



# CS 423

# Operating System Design

<https://cs423-uiuc.github.io>

Tianyin Xu

[tyxu@illinois.edu](mailto:tyxu@illinois.edu)

\* Thanks Adam Bates for the slides.



## Interrupt

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

# Discussion: Last Class

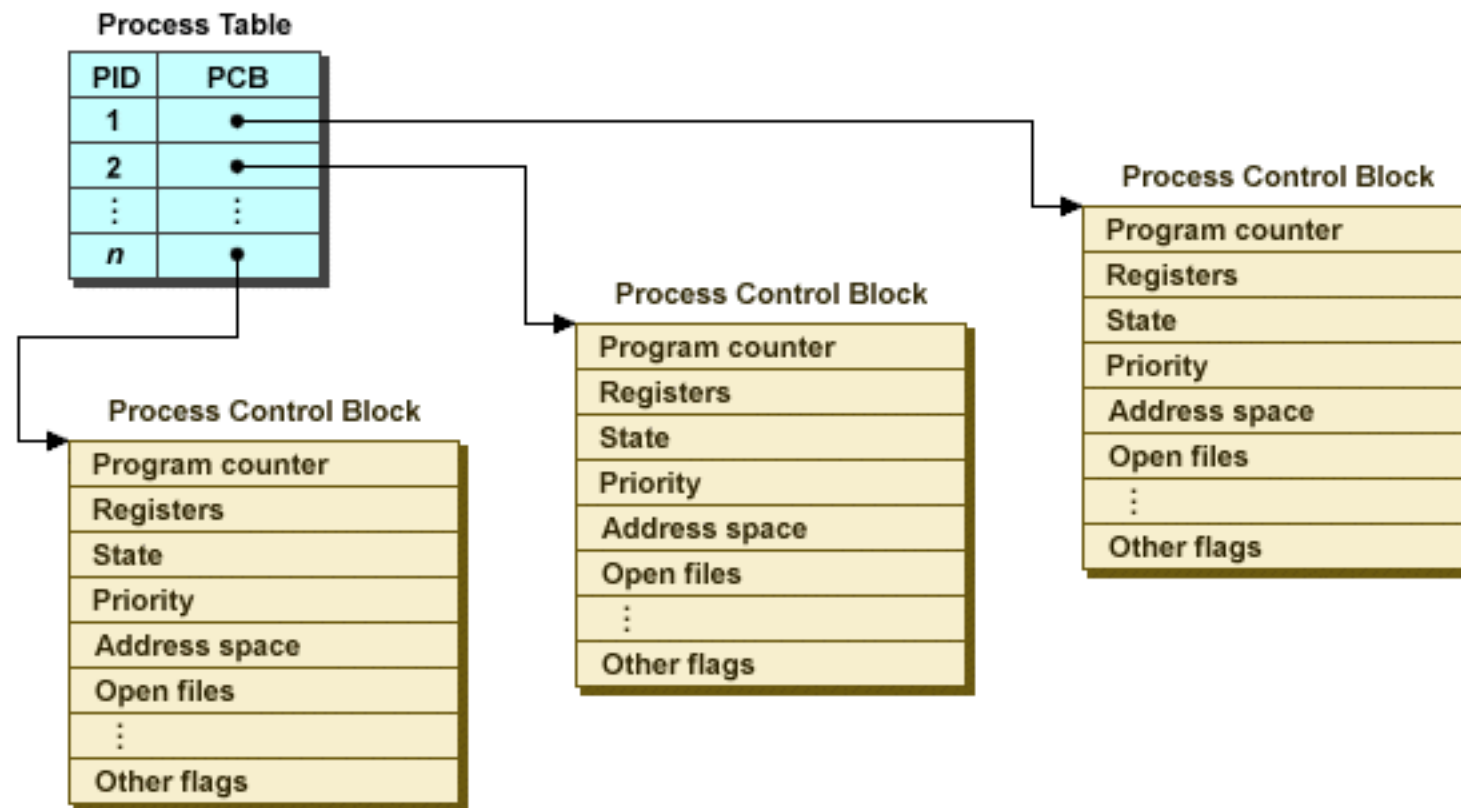


- Where is CPU State physically stored for active task?
  - Registers!
    - Program Counter is a register
    - Segment Registers
      - Code Segment
      - Data Segment
      - Stack Segment
- CPU has access to RAM and can save PC to stack before context switching.

