Operating System Design and Implementation

Lecture 9: Interrupt & exception

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

Acknowledgements and Disclaimer

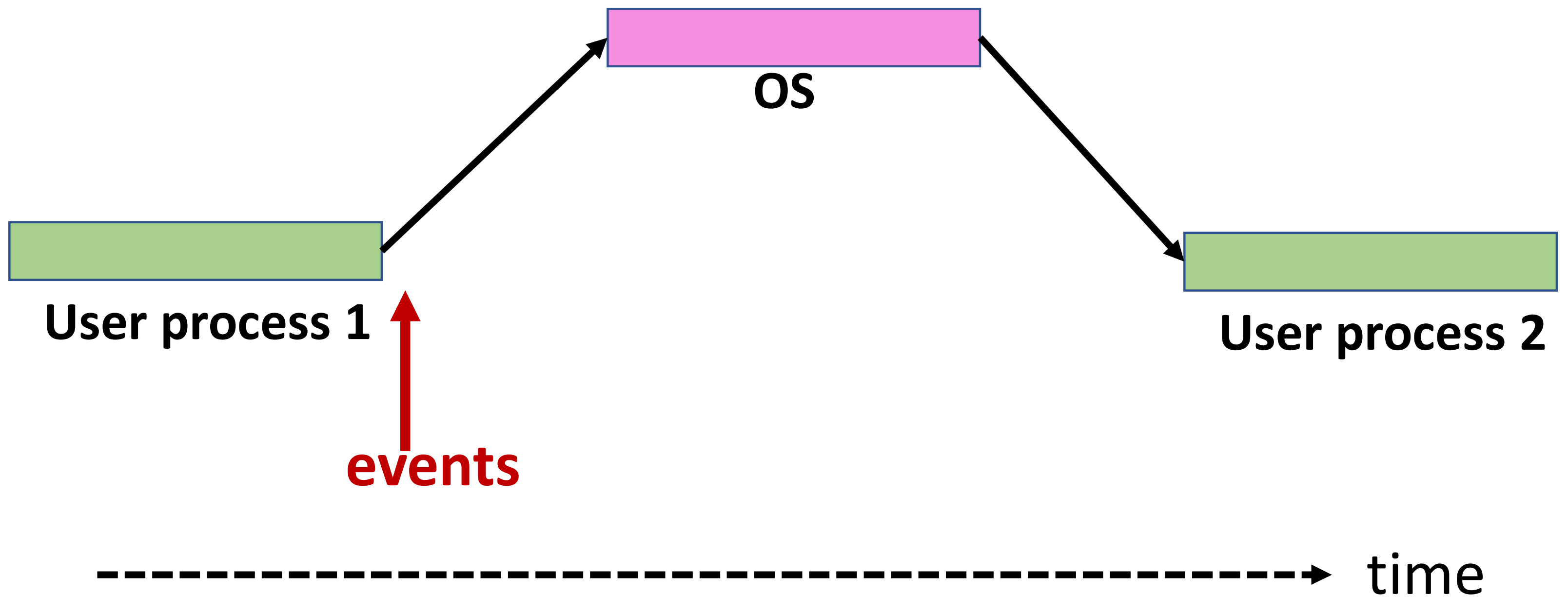
- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Interrupt
  - Hardware interrupt
  - Software interrupt
  - Programmable Interrupt Controller (PIC)
  - Interrupt vector
- Exception
- Interrupt handler
  - Top and bottom half (Linux)

Event driven OS

- OS is event driven
    - Executes only when there is an interrupt, trap, or system call



Why event driven design ?

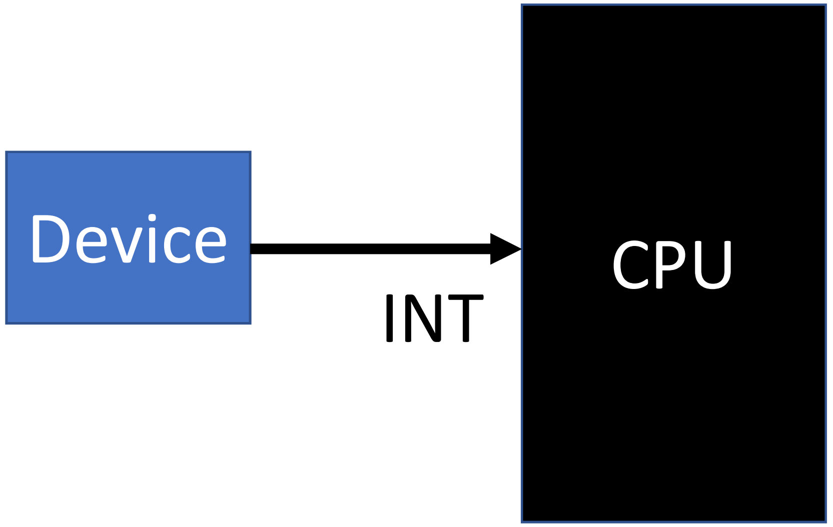
- OS cannot trust user processes
  - User processes may be buggy or malicious
  - User process crash should not affect OS
- OS needs to guarantee fairness to all user processes
  - One process cannot ‘hog’ CPU time
  - Timer interrupts

Interrupt

- **Interrupt**
  - An event that alters the sequence of instructions executed by a processor
  - Raised by hardware or programs to get OS attention
  - **Types**
    - **Hardware interrupts**: raised by external hardware
    - **Software interrupts**: raised by user programs
- **Exceptions**
  - Due to illegal operations

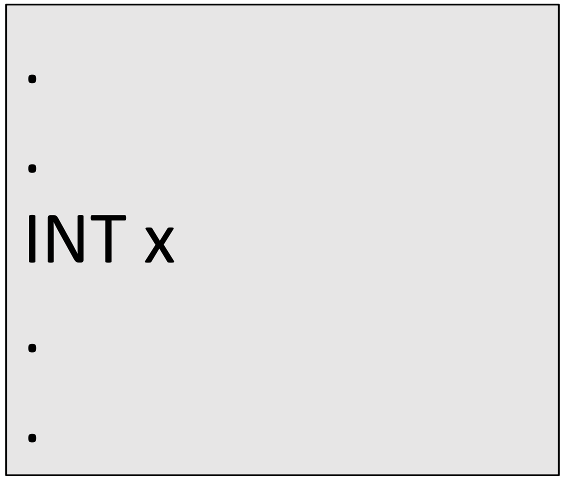
Interrupt

Hardware interrupt



A device (programmable interrupt controller (PIC)) asserts a pin in the CPU

Software interrupt



An executed instruction causes an interrupt

Why hardware interrupt ?

