CS 423
Operating System Design: The Kernel Abstraction

Jongyul Kim

* Thanks for Prof. Adam Bates for the slides.

What We will Learn Today

Interrupt

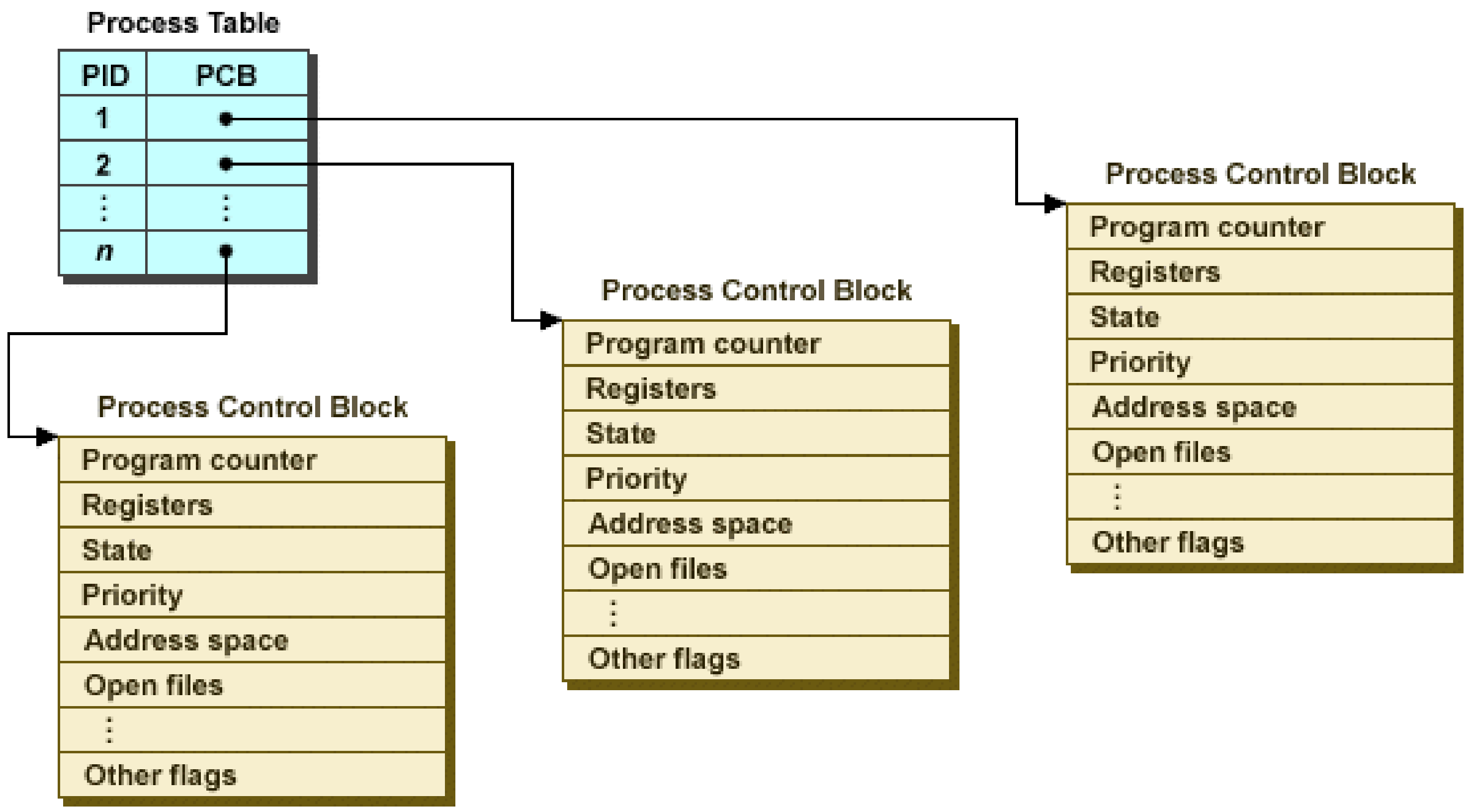
- Basic Interrupt Mechanism
- Hardware / Software Interrupts
- Interrupt Handlers
- Bottom halves
    - Bottom halves, Softirqs, Tasklets, Work queues

Discussion: Last Class

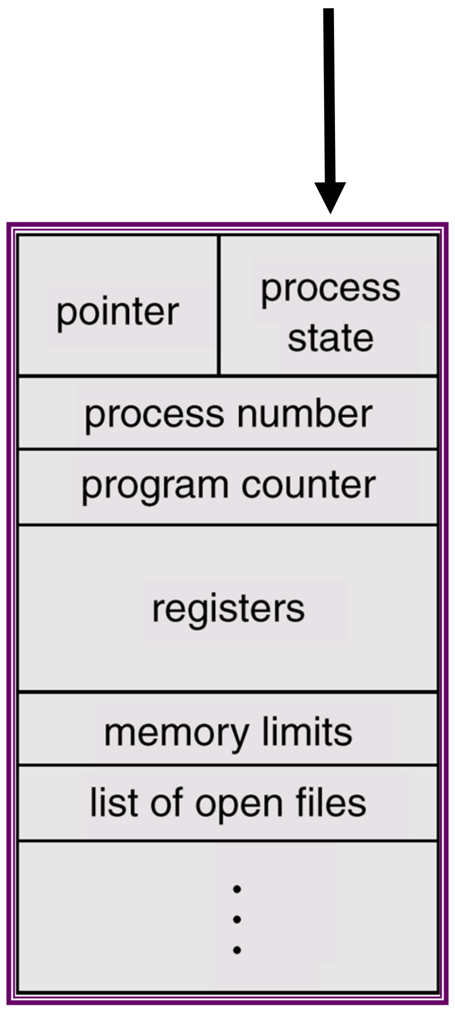
- Where is CPU State physically stored for active task?
  - Registers!
    - Program Counter is a register
    - Segment Registers
      - Code Segment
      - Data Segment
      - Stack Segment
- CPU has access to RAM and can save PC to stack before context switching.

Process Control Block

The state for processes that are not running on the CPU are maintained in the Process Control Block (PCB) data structure