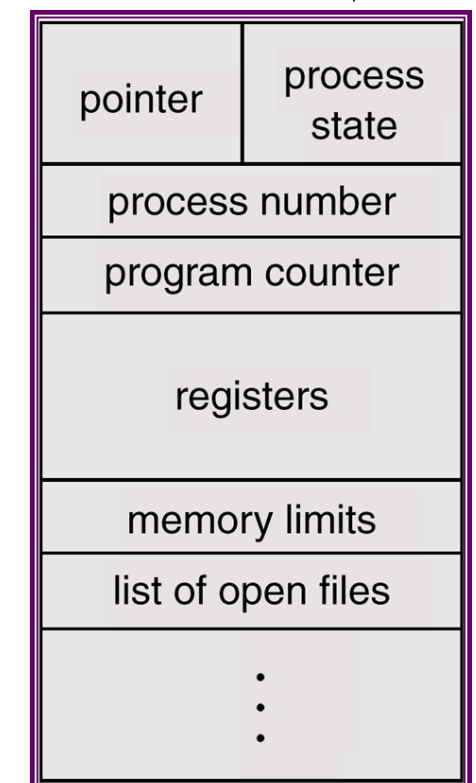
# Process Control Block



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



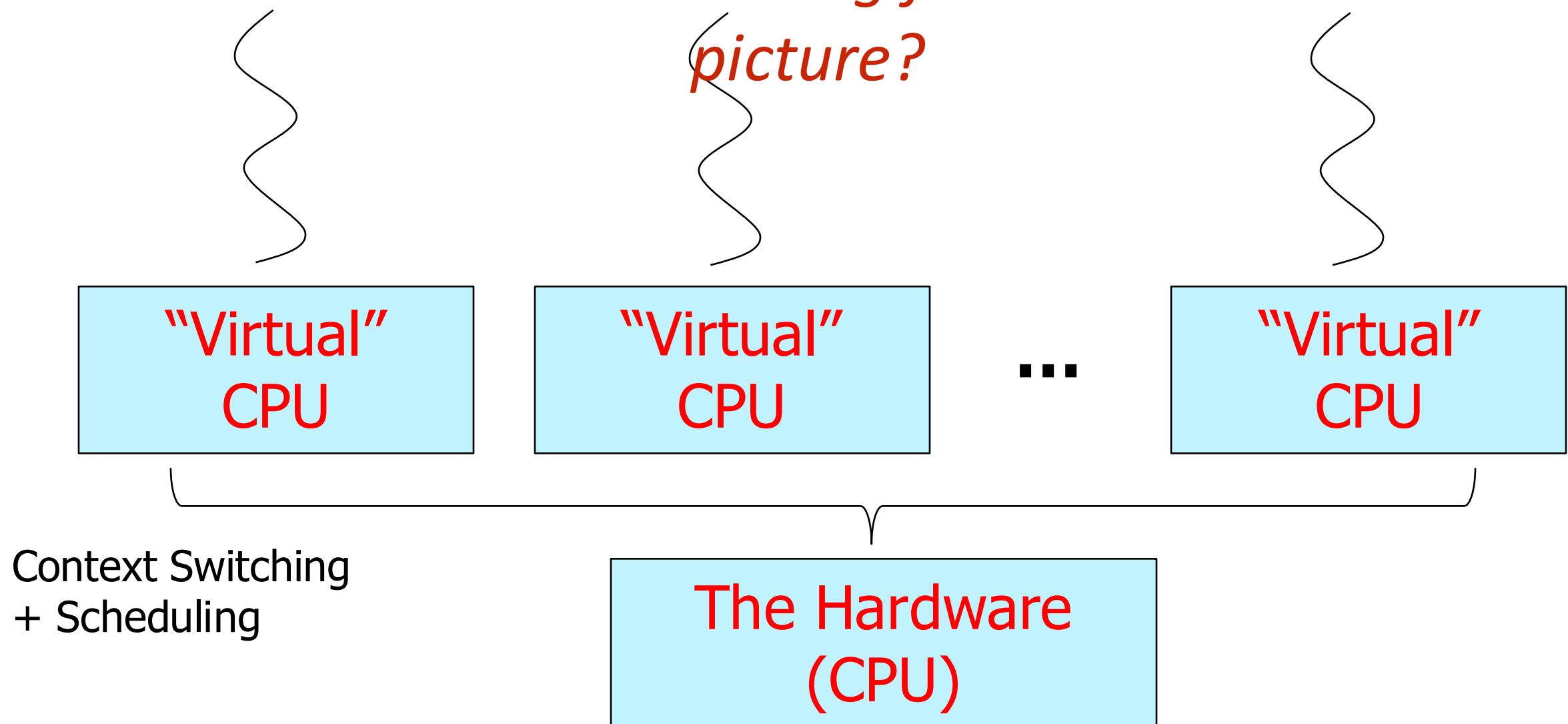
An alternate PCB diagram

# Where We Are:



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

