



CS 423

Operating System Design:

Interrupts

Professor Adam Bates

Goals for Today



- Learning Objectives:

- Understand the role and types of Interrupts



- Announcements:

- C4 weekly summaries! **Due TODAY (UTC-11)**
- HW0 is available on Compass! **Due TODAY (UTC-11)**
- MP0 is available on Compass! **Due Jan 28**

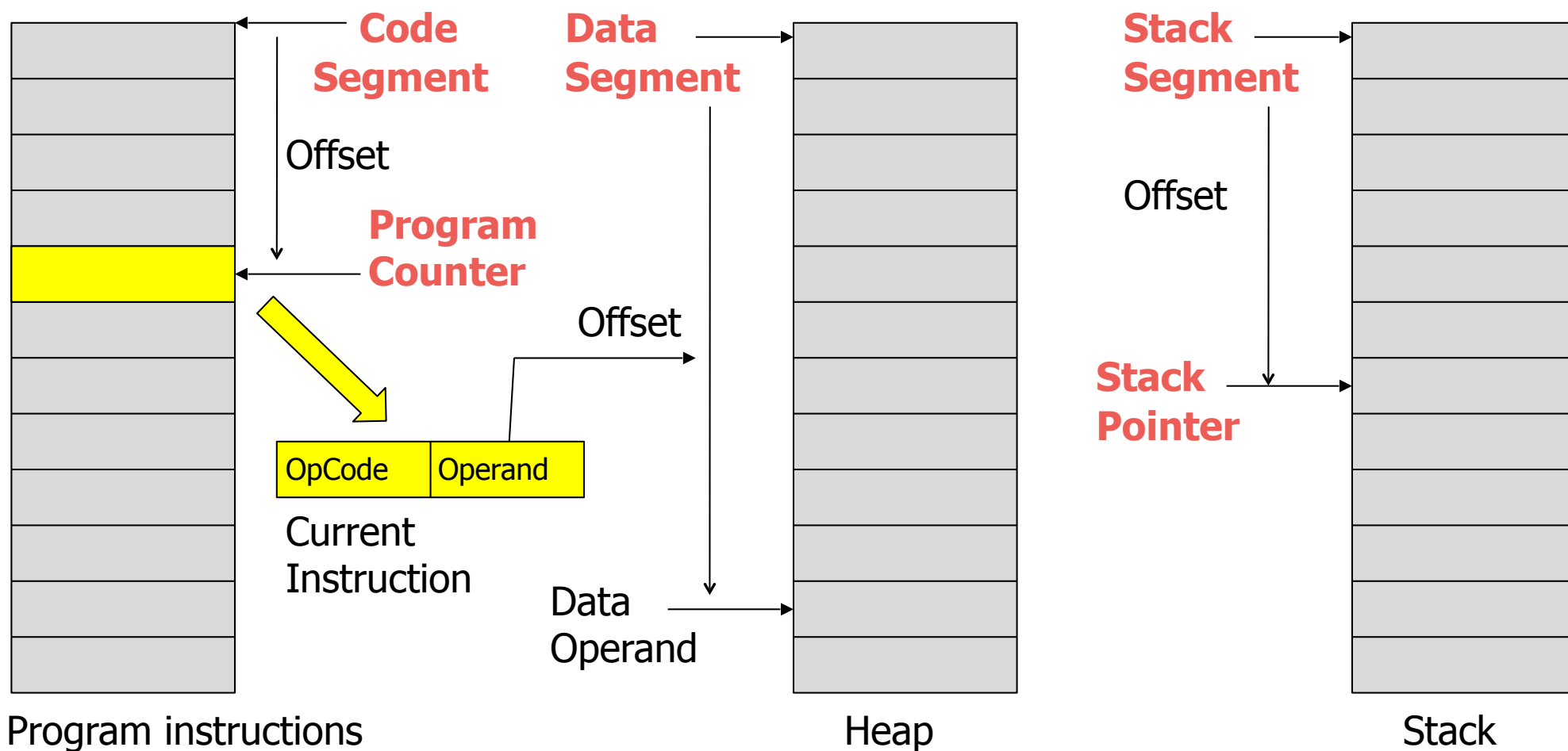
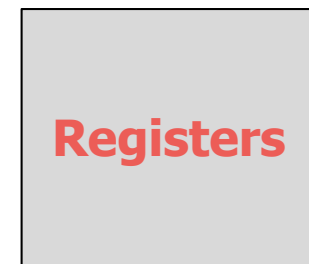


Reminder: Please put away devices at the start of class

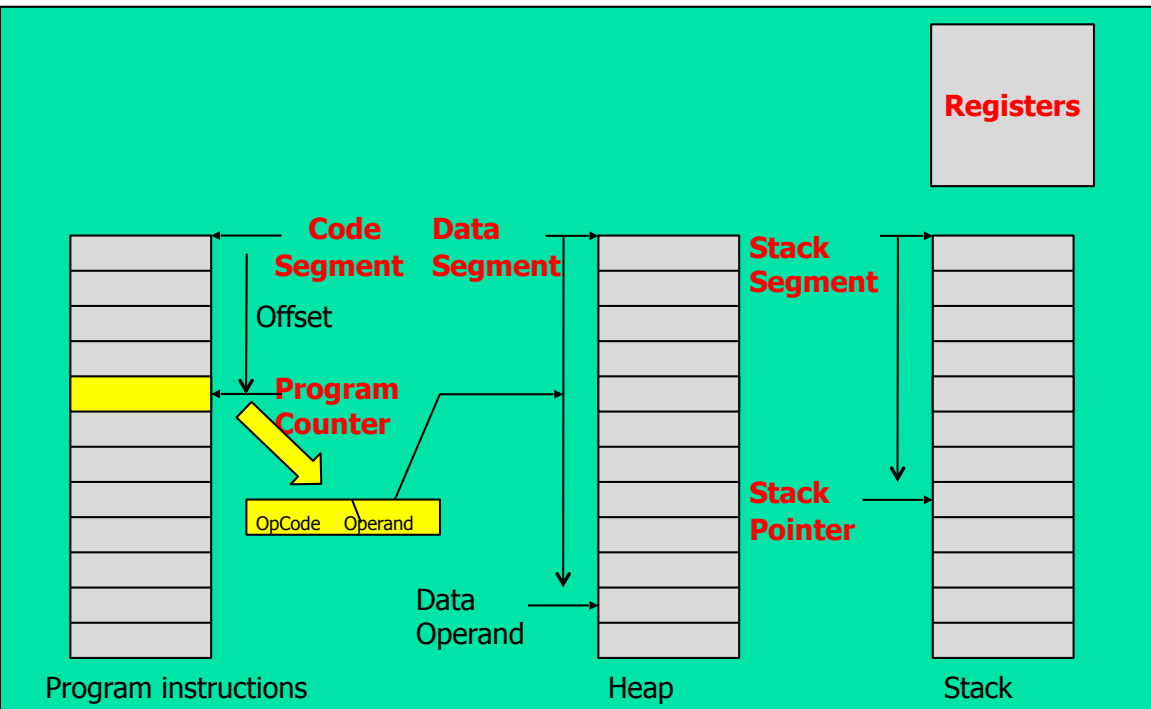
What's a 'real' CPU?