- Several devices connected to the CPU
  - E.g. keyboards, mouse, network card, etc.
- These devices occasionally need to be serviced by the CPU
  - For example: tell CPU that a key has be pressed
  - These events are asynchronous i.e. we cannot predict when they will happen
  - Need a way for the CPU to determine when a device needs attention

Interrupts

- Each device signals to the CPU that it wants to be serviced
- Generally CPUs have 2 pins
  - **INT: interrupt –** maskable interrupt, sent to the INTR pin of microprocessor
  - **NMI: Non maskable** – for very critical signal (NMI pin)
- Types
  - **Synchronous interrupt**
    - Produced by the CPU control unit while executing instructions
    - The control unit issues interrupts only after terminating the execution of an instruction
  - **Asynchronous interrupt**
    - Generated by other hardware devices at arbitrary times with respect to the CPU clock signals

Exception

- **Exception**
  - Generated when the CPU detects an anomalous condition while executing an instruction
- **Exception sources**
  - **Program-error exceptions**
    - Divide by zero
  - **Software generated exceptions**
    - INT 3: a break point exception
    - INTO: overflow instruction
  - **Machine-check exceptions**
    - Due to hardware error such as system bus error, parity errors in memory

Exception types -- Faults

- **Faults**
  - Exception that generally can be corrected
  - Once corrected, the program can continue execution
- **Examples:**
  - Divide by zero error
  - Invalid opcode
  - Device not available
  - Segment not present
  - Page not present

Exception types -- Traps

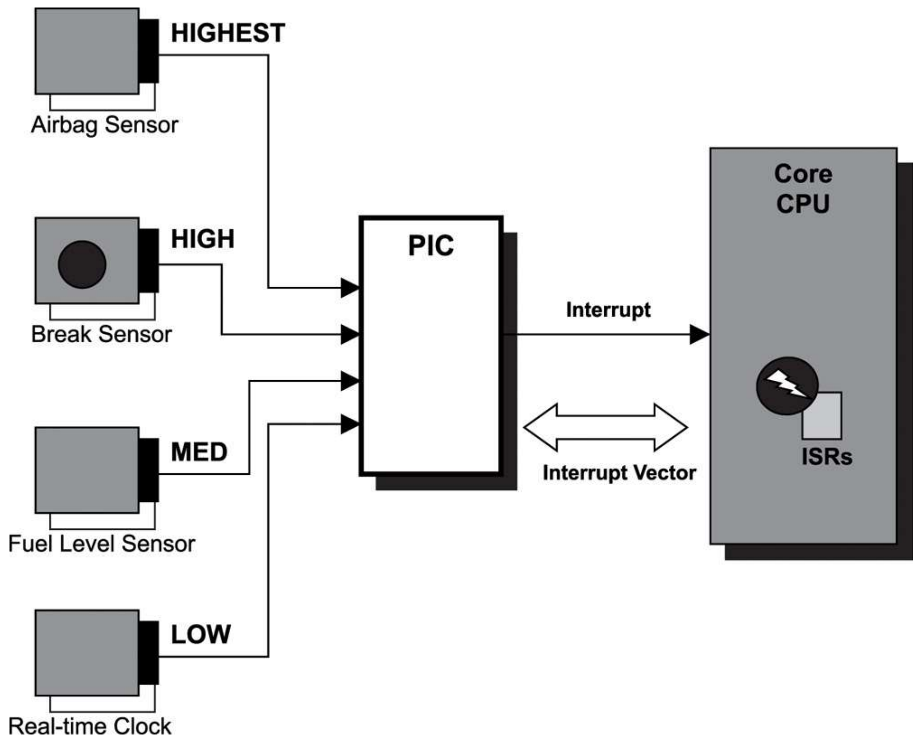
- **Traps**
  - Reported immediately after the execution of the trapping instruction
- **Examples:**
  - Breakpoint
  - Overflow
  - Debug instructions

Exception types -- Aborts

- **Aborts**
  - Severe unrecoverable errors
- **Examples**
  - **Double fault**
    - Occurs when an exception is unhandled or when an exception occurs while the CPU is trying to call an exception handler
  - **Machine check**
    - Internal errors in hardware detected
    - Such as bad memory, bus errors, cache errors, etc.

Exceptions and interrupts

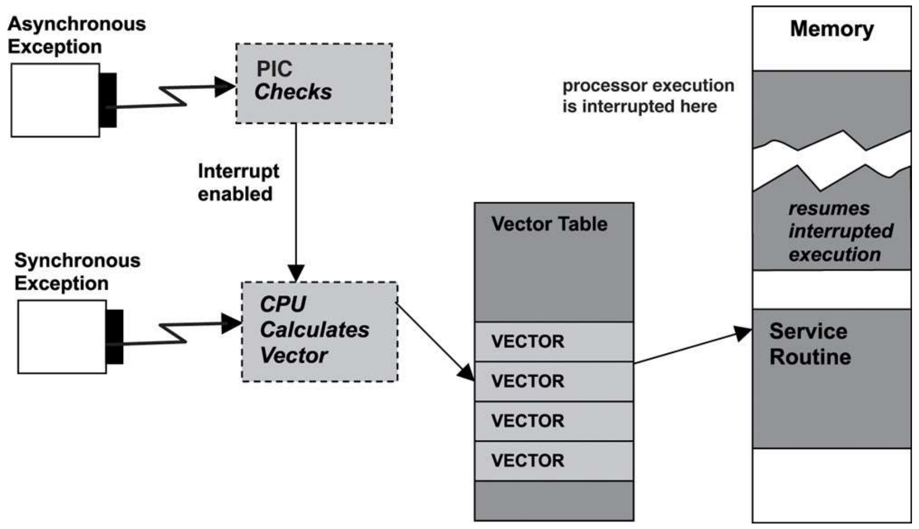
- How does exceptions and interrupts work ?
  - Programmable interrupt controller



Source: Qing Li “real-time concepts for embedded systems”

Exceptions and interrupts (Cont.)