*What's missing from this picture?*

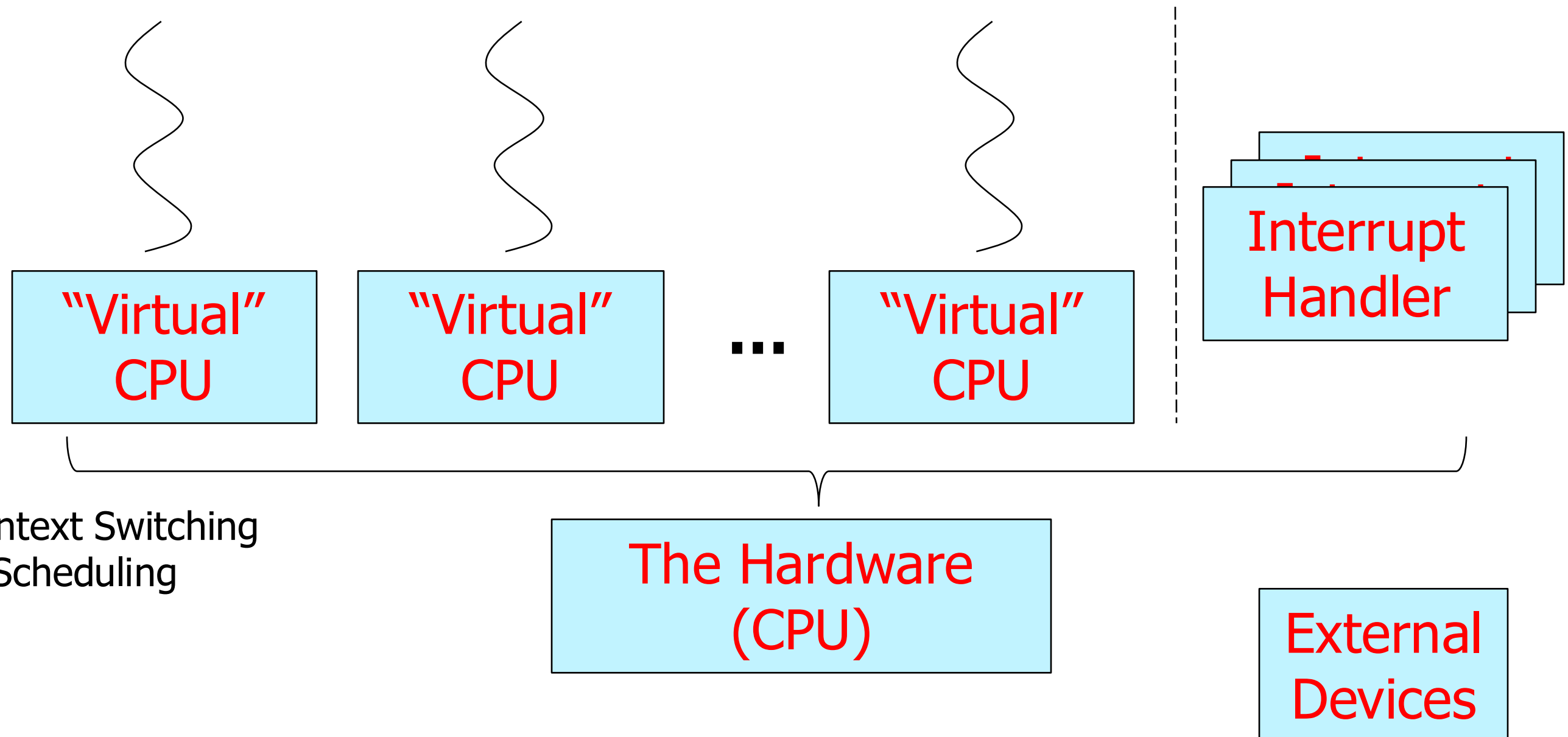


# Where We Are:

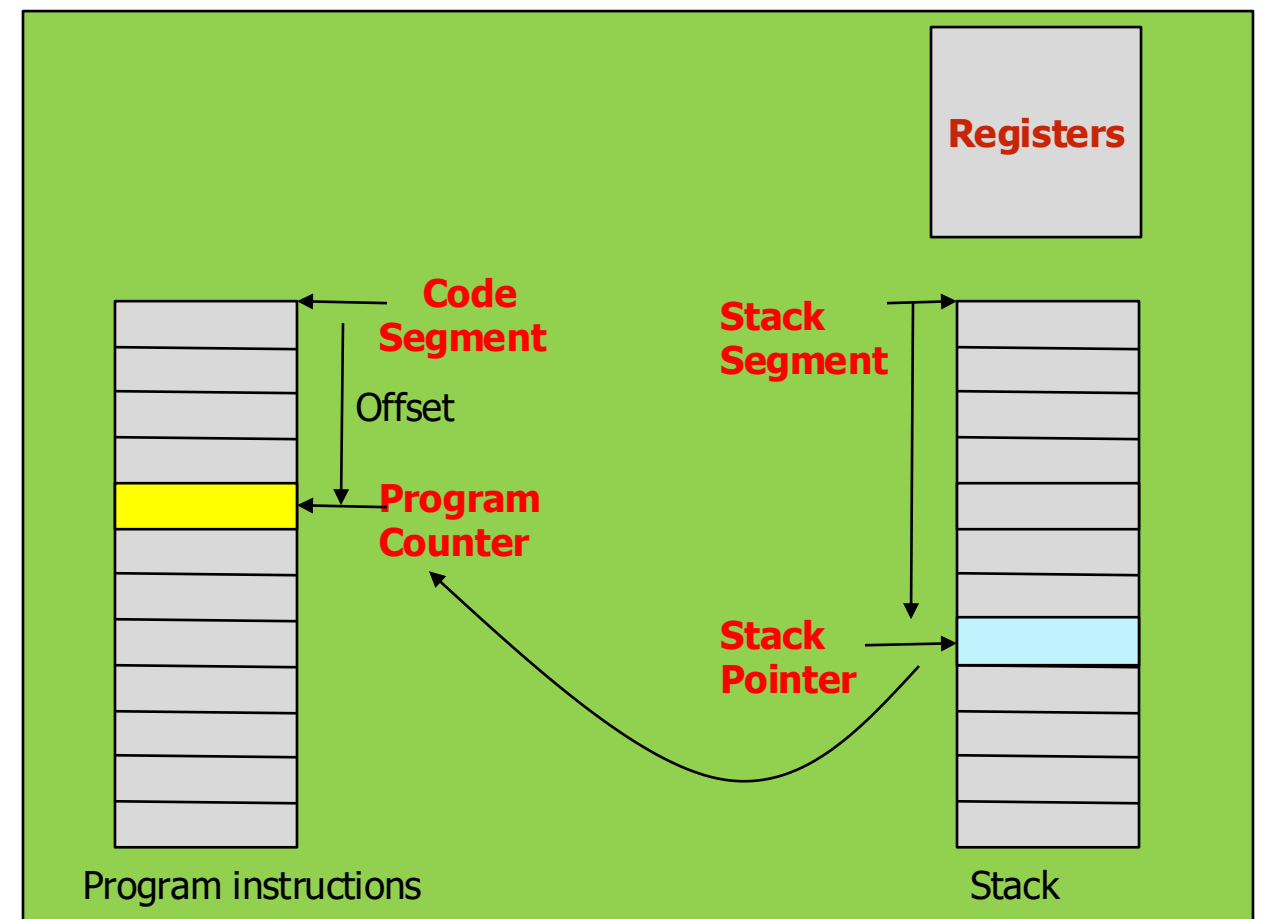
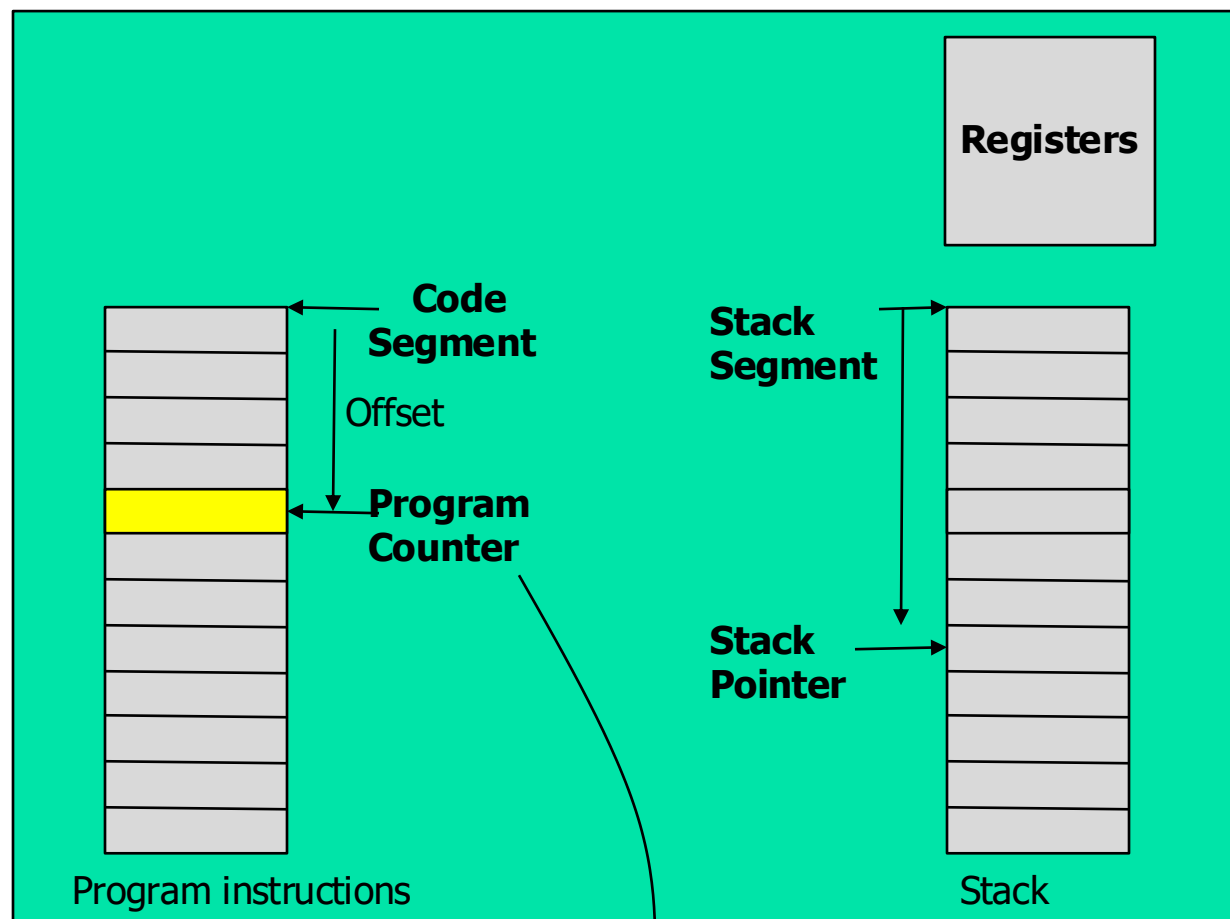


Interrupts to drive scheduling decisions!

Interrupt handlers are also tasks that share the CPU.