Updated during context switch

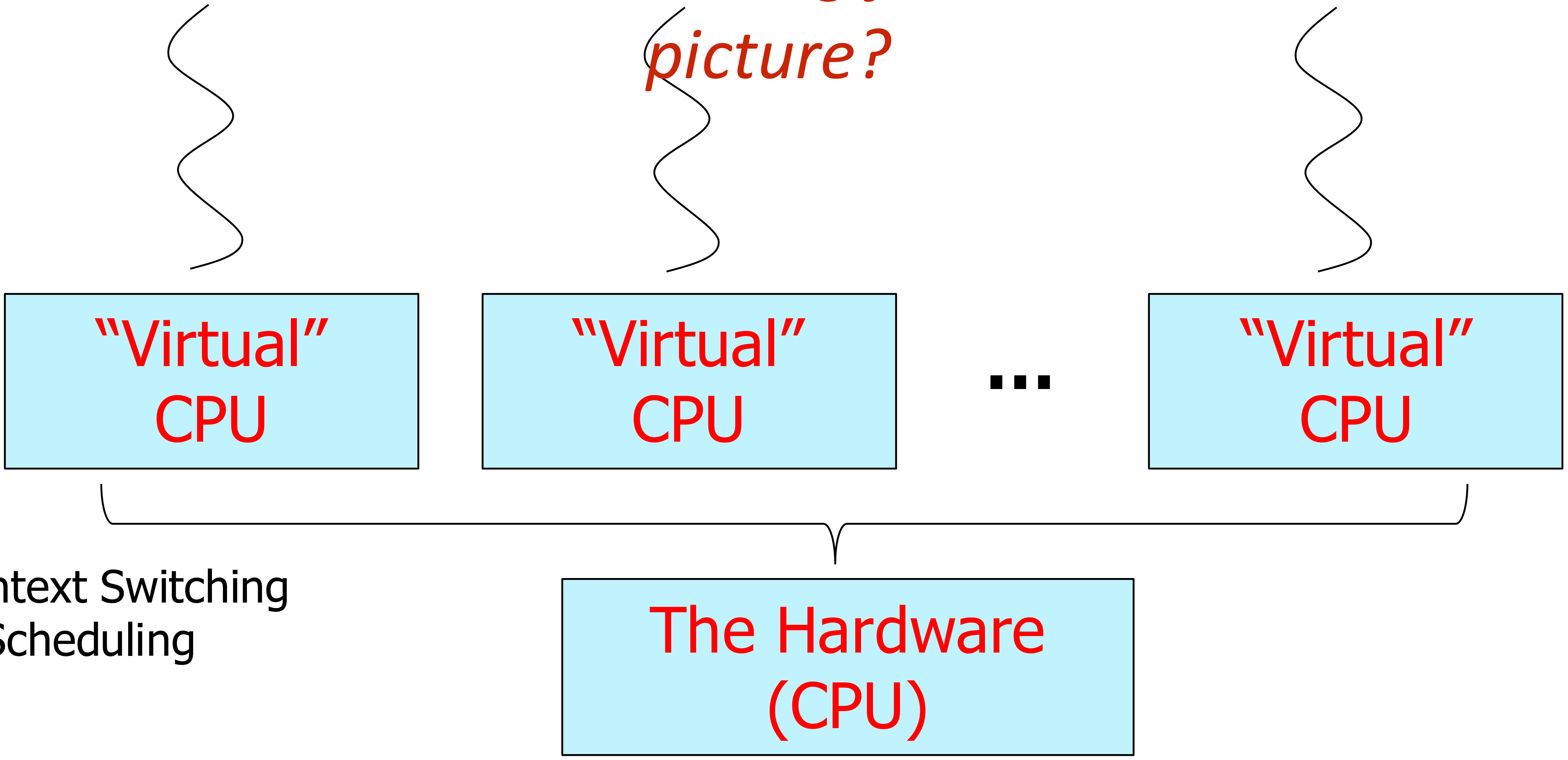


An alternate PCB diagram

Where We Are:

Last class, we discussed how context switches allow a single CPU to handle multiple tasks:

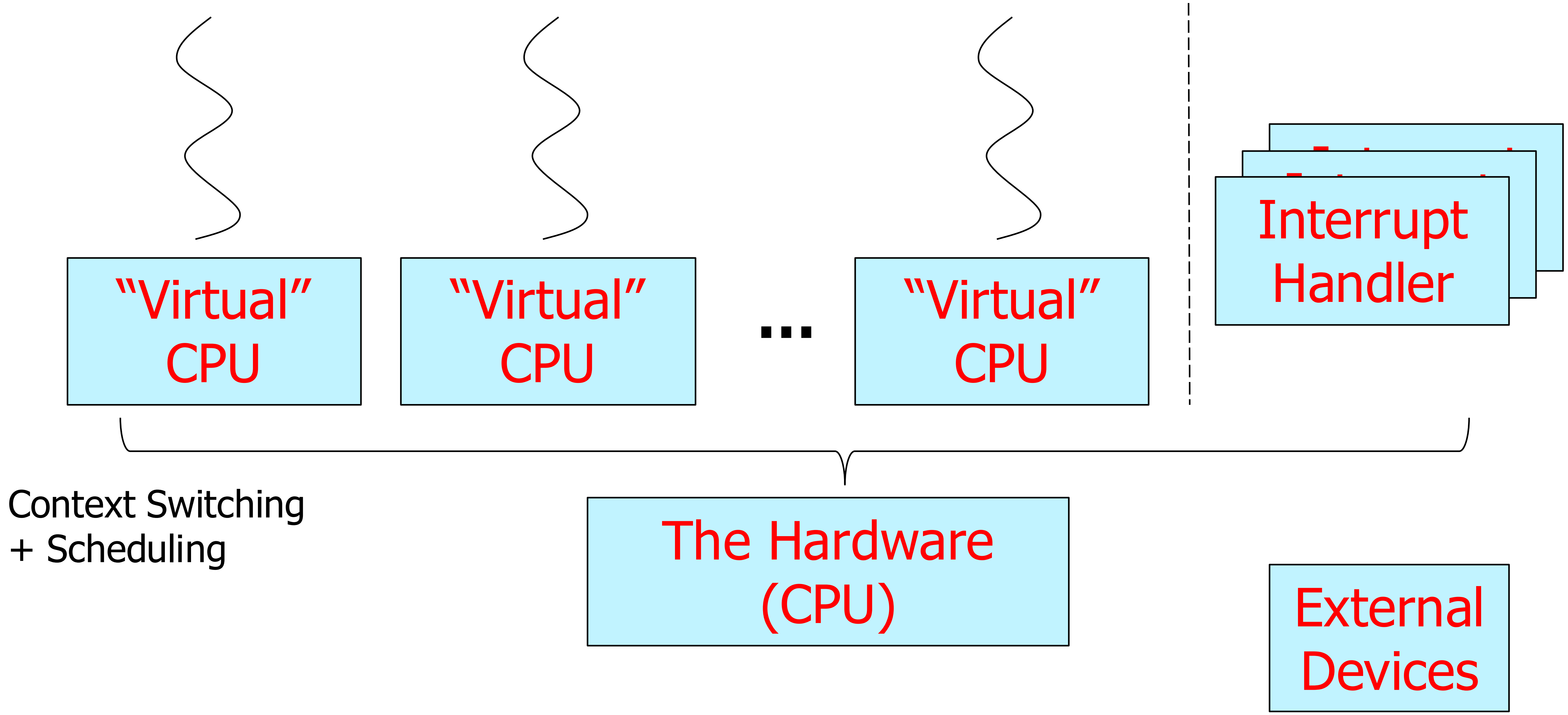
What's missing from this picture?