- Processing general exceptions (Cont.)
  - Loading and invoking exception handlers

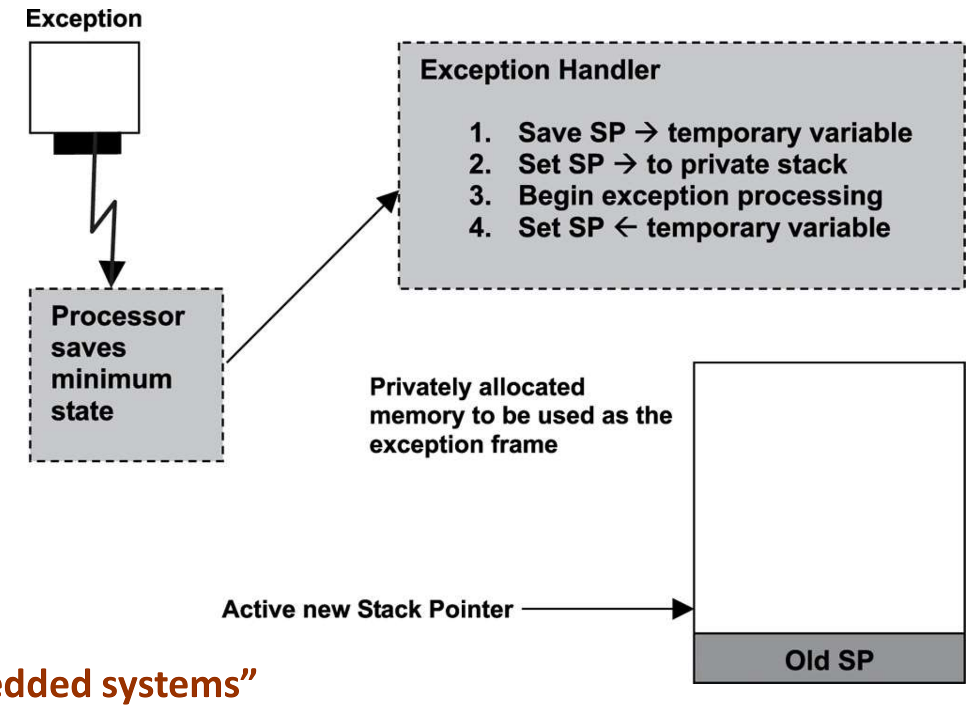


  - Nested exceptions and stack overflow

Source: Qing Li “real-time concepts for embedded systems”

Exceptions and interrupts (Cont.)

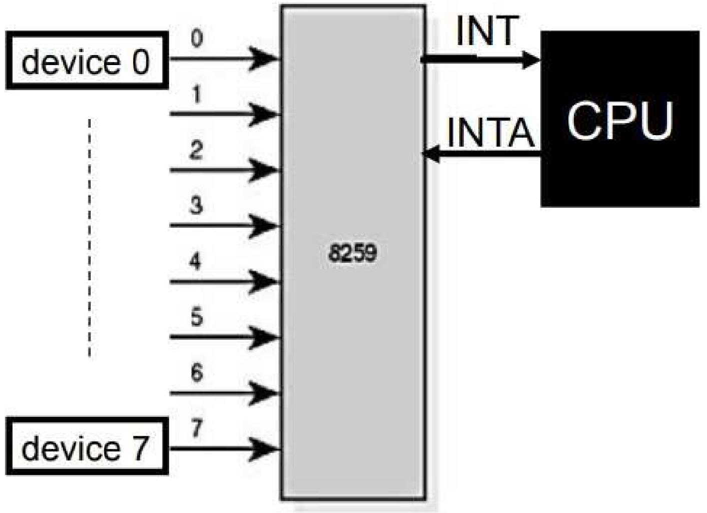
- Exception handlers
  - Exception frame
    - The exception frame is also called the interrupt stack in the context of asynchronous exceptions.


Exception
Exception Handler
1. Save SP → temporary variable
2. Set SP → to private stack
3. Begin exception processing
4. Set SP ← temporary variable
Processor saves minimum state
Privately allocated memory to be used as the exception frame
Active new Stack Pointer
Old SP

Source: Qing Li "real-time concepts for embedded systems"

