



CS 423

Operating System Design: The Kernel Abstraction

Tianyin Xu

* Thanks for Prof. Adam Bates for the slides.

Discussion: Last Class

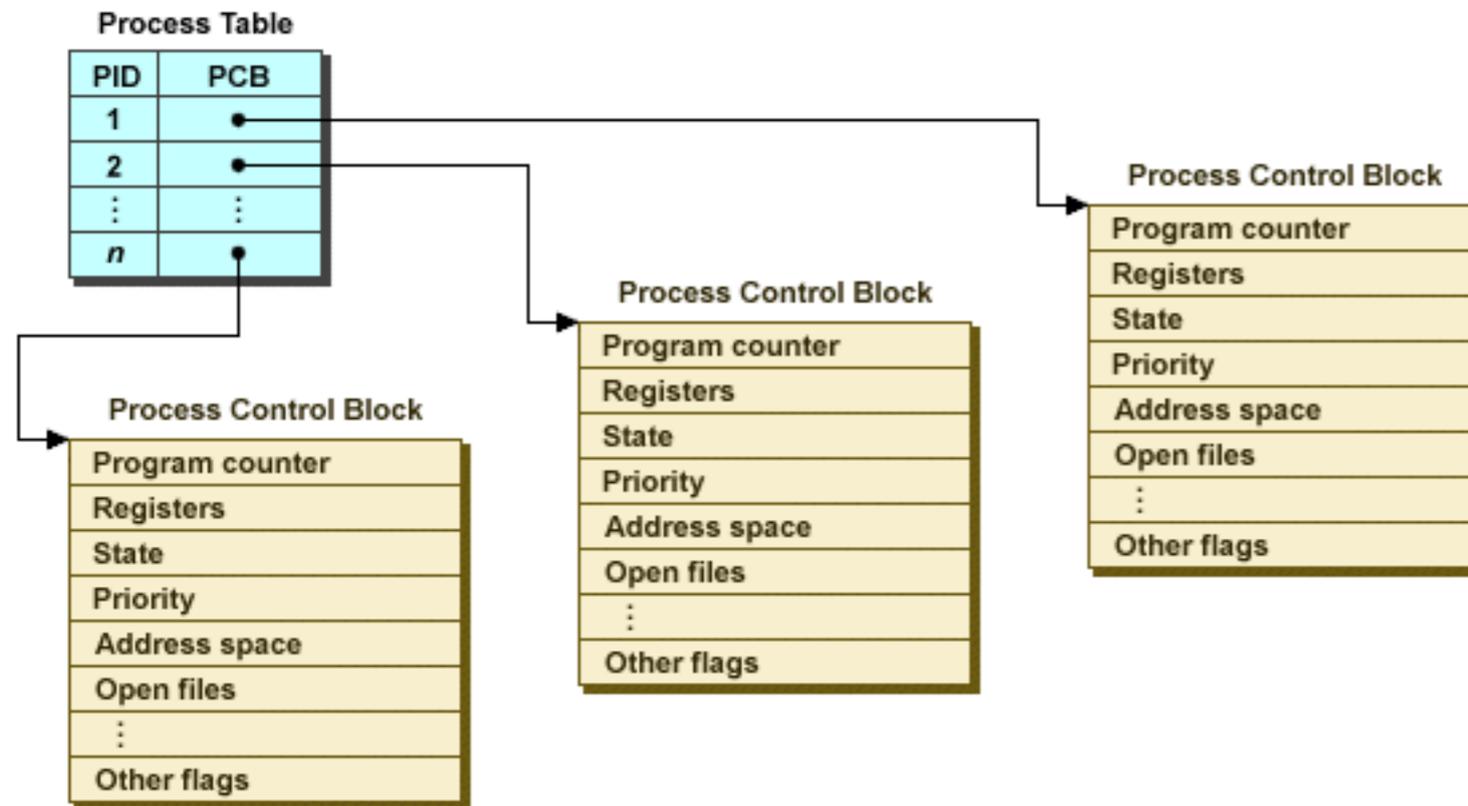


- Where is CPU State physically stored for active task?
 - Registers!
 - Program Counter is a register
 - Segment Registers
 - Code Segment
 - Data Segment
 - Stack Segment
- CPU has access to RAM and can save PC to stack before context switching.