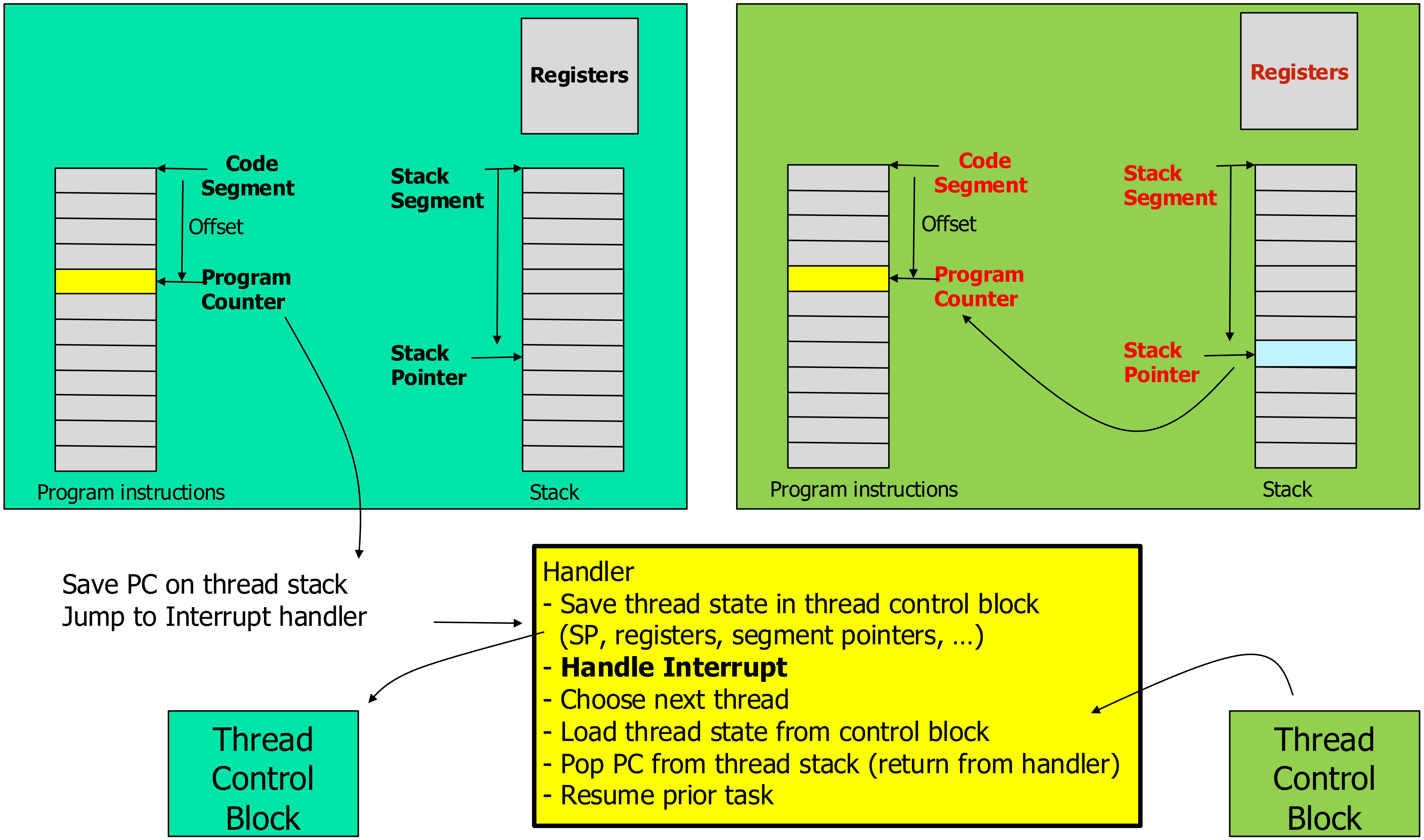
Where We Are:

Interrupts to drive scheduling decisions!

Interrupt handlers are also tasks that share the CPU.



CTX Switch: Interrupt



Save PC on thread stack
Jump to Interrupt handler

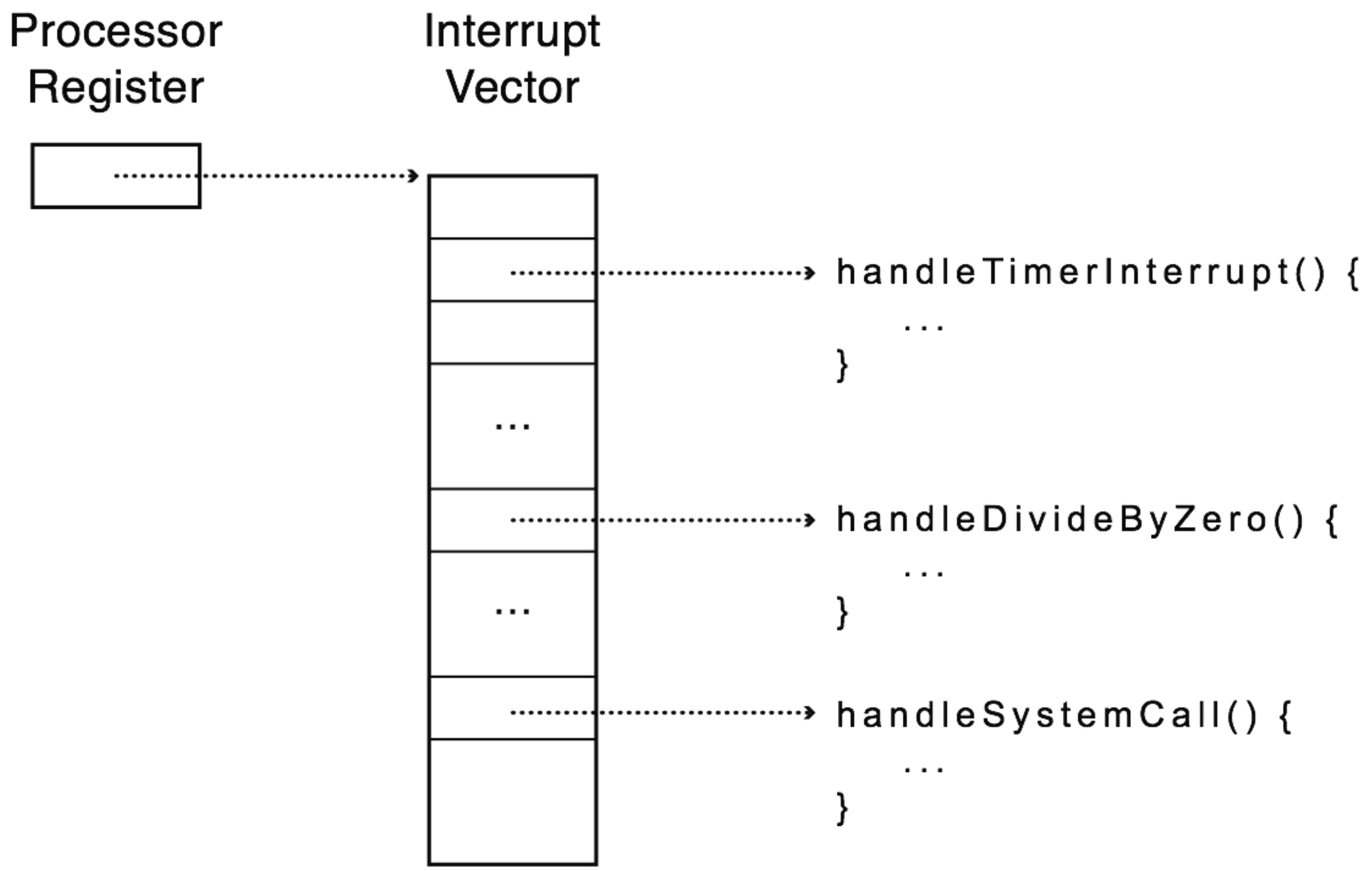
Handler
- Save thread state in thread control block
  (SP, registers, segment pointers, …)
- **Handle Interrupt**
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)
- Resume prior task

How do we take interrupts safely??