# CTX Switch: Interrupt



Save PC on thread stack  
Jump to Interrupt handler

Thread  
Control  
Block

## 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

Thread  
Control  
Block



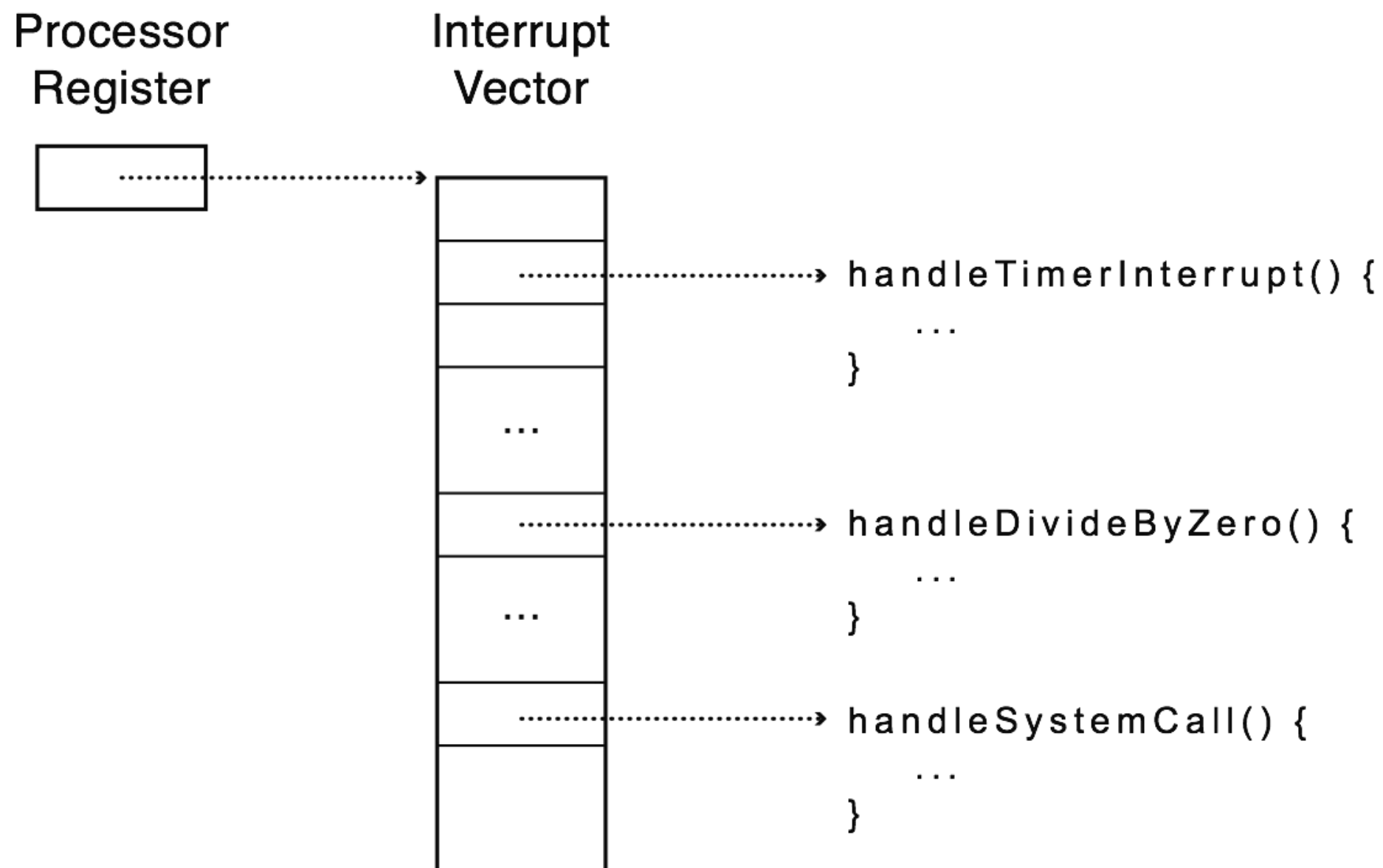
- **Interrupt Vector Table**
  - Where the processor looks for a handler
  - Limited number of entry points into kernel
  - Stored in RAM at a known address
- **Atomic transfer of control**
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user 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