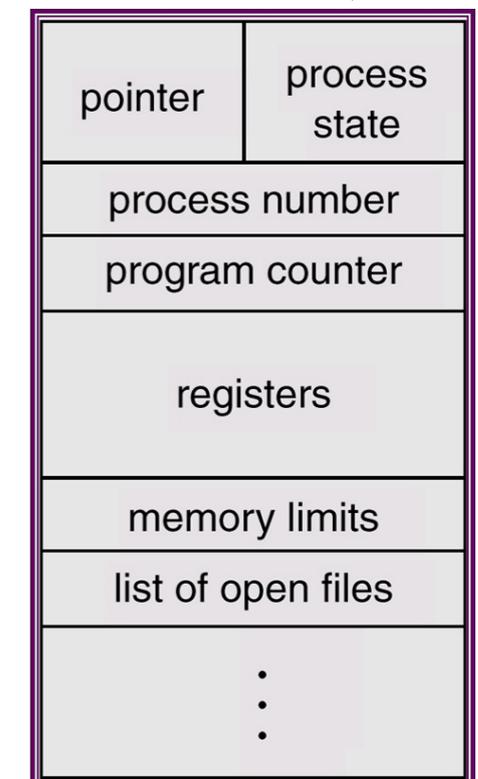
Process Control Block



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



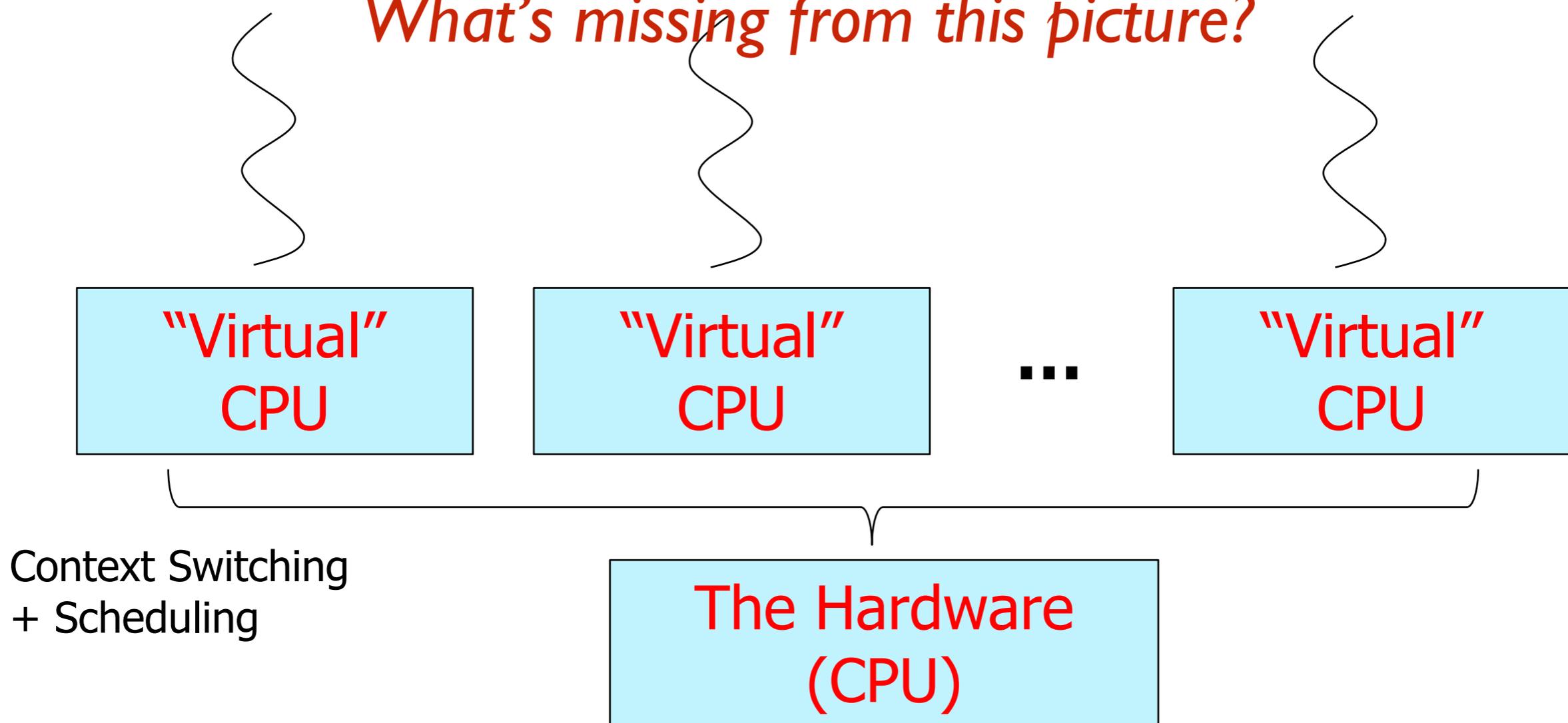
An alternate PCB diagram

Where We Are:



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

What's missing from this picture?