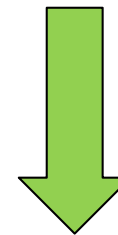
What's the **STATE** of a real CPU?



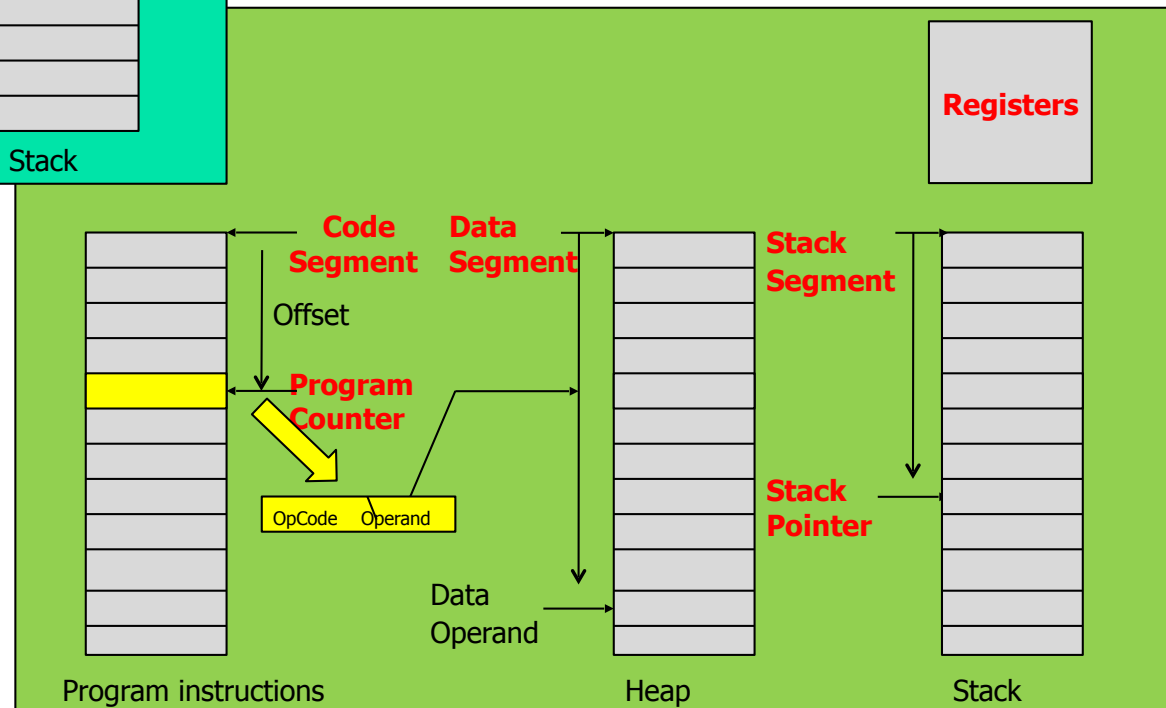
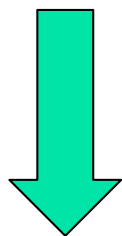
The Context Switch



**Load State
(Context)**



**Save State
(Context)**



Discussion: Last Class

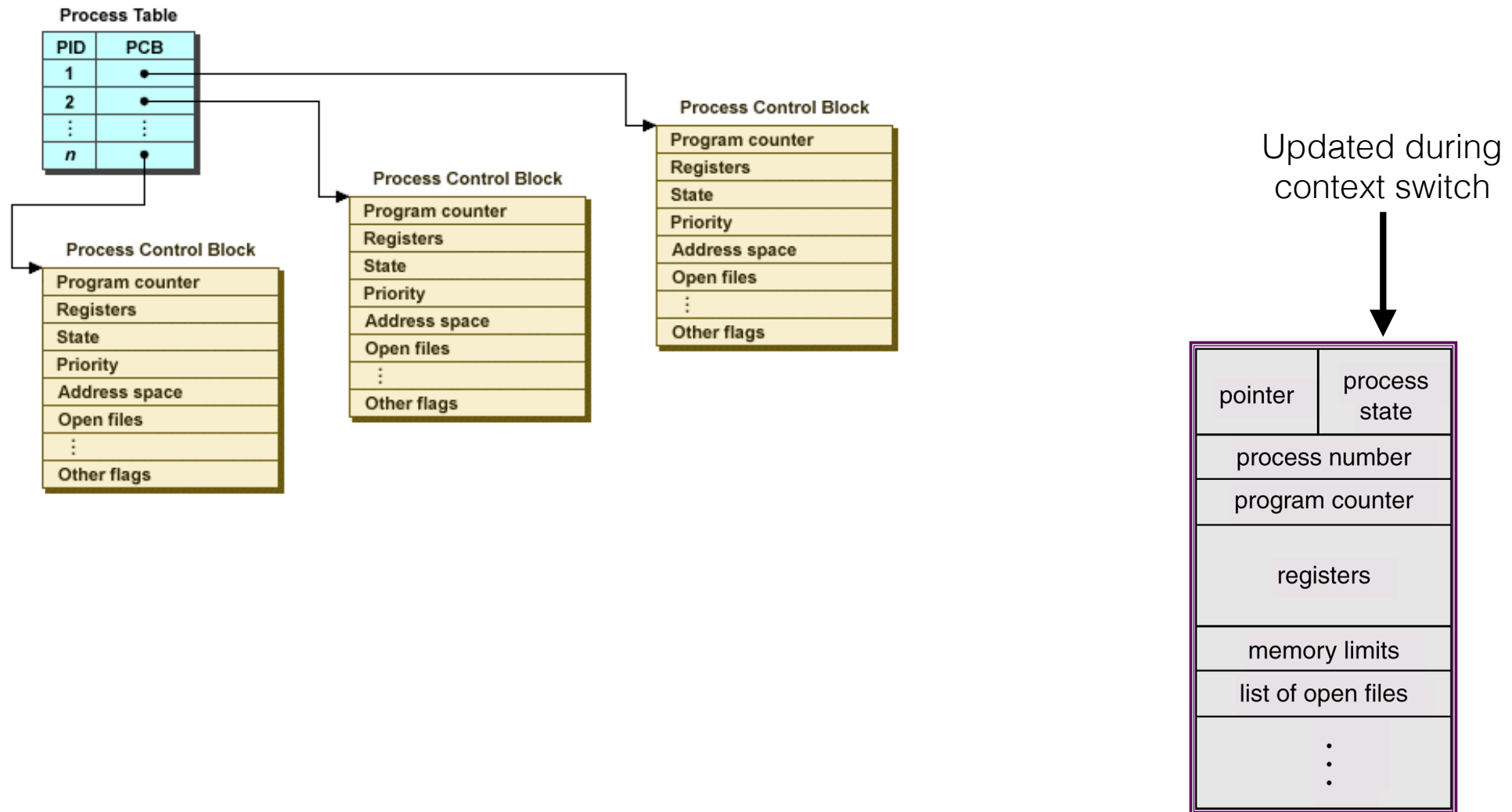


- Where is CPU State physically stored for active task?
 - Registers!
 - Program Counter is a register
 - Segment Registers
 - Code Segment
 - Data Segment
 - Stack Segment
- CPU has access to RAM and can save PC to stack before context switching.

