



CS 423

Operating System Design: The Kernel Abstraction

Jongyul Kim

* Thanks for Prof. 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

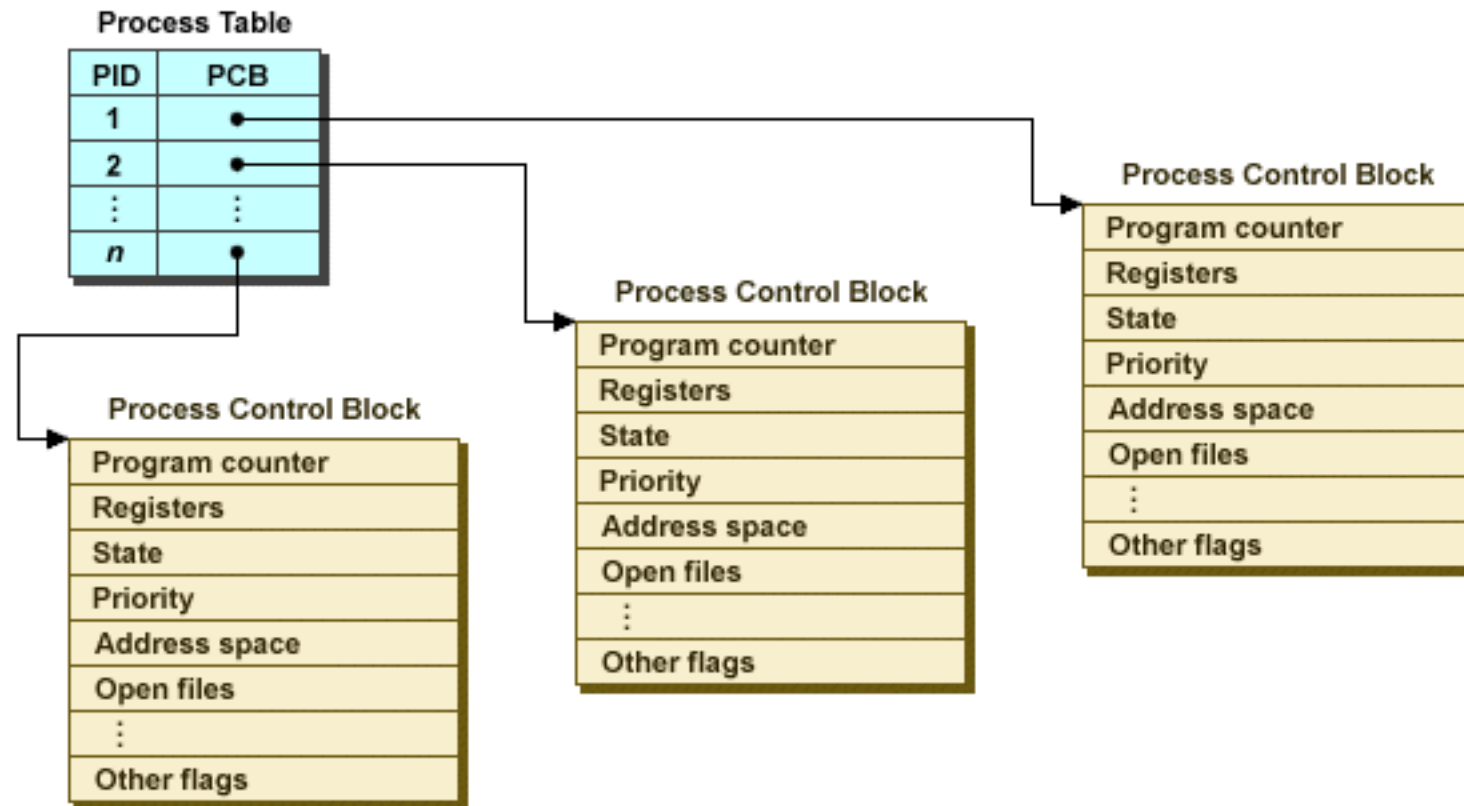


- Where is CPU State physically stored for active task?
 - Registers!
 - Program Counter is a register
 - Segment Registers
 - Code Segment
 - Data Segment
 - Stack Segment
- CPU has access to RAM and can save PC to stack before context switching.