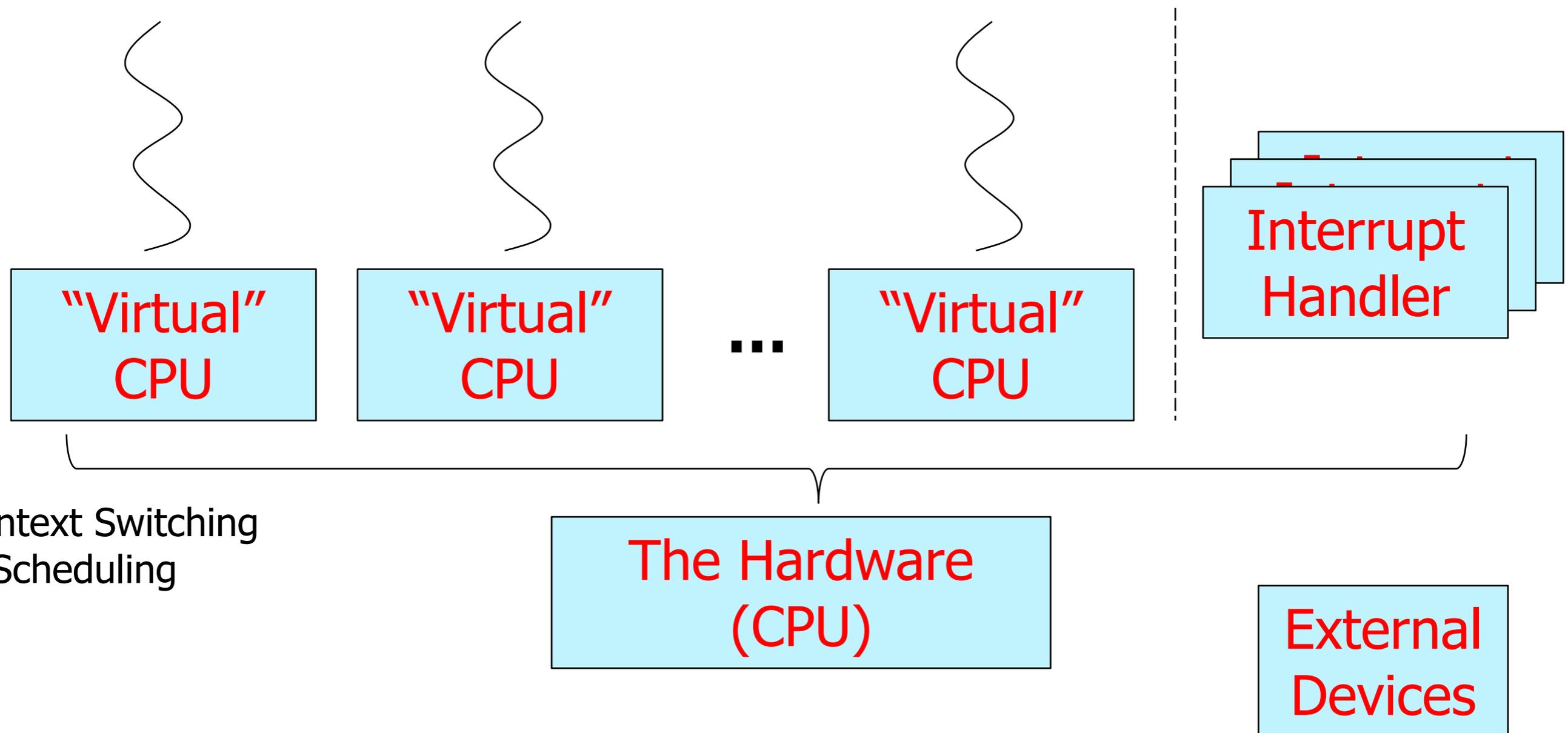


Where We Are:

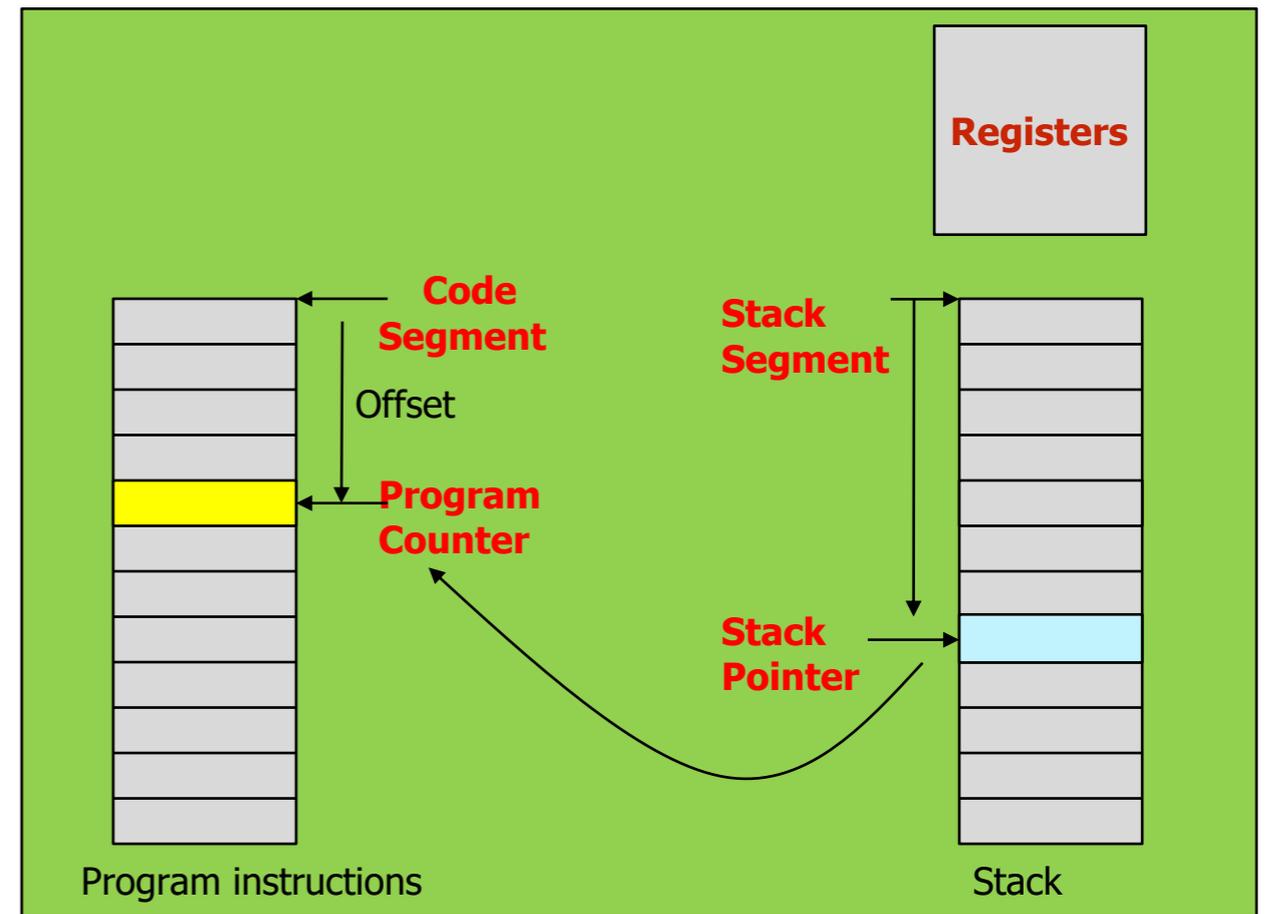
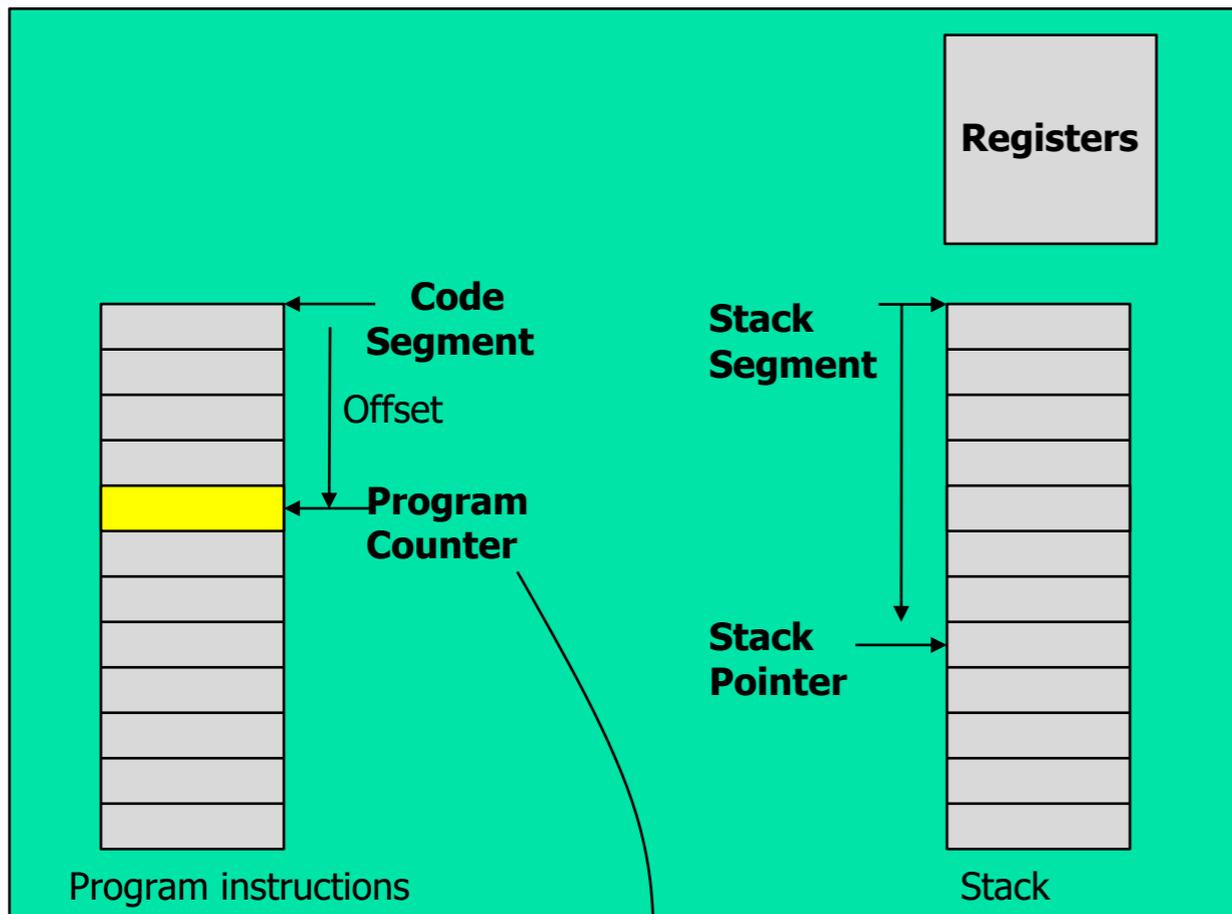