Process Control Block



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



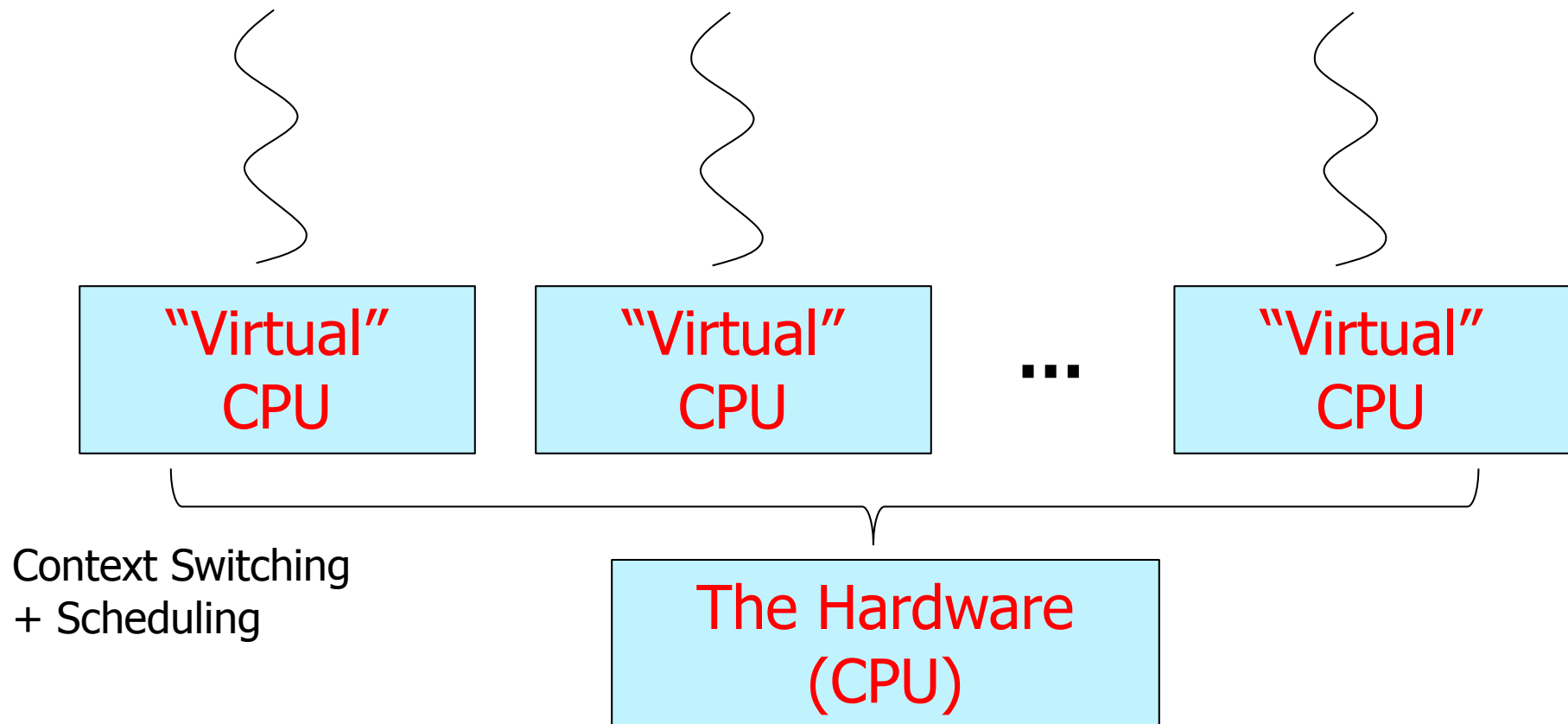
An alternate PCB diagram

Where We Are:



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

What's missing from this picture?

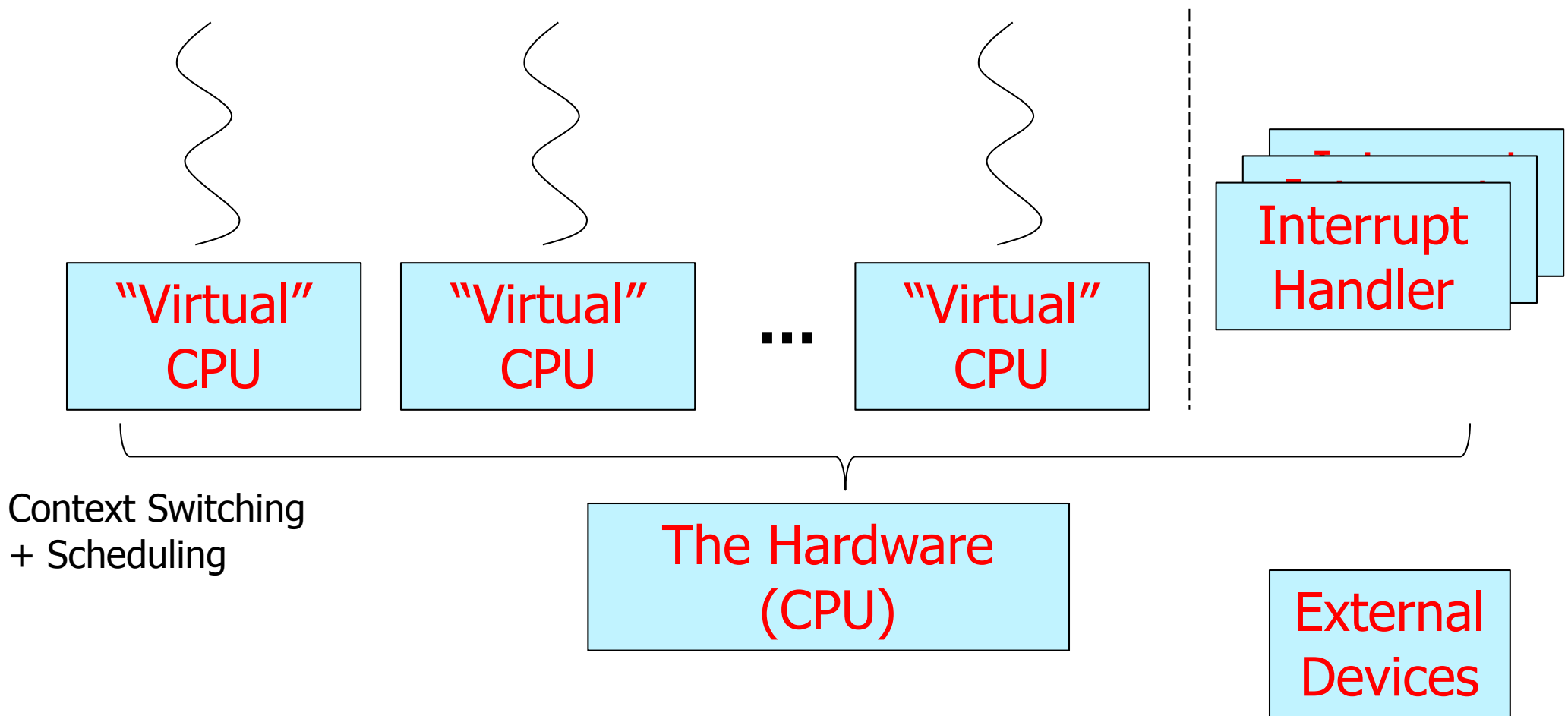


Where We Are:

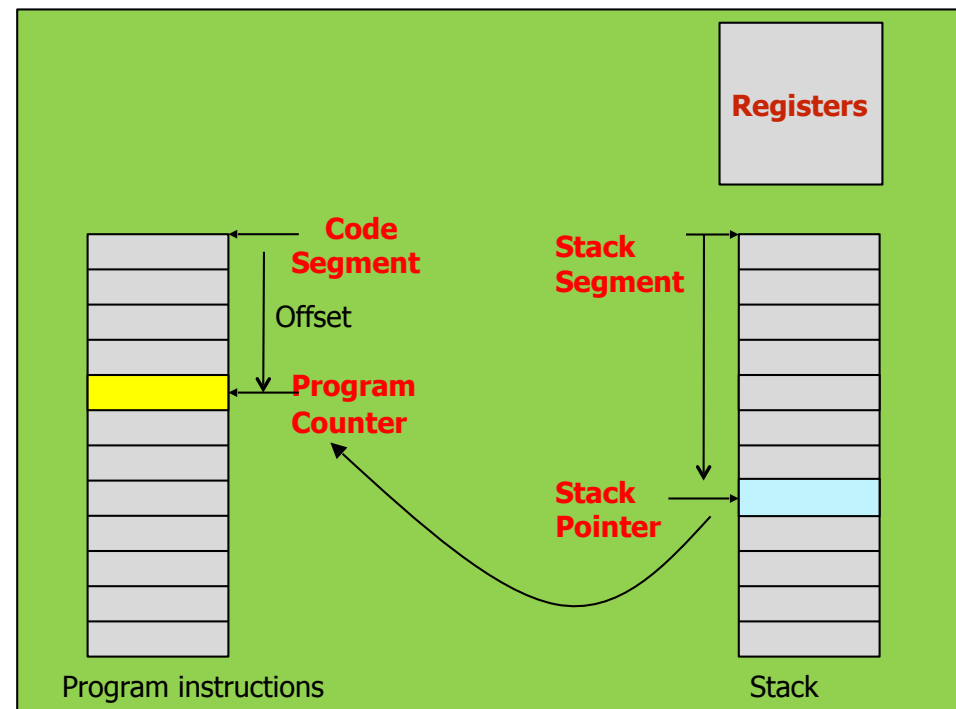
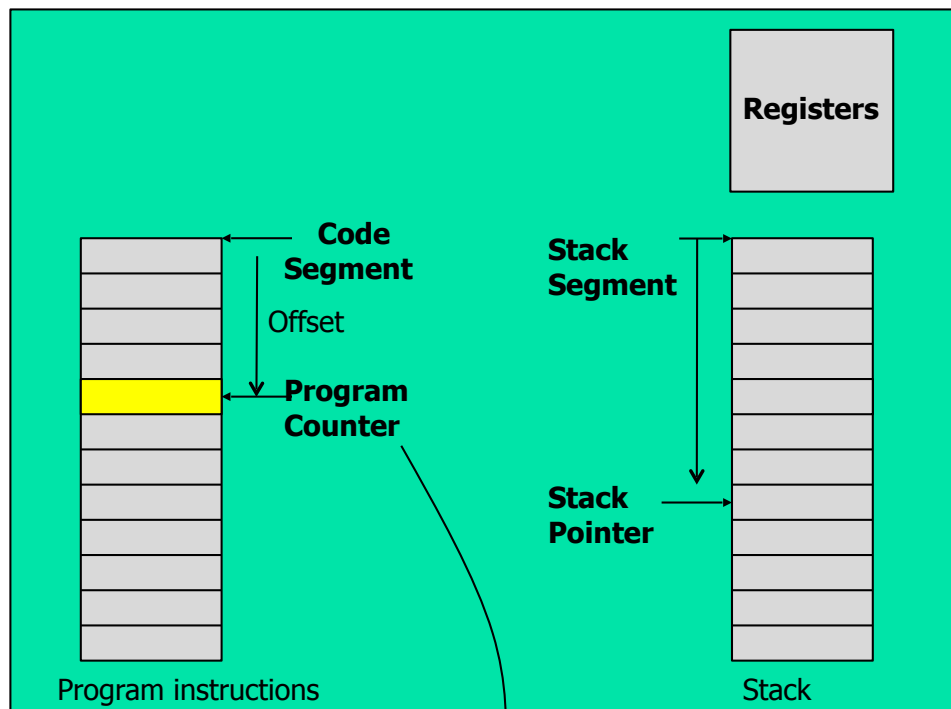


Interrupts to drive scheduling decisions!

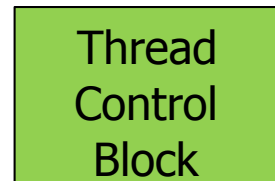
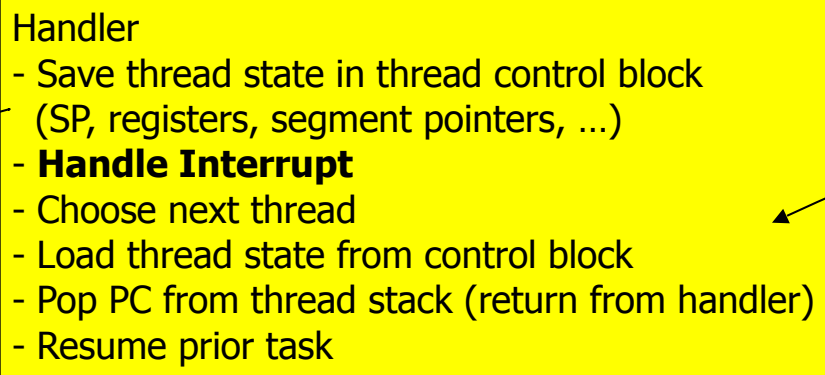
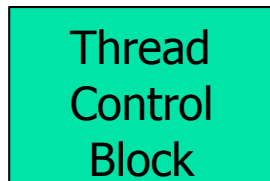
Interrupt handlers are also tasks that share the CPU.