8259 Programmable interrupt controller (PIC)

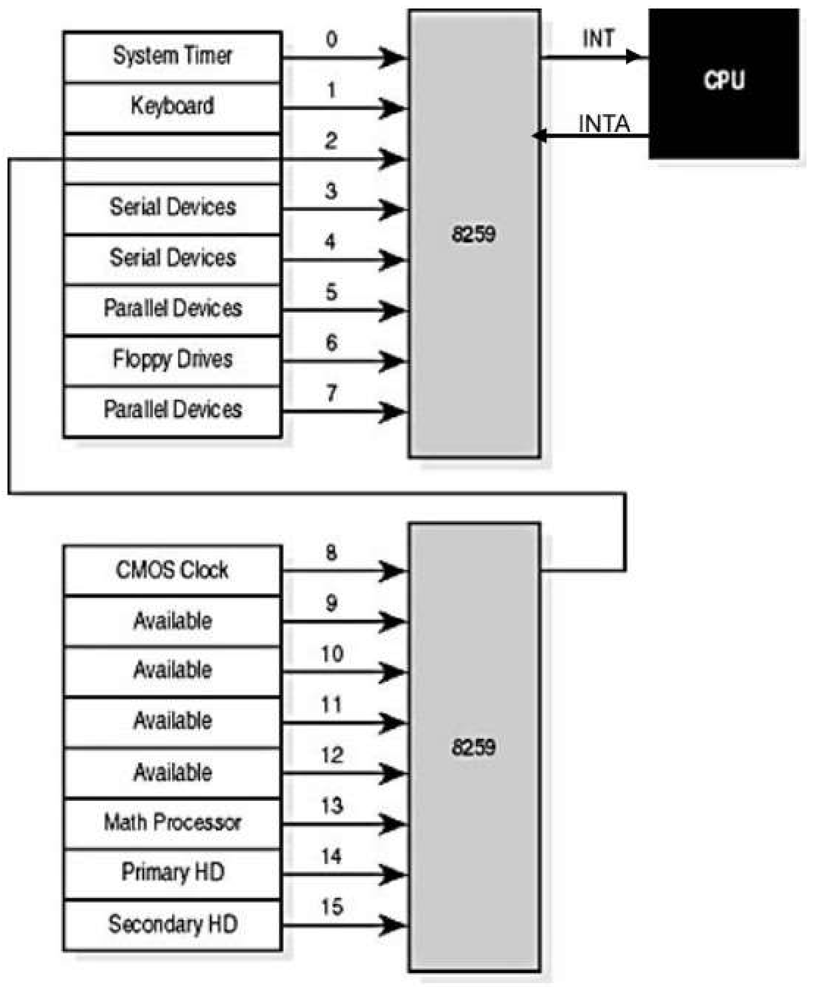
- 8259 PIC relays up to 8 interrupt to the CPU
  - Devices raise interrupts by an ‘interrupt request’ (IRQ)
  - CPU acknowledges and queries the 8259 to determine which device interrupted
  - Priorities can be assigned to each IRQ line
  - 8259s can be cascaded to support more interrupts
    - Two PICs and cascade buffer
    - IRQ2 -> IRQ9



INTA is a signal used to identify that a CPU has an interrupt made by the interrupt controller.

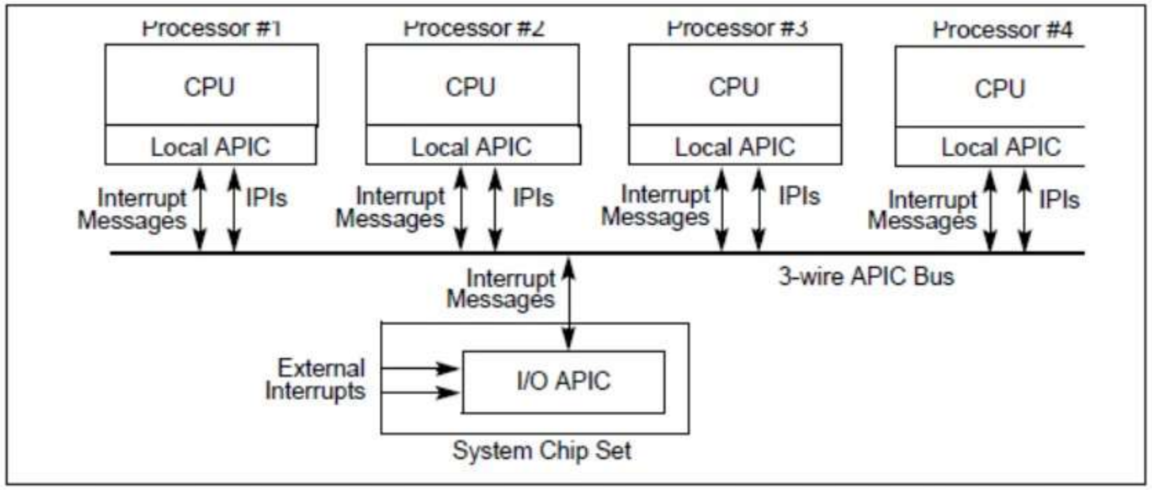
Interrupts in legacy CPUs

- IRQ 0 to IRQ 15, 15 possible devices
- Interrupt types
  - Edge
  - Level
- Limitations
  - Limited IRQs
  - Multi-processor support limited



Advanced PIC (APIC)

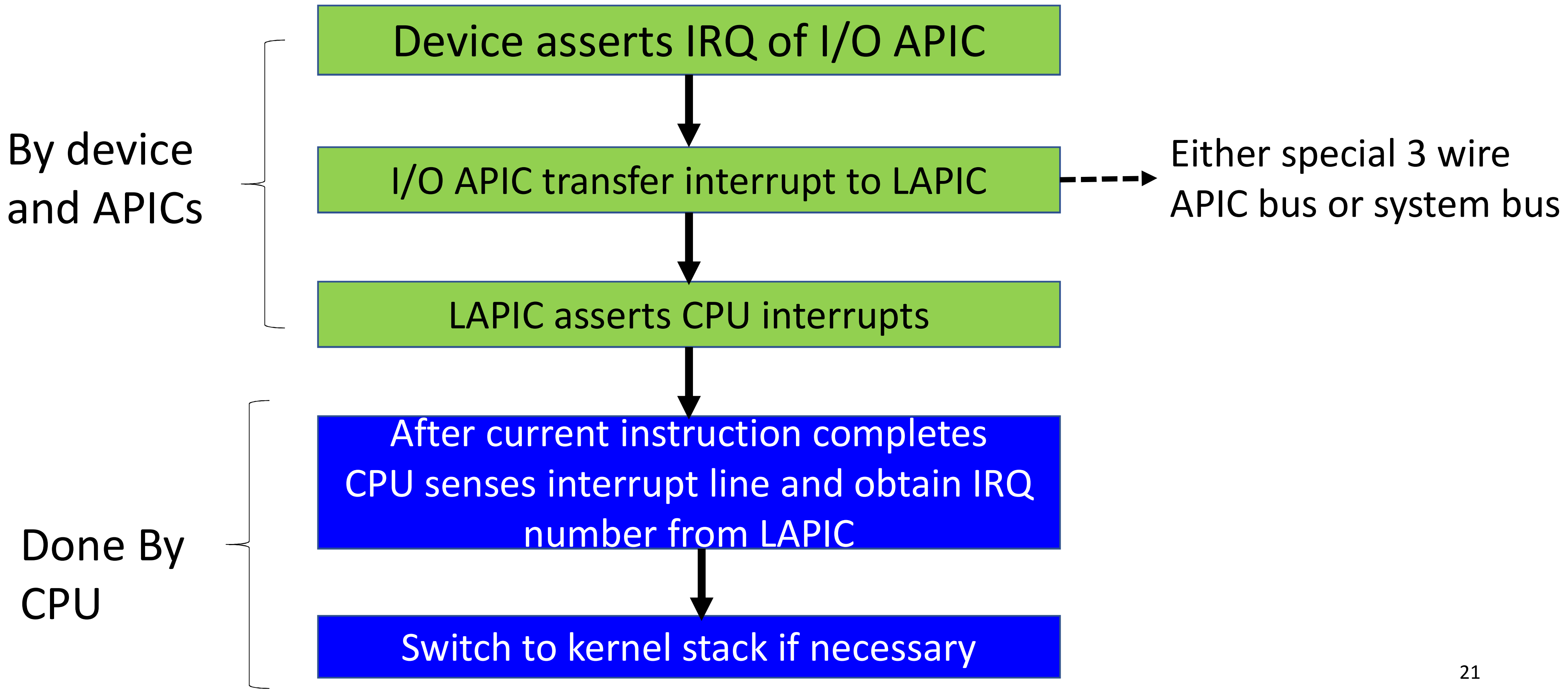
- External interrupts are routed from peripherals to CPUs in multi-processor systems through APIC
- APIC distributes and prioritizes interrupts to processors
- Comprises of two components
  - Local APIC (LAPIC)
  - I/O APIC
- APICs communicates through a special 3-wire APIC bus