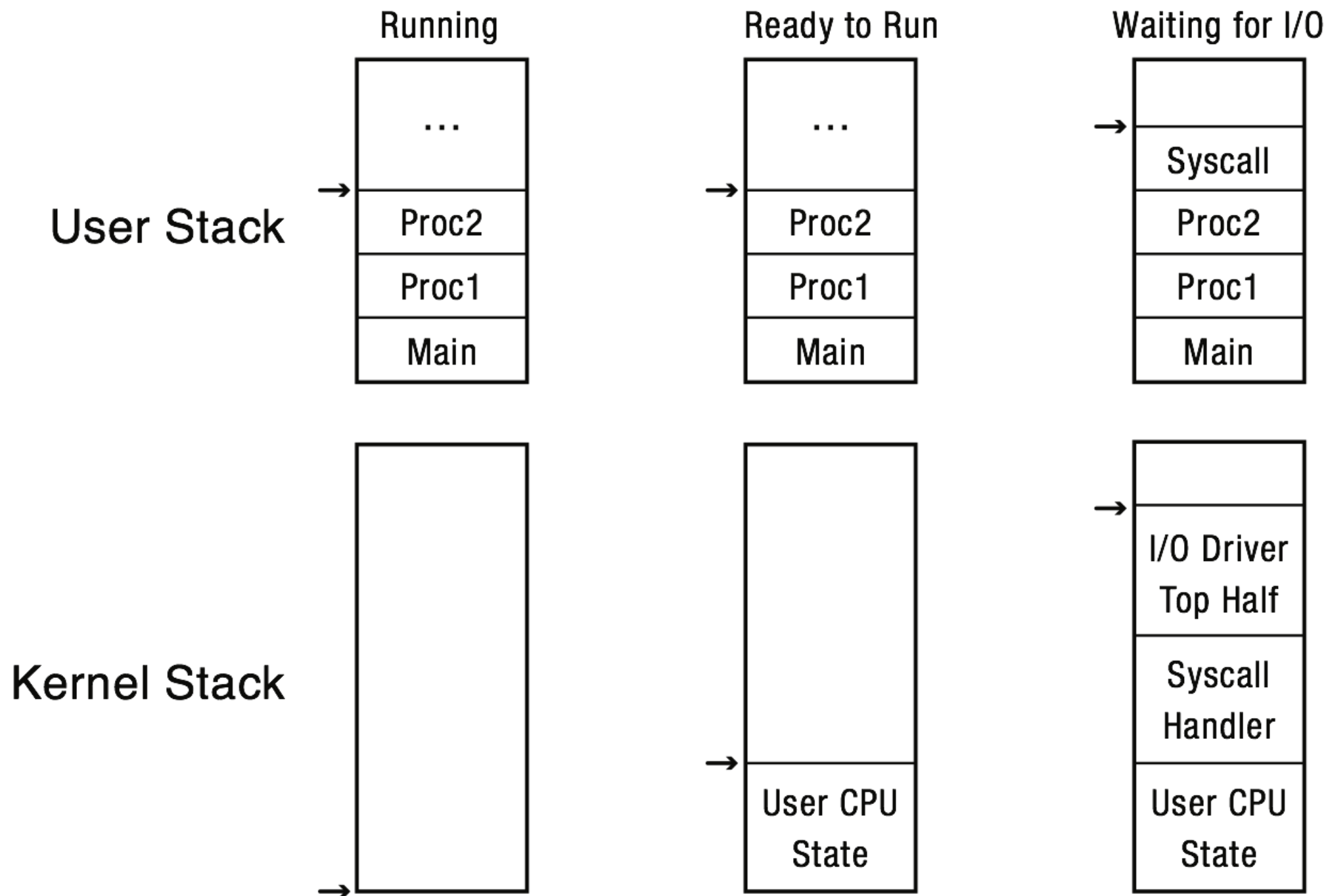


# Interrupt Stack



- 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?**

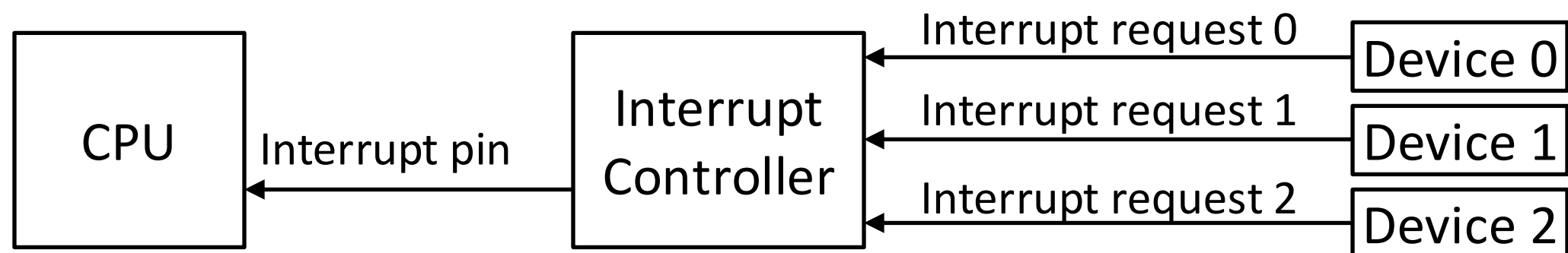
# Interrupt Stack



# 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



	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			
1	GND		GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	VC	GND	GND	GND	VC	GND	GND	GND	VC	GND	GND	GND	VC	GND	GND	1		
2		GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	VC	GND	GND	GND	VC	GND	GND	GND	VC	GND	GND	GND	GND	3.3V	GND	2		
3	TUNER(1)	TUNER(2)	TERME	GND	ID3#	GND	ID3#	GND	ID3#	GND	A08#	GND	A08#	GND	D25#	GND	D25#	GND	D17#	GND	D13#	GND	D14#	GND	D01#	GND	NC	GND	GND	3	
4		GND	OUTEN		ID4#		ID5#		A13#		A10#	GND	DEP3#	VC	GND	GND	GND	VC	GND	D18#	GND	D12#	GND	VC	STEN0#	GND	D03#	VC	NC	4	
5	NC	NC	GND	ID6#	GND	ID7#	GND	A11#	GND	A12#	GND	NC	GND	D29#	GND	STEN1#	GND	D19#	GND	DEP1#	GND	D08#	GND	STEN0#	GND	D02#	GND	GND	GND	5	
6		GND	RSP#		ID8#		ID9#		A08#		A08#	VC	DEP2#	GND	D08#	GND	D08#		VC	D18#	GND	D09#		D08#	VC	D09#	GND	NC	VC	6	
7	TD0	TD1	GND	RSP#	GND	ID9#	GND	DRDY0#	GND	A14#	GND	A08#	GND	D31#	GND	D22#	GND	D21#	GND	DEP0#	GND	D19#	GND	D10#	GND	D00#	GND	THRM	ALERT#	7	
8		GND	INT#		RS1#		RS2#		A17#		A19#		DEP8#	VC	D54#		D48#	VC	D49#	GND	D42#	VC	D48#	GND	D37#	VC	NC	GND	GND	8	
9	TMB	TDK	GND	REC0#	GND	DSSY#	GND	DSSY0#	GND	A21#	GND	A18#	GND	D63#	GND	D58#	GND	D53#	GND	D64#	GND	D38#	GND	D40#	GND	D39#	GND	VSSMON		9	
10		GND	REC1#		REC2#		HT#		A24#		A20#	VC	DEP7#	GND	D61#	VC	STEP3#		D50#	VC	D32#		STEN2#	VC	D3#	GND	GND	VC	10		
11	NC	NC	GND	REC3#	GND	DRDY#	GND	A23#	GND	A28#	GND	NC	GND	D60#	GND	STEN3#	GND	D58#	GND	DEP5#	GND	D41#	GND	STEP2#	GND	D33#	GND	VCCMON		11	
12		GND	REC4#		REC5#		HT#		A29#		A19#	GND	D62#	VC	D57#	D51#	VC	NC	GND	D48#	VC	D4#	GND	D38#	VC	GND				12	
