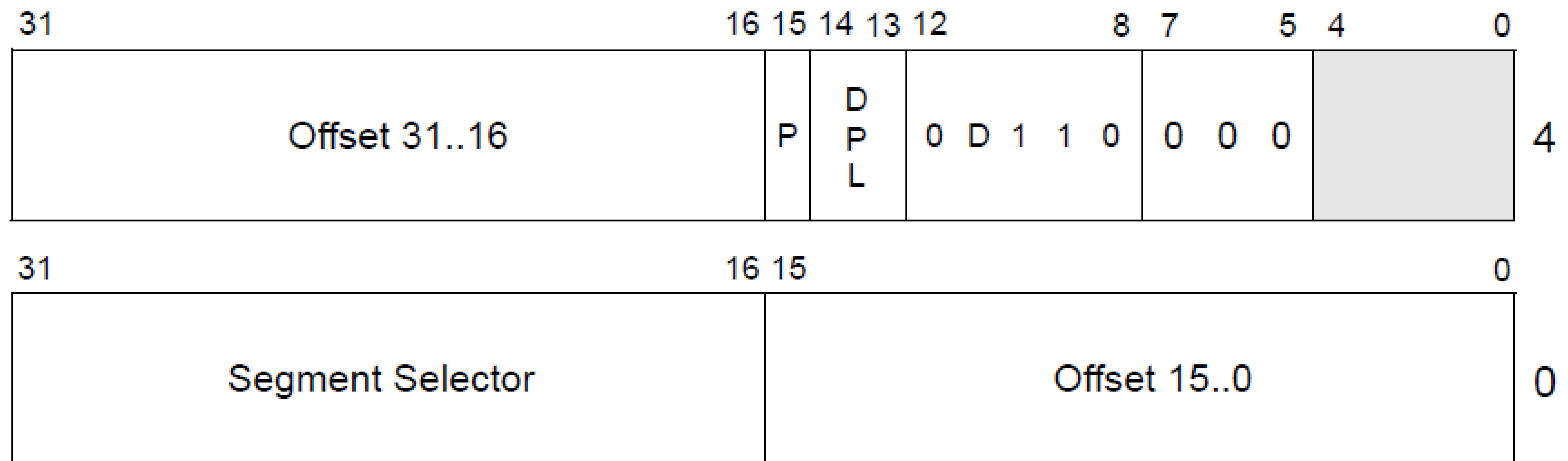
Handling interrupts and exceptions

- Same procedure
  - Stop execution of the current program
  - Start execution of a handler
  - Processor accesses the handler through an entry in the Interrupt Descriptor Table (IDT)
- Each interrupt is defined by a number
  - E.g., 14 is pagefault, 3 debug
  - This number is an index into the interrupt table (IDT)