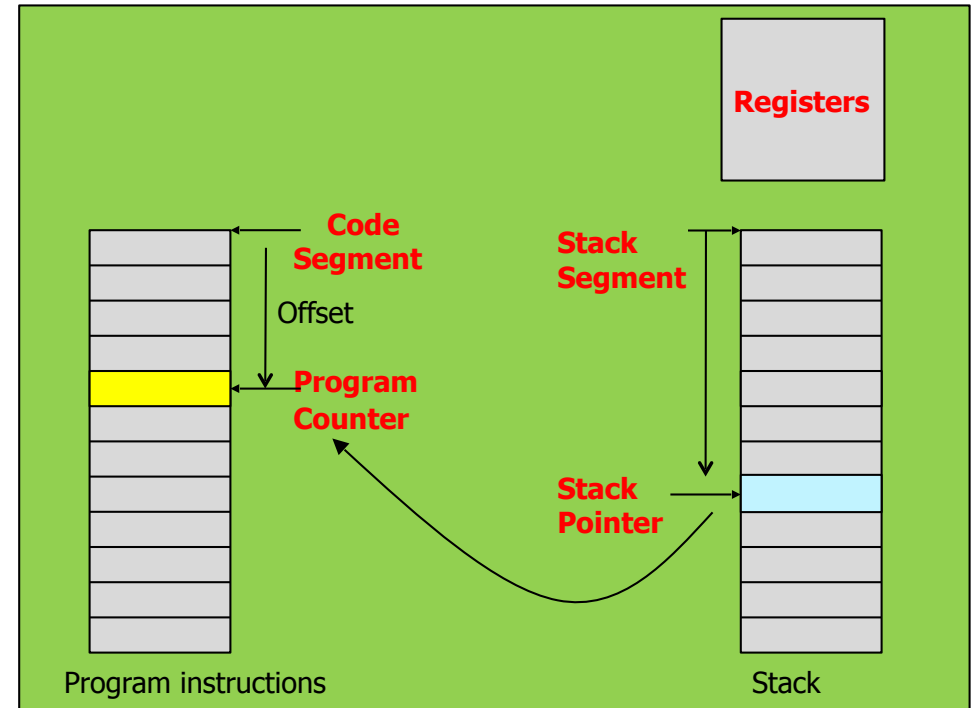
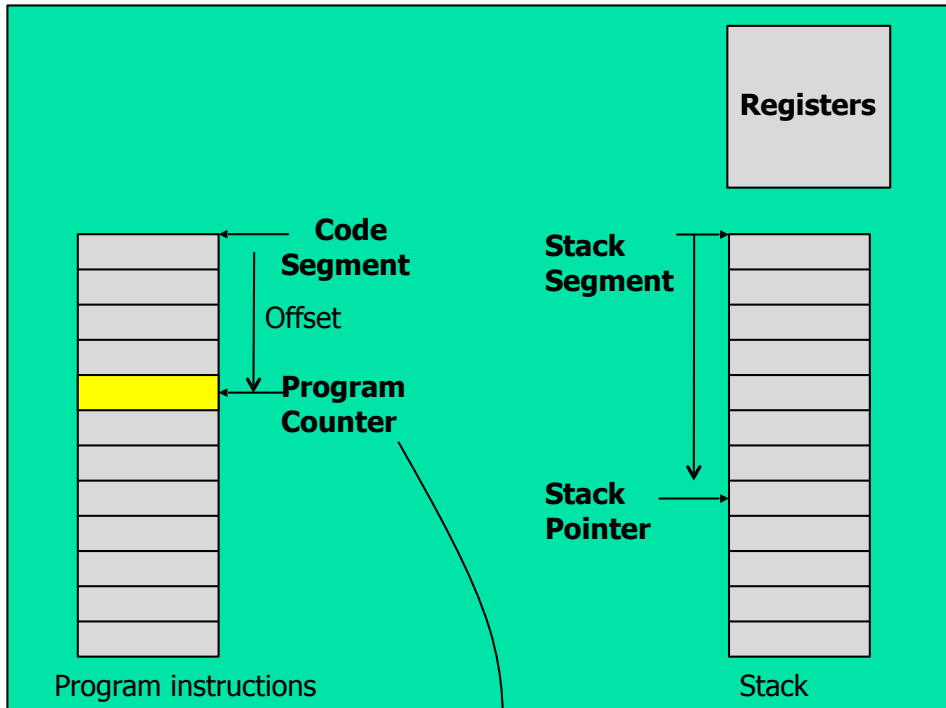
CTX Switch: Interrupt



Save PC on thread stack
Jump to Interrupt handler



Can also CTX Switch from Yield



Save PC on thread stack
Jump to yield() function

yield()

- Save thread state in thread control block (SP, registers, segment pointers, ...)
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)

Thread Control Block

Thread Control Block

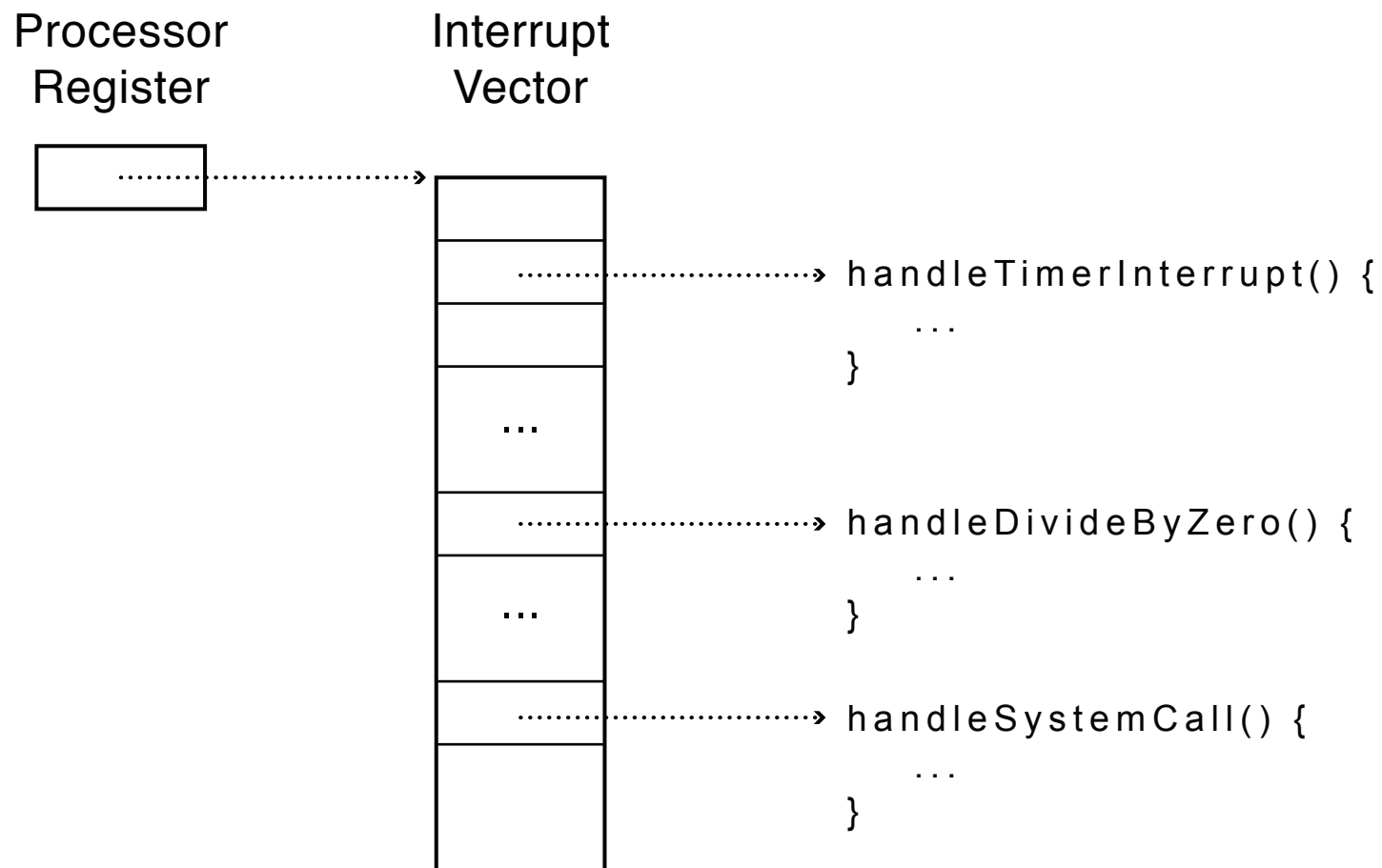


- **Interrupt Vector Table**
 - Where the processor looks for a handler
 - Limited number of entry points into kernel
 - Stored in RAM at a known address
- **Atomic transfer of control**
 - Single instruction to change:
 - Program counter
 - Stack pointer
 - Memory protection
 - Kernel/user mode
- **Transparent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