- **Interrupt Vector Table**
  - Where the processor looks for a handler
  - Limited number of entry points into kernel
  - Stored in RAM at a known address
- **Atomic transfer of control**
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- **Transparent restartable execution**
  - User program does not know interrupt occurred

Interrupt Vector Table

Table set up by OS kernel; pointers to code to run on different events

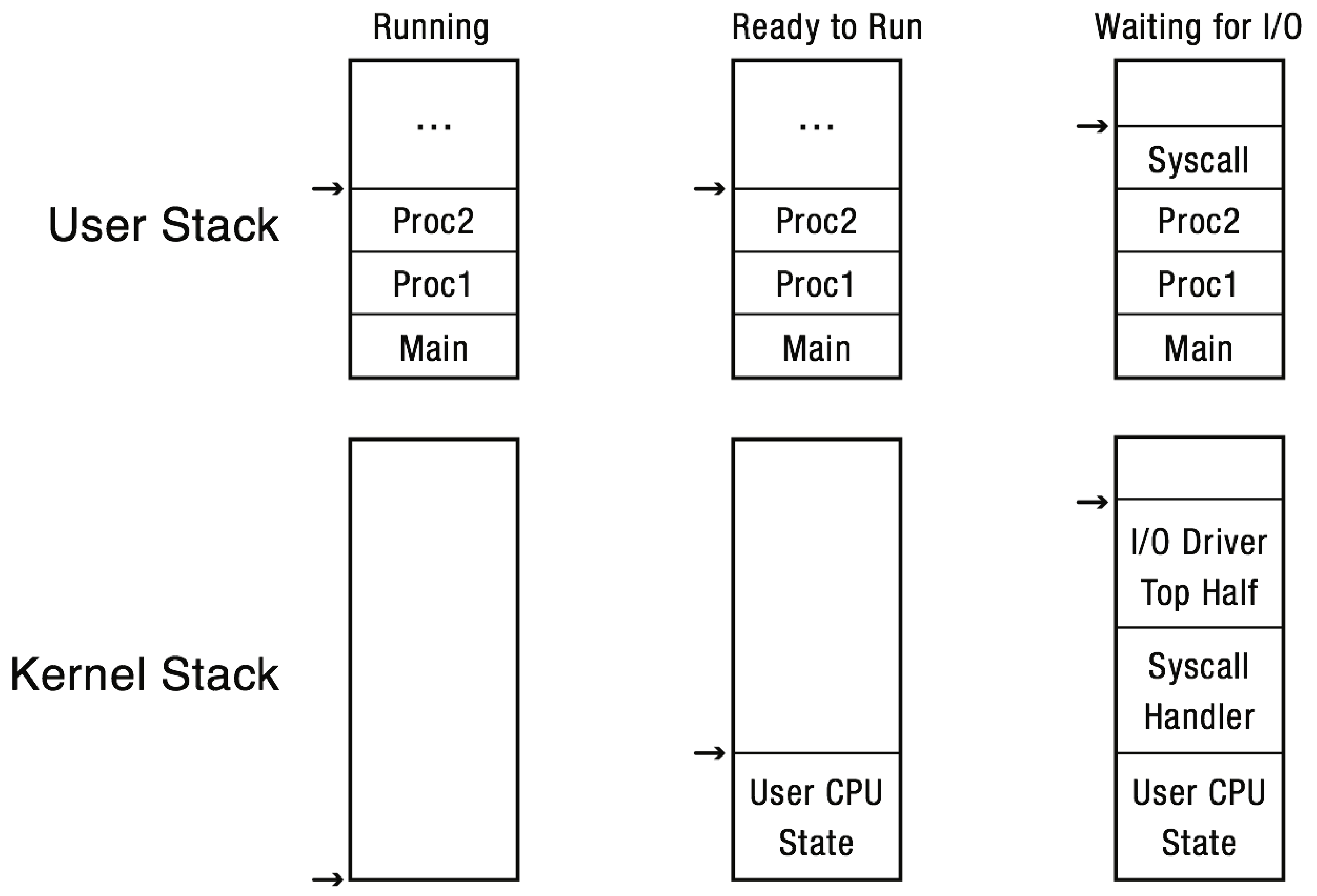


Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Fun fact! Usually a process/thread has both a kernel and user stack

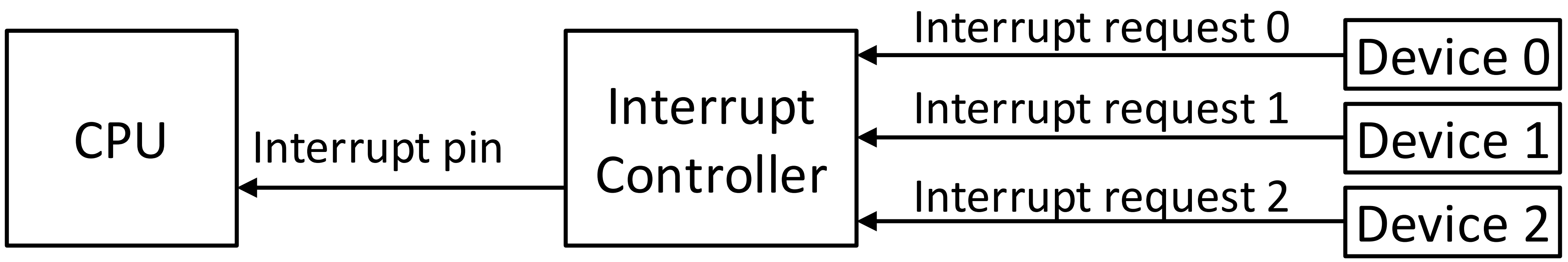
- **Can the interrupt handler run on the stack of the interrupted user process?**