Interrupts to drive scheduling decisions!

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

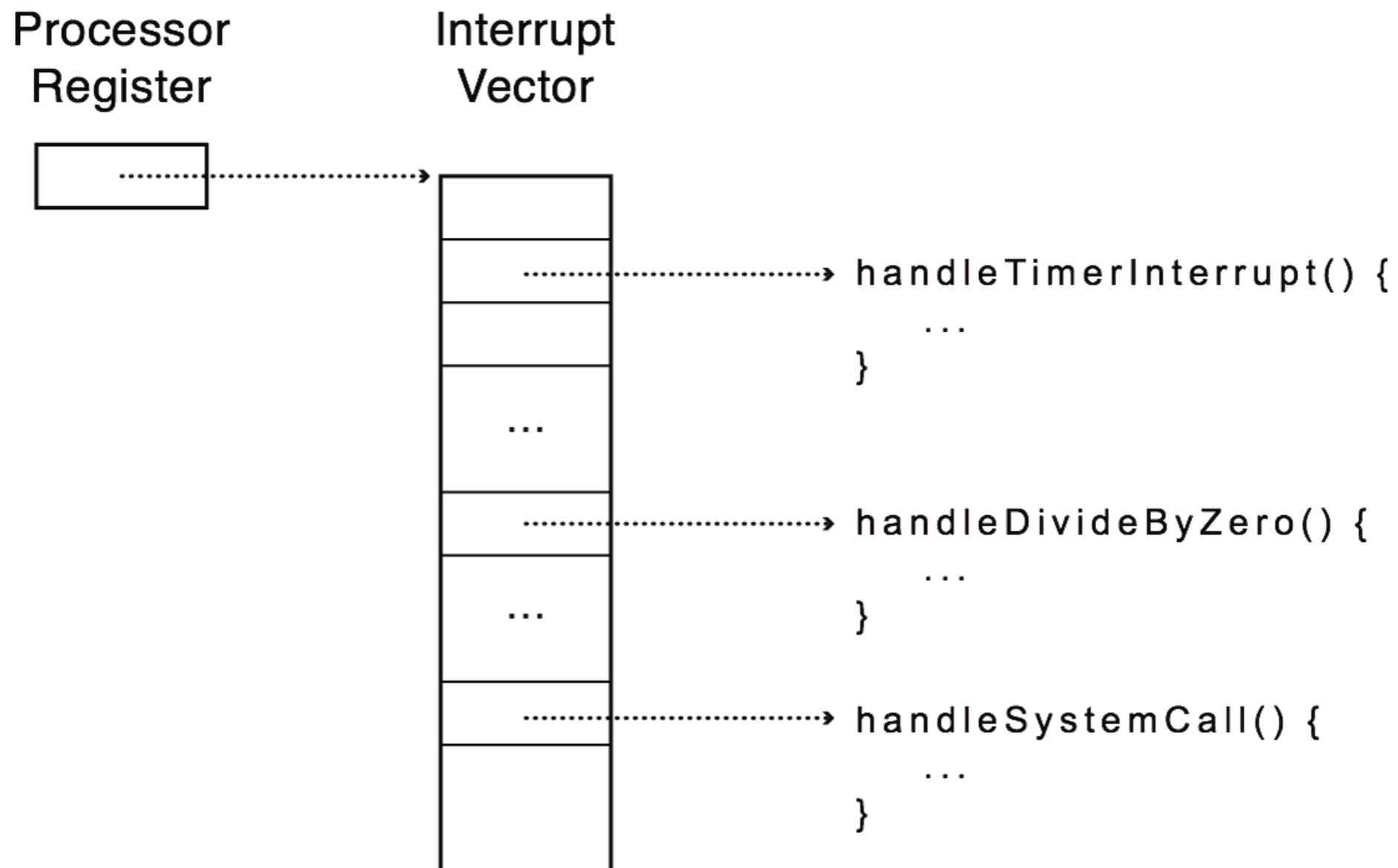


- **Interrupt Vector Table**
 - Where the processor looks for a handler
 - Limited number of entry points into kernel
 - Stored in RAM at a known address
- **Atomic transfer of control**
 - Single instruction to change:
 - Program counter
 - Stack pointer
 - Memory protection
 - Kernel/user mode
- **Transparent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

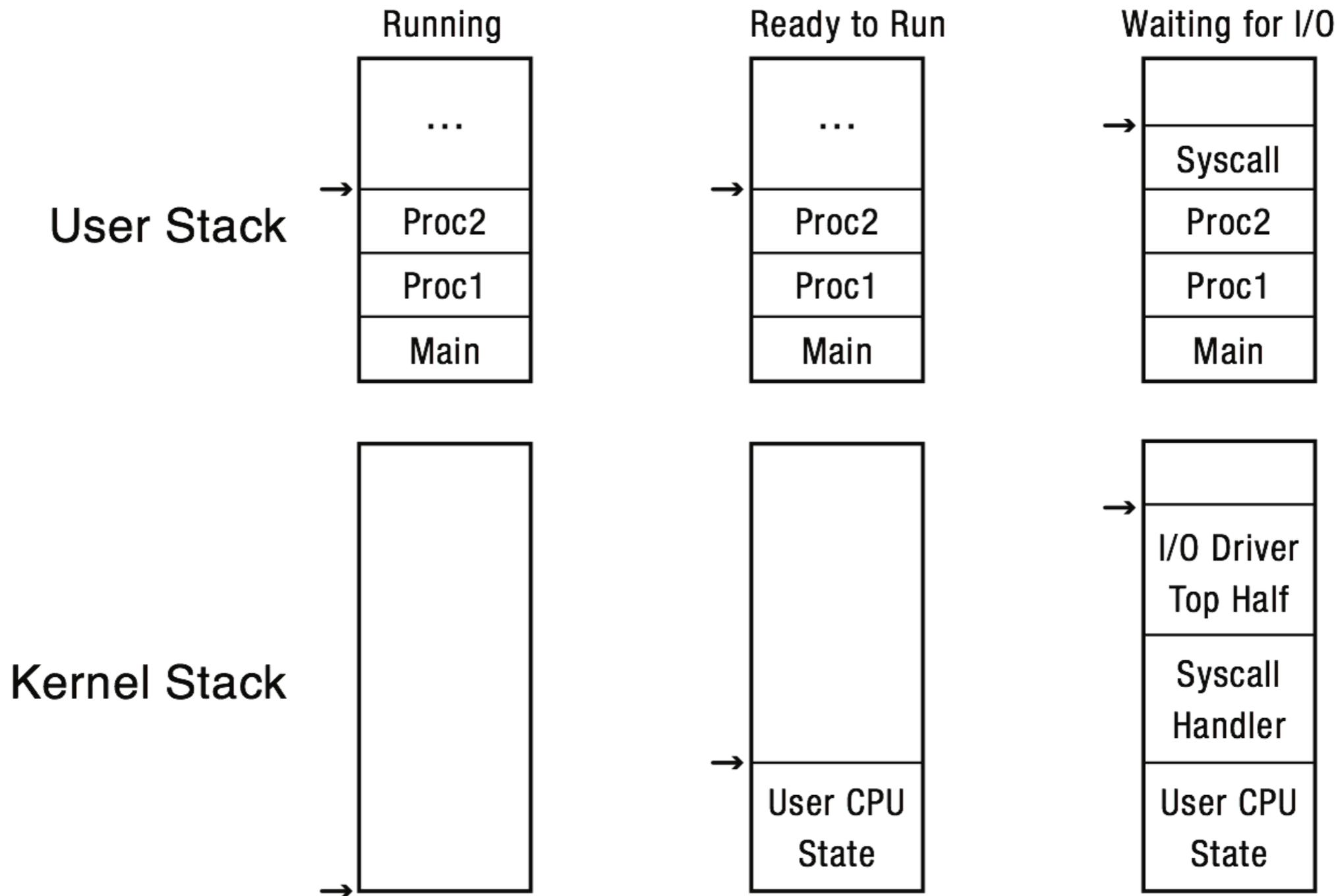


Interrupt Stack



- 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	TERM	GND	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	1
2		TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	TERM	2
3	TUNER(1)	TUNER(2)	TERM	TERM	3																								
4		OUTEN	TERM	TERM	4																								
5	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	5	
6		REF#	TERM	TERM	6																								
7	TDD	TDI	REF#	7																									
8		INT#	REF#	8																									
9	TMB	TCK	REF#	9																									
10		REQ#	REQ#	HT#	A2#	10																							
11	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	11	
12		REQ#	REQ#	HT#	A2#	12																							
13	BOLVN	BOLVP	SSSY#	13																									
14		TRDY#	SSSY#	14																									
15	PAR	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	15	
16	GOOD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	16	
17	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	17	
18		FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	18	
19	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	19	
20		FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	20	
21	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	21	
22		FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	22	
23	A20#	IGNB#	EPM#	23																									
24		FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	FRCD	24	
25	FEFF#	TH_TRIP#	PM#	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