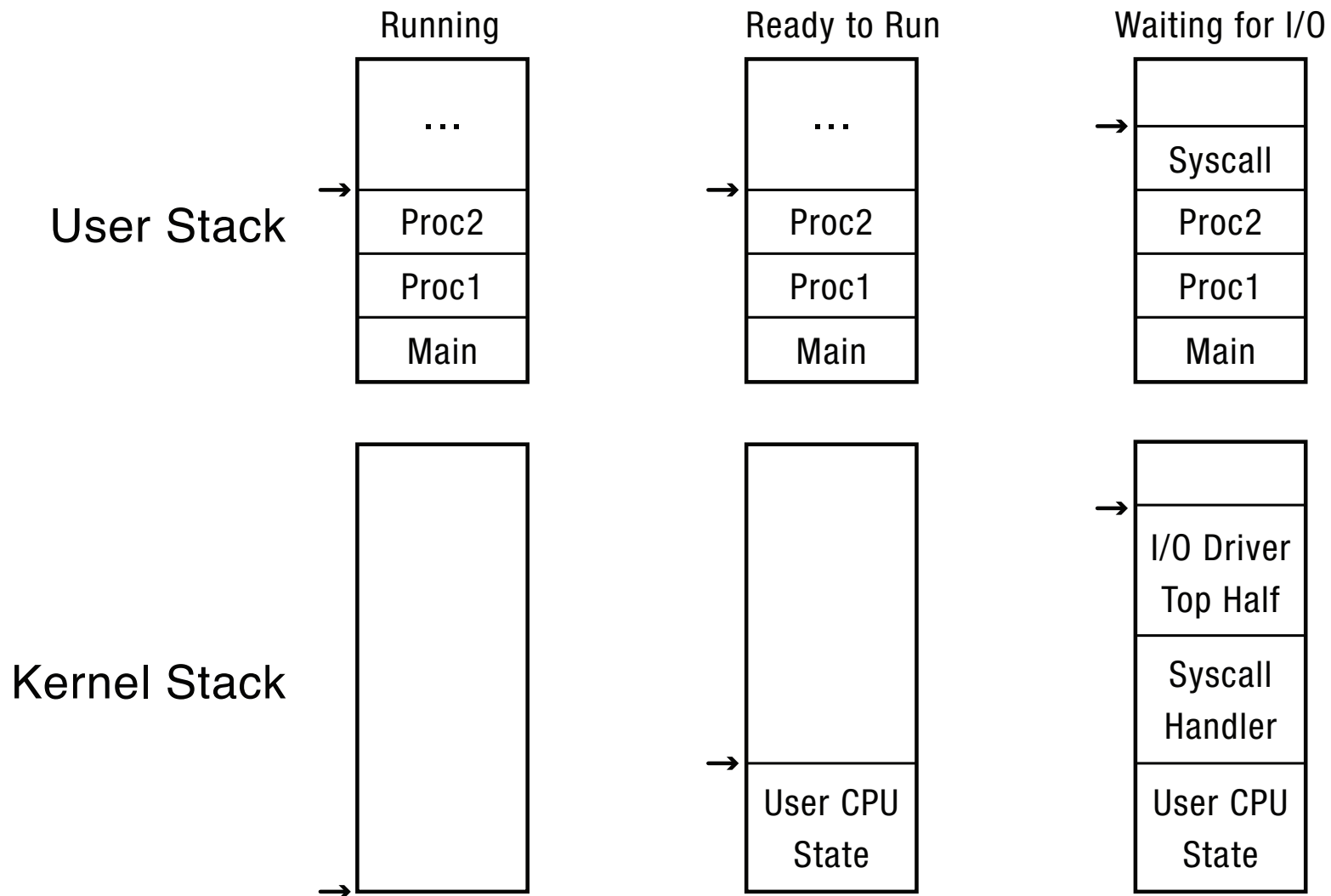


Interrupt Stack



- 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 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