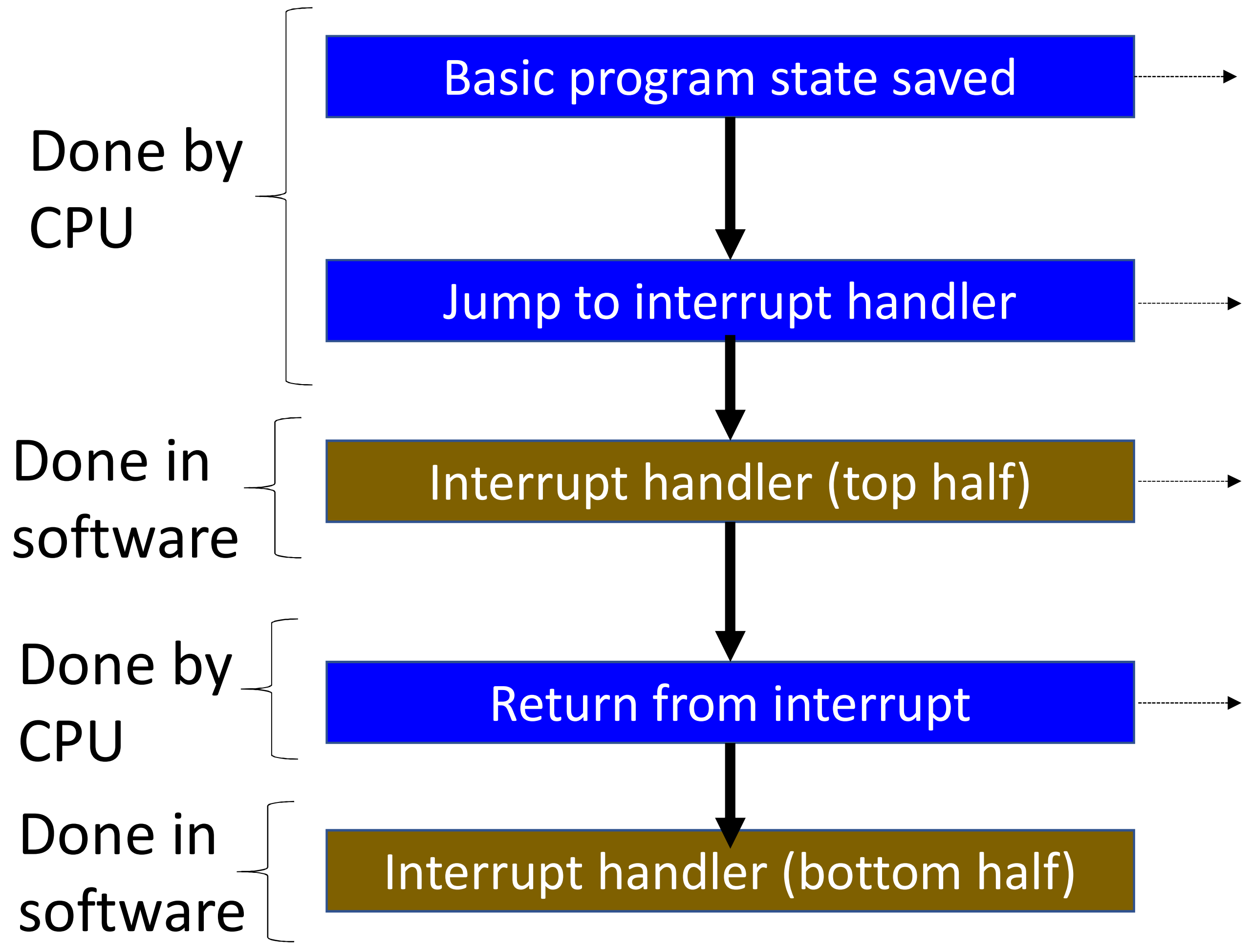
LAPIC and I/O APIC

- **LAPIC**
  - Receives interrupts from I/O APIC and routes it to the local CPU
  - Can also receive local interrupts such as from thermal sensor, internal timer, etc.
  - Send and receive IPIs (Inter processor interrupts)
    - IPIs used to distribute interrupts between processors or execute system wide functions like booting, load distribution, etc.
- **I/O APIC**
  - Present in chipset (north bridge)
  - Used to route external interrupts to local APIC

What happens when there is an interrupt ?