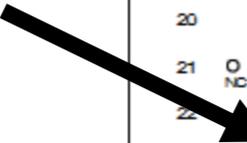
UUU638b

Ex: Itanium 2 Pinout

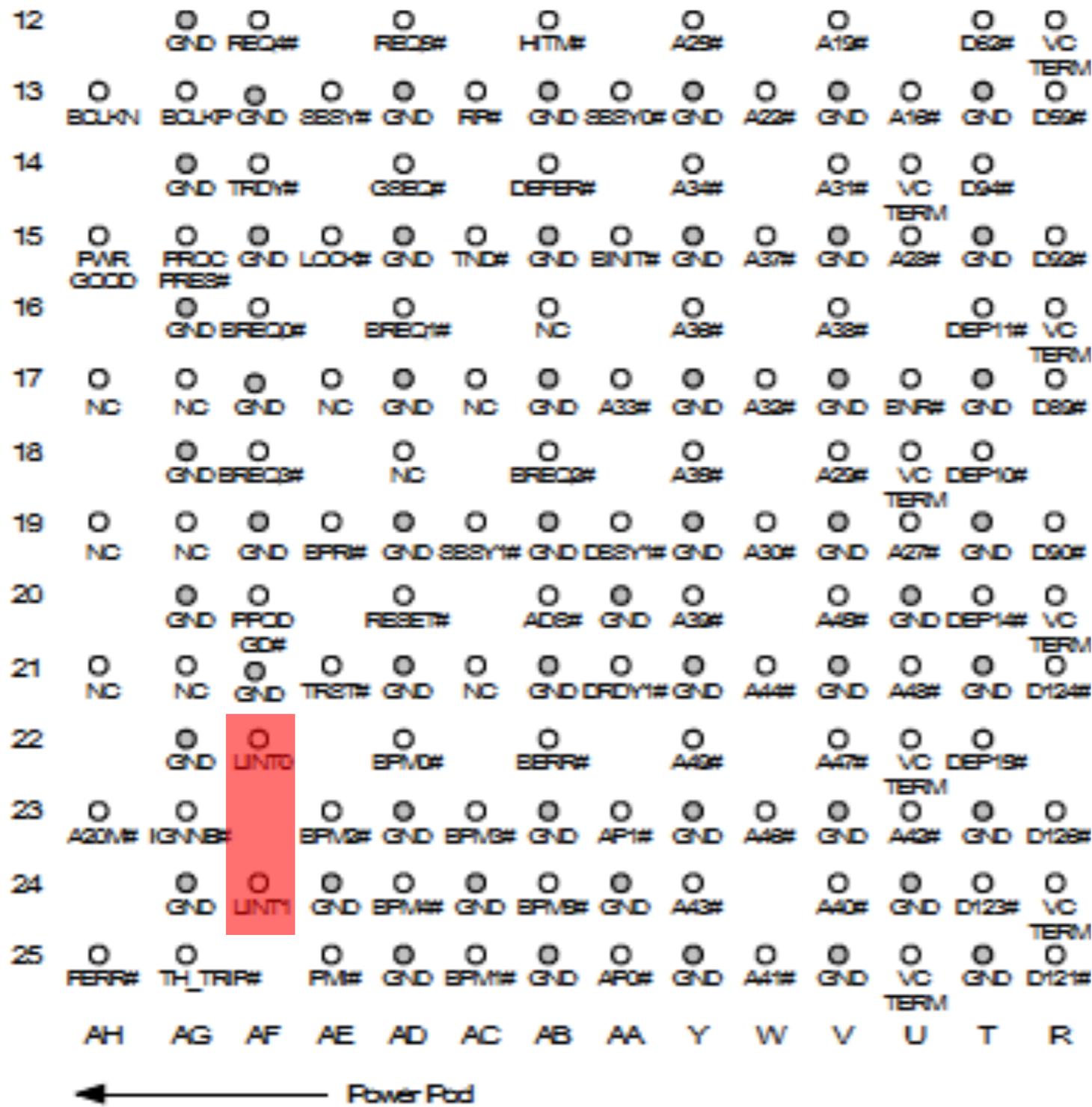


	AH	AG	AF	AE	AD	AC	AB	AA	Y	W	V	U	T	R	P	N	M	L	K	J	H	G	F	E	D	C	B	A	
1	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	VC	TERM	GND	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	GND	VC	TERM	GND	1
2		GND	TERMA	GND	IDG#	GND	ID1#	GND	A0#	GND	A0#	VC	D30#	GND	D27#	VC	D30#	GND	NC	VC	D1#	GND	D07#	VC	D04#	GND	3.3V	VC	2
3	TUNER(1)	TUNER(2)	TERMB	GND	IDG#	GND	ID3#	GND	A0#	GND	A0#	VC	D30#	GND	D29#	VC	D30#	GND	D13#	GND	D13#	GND	D14#	GND	D01#	GND	NC	GND	3
4		GND	OUTEN	GND	IDG#	GND	ID5#	A13#	A10#	GND	DEP3#	VC	D24#	GND	STEP1#	VC	D18#	GND	D12#	VC	STEN#	GND	D03#	VC	NC	NC	NC	4	
5	NC	NC	GND	IDG#	GND	ID7#	GND	A11#	GND	A12#	GND	NC	GND	D29#	GND	STEN#	GND	D19#	GND	DEP#	GND	D08#	GND	STEP6#	GND	D02#	GND	GND	5
6		GND	RFR#	GND	IDG#	GND	ID9#	A3#	A3#	VC	DEP2#	D28#	VC	D28#	D18#	VC	D09#	D08#	VC	D09#	D08#	VC	D09#	GND	NC	VC	NC	6	
7	TDD	TDI	GND	R80#	GND	IDG#	GND	DROV#	GND	A14#	GND	A0#	VC	D31#	GND	D22#	GND	D21#	GND	DEP0#	GND	D19#	GND	D10#	GND	D00#	GND	THRM	7
8		GND	INT#	GND	R81#	GND	R82#	A17#	A19#	GND	DEP#	VC	D54#	D48#	VC	D48#	GND	D42#	VC	D48#	GND	D48#	GND	D37#	VC	NC	GND	8	
9	TMB	TCK	GND	REC0#	GND	D8Y#	GND	D8Y0#	GND	A21#	GND	A13#	GND	D31#	GND	D28#	GND	D21#	GND	DEP#	GND	D38#	GND	D40#	GND	D38#	GND	VSSMON	9
10		GND	REC1#	GND	REC2#	GND	HT#	A24#	A20#	VC	DEP7#	D51#	VC	STEP2#	D50#	VC	D52#	STEP2#	VC	D52#	STEP2#	VC	D52#	GND	GND	VC	TERM	10	
11	NC	NC	GND	REC3#	GND	DROV#	GND	A23#	GND	A28#	GND	NC	GND	D30#	GND	STEN#	GND	D58#	GND	DEP5#	GND	D41#	GND	STEP2#	GND	D33#	GND	VDDMON	11