Interrupt Stack



Hardware Interrupts

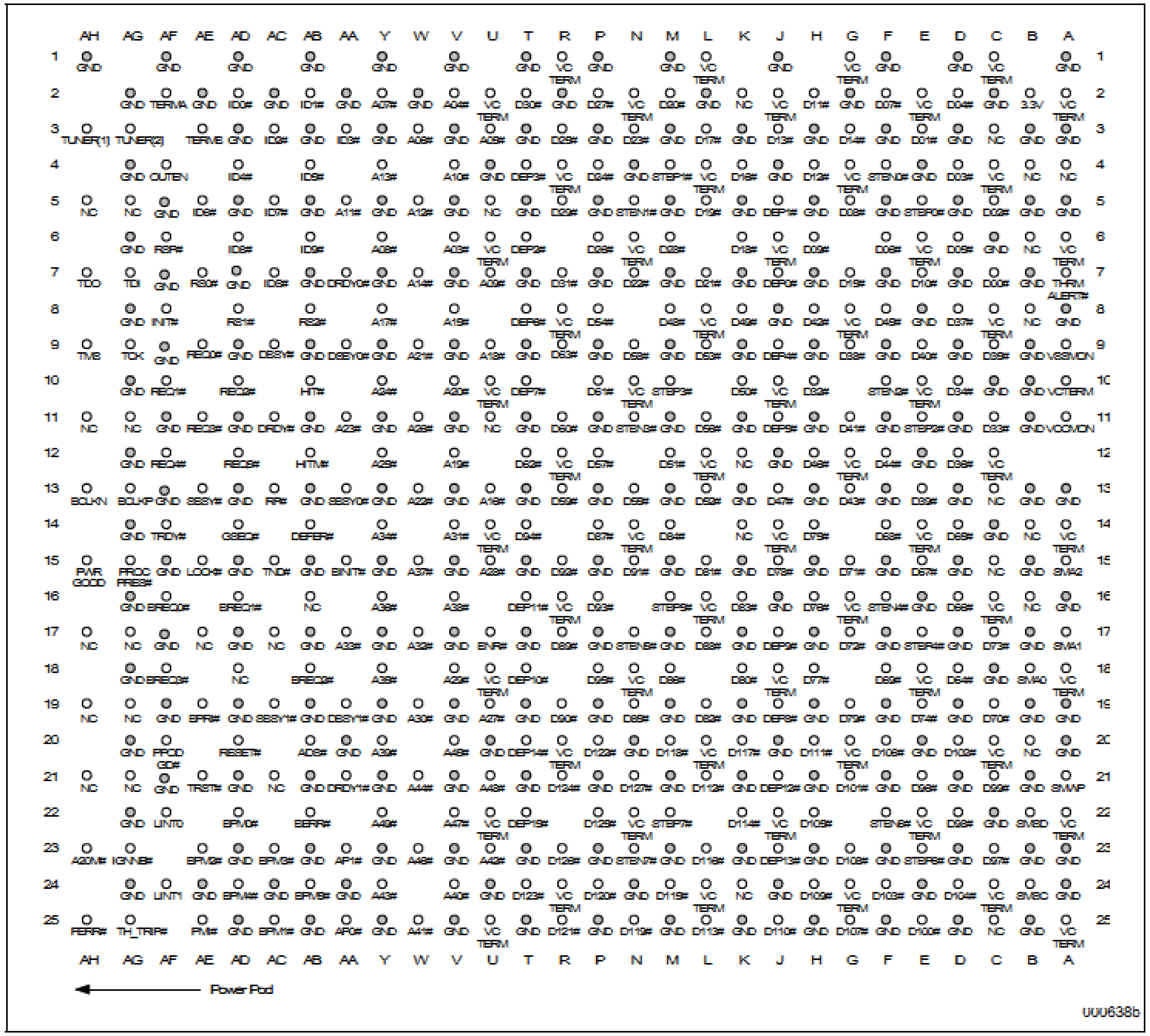
- Hardware generated:
  - Different I/O devices are connected to different physical lines (pins) of an "Interrupt controller"
  - Device hardware signals the corresponding line
  - Interrupt controller signals the CPU (by signaling the Interrupt pin and passing an interrupt number)
  - CPU saves return address after next instruction and jumps to corresponding interrupt handler



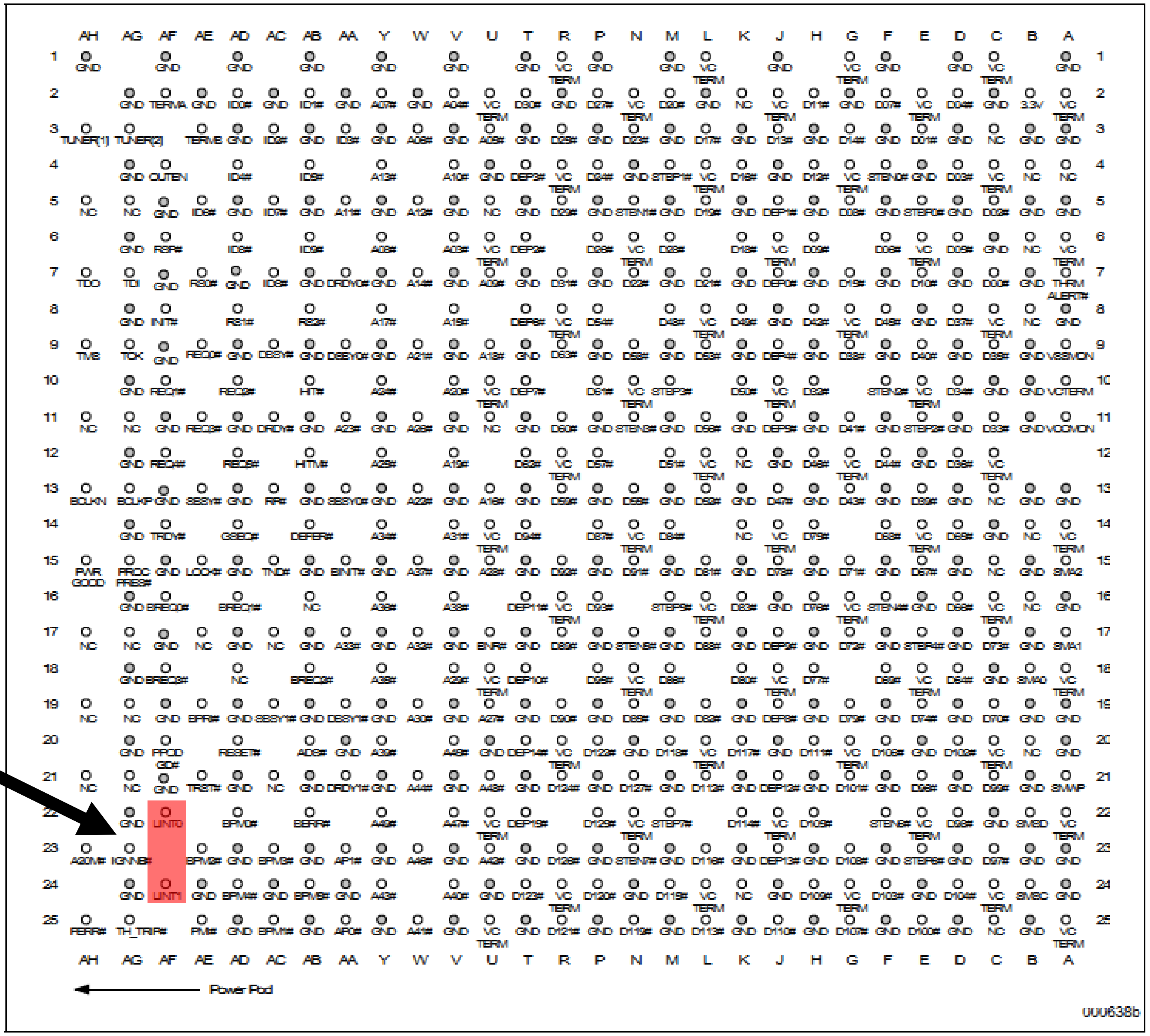
Why Hardware INTs?