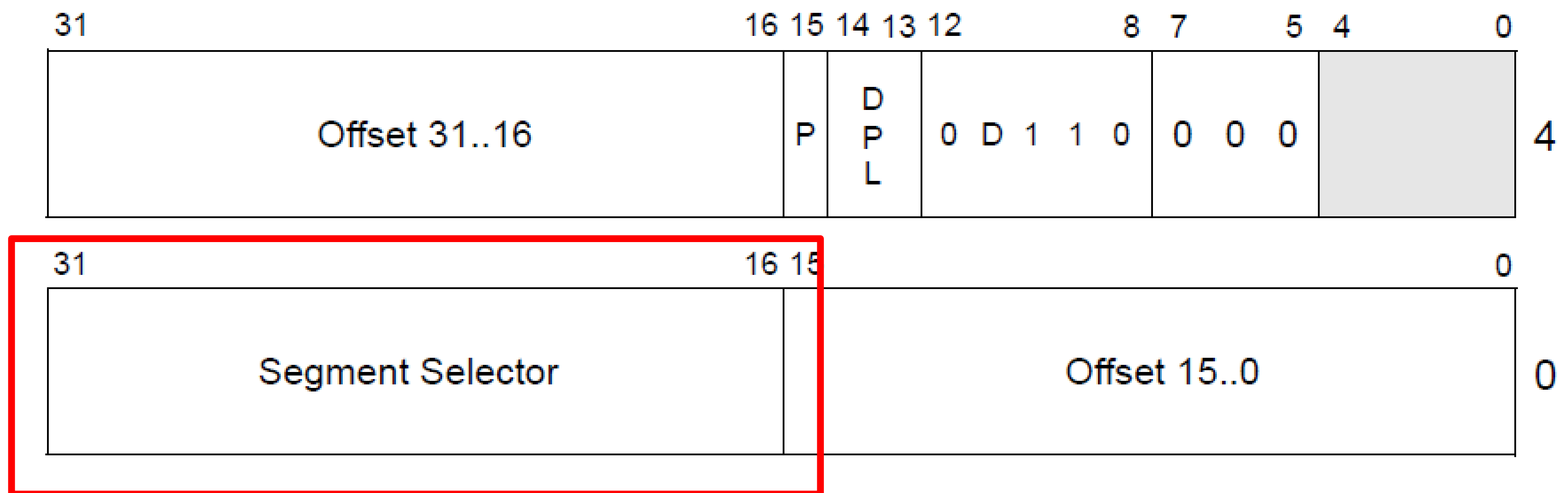

IDTR Register
47
16
15
0
IDT Base Address
IDT Limit
+
Interrupt
Descriptor Table (IDT)
Gate for
Interrupt #n
(n−1)*8
Gate for
Interrupt #3
16
Gate for
Interrupt #2
8
Gate for
Interrupt #1
0
31
0


IDTR Register
CPU
47
16 15
0
IDT Base Address
IDT Limit
Interrupt
Descriptor Table (IDT)
Gate for
Interrupt #n
(n−1)∗8
Gate for
Interrupt #3
16
Gate for
Interrupt #2
8
Gate for
Interrupt #1
0
31
0
Memory