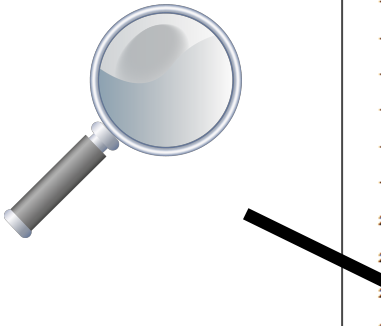


	AH	AG	AF	AE	AD	AC	AB	AA	Y	W	V	U	T	R	P	N	M	L	K	J	H	G	F	E	D	C	B	A		
1	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	GND	1	
2		GND	TERMI	GND	ID0#	GND	ID1#	GND	A07#	GND	A08#	GND	D03#	GND	D07#	GND	D08#	GND	NC	VC	D17#	GND	D07#	VC	D04#	GND	VC	3.3V	2	
3	TUNER(1)	TUNER(2)	TERM	GND	ID0#	GND	ID0#	GND	A08#	GND	A08#	GND	D09#	GND	D09#	GND	D09#	GND	D17#	GND	D13#	GND	D14#	GND	D07#	GND	NC	GND	3	
4		GND	OUTEN		ID#		ID#		A13#		A10#	GND	DEP#	VC	D8#	GND	STEP#	VC	D18#	GND	D12#	GND	VC	STEN#	GND	D03#	VC	NC	4	
5	NC	NC	GND	ID#	GND	ID7#	GND	A1#	GND	A12#	GND	NC	GND	D02#	GND	STEP#	GND	D19#	GND	DEP#	GND	D03#	GND	STEP#	GND	D02#	GND	GND	5	
6		GND	RSP#		ID#		ID#		A0#		A0#	VC	DEP#		D0#	GND	D0#		D18#	VC	D09#		D0#	VC	D08#	GND	NC	VC	6	
7	TDD	TD	GND	R0#	GND	ID0#	GND	DRDY#	GND	A14#	GND	A0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	NC	TERM	7
8		GND	INT#		R#		R#		A17#		A1#	GND	DEP#	VC	D5#	GND	D4#	VC	D4#	GND	D4#	GND	D4#	GND	D37#	VC	NC	GND	8	
9	TMB	TK	GND	REC0#	GND	D0#	GND	D0#	GND	A2#	GND	A0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	NC	TERM	9
10		GND	REC#		REC#		HT#		A2#		A2#	VC	DEP#		D#	GND	STEP#		D0#	VC	D0#		STEP#	VC	D0#	GND	GND	VC	10	
11	NC	NC	GND	REC#	GND	DRDY#	GND	A2#	GND	A2#	GND	NC	GND	D0#	GND	STEP#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	NC	11
12		GND	REC#		REC#		HT#		A2#		A1#	GND	D0#	VC	D5#	GND	D5#	VC	D4#	GND	D4#	GND	D4#	GND	D0#	VC	NC	VC	12	
13	BLUN	BLUP	GND	SSY#	GND	R#	GND	SSY#	GND	A2#	GND	A1#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	NC	GND	13
14		GND	TRD#		D0#		DEP#		A3#		A3#	VC	D0#		D0#	GND	D0#		NC	VC	D0#		D0#	VC	D0#	GND	NC	VC	14	
15	PAR GOOD	PRD	GND	LOCK#	GND	TND#	GND	BNTH#	GND	A3#	GND	A2#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	NC	GND	15
16		GND	EREC0#		EREC1#		NC		A3#		A3#	VC	DEP1#	VC	D0#	GND	STEP#		D0#	VC	D0#		STEP#	VC	D0#	GND	NC	GND	16	
17	NC	NC	GND	NC	GND	NC	GND	A3#	GND	A3#	GND	ENR#	GND	D0#	GND	STEP#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	NC	17
18		GND	EREC#		EREC#		A3#		A2#		A2#	VC	DEP1#	D0#	VC	D0#		D0#	VC	D0#		D0#	VC	D0#	GND	D0#	GND	NC	18	
19	NC	NC	GND	BPR#	GND	SSY#	GND	SSY#	GND	A3#	GND	A2#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	19
20		GND	PRD		RESET#		A0#	GND	A3#		A4#	GND	DEP1#	VC	D12#	GND	D11#	VC	D17#	GND	D11#	VC	D10#	GND	D10#	VC	NC	GND	20	
21	NC	NC	GND	TRST#	GND	NC	GND	DRDY#	GND	A4#	GND	A0#	GND	D12#	GND	D12#	GND	D12#	GND	D12#	GND	D12#	GND	D10#	GND	D0#	GND	D0#	GND	21
22		GND	LINT0		EPV0#		ERR#		A4#		A7#	VC	DEP1#	D12#	VC	STEP#		D14#	VC	D10#		STEP#	VC	D0#	GND	NC	VC	22		
23	A0M#	IGN#		EPV#	GND	EPV#	GND	AF#	GND	A4#	GND	A0#	GND	D12#	GND	STEP#	GND	D11#	GND	D13#	GND	D10#	GND	STEP#	GND	D0#	GND	GND	23	
24		GND	LINT1		EPV#	GND	EPV#	GND	A4#		A4#	GND	D12#	VC	D12#	GND	D11#	VC	NC	GND	D10#	VC	D10#	GND	D10#	VC	GND	NC	GND	24
25	ERR#	TH_TRP#		PL#	GND	EPV#	GND	AP#	GND	A4#	GND	VC	DEP1#	D12#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	D10#	GND	D10#	GND	NC	VC	25

← 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	TERM	GND	GND	GND	GND	GND	TERM	GND	GND	TERM	GND	GND	GND	GND	GND	GND	GND	1
2	GND	TERM	GND	IO0#	IO1#	IO2#	IO3#	IO4#	IO5#	IO6#	IO7#	IO8#	IO9#	IO10#	IO11#	IO12#	IO13#	IO14#	IO15#	IO16#	IO17#	IO18#	IO19#	IO20#	IO21#	IO22#	IO23#	IO24#	2
3	TUNER(1)	TUNER(2)	TERM	GND	IO0#	IO1#	IO2#	IO3#	IO4#	IO5#	IO6#	IO7#	IO8#	IO9#	IO10#	IO11#	IO12#	IO13#	IO14#	IO15#	IO16#	IO17#	IO18#	IO19#	IO20#	IO21#	IO22#	IO23#	3
4	GND	OUTEN	IO0#	IO1#	IO2#	A13#	A10#	GND	DEP3#	VC	D8#	GND	STEP1#	VC	D18#	GND	D12#	GND	STEN0#	GND	D03#	GND	NC	NC	NC	NC	NC	NC	4
5	NC	NC	GND	IO0#	IO1#	IO2#	A1#	GND	A12#	GND	NC	GND	DEP2#	GND	STEP1#	GND	D19#	GND	DEP1#	GND	D02#	GND	STEP0#	GND	D02#	GND	GND	GND	5
6	GND	RSP#	IO0#	IO1#	IO2#	A0#	A0#	VC	DEP2#	GND	D0#	GND	D0#	GND	D1#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	D0#	GND	NC	NC	6
7	TD0	TD1	GND	RSP#	IO0#	IO1#	DRDY0#	GND	A14#	GND	A0#	GND	D3#	GND	D2#	GND	D2#	GND	D2#	GND	D2#	GND	D2#	GND	D2#	GND	D2#	GND	7
8	GND	INT#	RSP#	RSP#	A17#	A1#	A1#	DEP3#	VC	D5#	D4#	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	NC	GND	8
9	TIME	TDK	GND	REC0#	GND	DEP1#	GND	DEP1#	GND	A2#	GND	A1#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	9
10	GND	REC#	REC#	HT#	A2#	A2#	VC	DEP1#	GND	D6#	GND	STEP2#	VC	D3#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	GND	D3#	GND	VC	10
11	NC	NC	GND	REC3#	GND	DRDY#	GND	A2#	GND	A2#	GND	NC	GND	D6#	GND	STEP2#	GND	D5#	GND	D5#	GND	D5#	GND	D5#	GND	D5#	GND	D5#	11
12	GND	REC#	REC#	HT#	A2#	A1#	DEP2#	VC	D5#	D5#	GND	NC	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	D4#	GND	VC	12
13	BCLK#	BCLK#	GND	SSY#	GND	RFP#	GND	SSY#	GND	A2#	GND	A1#	GND	D5#	GND	D5#	GND	D5#	GND	D5#	GND	D5#	GND	D5#	GND	D5#	GND	NC	13
14	GND	TRDY#	GND	DEP#	A3#	A3#	VC	D9#	D8#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	NC	NC	14
15	PAR	PROC	GND	LOCK#	GND	TND#	GND	BNTH#	GND	A3#	GND	A2#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	NC	15
16	GOOD	FRS#	GND	EPR0#	GND	EPR1#	NC	A3#	A3#	DEP1#	VC	D9#	GND	STEP2#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	D8#	GND	NC	16
17	NC	NC	GND	NC	NC	NC	GND	A3#	GND	A3#	GND	ENR#	GND	D9#	GND	STEP2#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	17
18	GND	EPR0#	NC	EPR0#	A3#	A2#	VC	DEP1#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	18
19	NC	GND	BPR#	GND	SSY#	GND	SSY#	GND	A3#	GND	A2#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	D9#	GND	19
20	GND	PROC	GND	RESET#	A0#	GND	A3#	A4#	GND	DEP1#	VC	D12#	GND	D11#	VC	D11#	GND	D11#	VC	D11#	GND	D11#	VC	D11#	GND	D11#	VC	NC	20
21	NC	GND	TRDY#	GND	NC	GND	DRDY1#	GND	A4#	GND	A2#	GND	D12#	GND	D12#	GND	D12#	GND	D12#	GND	D12#	GND	D12#	GND	D12#	GND	D12#	GND	21
22	GND	UNIT0	EPV0#	EPV0#	A4#	A7#	VC	DEP1#	GND	D12#	GND	STEP1#	D14#	VC	D10#	GND	D10#	GND	D10#	GND	D10#	GND	D10#	GND	D10#	GND	D10#	GND	22
23	AQ0#	IGNB#	EPV0#	GND	EPV0#	GND	AF1#	GND	A4#	GND	A4#	GND	D12#	GND	STEP1#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	23
24	GND	UNIT1	EPV0#	GND	EPV0#	GND	A4#	A4#	GND	D12#	GND	D12#	GND	D11#	VC	NC	GND	D10#	VC	D10#	GND	D10#	GND	D10#	GND	D10#	GND	VC	24
25	FERR#	TH_TRP#	PV#	GND	EPV1#	GND	AP0#	GND	A4#	GND	VC	D12#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	D11#	GND	NC	25