12		GND	REC4#	GND	REC5#	GND	HT#	A23#	A19#	GND	D52#	VC	D57#	D51#	VC	NC	GND	D48#	VC	D48#	GND	D48#	GND	D38#	GND	VC	NC	12	
13	BOLVN	BOLVP	GND	S8Y#	GND	RFR#	GND	S8Y0#	GND	A22#	GND	A18#	GND	D59#	GND	D58#	GND	D52#	GND	D47#	GND	D43#	GND	D38#	GND	NC	GND	GND	13
14		GND	TROY#	GND	G8EC#	GND	DEFR#	A34#	A31#	VC	D94#	D87#	VC	D84#	NC	VC	D79#	D68#	VC	D79#	D68#	VC	D79#	GND	NC	VC	NC	14	
15	PAR	GOOD	GND	LOOK#	GND	TND#	GND	SINT#	GND	A37#	GND	A23#	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#	A38#	VC	DEP11#	D93#	VC	STEP5#	D83#	GND	D78#	VC	STEN#	GND	D68#	VC	NC	NC	NC	NC	GND	16	
17	NC	NC	GND	NC	GND	NC	GND	A33#	GND	A32#	GND	ENR#	GND	D89#	GND	STEN#	GND	D83#	GND	DEP3#	GND	D72#	GND	STEP4#	GND	D73#	GND	SMA1	17
18		GND	EPREC3#	GND	NC	EPREC2#	GND	A38#	A22#	VC	DEP10#	D95#	VC	D88#	D80#	VC	D77#	D69#	VC	D64#	GND	SMA0	VC	NC	NC	VC	NC	18	
19	NC	NC	GND	EPFR#	GND	S8Y1#	GND	D8Y1#	GND	A30#	GND	A27#	GND	D90#	GND	D89#	GND	D82#	GND	DEP8#	GND	D75#	GND	D74#	GND	D70#	GND	GND	19
20		GND	FFOD	GND	RESET#	GND	A08#	GND	A36#	A48#	GND	DEP14#	VC	D122#	GND	D113#	VC	D117#	GND	D111#	VC	D108#	GND	D102#	VC	NC	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	SAMP	21
22		GND	UNTI	GND	EPM0#	GND	EBFR#	A49#	A47#	VC	DEP15#	D125#	VC	STEP7#	D114#	VC	D109#	STEP6#	VC	D98#	GND	SMD	VC	NC	NC	VC	NC	22	
23	A20#	IGNB#	EPM#	GND	EPM#	GND	EPM#	GND	AF1#	GND	A48#	GND	A42#	GND	D128#	GND	STEN7#	GND	D118#	GND	DEP13#	GND	D108#	GND	STEP5#	GND	D97#	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	24	
25	FERR#	TH_TRIP#	PM#	GND	EPM1#	GND	AP0#	GND	A41#	GND	VC	D121#	GND	D119#	GND	D113#	GND	D110#	GND	D107#	GND	D100#	GND	NC	NC	VC	NC	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