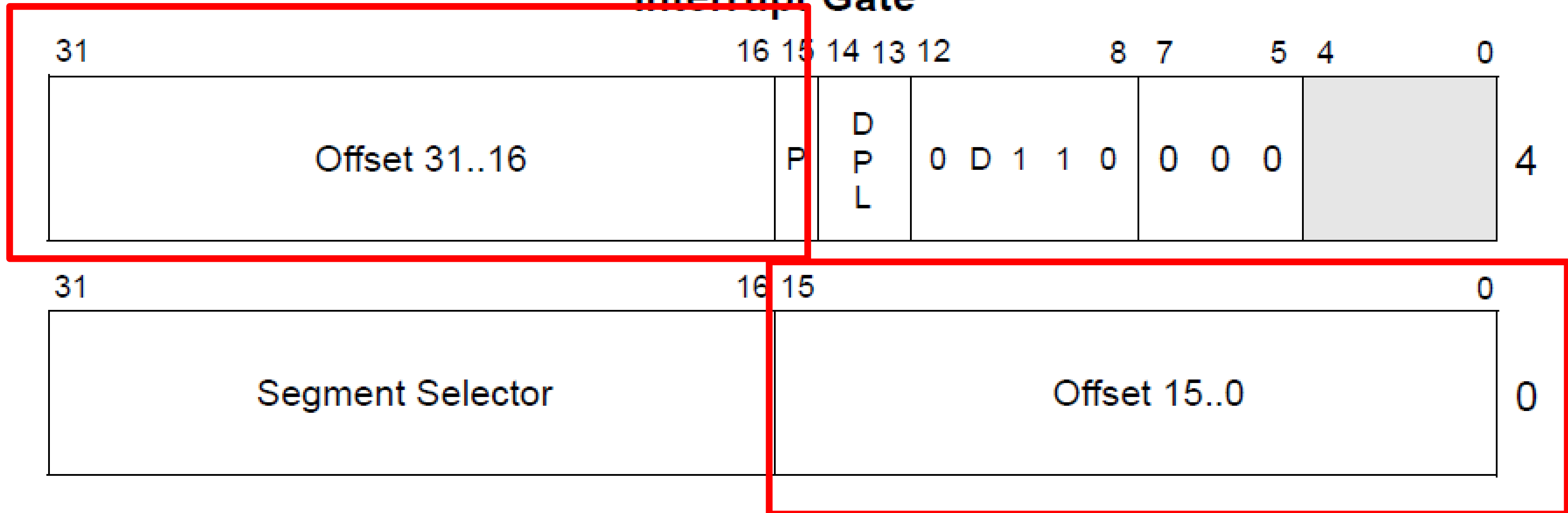
Interrupt descriptor

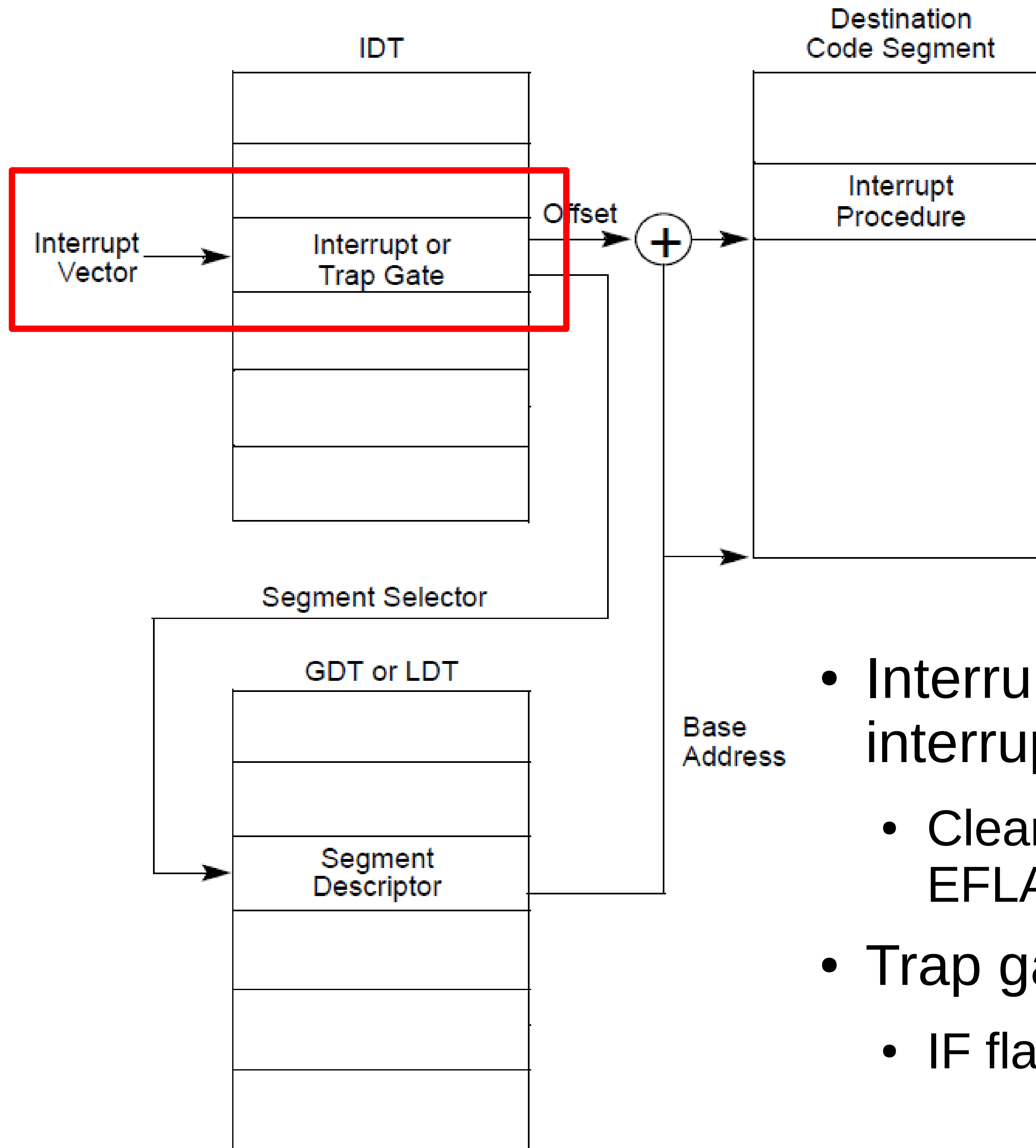


Interrupt descriptor

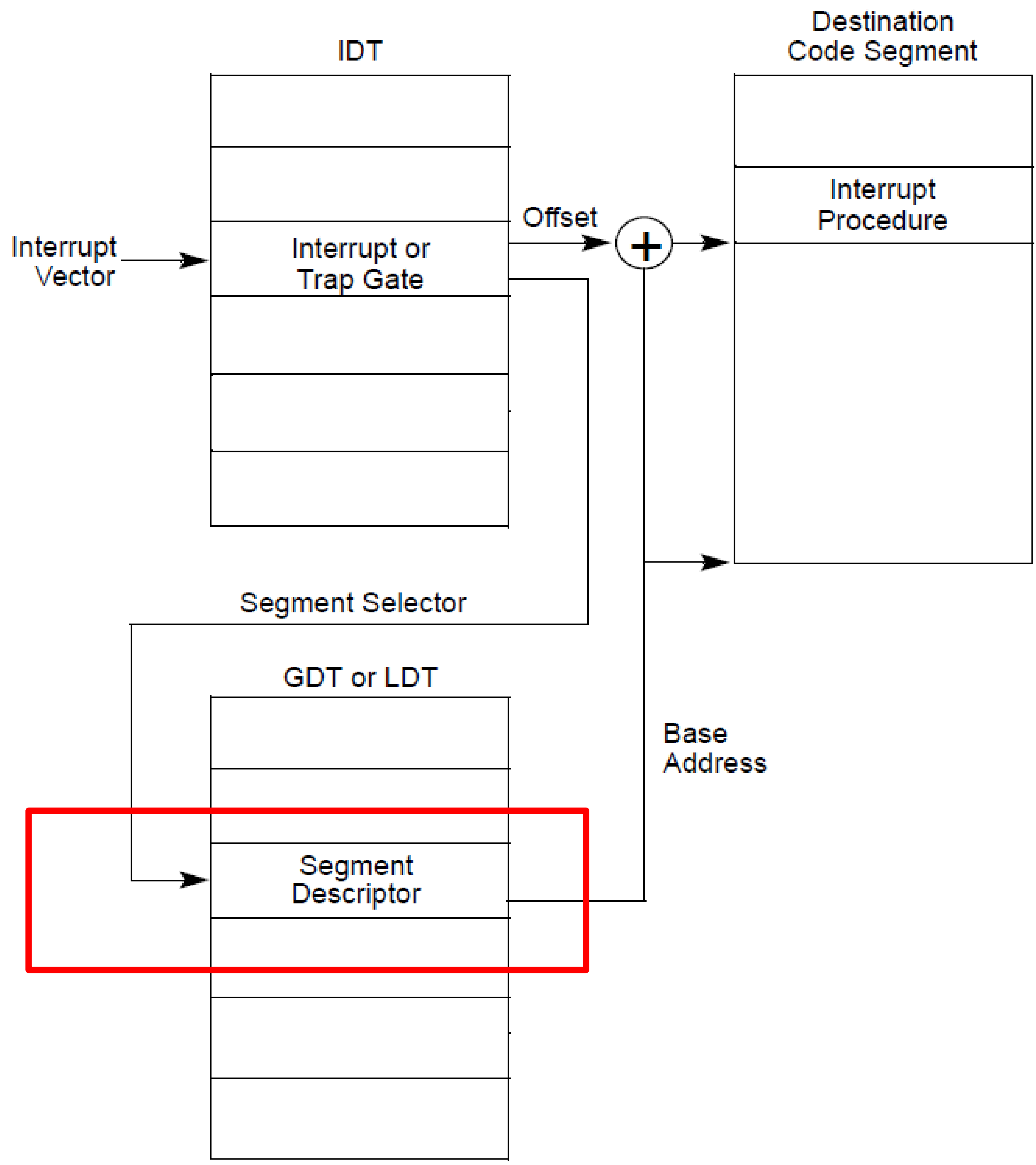


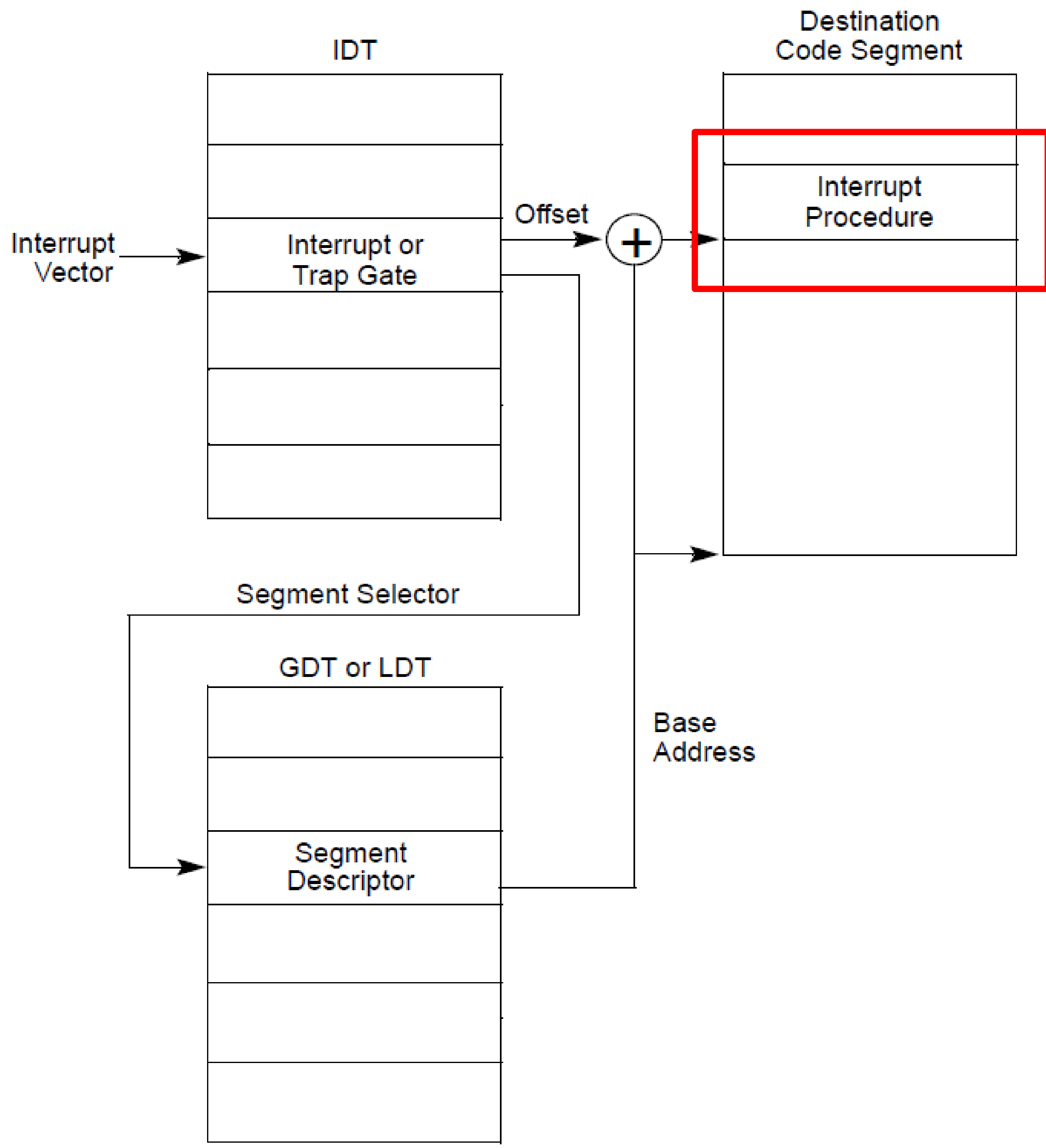
Interrupt descriptor



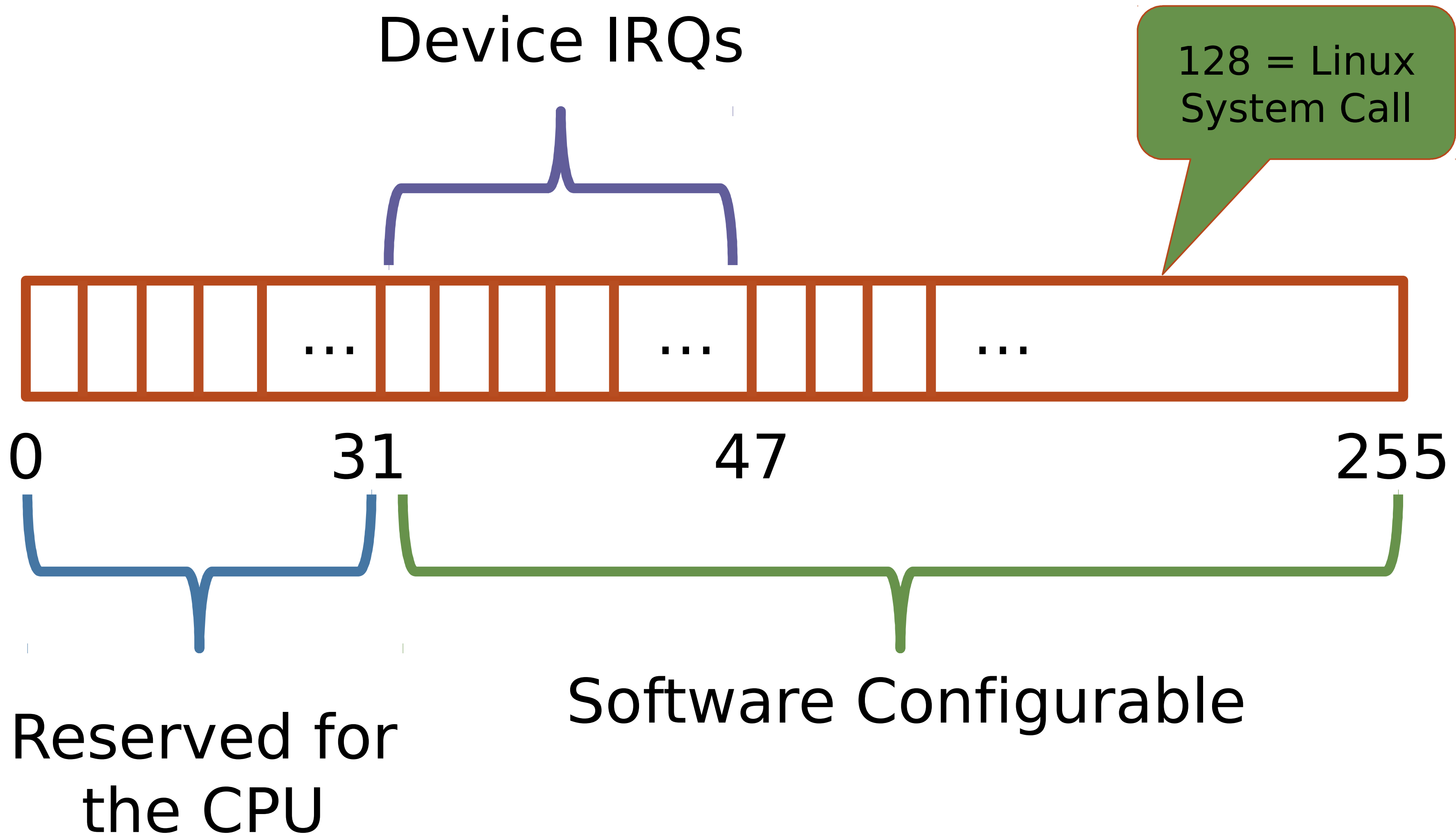


- Interrupt gate disables interrupts
  - Clears the IF flag in EFLAGS register
- Trap gate doesn't
  - IF flag is unchanged


IDT
Destination
Code Segment
Interrupt
Procedure
Offset
Interrupt
Vector
Interrupt or
Trap Gate
Segment Selector
GDT or LDT
Base
Address
Segment
Descriptor


IDT
Destination
Code Segment
Interrupt
Vector