← Power Pad

UUU638b

A Note on Multicore

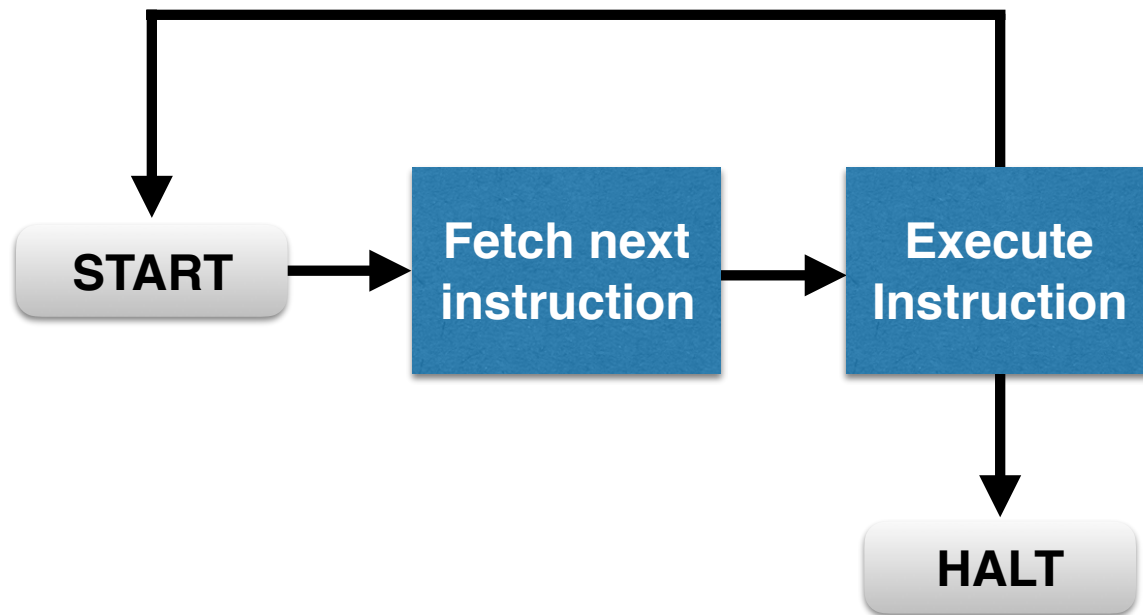


- How are interrupts handled on multicore machines?
 - On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
 - The OS can program the APICs to determine which interrupts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

Instruction Cycle



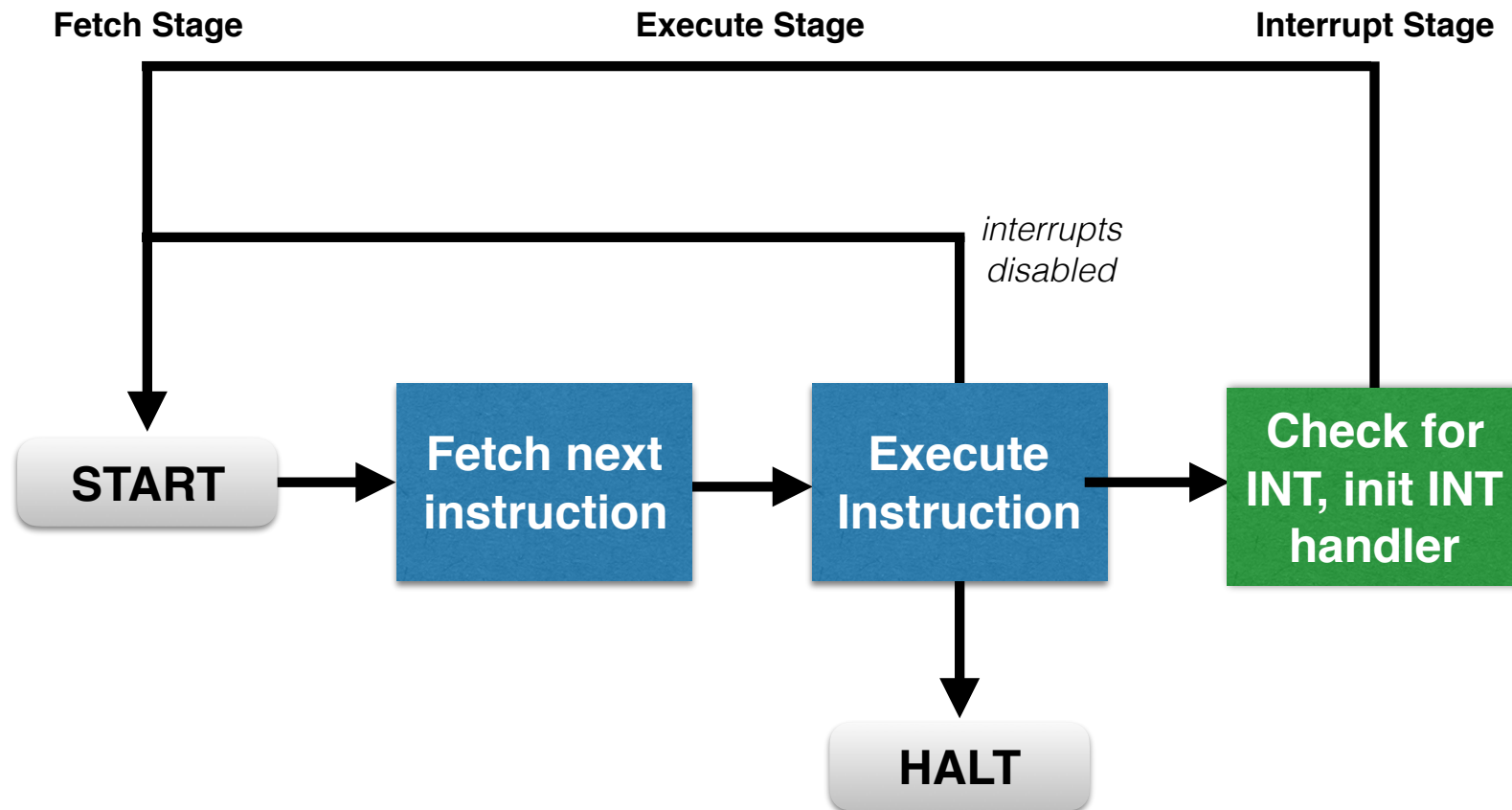
How does interrupt handling change the instruction cycle?



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 `malloc()`?
- 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”



- 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.



- 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
- ...



- 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);`