By device and APICs
Device asserts IRQ of I/O APIC
I/O APIC transfer interrupt to LAPIC
Either special 3 wire APIC bus or system bus
LAPIC asserts CPU interrupts
Done By CPU
After current instruction completes CPU senses interrupt line and obtain IRQ number from LAPIC
Switch to kernel stack if necessary

What happens when there is an interrupt ?


Done by CPU
Basic program state saved
Jump to interrupt handler
Done in software
Interrupt handler (top half)
Done by CPU
Return from interrupt
Done in software
Interrupt handler (bottom half)

x86 saves the SS, ESP, EFLAGS, CS, EIP, error code on stack
(restored by **iret** instruction)
suspends current task

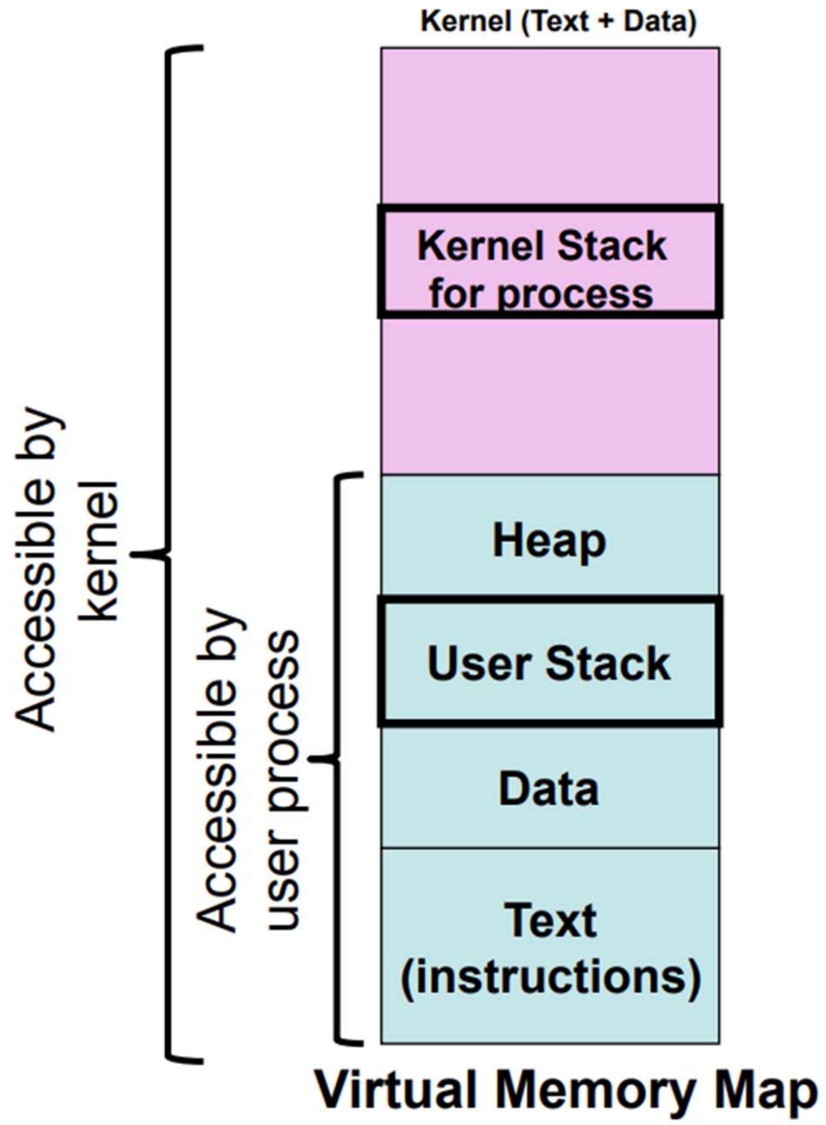
How does hardware find the OS interrupt handler ?

1. Respond to interrupt
2. More storing of program state
3. Schedule bottom half
4. IRET

Restore flags and registers saved earlier. Restore running task

Stack

- Each process has two stacks
  - A user space stack
  - A kernel space stack



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/6_Interrupts.pdf

Switch stack (to switch or not to switch)

- **When to switch stack ?**
  - If a process executes in user space -> switch stack to a kernel switch
  - No stack switch when executing in kernel space
- **Why switch stack ?**
  - OS cannot trust stack (SS and ESP) of user process
  - Stack switch needed **only when moving from user to kernel mode**
- **How to switch stack ?**
  - CPU should know locations of the new SS and ESP
  - Done by **task segment descriptor**

How to switch stack ?

- **Task state segment (TSS)**
  - Specialized segment for hardware support for multitasking
  - **TSS stored in memory**
    - Processor register states -> used for task switching
    - I/O port permission bitmap -> specifies individual ports to accessible program
    - Inner-level stack pointer -> specifies the new stack pointer when a privilege level change occurs
    - Previous TSS link
  - **TSS is used to find the new stack**
    - SS0: the stack segment (in kernel)
    - ESP0: stack pointer (in kernel)

Saving program state

- Why does the OS need to saving program state ?
  - Current program being executed must be able to resume after interrupt service completed
- When stack switch occurs
  - Also save the previous SS and ESP
- When no stack switch occurs
  - Using existing stack

Interrupt vectors

- When exception or interrupt occur, what to execute next ?
  - Each interrupt/exception provided a number
  - Number used to index into an **interrupt descriptor table (IDT)**
  - IDT provides the entry point into an interrupt/exception handler
  - 0 to 255 vectors possible
    - **0 to 31** correspond to **exception and nonmaskable interrupts**
    - **32 – 47** are assigned to **maskable interrupts caused by IRQs**
    - **48 – 255** may be used to identify **software interrupts**
    - For example, Linux uses 128 (0x80) vector to implement system calls
    - When int 0x80 assembly instruction is executed by a process in user mode, the CPU switches into kernel mode and starts executing the system_call() kernel function

Interrupt vector (cont.)