Interrupt or
Trap Gate
Offset
Interrupt
Procedure
Segment Selector
GDT or LDT
Base
Address
Segment
Descriptor

x86 interrupt table



Interrupts

- Each type of interrupt is assigned an index from 0—255.
  - 0—31 are for processor interrupts fixed by Intel
  - E.g., 14 is always for page faults
- 32—255 are software configured
  - 32—47 are often for device interrupts (IRQs)
  - Most device’s IRQ line can be configured
  - Look up APICs for more info (Ch 4 of Bovet and Cesati)
  - 0x80 issues system call in Linux (more on this later)

Sources

- Interrupts
  - External
    - Through CPU pins connected to APIC
  - Software generated with INT n instruction
- Exceptions
  - Processor generated, when CPU detects an error in the program
    - Fault, trap, abort
  - Software generated with INTO, INT 3, BOUND

Software interrupts

- The INT n instruction allows software to raise an interrupt
  - 0x80 is just a Linux convention
  - You could change it to use 0x81!
- There are a lot of spare indexes

- OS sets ring level required to raise an interrupt
  - Generally, user programs can’t issue an `int 14` (page fault manually)
  - An unauthorized `int` instruction causes a general protection fault
  - Interrupt 13

Disabling interrupts

- Delivery of maskable interrupts can be disabled with IF (interrupt flag) in EFLAGS register
- Exceptions
  - Non-maskable interrupts (see next slide)
  - INT n – cannot be masked as it is synchronous

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Nonmaskable interrupts (NMI)

- Delivered even if IF is clear, e.g. interrupts disabled
  - CPU blocks subsequent NMI interrupts until IRET
- Sources
  - External hardware asserts the NMI pin
  - Processor receives a message on the system bus, or the APIC serial bus with NMI delivery mode
- Delivered via vector #2