- Hardware devices may need asynchronous and immediate service. For example:
  - Timer interrupt: Timers and time-dependent activities need to be updated with the passage of time at precise intervals
  - Network interrupt: The network card interrupts the CPU when data arrives from the network
  - I/O device interrupt: I/O devices (such as mouse and keyboard) issue hardware interrupts when they have input (e.g., a new character or mouse click)

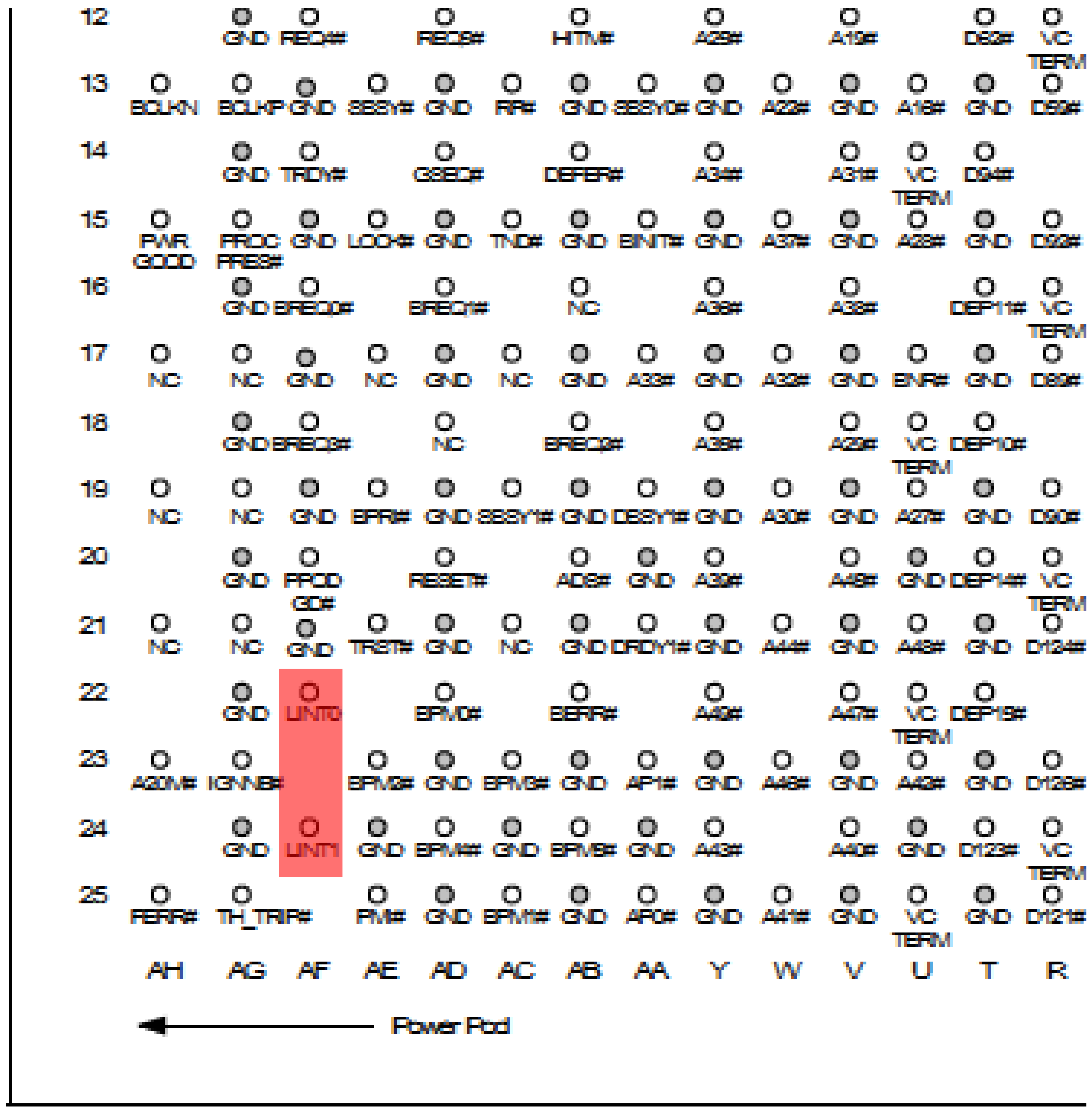
Ex: Itanium 2 Pinout



Ex: Itanium 2 Pinout


AH AG AF AE AD AC AB AA Y W V U T R P N M L K J H G F E D C B A
LINT0
LINT1
IGNNE#
A20M#
Power Pod
000638b

Ex: Itanium 2 Pinout



12 GND REQ4# REQ3# HITM# A29# A19# D62# VCTERM
13 BCLKN BCLKP GND SBSY# GND RP# GND SBSY0# GND A22# GND A18# GND D59#
14 GND TRDY# GSEQ# DEFER# A34# A31# VCTERM D94#
15 PWRGOOD PROCPRES# GND LOCK# GND TND# GND BINIT# GND A37# GND A23# GND D92#
16 GND BREQ0# BREQ1# NC A36# A38# DEP11# VCTERM
17 NC NC GND NC GND NC GND A33# GND A32# GND BNR# GND D89#
18 GND BREQ3# NC BREQ2# A39# A29# VCTERM DEP10#
19 NC NC GND BPRI# GND SBSY1# GND DBSY1# GND A30# GND A27# GND D90#
20 GND PROCGD# RESET# ADS# GND A39# A48# GND DEP14# VCTERM
21 NC NC GND TRST# GND NC GND DRDY1# GND A44# GND A43# GND D124#
22 GND LINT0 BPM0# BERR# A49# A47# VCTERM DEP15#
23 A20M# IGNNE# BPM2# GND BPM3# GND AP1# GND A46# GND A42# GND D126#
24 GND LINT1 GND BPM4# GND BPM5# GND A43# A40# GND D123# VCTERM
25 FERR# TH_TRIP# PMI# GND BPM1# GND AP0# GND A41# GND VCTERM GND D121#
AH AG AF AE AD AC AB AA Y W V U T R
Power Pod

LINTx — lines/pins for hardware interrupts.