- Processor generated exception

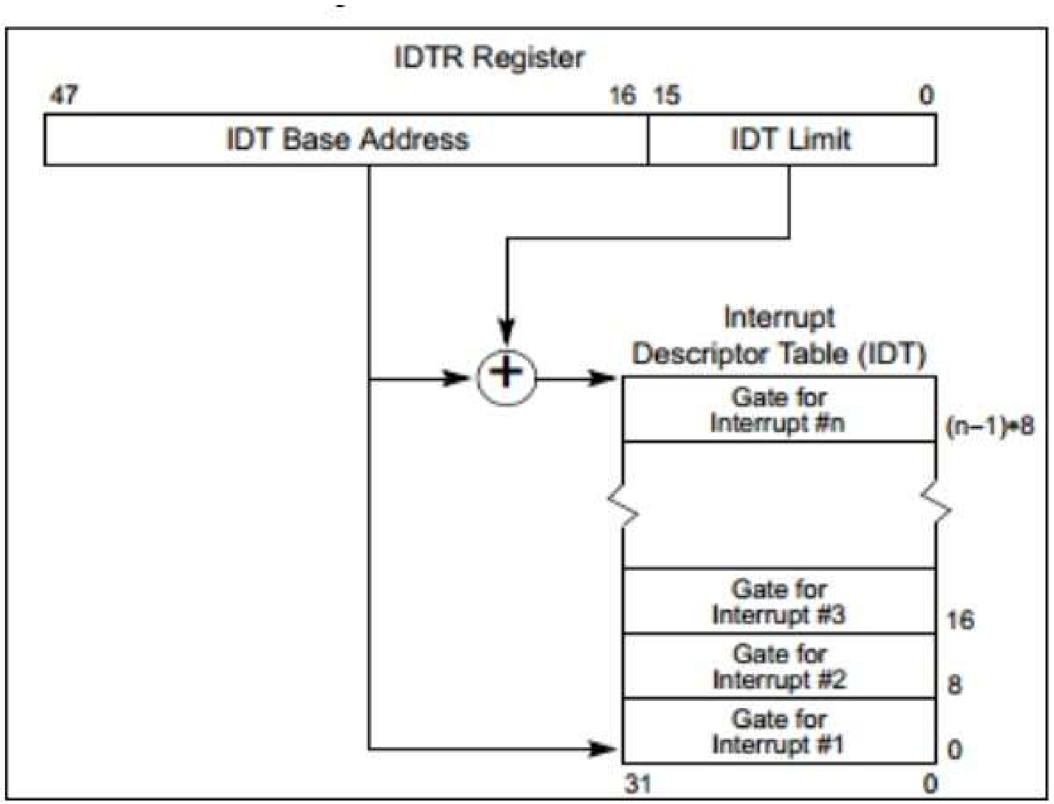
INT_NUM	Short description
0x00	Division by zero
0x01	Single-step interrupt
0x02	NMI
0x03	Breakpoint
0x04	Overflow
0x05	Bound range exceeded
0x06	Invalid opcode

https://en.wikipedia.org/wiki/Interrupt_descriptor_table

Finding the interrupt service routine

- **IDT: interrupt descriptor table**
  - Also called interrupt vectors
  - Stored in memory and pointed to by IDTR (register)
  - Initialized by OS at boot

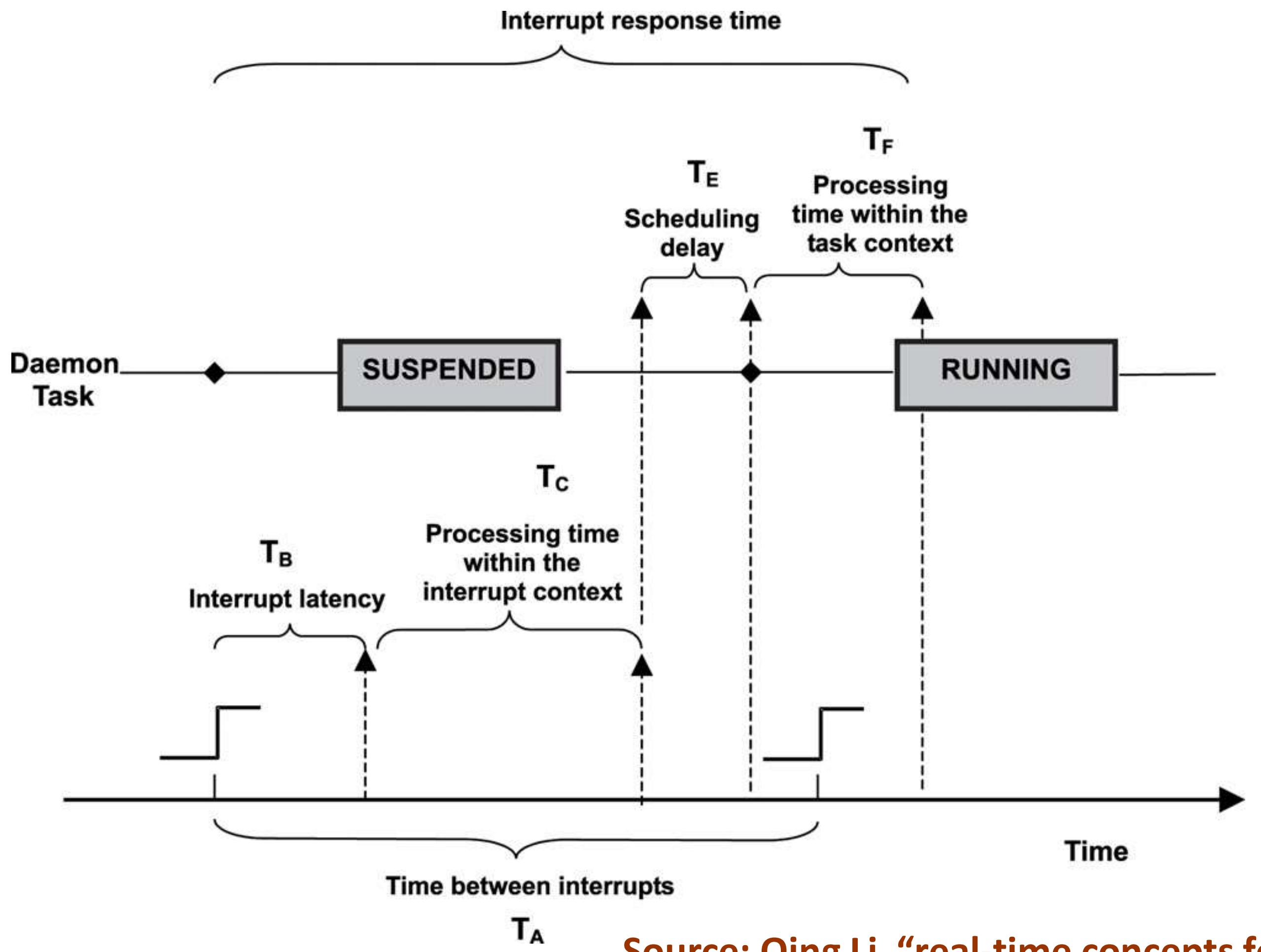
Selected descriptor =
base address + (vector * 8)
// 8 is the entry size in protected mode