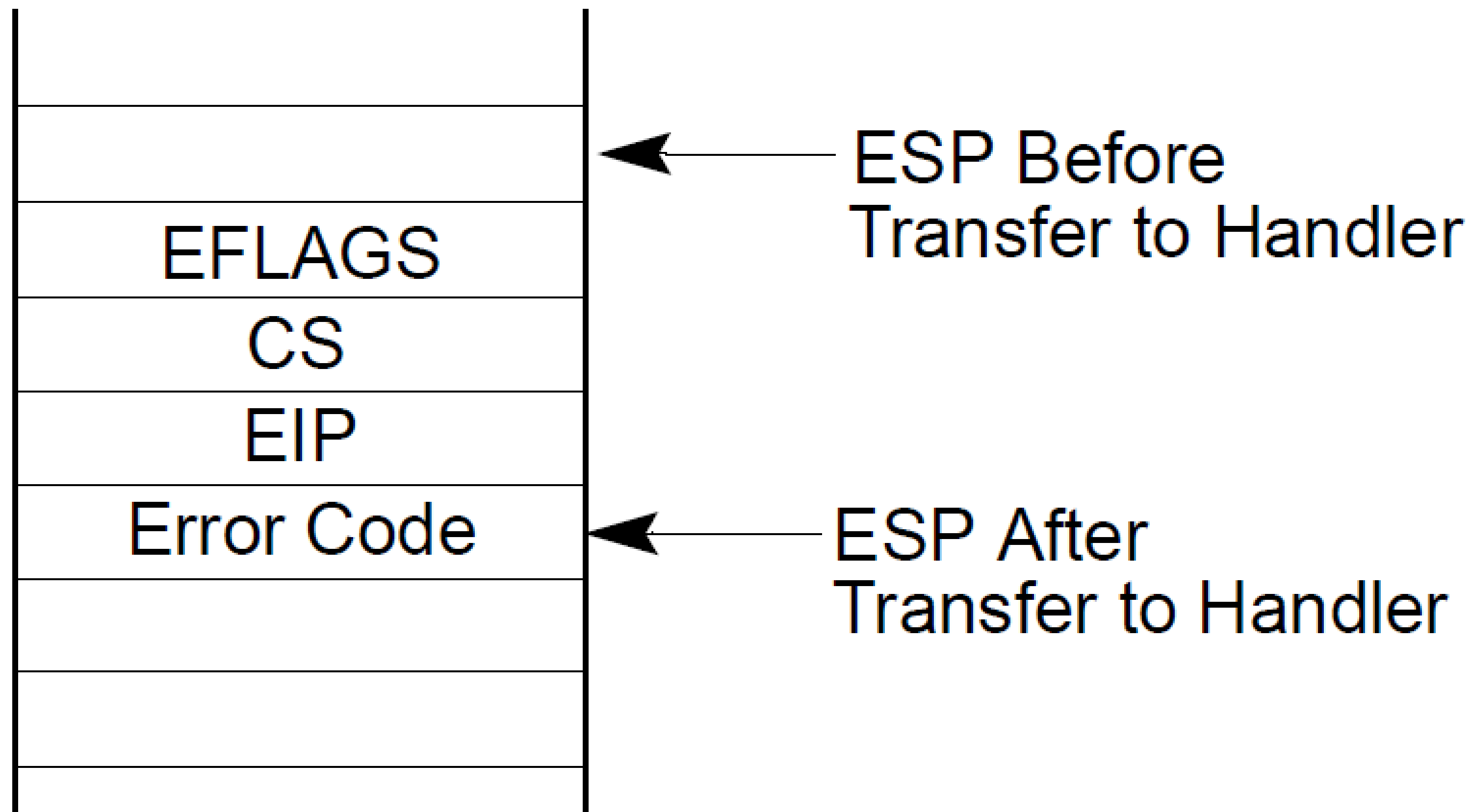
Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Processing of interrupt (same PL)

1. Push the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack
2. Push an error code (if appropriate) on the stack
3. Load the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers
4. If the call is through an interrupt gate, clear the IF flag in the EFLAGS register
5. Begin execution of the handler

Stack Usage with No Privilege-Level Change

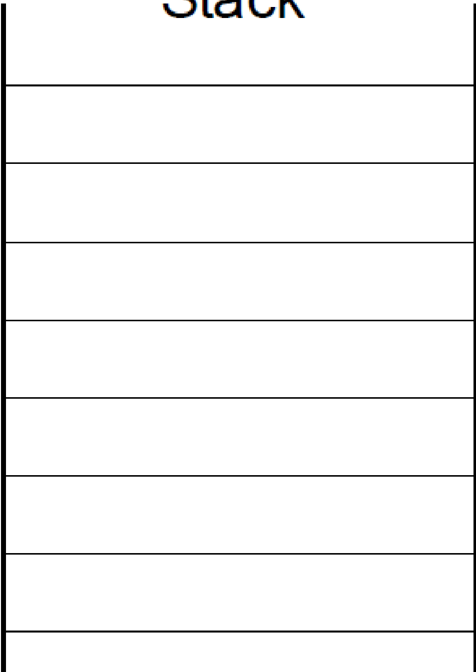
Interrupted Procedure's and Handler's Stack