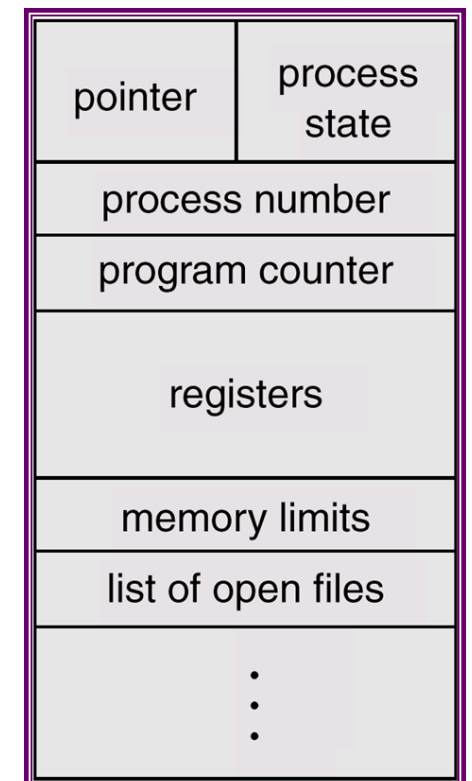
Process Control Block



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



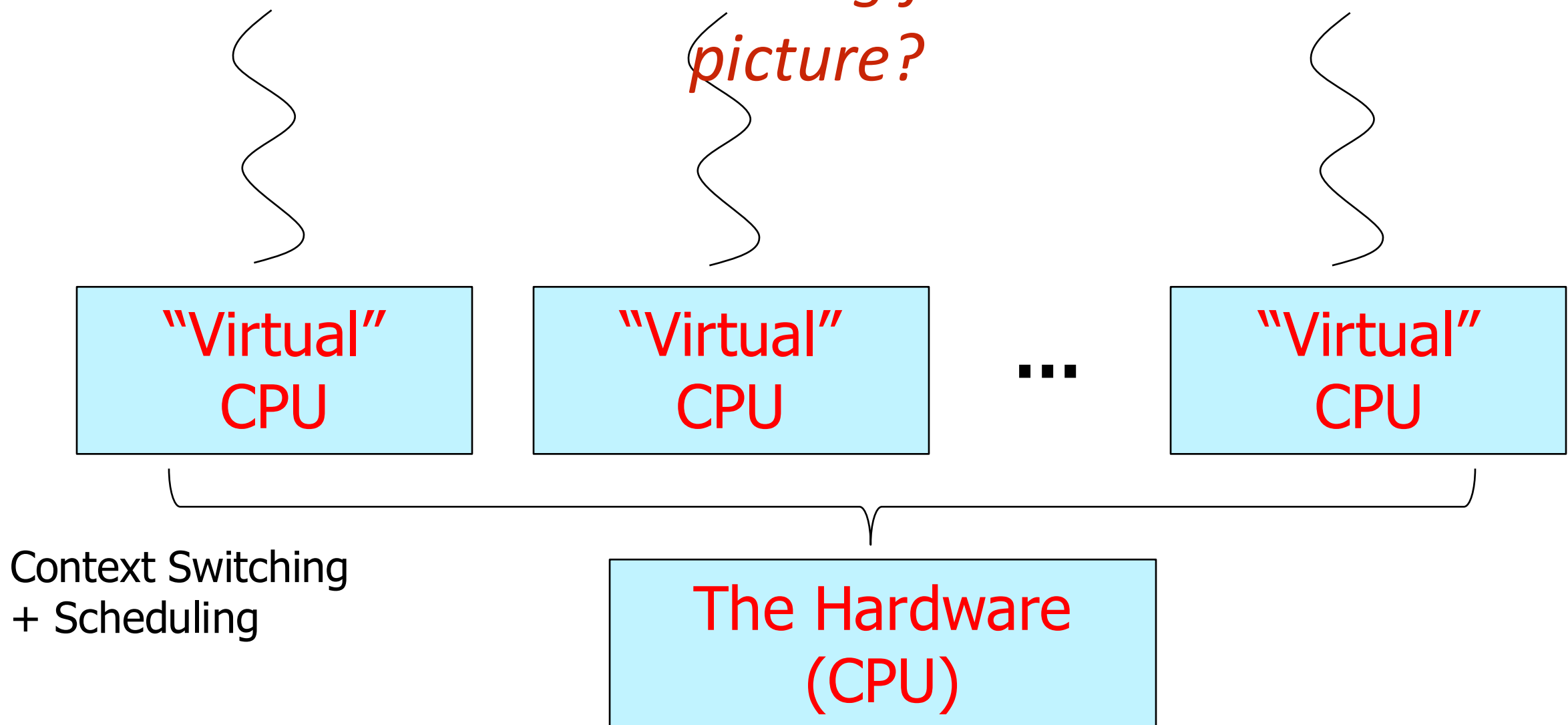
An alternate PCB diagram

Where We Are:



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

What's missing from this picture?

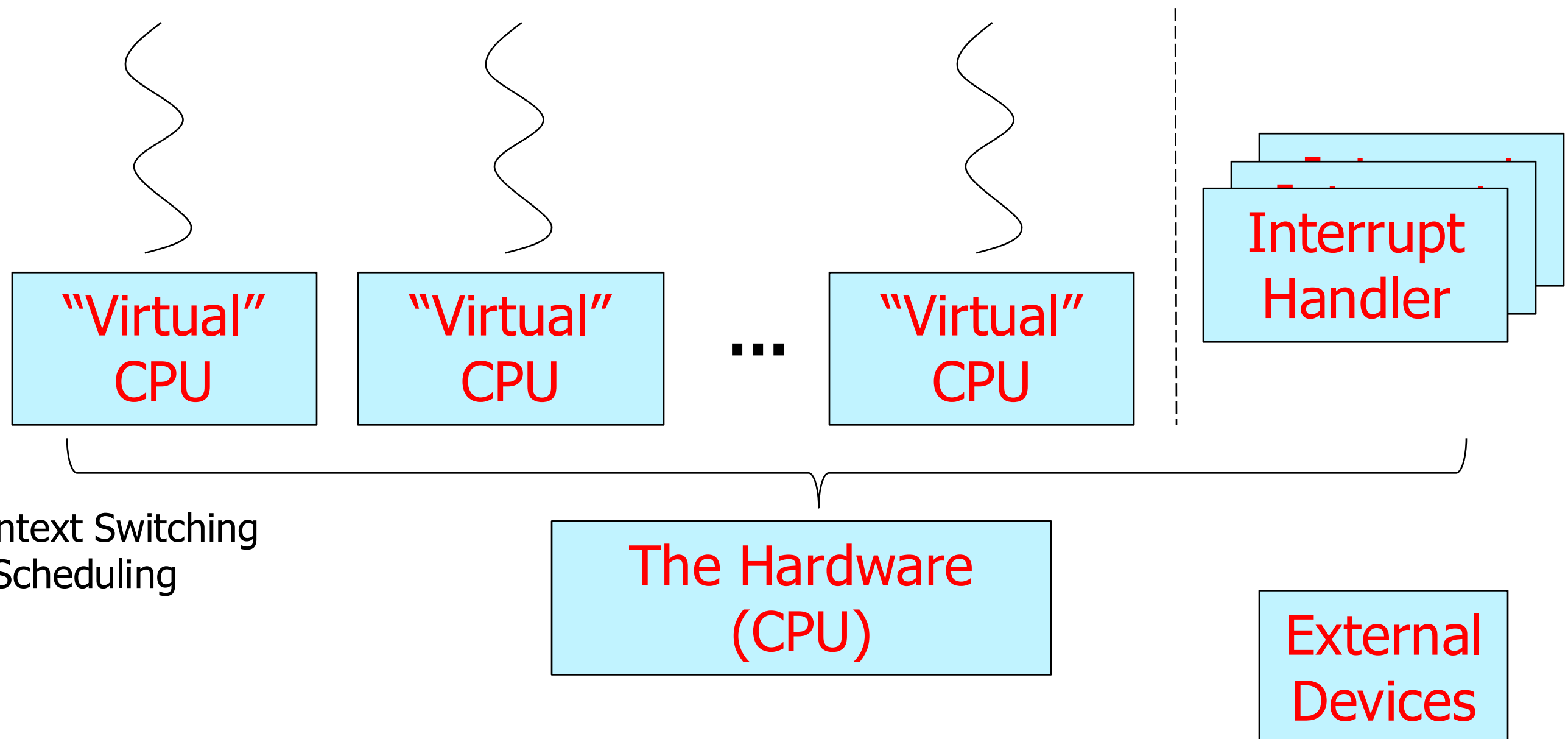


Where We Are:

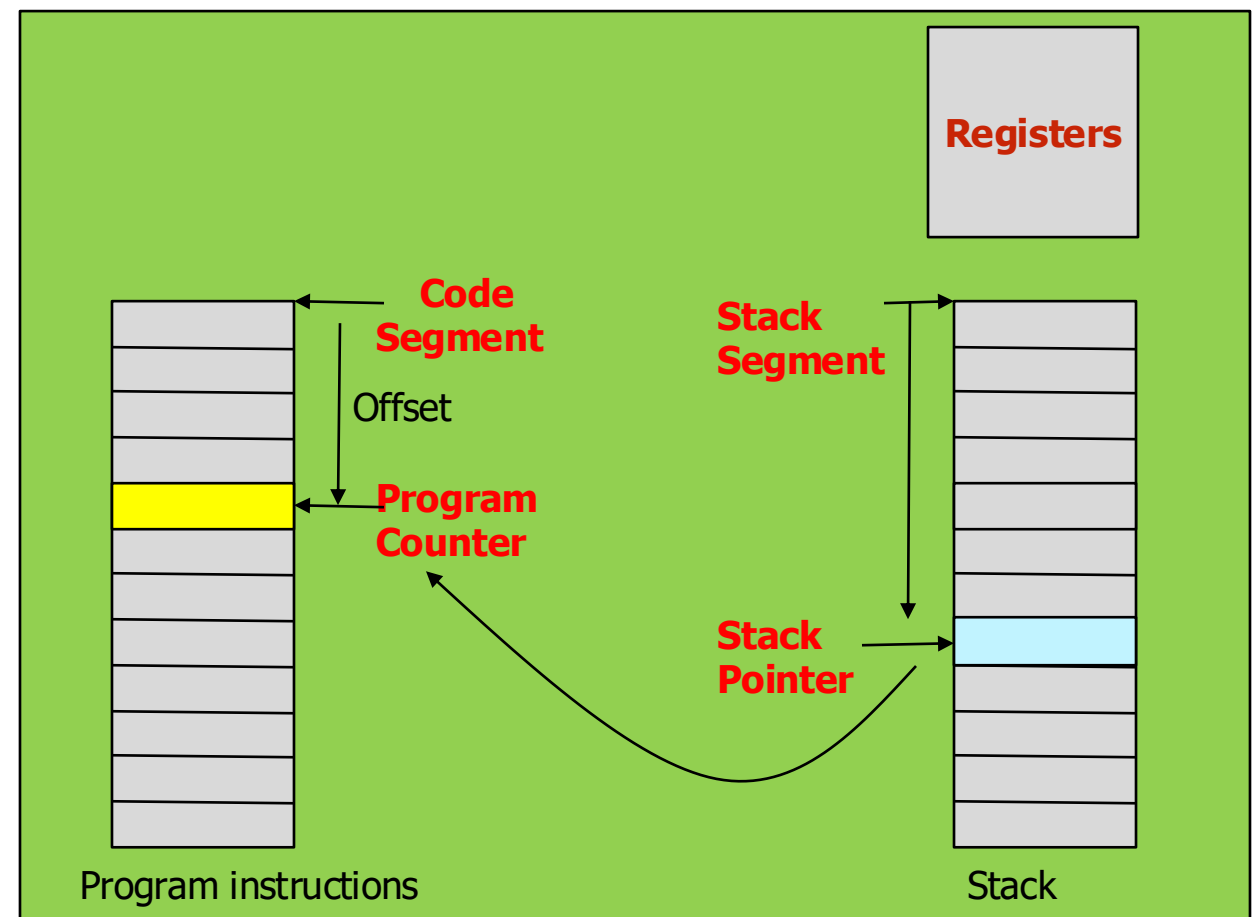
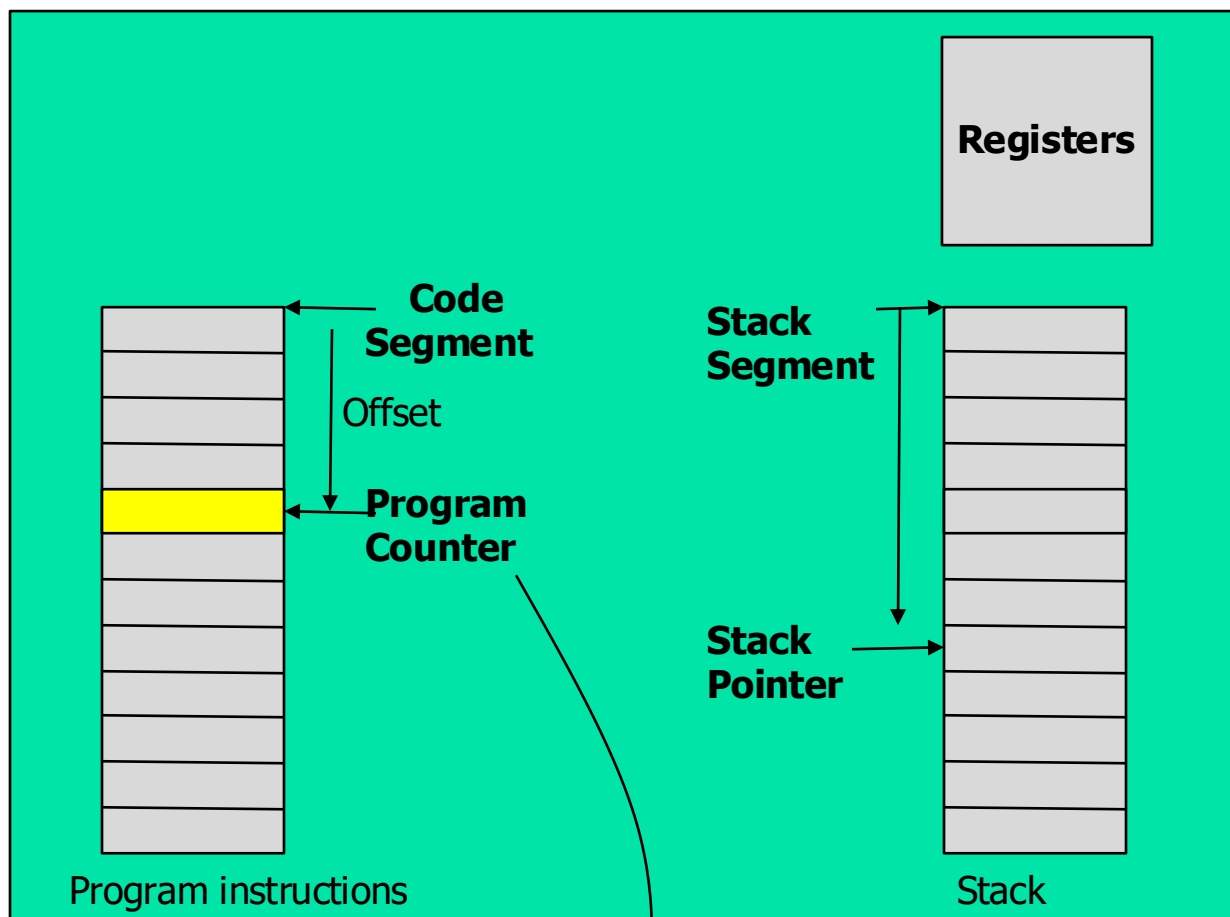


Interrupts to drive scheduling decisions!