13	BCLKN	BCLKP	GND	SBSY#	GND	RFP#	GND	SBSY0#	GND	A22#	GND	A18#	GND	D59#	GND	D59#	GND	D58#	GND	D47#	GND	D43#	GND	D39#	GND	NC	GND	GND		13	
14		GND	TRDY#		GSEC#		DEP8#		A34#		A31#	VC	D94#	GND	D87#	VC	D84#		VC	D79#		D68#	VC	D69#	GND	NC	VC			14	
15	PWR	GOOD	GND	LOCK#	GND	TND#	GND	BNIT#	GND	A37#	GND	A28#	GND	D92#	GND	D91#	GND	D81#	GND	D78#	GND	D71#	GND	D67#	GND	NC	GND	SMA2		15	
16		GND	SREC0#		SREC1#		NC		A38#		A38#	VC	DEP11#	VC	D93#		STEP5#	VC	D83#	GND	D78#	VC	STEN4#	GND	D68#	VC	NC	GND		16	
17	NC	NC	GND	NC	GND	NC	GND	A33#	GND	A32#	GND	BNF#	GND	D89#	GND	GND	STEN5#	GND	D88#	GND	DEP9#	GND	D72#	GND	GND	STEP4#	GND	D73#	GND	SMA1	17
18		GND	SREC3#		NC		SREC3#		A38#		A29#	VC	DEP10#	GND	D95#	VC	D88#		D80#	VC	D77#		D69#	VC	D64#	GND	SMA0	VC		18	
19	NC	NC	GND	SRR#	GND	SBSY1#	GND	DSSY1#	GND	A30#	GND	A27#	GND	D90#	GND	D89#	GND	D82#	GND	DEP8#	GND	D75#	GND	D74#	GND	D70#	GND	GND		19	
20		GND	PROC		RESET#		A06#	GND	A39#		A48#	GND	DEP14#	VC	D122#	GND	D118#	VC	D117#	GND	D111#	VC	D108#	GND	D102#	VC	NC	GND		20	
21	NC	NC	GND	TRST#	GND	NC	GND	DRDY1#	GND	A44#	GND	A48#	GND	D124#	GND	D127#	GND	D112#	GND	DEP12#	GND	D101#	GND	D09#	GND	D02#	GND	SWAP		21	
22		GND	UNTO		BPV0#		BPFR#		A40#		A47#	VC	DEP15#	D125#	VC	STEP7#		D114#	VC	D109#		STEN6#	VC	D08#	GND	SMD	VC		22		
23	A20V#	IGNB#		BPV6#	GND	BPV6#	GND	AP1#	GND	A48#	GND	A42#	GND	D126#	GND	STEN7#	GND	D118#	GND	DEP13#	GND	D108#	GND	STEP6#	GND	D07#	GND	GND		23	
24		GND	UNTI	GND	BPV4#	GND	BPV6#	GND	A43#		A40#	GND	D123#	VC	D120#	GND	D119#	VC	NC	GND	D109#	VC	D103#	GND	D104#	VC	SMD	GND		24	
25	PERF#	TH_TRIP#		BPV#	GND	BPV#	GND	AP0#	GND	A41#	GND	VC	GND	D121#	GND	D119#	GND	D113#	GND	D110#	GND	D107#	GND	D100#	GND	NC	GND	VC		25	
	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			
	← Power Pad																														