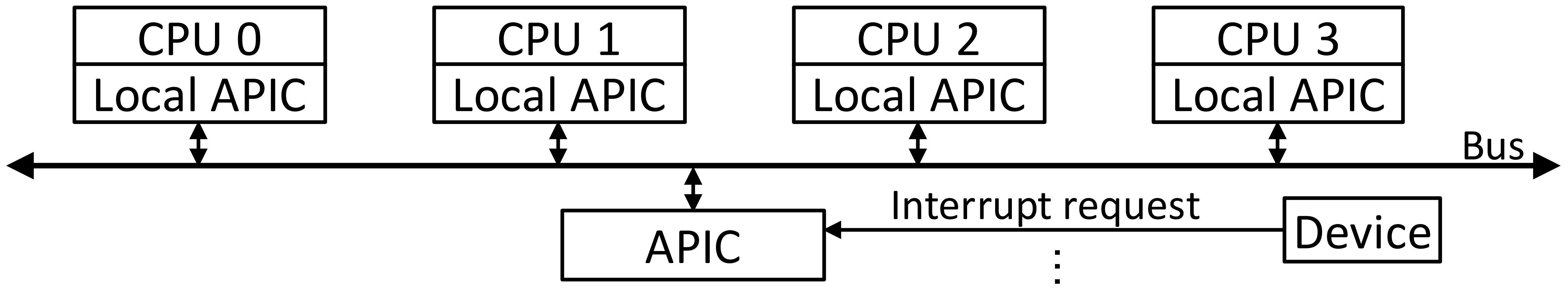
In this case…

LINT0 — line for unmaskable interrupts

LINT1 — line for maskable interrupts

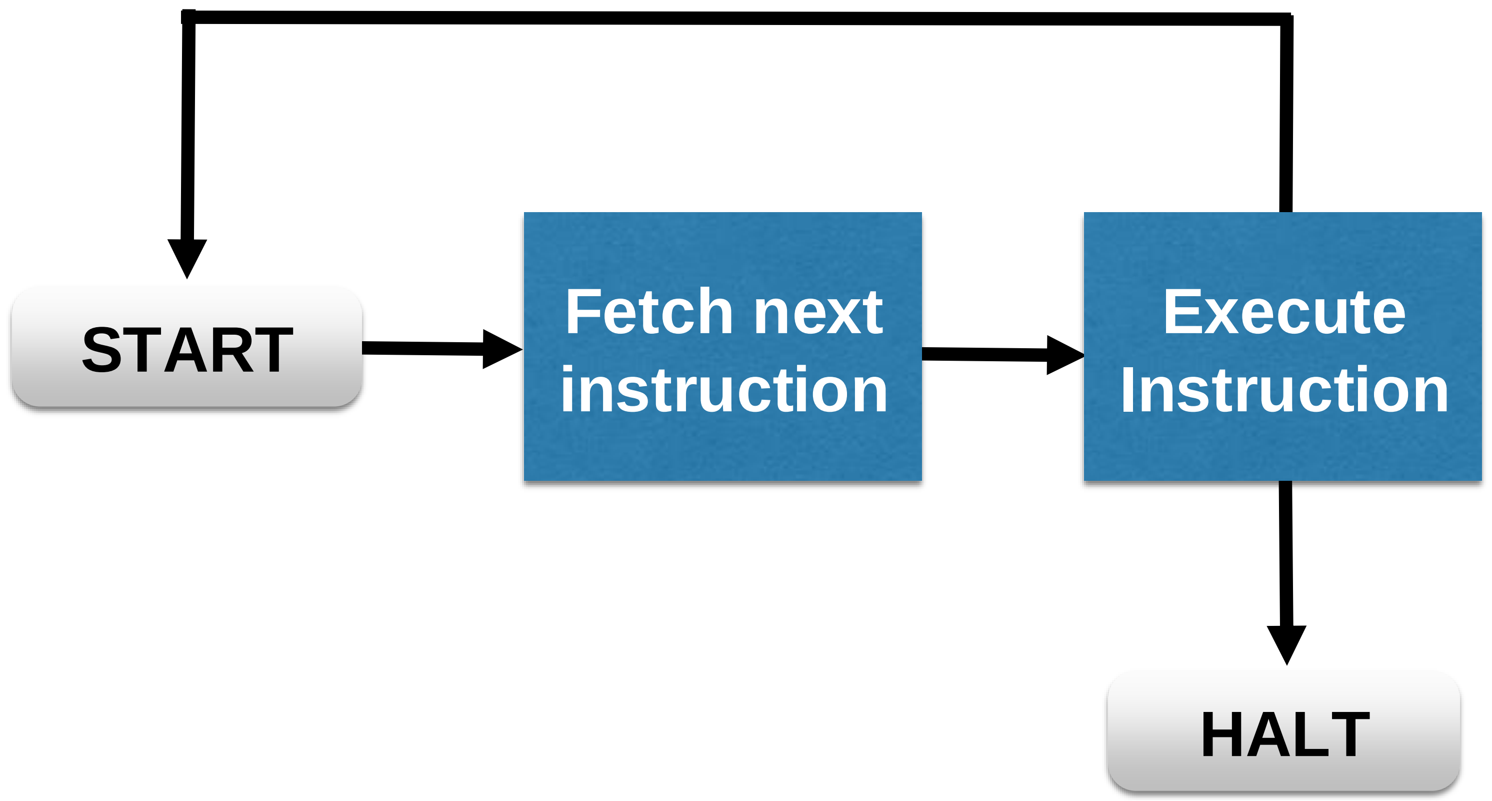
A Note on Multicore

- How are interrupts handled on multicore machines?
  - On x86 systems each CPU gets its own local **Advanced Programmable Interrupt Controller (APIC)**. They are wired in a way that allows routing device interrupts to any selected local APIC.
  - The OS can program the APICs to determine which interrupts get routed to which CPUs.
    - The default (unless OS states otherwise) is to route all interrupts to processor 0



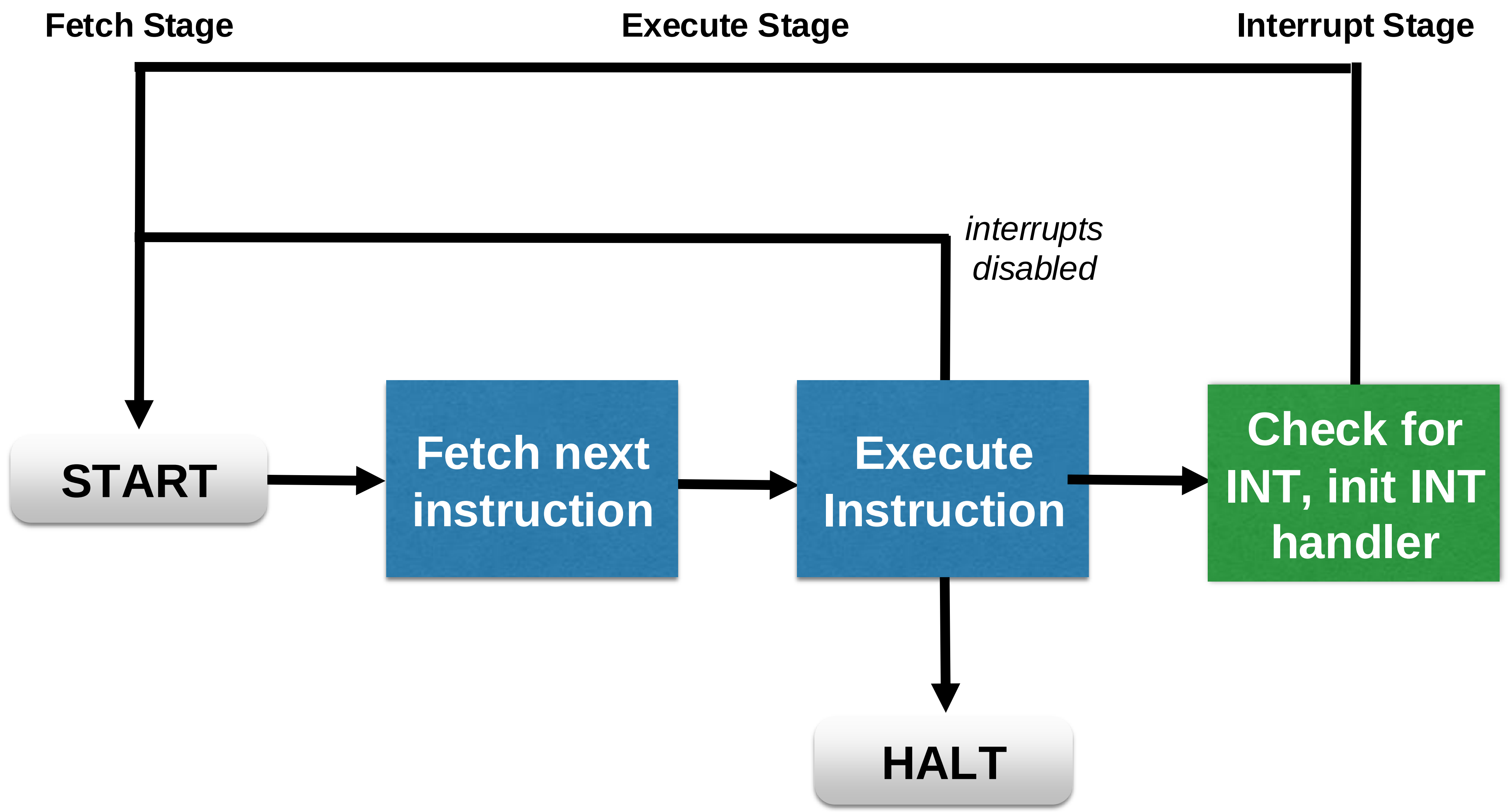
Instruction Cycle

How does interrupt handling change the instruction cycle?