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

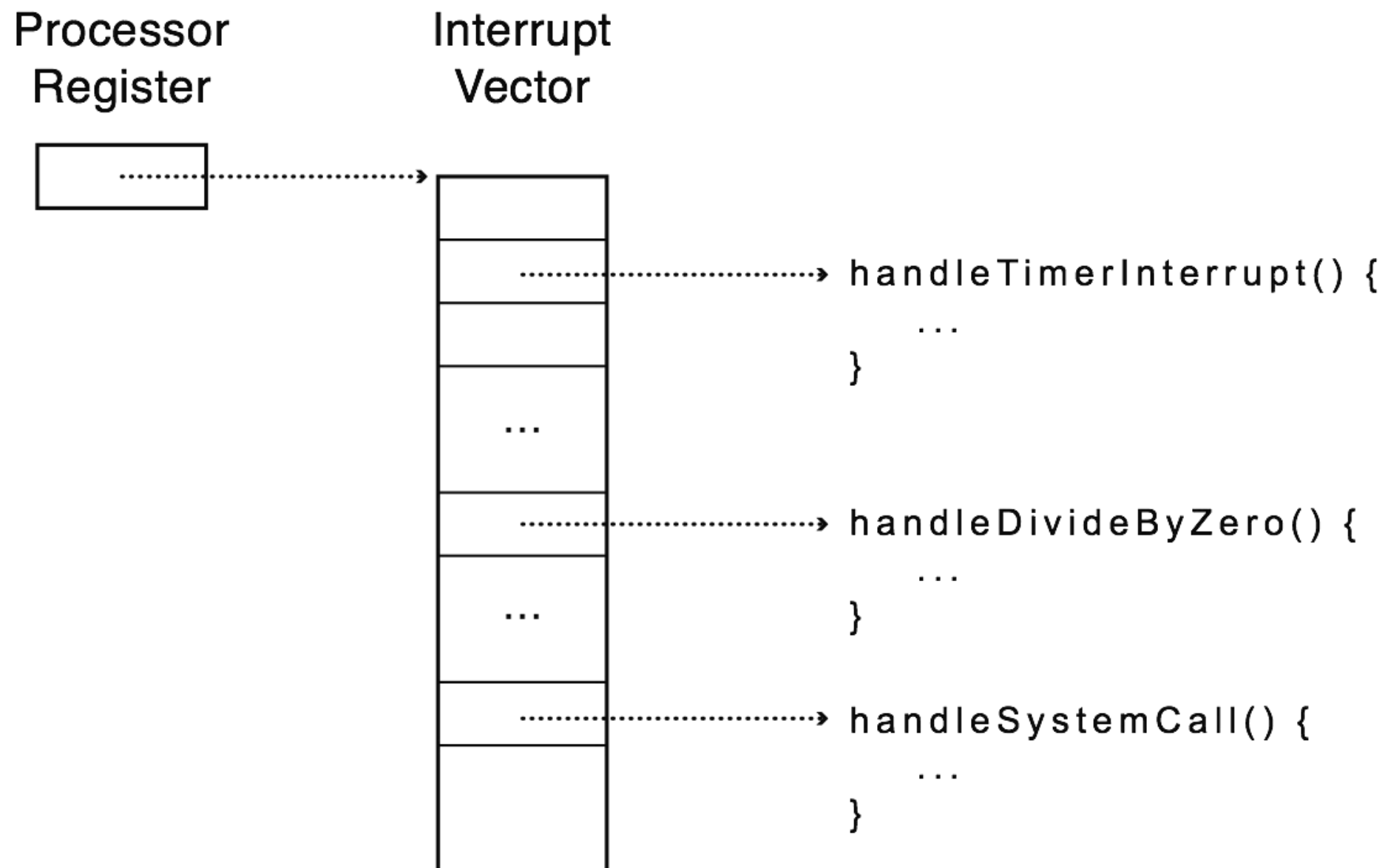


- **Interrupt Vector Table**
 - Where the processor looks for a handler
 - Limited number of entry points into kernel
 - Stored in RAM at a known address
- **Atomic transfer of control**
 - Single instruction to change:
 - Program counter
 - Stack pointer
 - Memory protection
 - Kernel/user mode
- **Transparent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