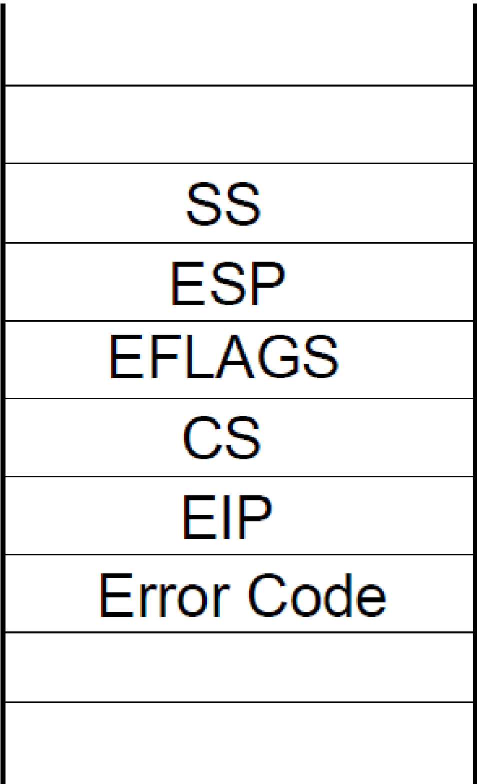


Processing of interrupt (cross PL)

1. Save ESP and SS in a CPU-internal register
2. Load SS and ESP from TSS
3. Push user SS, user ESP, user EFLAGS, user CS, user EIP onto new stack (kernel stack)
4. Set CS and EIP from IDT descriptor's segment selector and offset
5. If the call is through an interrupt gate clear some EFLAGS bits
6. Begin execution of a handler

Stack Usage with Privilege-Level Change





Task State Segment

- Another magic control block
  - Pointed to by special task register (TR)
    - Selector
    - Actually stored in the GDT
  - Hardware-specified layout
- Lots of fields for rarely-used features
- Two features we care about in a modern OS:
  - Location of kernel stack (fields SS/ESP)
  - I/O Port privileges (more in a later lecture)

Interrupt handlers

- Just plain old code in the kernel
- The IDT stores a pointer to the right handler routine

Return from an interrupt

- Starts with IRET

1. Restore the CS and EIP registers to their values prior to the interrupt or exception
2. Restore EFLAGS
3. Restore SS and ESP to their values prior to interrupt
   - This results in a stack switch
4. Resume execution of interrupted procedure

Short overview of QEMU

What needs to be emulated?

- CPU and memory
  - Register state
  - Memory state
- Memory management unit
  - Page tables, segments
- Platform
  - Interrupt controller, timer, buses
- BIOS
- Peripheral devices
  - Disk, network interface, serial line

Dis-
play

Disk

IO ports
(65536)

CPU

DMA

Main
loop

Inject exceptions;
Run guest;
Process exit;
Process I/O;

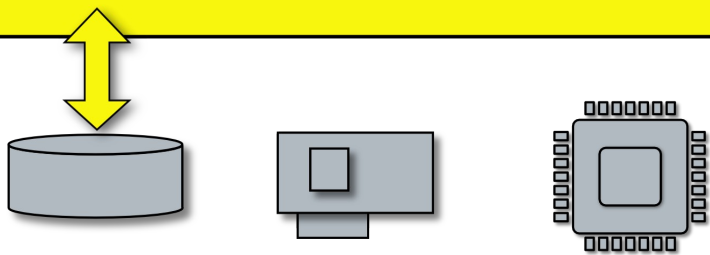
RAM

initrd

Kernel

File I/O

Qemu



x86 is not virtualizable

- Some instructions (*sensitive*) read or update the state of virtual machine and don't trap (*non-privileged*)
  - 17 sensitive, non-privileged instructions [Robin et al 2000]

x86 is not virtualizable (II)

Group	Instructions
Access to interrupt flag	pushf, popf, iret
Visibility into segment descriptors	lar, verr, verw, lsl
Segment manipulation instructions	pop <seg>, push <seg>, mov <seg>
Read-only access to privileged state	sgdt, sldt, sidt, smsw
Interrupt and gate instructions	fcall, longjump, retfar, str, int <n>

- Examples
  - `popf` updates interrupt flag (IF)
    - Impossible to detect when guest disables interrupts
  - `push %cs` can read code segment selector (%cs) and learn its CPL
    - Guest gets confused

Solution space

- Parse the instruction stream and detect all sensitive instructions dynamically
  - Interpretation (BOCHS, JSLinux)
  - Binary translation (VMWare, QEMU)
- Change the operating system
  - Paravirtualization (Xen, L4, Denali, Hyper-V)
- Make all sensitive instructions privileged!
  - Hardware supported virtualization (Xen, KVM, VMWare)
    - Intel VT-x, AMD SVM

Thank you.