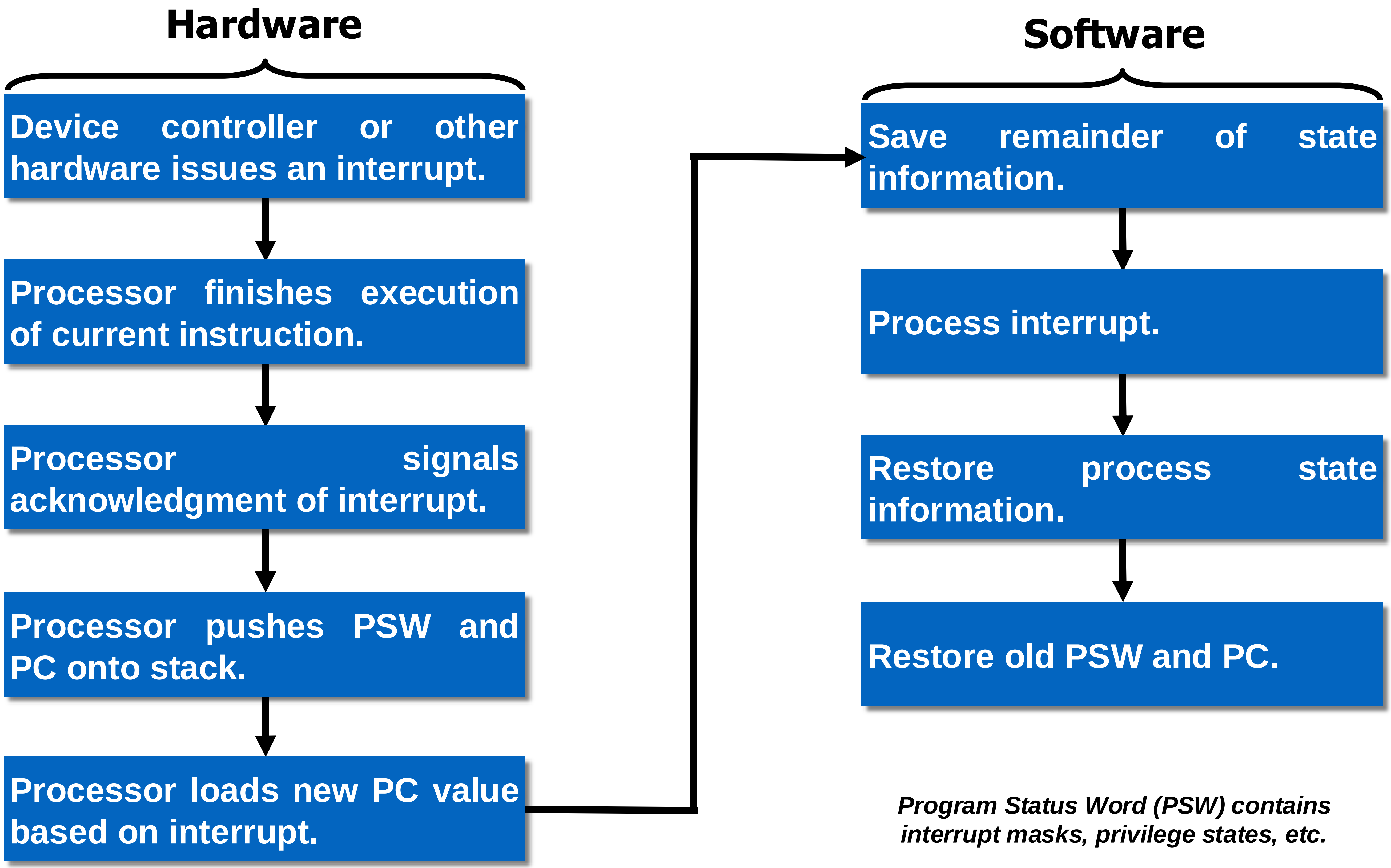


Instruction Cycle w/ INTs

How does interrupt handling change the instruction cycle?



Processing HW INT's


Hardware
Device controller or other hardware issues an interrupt.
Processor finishes execution of current instruction.
Processor signals acknowledgment of interrupt.
Processor pushes PSW and PC onto stack.
Processor loads new PC value based on interrupt.
Software
Save remainder of state information.
Process interrupt.
Restore process state information.
Restore old PSW and PC.

Program Status Word (PSW) contains interrupt masks, privilege states, etc.

Other Interrupts

- Software Interrupts:
  - Interrupts caused by the execution of a software instruction:
    - `INT <interrupt_number>`
  - Used by the system call `interrupt()`
- Initiated by the running (user level) process
- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel

Other Interrupts

- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception

They're all interrupts

- HW -> CPU -> Kernel: Classic HW Interrupt
- User -> Kernel: SW Interrupt
- CPU -> Kernel: Exception
- Interrupt Handlers used in all 3 scenarios

INTs, Priorities, & Blocking

- Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first
  - What are the implications on regular program execution?
    - Must keep interrupt code short in order not to keep other processing stopped for a long time
    - Cannot block (regular processing does not resume until interrupt returns, so if the interrupt blocks in the middle the system “hangs”)

INTs, Priorities, & Blocking

- Can an interrupt handler use kmalloc()?
- Can an interrupt handler write data to disk?
- Can an interrupt handler use busy wait?
  - E.G. — `while (!event) loop;`

Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run

Interrupt Handlers

Designing an Interrupt Handler:

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes
- The deferred portion of interrupt processing is called the "Bottom Half"

Bottom Halves

- Method for deferring portion of interrupt processing
- Globally serialized
    - When one bottom half is executing, no other bottom half can execute (even different type) on any CPU.
- Obvious performance limitations; primarily available for legacy support.
- Note: other mechanisms for deferred work are also sometimes referred to as bottom half mechanisms.

soft_irq's

- A hardware interrupt handler (before returning) uses raise_softirq() to mark that a given soft_irq must execute deferred work
- At a later time, when scheduling permits, the marked soft_irq handler is executed
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled
- Handlers that, like bottom halves, must be statically defined/allocated in the Linux kernel at compile time.
- Unlike bottom halves, softirqs are reentrant and can be executed concurrently on several CPUs
  - How to protect data??

soft_irq types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- …

soft_irq types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- …

Tasklets

- Another Deferred work mechanism multiplexed on top of soft_irq's
- Scheduled using
  - tasklet_schedule()
  - tasklet_hi_schedule()
- Typically, a tasklet is serialized with respect to itself.
  - Non-reentrant == easier to code
  - Different tasklets can be executed concurrently on different CPUs.
- Tasklets can be created or removed dynamically
- Cannot sleep (cannot save their context)

Work Queues

- A different mechanism for (non-interrupt) deferred work
- Work deferred to its own thread
  - Does not run in interrupt concept
- Can be scheduled together with other threads according to priorities set by a scheduling policy
- Associated with its thread control block and hence can block (and save context)
  - DECLARE_WORK(name, void (*func)(void *), void *data);
  - INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
  - schedule_work(&work);