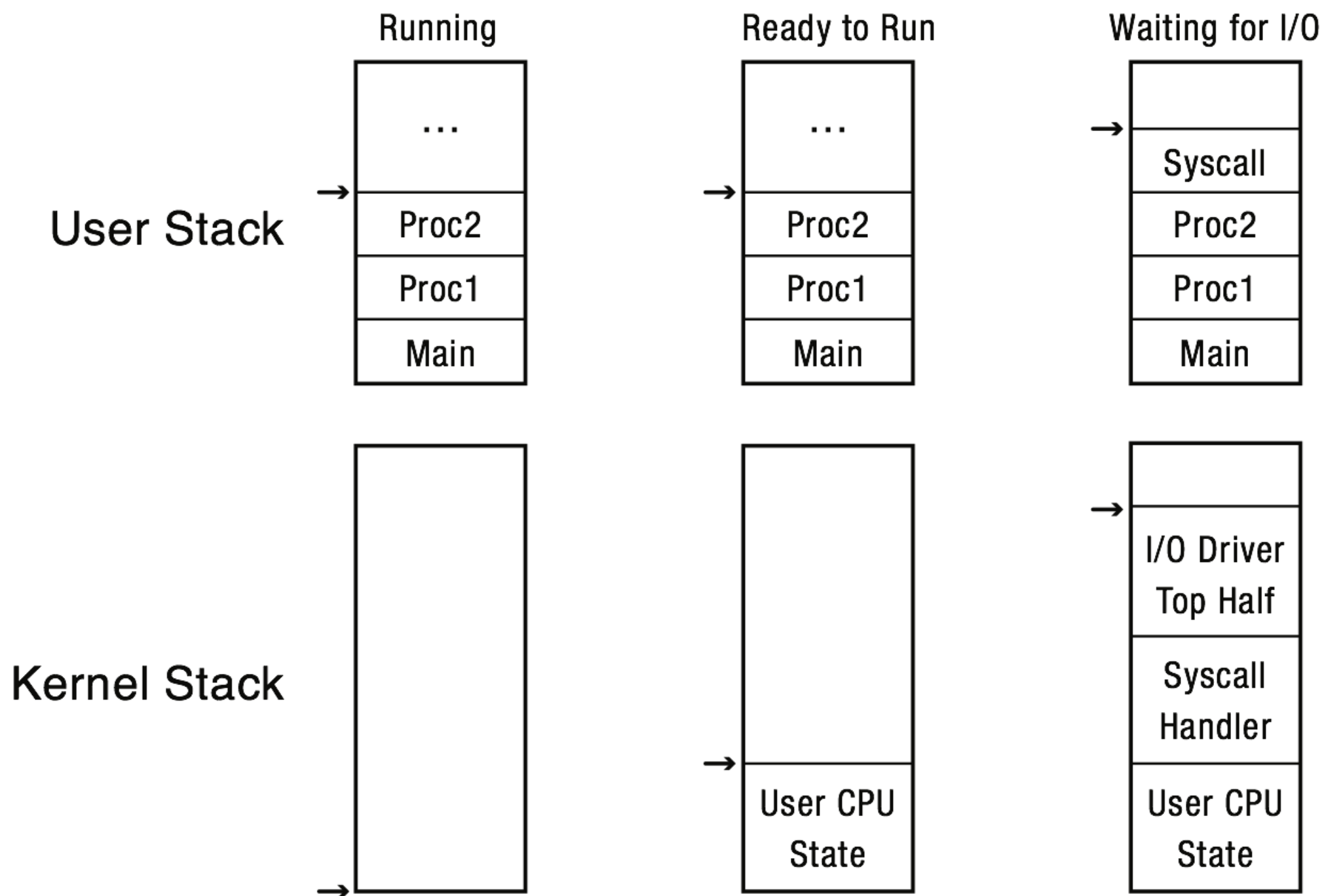


Interrupt Stack



- 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?**

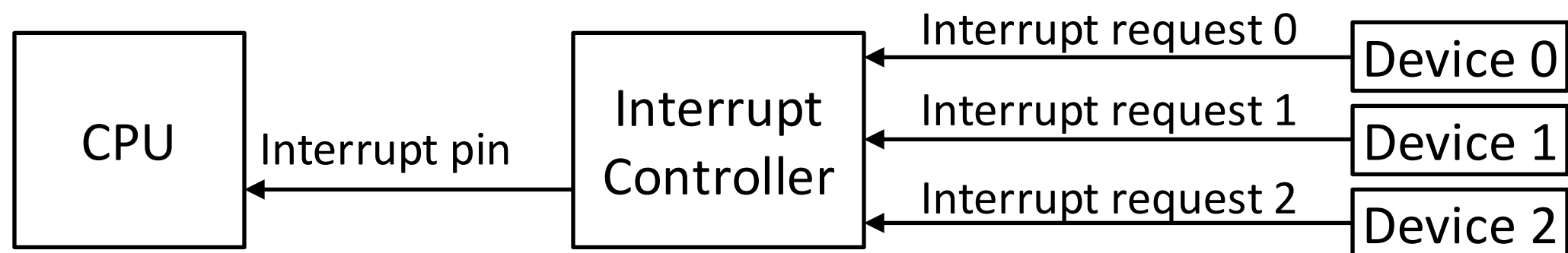
Interrupt Stack



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	GND	VC	GND	GND	GND	VC	GND	GND	GND	VC	GND	GND	GND	VC	GND	GND	1		
2		GND	TERMA	GND	ID0#	GND	ID1#	GND	A07#	GND	A04#	VC	D30#	GND	D27#	VC	D30#	GND	NC	VC	D11#	GND	D07#	VC	D04#	GND	3.3V	VC	2		
3	TUNER(1)	TUNER(2)	TERMB	GND	ID2#	GND	ID3#	GND	A08#	GND	A05#	TERMA	GND	D29#	GND	D26#	VC	D29#	GND	D17#	GND	D13#	GND	D09#	GND	NC	GND	GND	3		
4		GND	OUTEN	GND	ID4#	GND	ID5#	A13#	A10#	GND	DEP3#	VC	D24#	GND	D21#	VC	D24#	GND	D18#	GND	D12#	VC	STEN#	GND	D03#	VC	NC	NC	4		
5	NC	NC	GND	ID6#	GND	ID7#	GND	A11#	GND	A12#	GND	NC	GND	D25#	GND	D22#	VC	D25#	GND	D19#	GND	DEP#	GND	D08#	GND	STEP6#	GND	D02#	GND	5	
6		GND	RFR#	GND	ID8#	GND	ID9#	A09#	A06#	VC	DEP2#	D28#	VC	D28#	GND	D25#	VC	D28#	GND	D18#	VC	D09#	D08#	VC	D05#	GND	NC	VC	6		
7	TDD	TDI	GND	R80#	GND	ID10#	GND	DROV0#	GND	A14#	GND	A09#	TERMA	GND	D31#	GND	D28#	VC	D28#	GND	D21#	GND	D15#	GND	D10#	GND	D00#	GND	THRM	7	
8		GND	INT#	GND	R81#	GND	R82#	A17#	A15#	DEP#	VC	D54#	D48#	VC	D48#	GND	D48#	VC	D48#	GND	D42#	VC	D48#	GND	D37#	VC	NC	GND	8		
9	TMB	TCK	GND	REC0#	GND	D85#	GND	D86#	GND	A21#	GND	A13#	GND	D32#	GND	D29#	VC	D32#	GND	D26#	GND	D20#	GND	D14#	GND	D08#	GND	D02#	GND	VSSMON	9
10		GND	REC1#	GND	REC2#	GND	HT#	A24#	A22#	VC	DEP7#	D51#	VC	STEP3#	D50#	VC	D52#	STEP#	VC	D52#	STEP#	VC	D52#	STEP#	VC	D52#	GND	GND	VTERM	10	
11	NC	NC	GND	REC3#	GND	DROV#	GND	A23#	GND	A23#	GND	NC	GND	D30#	GND	D27#	VC	D30#	GND	D21#	GND	D15#	GND	D09#	GND	D03#	GND	GND	VOCMON	11	
12		GND	REC4#	GND	REC5#	GND	HTM#	A25#	A19#	D52#	VC	D57#	D51#	VC	NC	GND	D48#	VC	D48#	GND	D42#	VC	D48#	GND	D37#	VC	NC	GND	12		