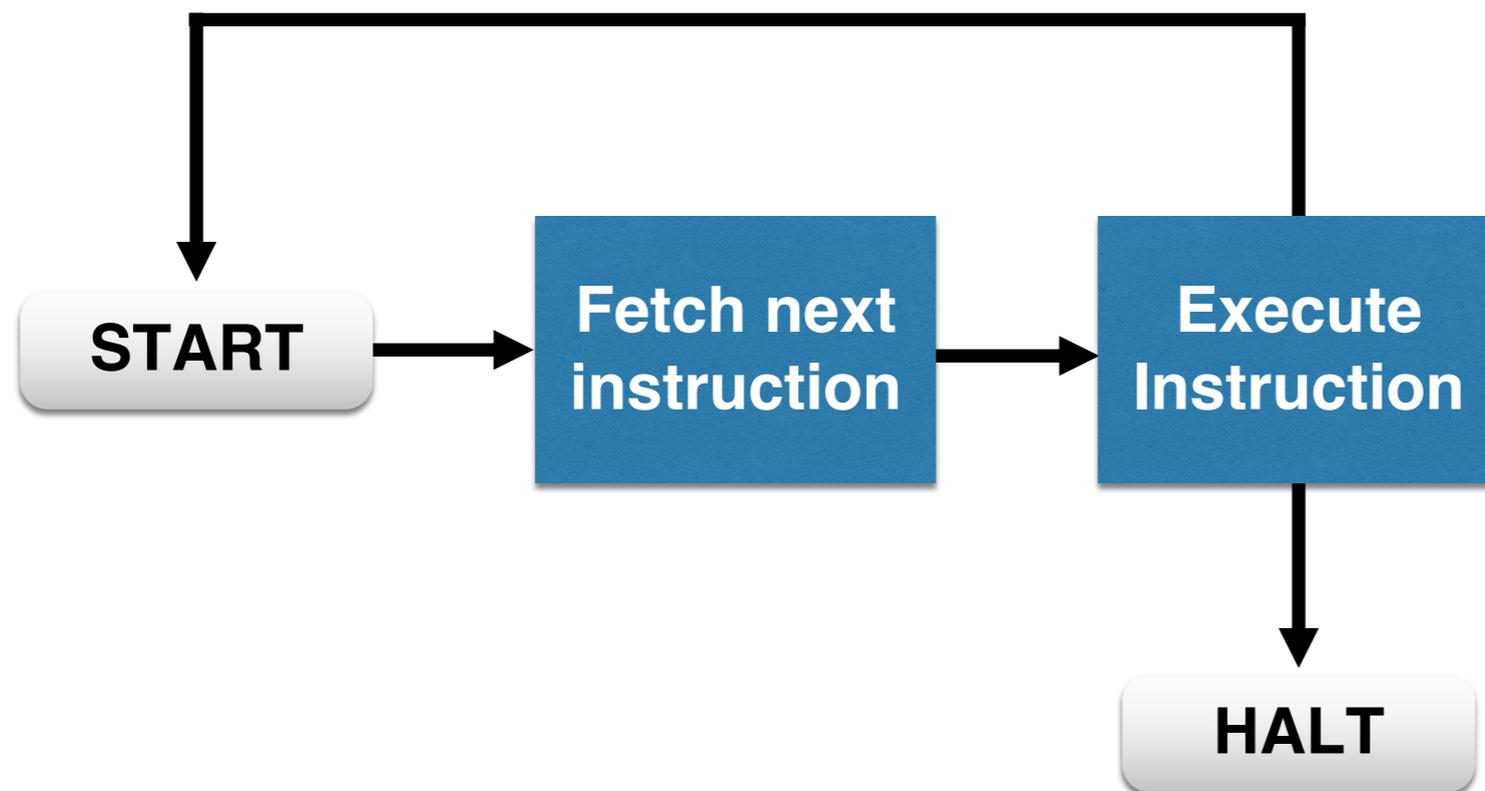


- How are interrupts handled on multicore machines?
 - On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
 - The OS can program the APICs to determine which interrupts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

Instruction Cycle



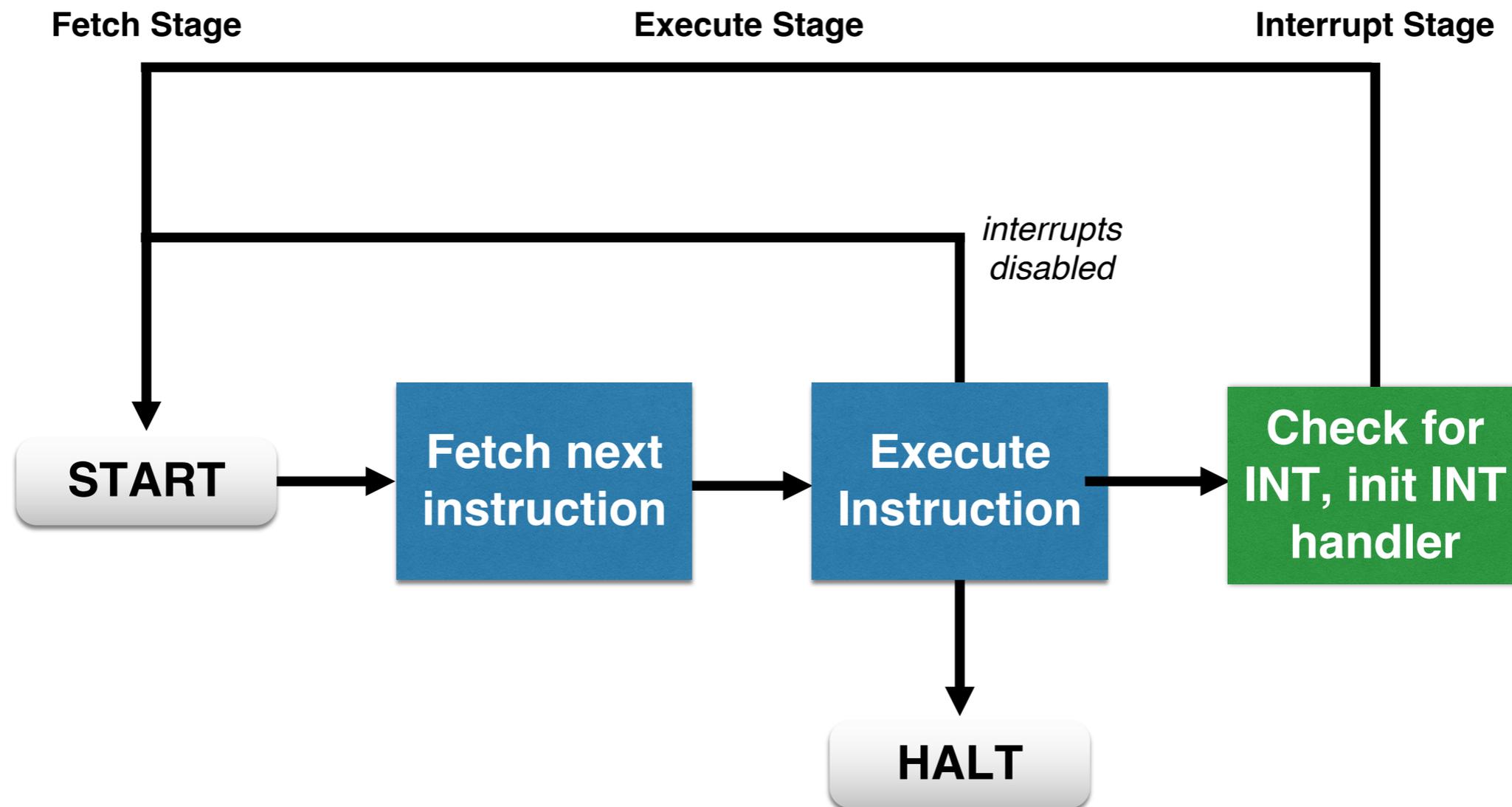
How does interrupt handling change the instruction cycle?



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



- 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



- Handlers that, like bottom halves, must be statically defined/allocated in the Linux kernel at compile time.
- 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
- 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 task lets 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);`