UUU638b

UUU638b

# Ex: Itanium 2 Pinout



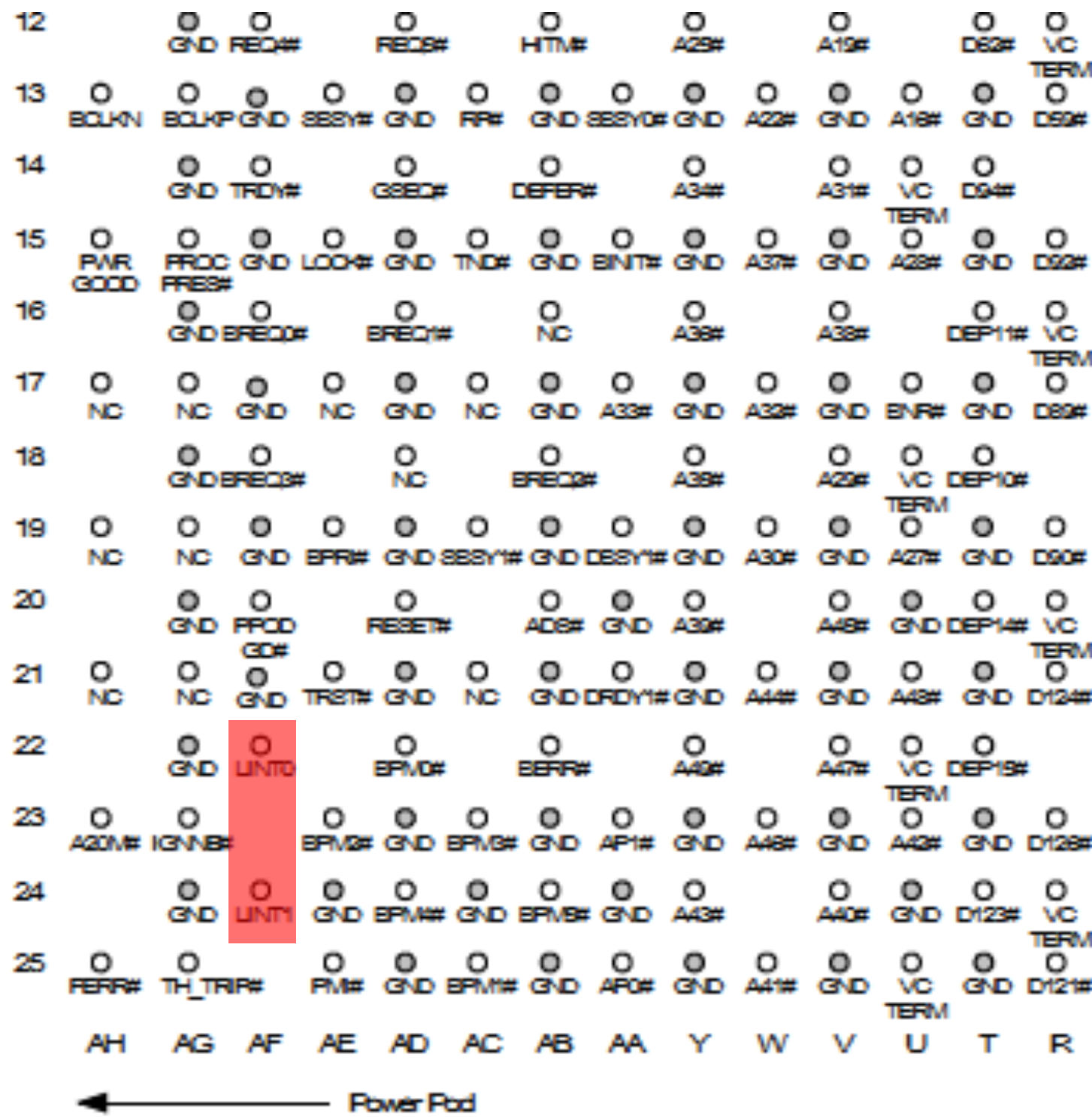
	AH	AG	AF	AE	AD	AC	AB	AA	Y	W	V	U	T	R	P	N	M	L	K	J	H	G	F	E	D	C	B	A			
1	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	VC	TERM	GND	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	GND	1	
2		GND	TERMA	GND	IO0#	GND	IO1#	GND	A07#	GND	A04#	VC	D00#	GND	D07#	VC	D00#	GND	NC	VC	D11#	GND	D07#	VC	D00#	GND	3.3V	VC	TERM	2	
3	TUNER(1)	TUNER(2)	TERMB	GND	IO2#	GND	IO3#	GND	A08#	GND	A05#	VC	D01#	GND	D08#	VC	D01#	GND	D12#	GND	D08#	VC	STEN0#	GND	D03#	GND	NC	GND	GND	3	
4		GND	OUTEN		IO4#	GND	IO5#	GND	A13#	GND	A10#	GND	DEP3#	VC	D02#	GND	STEP1#	VC	D18#	GND	D12#	VC	STEN0#	GND	D03#	GND	NC	NC	4		
5	NC	NC	GND	IO6#	GND	IO7#	GND	A11#	GND	A12#	GND	NC	GND	D09#	GND	STEN1#	GND	D19#	GND	DEP1#	GND	D08#	GND	STEN0#	GND	D02#	GND	GND	5		
6		GND	RSP#		IO8#	GND	IO9#	GND	A09#	GND	A03#	VC	DEP2#	GND	D09#	GND	D03#	GND	D18#	GND	D09#	GND	D08#	GND	D09#	GND	NC	VC	TERM	6	
7	TD0	TD1	GND	RSP#	GND	IO9#	GND	D00#	GND	A14#	GND	A09#	VC	D01#	GND	D02#	GND	D01#	GND	D09#	GND	D10#	GND	D00#	GND	THRM	ALERT#	GND	7		
8		GND	INT#		RSP#	GND	RSP#	GND	A17#	GND	A19#	GND	DEP8#	VC	D04#	GND	D08#	VC	D48#	GND	D42#	VC	D48#	GND	D37#	VC	NC	GND	8		
9	TMB	TDK	GND	REC0#	GND	D00#	GND	D00#	GND	A21#	GND	A13#	VC	D03#	GND	D03#	GND	D03#	GND	D04#	GND	D03#	GND	D03#	GND	D03#	GND	D03#	GND	VSSEN0N	9
10		GND	REC1#		REC2#	GND	HT#	GND	A2#	GND	A20#	VC	DEP7#	GND	D01#	VC	STEP3#	GND	D00#	VC	D02#	STEN2#	VC	D03#	GND	GND	VC	TERM	10		
11	NC	NC	GND	REC3#	GND	D00#	GND	A22#	GND	A22#	GND	NC	GND	D00#	GND	STEN3#	GND	D08#	GND	D09#	GND	D41#	GND	STEN2#	GND	D03#	GND	VC	TERM	11	
12		GND	REC4#		REC5#	GND	HT#	GND	A2#	GND	A19#	GND	D02#	VC	D07#	GND	D01#	VC	NC	GND	D48#	VC	D41#	GND	D03#	GND	VC	TERM	12		
13	BOLVN	BOLVP	GND	SSSY#	GND	RFP#	GND	SSSY0#	GND	A22#	GND	A18#	VC	D09#	GND	D08#	GND	D07#	GND	D47#	GND	D43#	GND	D09#	GND	NC	GND	GND	13		
14		GND	TRDY#		D00#	GND	DEP8#	GND	A3#	GND	A31#	VC	D04#	GND	D07#	VC	D08#	GND	NC	VC	D79#	GND	D08#	VC	D08#	GND	NC	VC	TERM	14	
15	PAR	GOOD	GND	LOCK#	GND	TND#	GND	BNIT#	GND	A37#	GND	A28#	VC	D02#	GND	D01#	GND	D01#	GND	D78#	GND	D71#	GND	D07#	GND	NC	GND	SM42	15		
16		GND	EPREC0#		EPREC1#	GND	NC	GND	A38#	GND	A38#	VC	DEP11#	VC	D03#	GND	STEP5#	VC	D03#	GND	D78#	VC	STEN4#	GND	D08#	VC	NC	GND	GND	16	
17	NC	NC	GND	NC	GND	NC	GND	A33#	GND	A32#	GND	BNF#	GND	D09#	GND	STEN5#	GND	D08#	GND	DEP9#	GND	D72#	GND	STEN4#	GND	D73#	GND	SM41	17		
18		GND	EPREC3#		NC	GND	EPREC3#	GND	A38#	GND	A23#	VC	DEP10#	GND	D09#	VC	D08#	GND	D00#	VC	D77#	GND	D09#	VC	D04#	GND	SM40	VC	TERM	18	
19	NC	NC	GND	EPRE#	GND	SSSY1#	GND	D00#	GND	A30#	GND	A27#	GND	D00#	GND	D09#	GND	D03#	GND	DEP8#	GND	D79#	GND	D74#	GND	D70#	GND	GND	19		
20		GND	FFOD	GND	RESET#	GND	A06#	GND	A39#	GND	A48#	GND	DEP14#	VC	D122#	GND	D118#	VC	D117#	GND	D111#	VC	D108#	GND	D102#	VC	NC	GND	20		
21	NC	NC	GND	TRST#	GND	NC	GND	D00#	GND	A44#	GND	A48#	GND	D124#	GND	D127#	GND	D112#	GND	DEP12#	GND	D101#	GND	D09#	GND	D02#	GND	SWAP	21		
22		GND	UNTO		EPV0#	GND	EPRE#	GND	A40#	GND	A47#	VC	DEP15#	D125#	VC	STEP7#	GND	D114#	VC	D109#	STEN6#	VC	D08#	GND	SVED	VC	TERM	22			
23	A20V#	IGNNB#		EPV6#	GND	EPV6#	GND	AP1#	GND	A48#	GND	A42#	GND	D126#	GND	STEN7#	GND	D118#	GND	DEP13#	GND	D108#	GND	STEN6#	GND	D07#	GND	GND	23		
24		GND	UNTN		GND	EPV4#	GND	EPV6#	GND	A43#	GND	A40#	GND	D123#	VC	D120#	GND	D119#	VC	NC	GND	D109#	VC	D103#	GND	D104#	VC	SM5C	GND	24	
25	PERF#	TH_TRIP#		PV#	GND	EPV1#	GND	AP0#	GND	A41#	GND	VC	GND	D121#	GND	D119#	GND	D113#	GND	D110#	GND	D107#	GND	D100#	GND	NC	GND	VC	TERM	25	
	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			
	← Power Pad																														