13	BOLVN	BOLVP	GND	SSS0#	GND	RFR#	GND	SSS0#	GND	A22#	GND	A16#	GND	D39#	GND	D36#	VC	D39#	GND	D33#	GND	D27#	GND	D21#	GND	D15#	GND	NC	GND	13	
14		GND	TRDY#	GND	C8EC#	GND	DEFR#	A34#	A31#	VC	D94#	D87#	VC	D87#	NC	VC	D79#	VC	D79#	GND	D73#	VC	D79#	GND	D68#	VC	NC	VC	14		
15	PAR	GOOD	FFOD	GND	LOCK#	GND	TND#	GND	BINT#	GND	A37#	GND	A28#	GND	D32#	GND	D29#	VC	D32#	GND	D26#	GND	D20#	GND	D14#	GND	NC	GND	SMA2	15	
16		GND	EFREC0#	GND	EFREC1#	GND	NC	A36#	A33#	DEP11#	VC	D23#	STEP#	VC	D23#	GND	D17#	VC	D17#	GND	D11#	VC	STEN#	GND	D06#	VC	NC	GND	16		
17	NC	NC	GND	NC	GND	NC	GND	A33#	GND	A32#	GND	ENR#	GND	D38#	GND	D35#	VC	D38#	GND	D32#	GND	D26#	GND	D20#	GND	D14#	GND	D08#	GND	SMA1	17
18		GND	EFREC3#	GND	NC	EFREC2#	GND	A35#	A29#	VC	DEP10#	D25#	VC	D25#	D80#	VC	D77#	D89#	VC	D89#	GND	D83#	VC	D89#	GND	D83#	VC	SMA0	VC	18	
19	NC	NC	GND	EFRR#	GND	SSS1#	GND	SSS1#	GND	A30#	GND	A27#	GND	D30#	GND	D27#	VC	D30#	GND	D24#	GND	D18#	VC	D24#	GND	D12#	VC	GND	GND	19	
20		GND	FFOD	GND	RESET#	GND	A06#	GND	A32#	A48#	GND	DEP14#	VC	D122#	GND	D118#	VC	D118#	GND	D112#	GND	D106#	VC	D102#	VC	D102#	VC	NC	GND	20	
21	NC	NC	GND	TRST#	GND	NC	GND	DROV1#	GND	A44#	GND	A48#	GND	D124#	GND	D120#	GND	D114#	GND	D108#	GND	D102#	GND	D06#	GND	D00#	GND	GND	SAMP	21	
22		GND	UNTO	GND	EPM0#	GND	EBFR#	A49#	A47#	VC	DEP15#	D125#	VC	STEP7#	D114#	VC	D109#	STEP#	VC	D109#	STEP#	VC	D109#	STEP#	VC	D109#	GND	SMBD	VC	22	
23	A20M#	IGNB#	EPM#	GND	EPM#	GND	EPM#	GND	AF1#	GND	A48#	GND	A42#	GND	D126#	GND	D122#	GND	D116#	GND	D110#	GND	D104#	GND	D08#	GND	D02#	GND	GND	23	
24		GND	UNTI	GND	EPM#	GND	EPM#	GND	A43#	A40#	GND	D123#	VC	D120#	GND	D116#	VC	NC	GND	D109#	VC	D109#	GND	D103#	VC	D103#	VC	SMB0	GND	24	
25	FEFR#	TH_TRIP#	PM#	GND	EPM#	GND	AP0#	GND	A41#	GND	VC	GND	D121#	GND	D117#	GND	D113#	GND	D107#	GND	D101#	GND	D05#	GND	D00#	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