Interrupt handlers

- **Typical interrupt handler**
  - Save additional CPU context (written in assembly)
  - Process interrupt (communicate with I/O devices)
  - Invoke kernel scheduler
  - Restore CPU context and return (written in assembly)

Interrupt latency

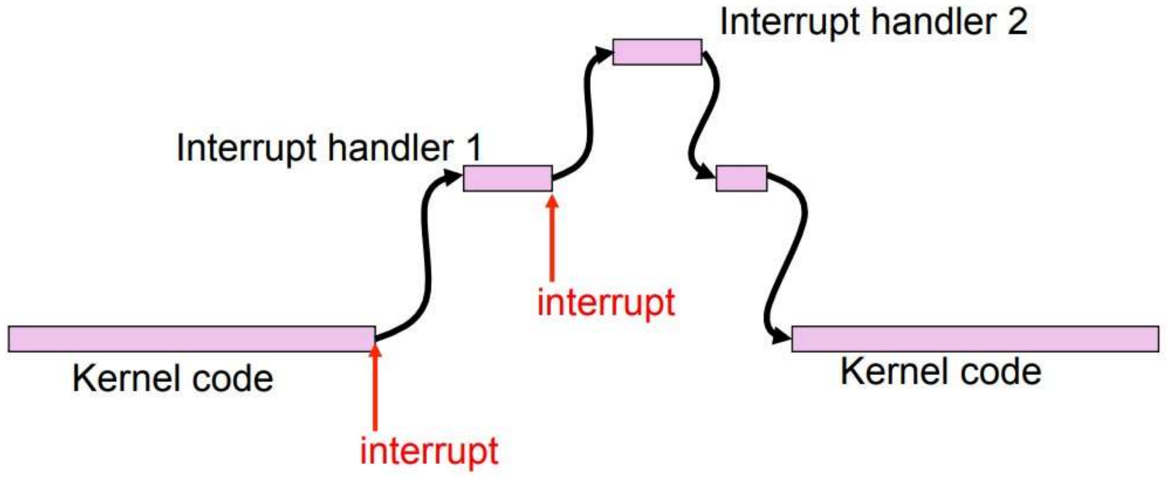


Source: Qing Li “real-time concepts for embedded systems”

Importance of interrupt latency

- **Real time systems**
  - OS should guarantee interrupt latency is less than a specified value
- **Minimum interrupt latency**
  - Mostly due to the interrupt controller
- **Maximum interrupt latency**
  - Due to the OS
  - Occurs when interrupt handler cannot serviced immediately
  - E.g. when OS executing atomic operations, interrupt handler would need to wait till completion of atomic operations

Nested interrupts



- Typically interrupt disabled during handler executes
  - This reduces system responsiveness
- To improve responsiveness, enable interrupts within handlers
  - This often causes nested interrupts but difficult to develop and validate
- Linux interrupt handler approach
  - Design interrupt handlers to be small so that nested interrupts are less likely

Small interrupt handlers

- Do as little as possible in the interrupt handler
  - Often just queue a work item or set a flag
  - Defer non-critical actions till later
- Top and bottom half technique (Linux)
  - **Top half:** do minimum work and return from interrupt handler
    - Should complete as quickly as possible since all interrupts are disabled
  - **Bottom half:** deferred processing
    - Implemented in Linux as softirqs, tasklets or workqueues

Softirqs

- Softirqs are a form of bottom half processing
  - The softirqs handlers are executed with all interrupts enabled
  - A given softirq handler can run simultaneously on multi-CPUs
  - Softirqs are executed once all interrupt handlers have completed before the kernel resumes scheduling processes
  - The number of softirqs is fixed in the system, are not directly used by drivers, but by kernel subsystems
  - This list of softirqs is defined in “include/linux/interrupt.h”

Tasklets

- **Tasklets**
  - Executed withing the **HI_SOFTIRQ** and **TASKLET_SOFTIRQ** softirqs
  - Executed with all interrupts enabled
  - A given tasklet is guaranteed to execute on a single CPU at a time
  - Created with the tasklet_init() function
  - A tasklet is simply implemented as a function
  - Tasklets can easily be used by individual device drivers
  - The interrupt handler can schedule tasklet execution with:
    - Tasklet_schedule() to get it executed in TASKLET_SOFTIRQ
    - Tasklet_hi_schedule() to get it executed in HI_SOFTIRQ (high priority)

Workqueues

- **Workqueue**
  - Aims to defer work
  - Not limited in usage to handle interrupts
  - Usually allocated in a per-device structure
  - Registered with INIT_WORK()
  - Triggered with queue_work() when using a dedicated queue
  - The complete API in “include/linux/workqueue.h”
  - Example (drivers/crypto/atmel-i2c)
    - INIT_WORK(&work_data->work, atmel_i2c_work_handler);
    - schedule_work(&work_data->work)

Summary

- Interrupt changes the sequence of instruction execution
- Exception occurs since the illegal operation
- Hardware interrupt – programmable interrupt controller
- Interrupt vector records interrupt commands
- Interrupt latency
- Interrupt handler – top half and bottom half