UUU6381

UUU638b



# Ex: Itanium 2 Pinout



LINTx — lines/pins for hardware interrupts.

In this case...

LINT0 — line for unmaskable interrupts

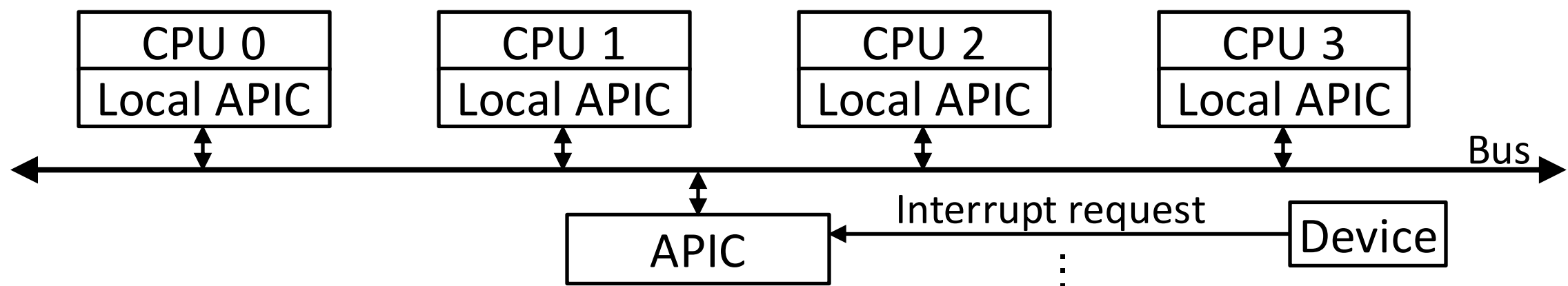
LINT1 — line for maskable interrupts



# A Note on Multicore



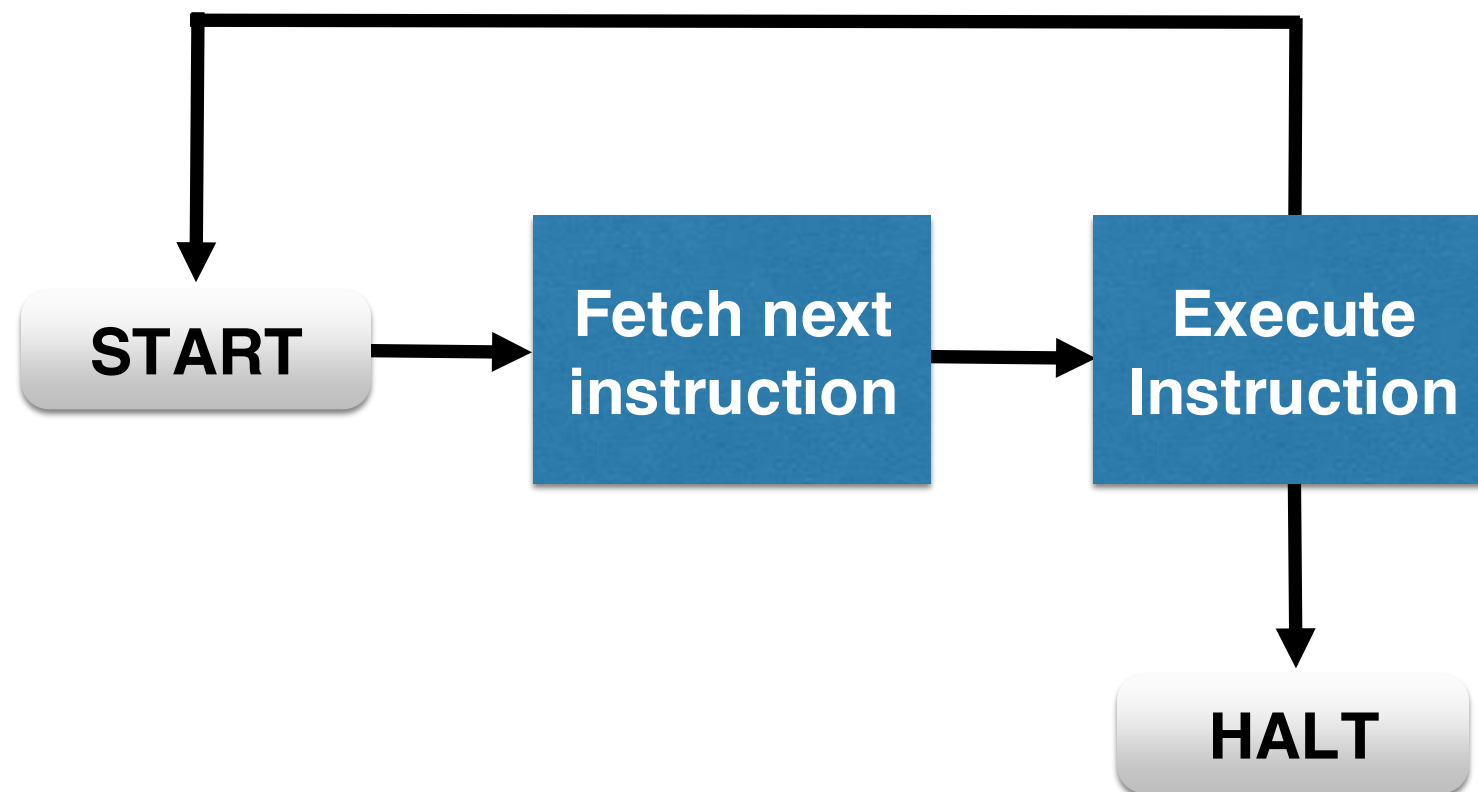
- How are interrupts handled on multicore machines?
  - On x86 systems each CPU gets its own local **Advanced Programmable Interrupt Controller (APIC)**. They are wired in a way that allows routing device interrupts to any selected local APIC.
  - The OS can program the APICs to determine which interrupts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



# Instruction Cycle



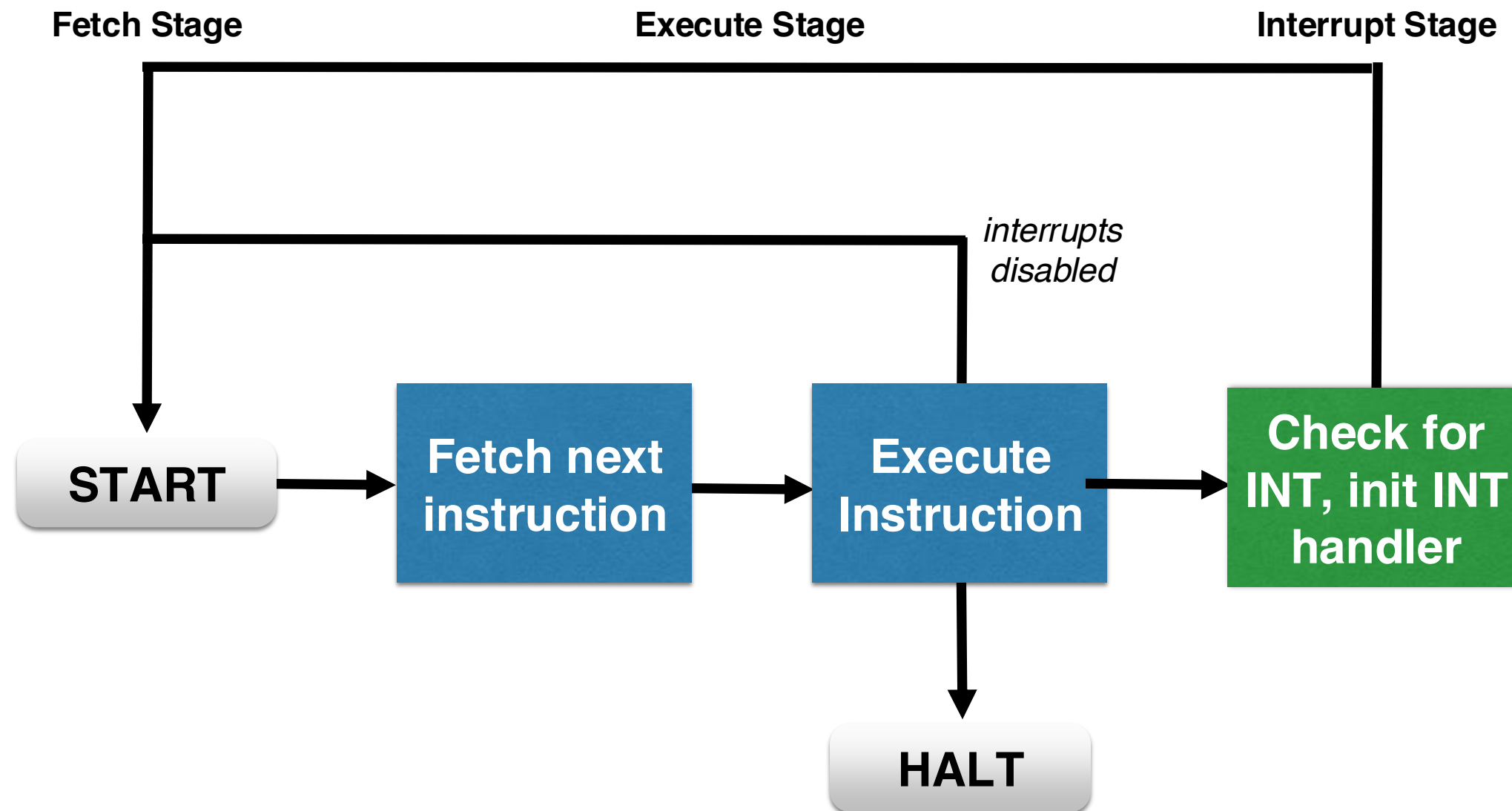
How does interrupt handling change the instruction cycle?



# 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



- 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”)





- 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



## 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.



- 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\_SOFTIRQ
- NET\_RX\_SOFTIRQ
- BLOCK\_SOFTIRQ
- TASKLET\_SOFTIRQ
- SCHED\_SOFTIRQ
- ...

# soft\_irq types



- HI\_SOFTIRQ
- TIMER\_SOFTIRQ
- NET\_TX\_SOFTIRQ
- NET\_RX\_SOFTIRQ
- BLOCK\_SOFTIRQ
- TASKLET\_SOFTIRQ
- SCHED\_SOFTIRQ
- ...



- 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);`