Ex: Itanium 2 Pinout



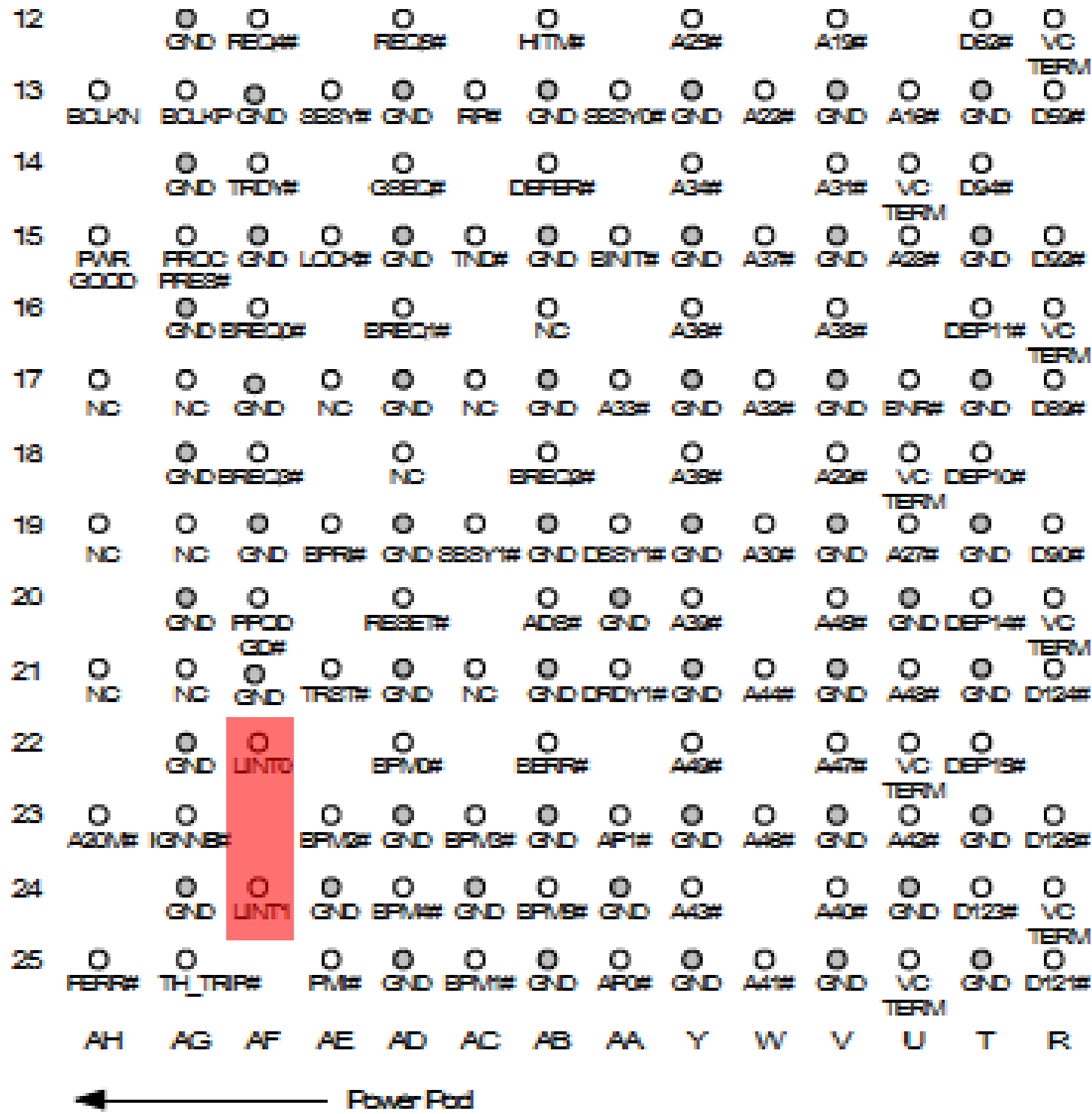
	AH	AG	AF	AE	AD	AC	AB	AA	Y	W	V	U	T	R	P	N	M	L	K	J	H	G	F	E	D	C	B	A			
1	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	VC	TERM	GND	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	GND	1	
2		GND	TERMA	GND	ID0#	GND	ID1#	GND	A07#	GND	A04#	VC	D30#	GND	D27#	VC	D30#	GND	NC	VC	D11#	GND	D07#	VC	D04#	GND	3.3V	VC	TERM	2	
3	TUNER(1)	TUNER(2)	TERMB	GND	ID2#	GND	ID3#	GND	A08#	GND	A05#	TERM	D31#	GND	D28#	VC	D31#	GND	D12#	GND	D13#	GND	D14#	GND	D01#	GND	NC	GND	GND	3	
4		GND	OUTEN	GND	ID4#	GND	ID5#	A13#	GND	A10#	GND	DEP3#	VC	D34#	GND	D31#	VC	D34#	GND	D18#	GND	D12#	VC	STEN#	GND	D03#	VC	NC	NC	4	
5	NC	NC	GND	ID6#	GND	ID7#	GND	A11#	GND	A12#	GND	NC	GND	D32#	GND	STEN#	GND	D19#	GND	DEP#	GND	D08#	GND	STEP6#	GND	D02#	GND	GND	GND	5	
6		GND	RFR#	GND	ID8#	GND	ID9#	A2#	GND	A3#	VC	DEP2#	D35#	GND	D32#	VC	D35#	GND	D19#	VC	D09#	GND	D08#	VC	D05#	GND	NC	VC	TERM	6	
7	TDD	TDI	GND	R80#	GND	ID10#	GND	DROV0#	GND	A14#	GND	A09#	TERM	D36#	GND	D33#	VC	D36#	GND	D21#	GND	DEP0#	GND	D19#	GND	D10#	GND	D00#	GND	THRM	7
8		GND	INT#	GND	R81#	GND	R82#	A17#	GND	A18#	DEP#	VC	D54#	D58#	VC	D58#	GND	D48#	GND	D42#	VC	D48#	GND	D48#	GND	D37#	VC	NC	GND	8	
9	TMB	TKX	GND	REC0#	GND	D85#	GND	D86#	GND	A21#	GND	A13#	GND	D37#	GND	D34#	VC	D37#	GND	D52#	GND	DEP4#	GND	D38#	GND	D40#	GND	D39#	GND	VSSMON	9
10		GND	REC1#	GND	REC2#	GND	HT#	A24#	GND	A23#	VC	DEP7#	D51#	VC	STEP3#	D50#	VC	D52#	GND	STEN#	VC	D52#	GND	D3#	GND	GND	VC	TERM	10		
11	NC	NC	GND	REC3#	GND	DROV#	GND	A23#	GND	A22#	GND	NC	GND	D60#	GND	STEN#	GND	D59#	GND	DEP5#	GND	D41#	GND	STEP2#	GND	D33#	GND	VCCMON	11		
12		GND	REC4#	GND	REC5#	GND	HTM#	A25#	GND	A19#	D62#	VC	D57#	D51#	VC	NC	GND	D48#	VC	D44#	GND	D38#	GND	D3#	GND	VC	TERM	12			
13	BOLVN	BOLVP	GND	SSS0#	GND	RFR#	GND	SSS0#	GND	A22#	GND	A18#	GND	D59#	GND	D56#	VC	D59#	GND	D47#	GND	D43#	GND	D39#	GND	NC	GND	GND	GND	13	
14		GND	TRDY#	GND	C8EC#	GND	DEFR#	A3#	GND	A31#	VC	D94#	D97#	VC	D94#	NC	VC	D79#	GND	D68#	VC	D68#	GND	D6#	GND	NC	VC	TERM	14		
15	PAR	GOOD	GND	FFOC	GND	LOCK#	GND	TND#	GND	BNTH#	GND	A37#	GND	A28#	GND	D92#	GND	D91#	GND	D81#	GND	D78#	GND	D71#	GND	D57#	GND	NC	GND	SMA2	15
16		GND	EPREC0#	GND	EPREC1#	GND	NC	A38#	GND	A38#	DEP11#	VC	D93#	STEP5#	VC	D83#	GND	D78#	VC	STEN#	GND	D68#	VC	D6#	GND	NC	GND	GND	GND	16	
17	NC	NC	GND	NC	GND	NC	GND	A33#	GND	A32#	GND	ENR#	GND	D89#	GND	STEN#	GND	D88#	GND	DEP3#	GND	D72#	GND	STEP4#	GND	D73#	GND	SMA1	GND	17	
18		GND	EPREC3#	GND	NC	EPREC2#	GND	A39#	GND	A29#	VC	DEP10#	D95#	VC	D88#	D80#	VC	D77#	D89#	VC	D64#	GND	D6#	GND	SMA0	VC	TERM	18			
19	NC	NC	GND	EPFR#	GND	SSS1#	GND	SSS1#	GND	A30#	GND	A27#	GND	D90#	GND	D89#	GND	D82#	GND	DEP8#	GND	D75#	GND	D74#	GND	D70#	GND	GND	GND	19	
20		GND	FFOC	GND	RESET#	GND	A06#	GND	A36#	A48#	GND	DEP14#	VC	D122#	GND	D118#	VC	D117#	GND	D111#	VC	D108#	GND	D102#	VC	NC	GND	GND	GND	20	
21	NC	NC	GND	TRST#	GND	NC	GND	DROV1#	GND	A44#	GND	A48#	GND	D124#	GND	D122#	GND	D112#	GND	DEP12#	GND	D101#	GND	D98#	GND	D92#	GND	GND	SAMP	21	
22		GND	UNTI	GND	EPM0#	GND	EBFR#	A49#	GND	A47#	VC	DEP15#	D125#	VC	STEP7#	D114#	VC	D109#	STEN#	VC	D98#	GND	SMBD	VC	TERM	22					
23	A20M#	IGNB#	GND	EPM#	GND	EPM#	GND	AF1#	GND	A48#	GND	A42#	GND	D126#	GND	STEN#	GND	D116#	GND	DEP13#	GND	D108#	GND	STEP6#	GND	D97#	GND	GND	GND	23	
24		GND	UNTI	GND	EPM#	GND	EPM#	GND	A43#	A40#	GND	D123#	VC	D120#	GND	D119#	VC	NC	GND	D109#	VC	D103#	GND	D104#	VC	SMB0	GND	GND	GND	24	
25	FEFR#	TH_TRIP#	GND	PM#	GND	EPM#	GND	AP0#	GND	A41#	GND	VC	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

UUU638b



Ex: Itanium 2 Pinout



LINTx — lines/pins for hardware interrupts.

In this case...

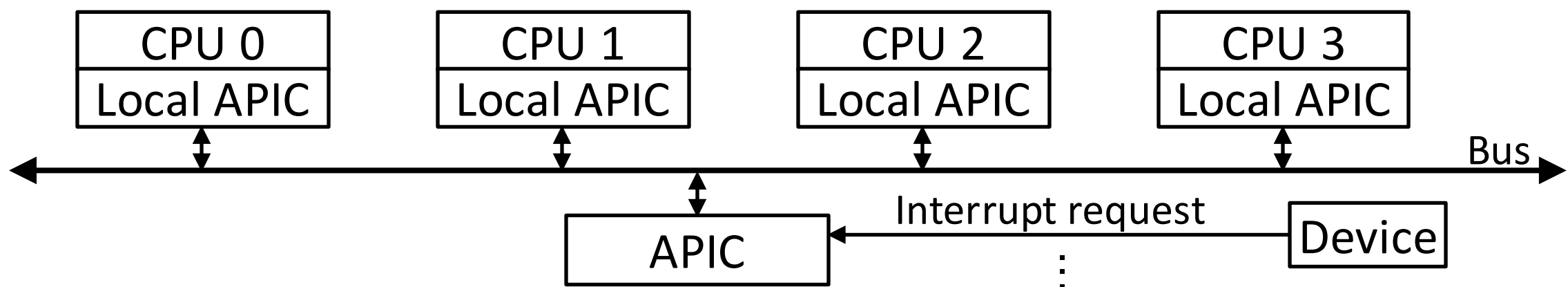
LINT0 — line for unmaskable interrupts

LINT1 — line for maskable interrupts

A Note on Multicore



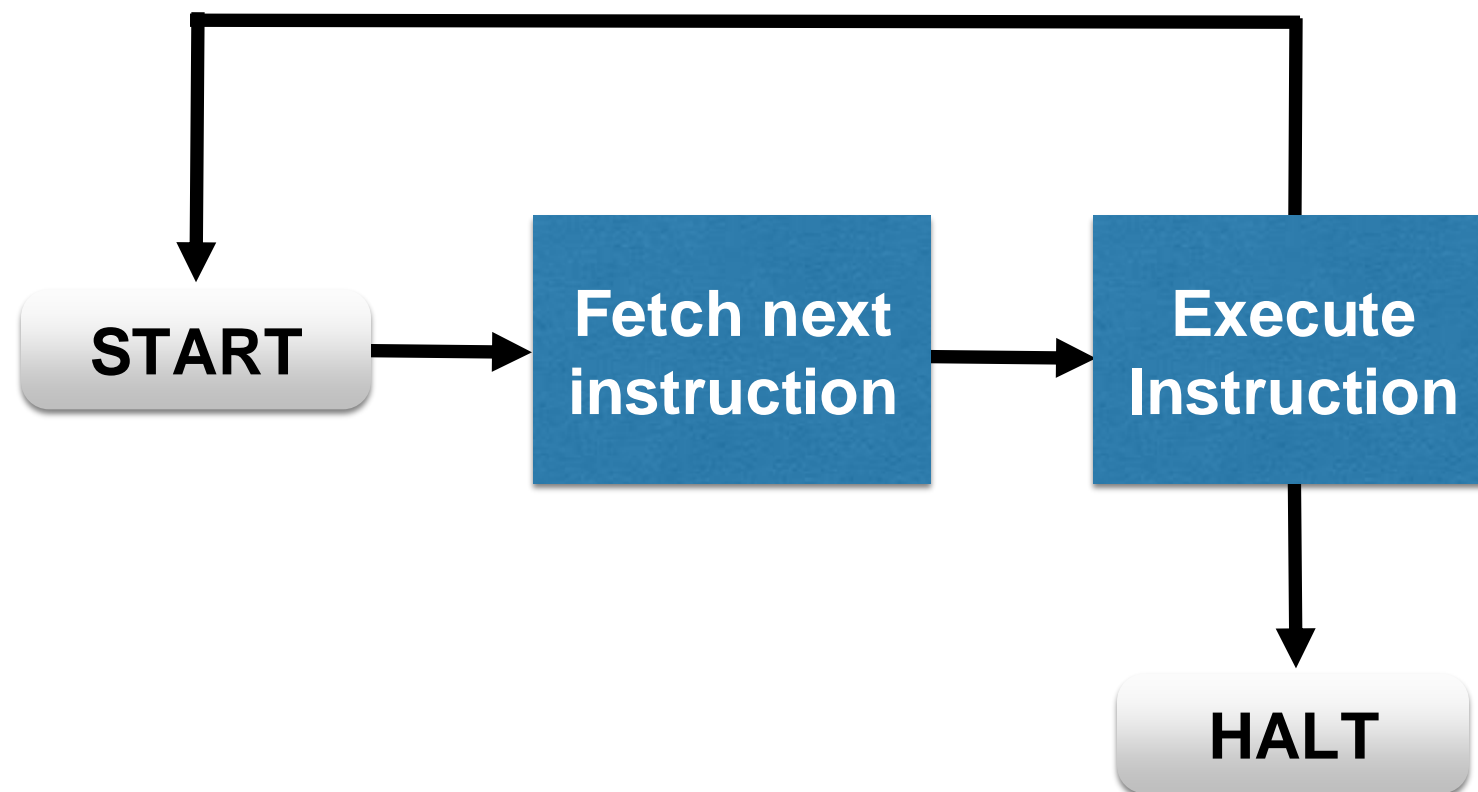
- How are interrupts handled on multicore machines?
 - On x86 systems each CPU gets its own local **Advanced Programmable Interrupt Controller (APIC)**. They are wired in a way that allows routing device interrupts to any selected local APIC.
 - The OS can program the APICs to determine which interrupts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



Instruction Cycle



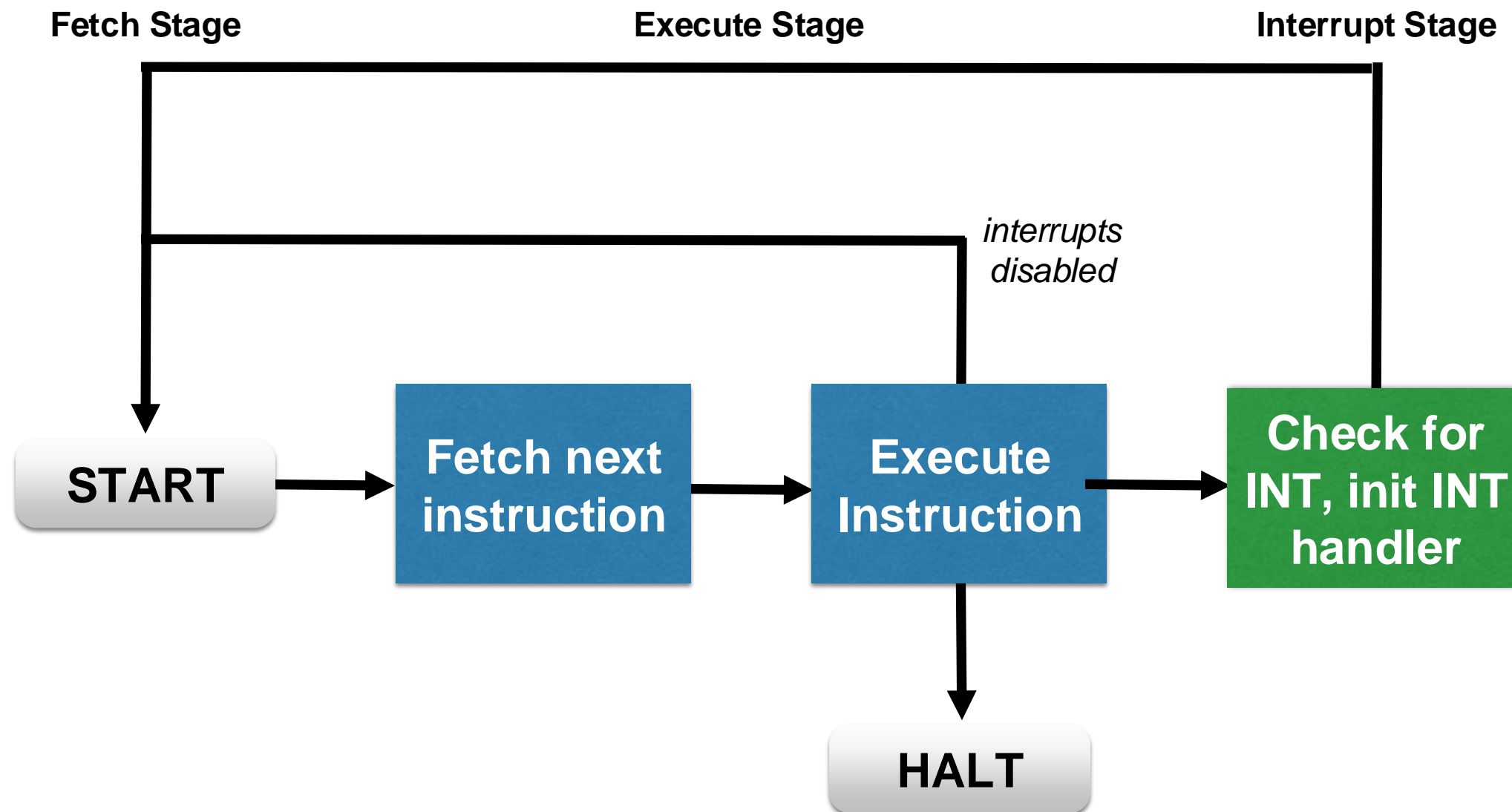
How does interrupt handling change the instruction cycle?



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.

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_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
- ...

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);`