



CS 423

Operating System Design: Interrupts

Tianyin Xu

* Thanks for Prof. Adam Bates for the slides.

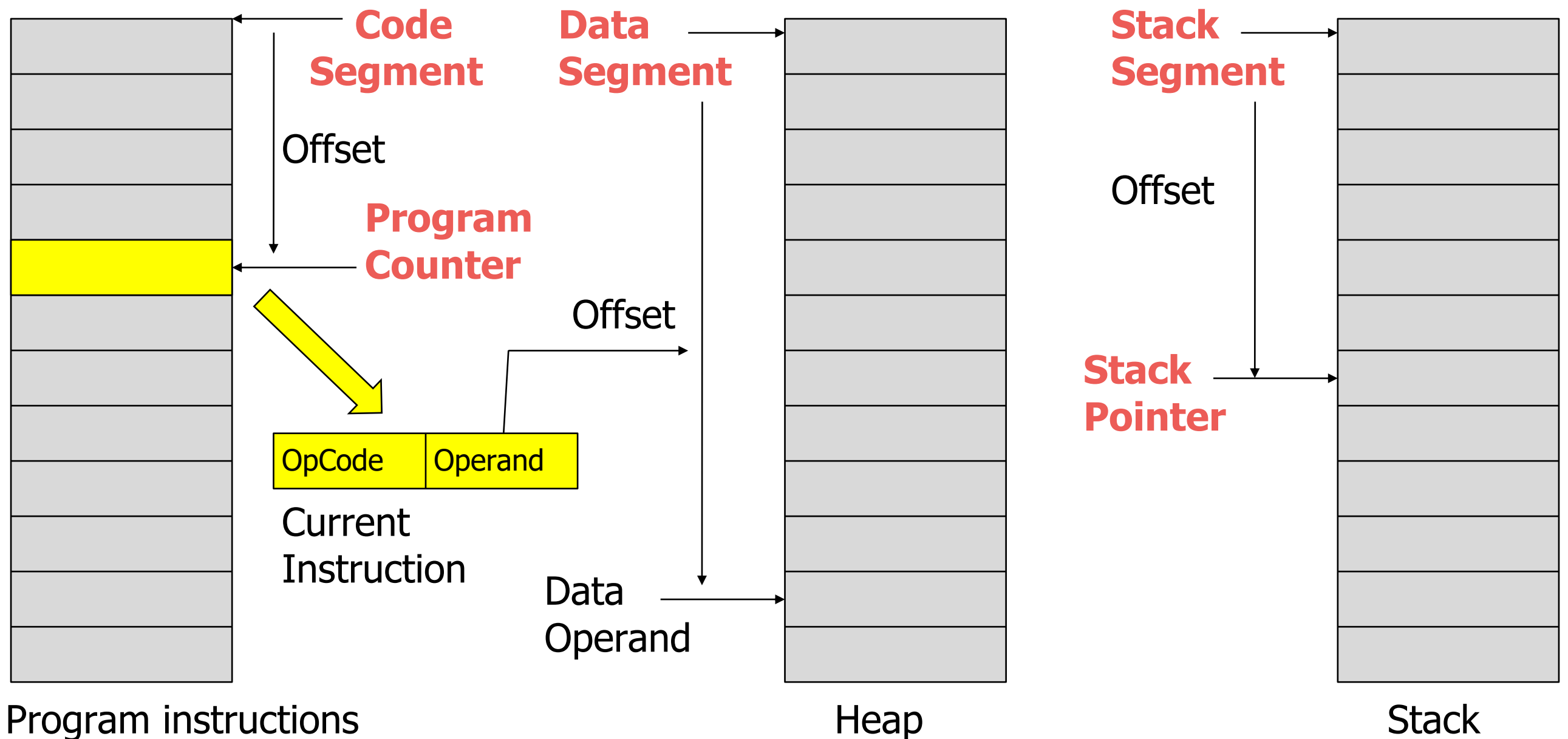
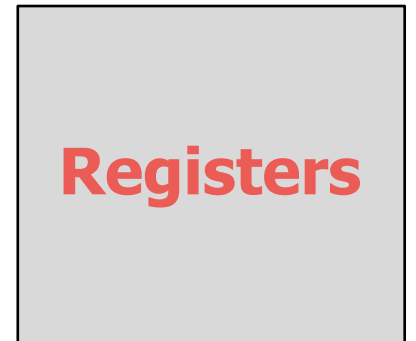


- MP 1 should be out today.
- Please send us your Github ID. The Google form is online.

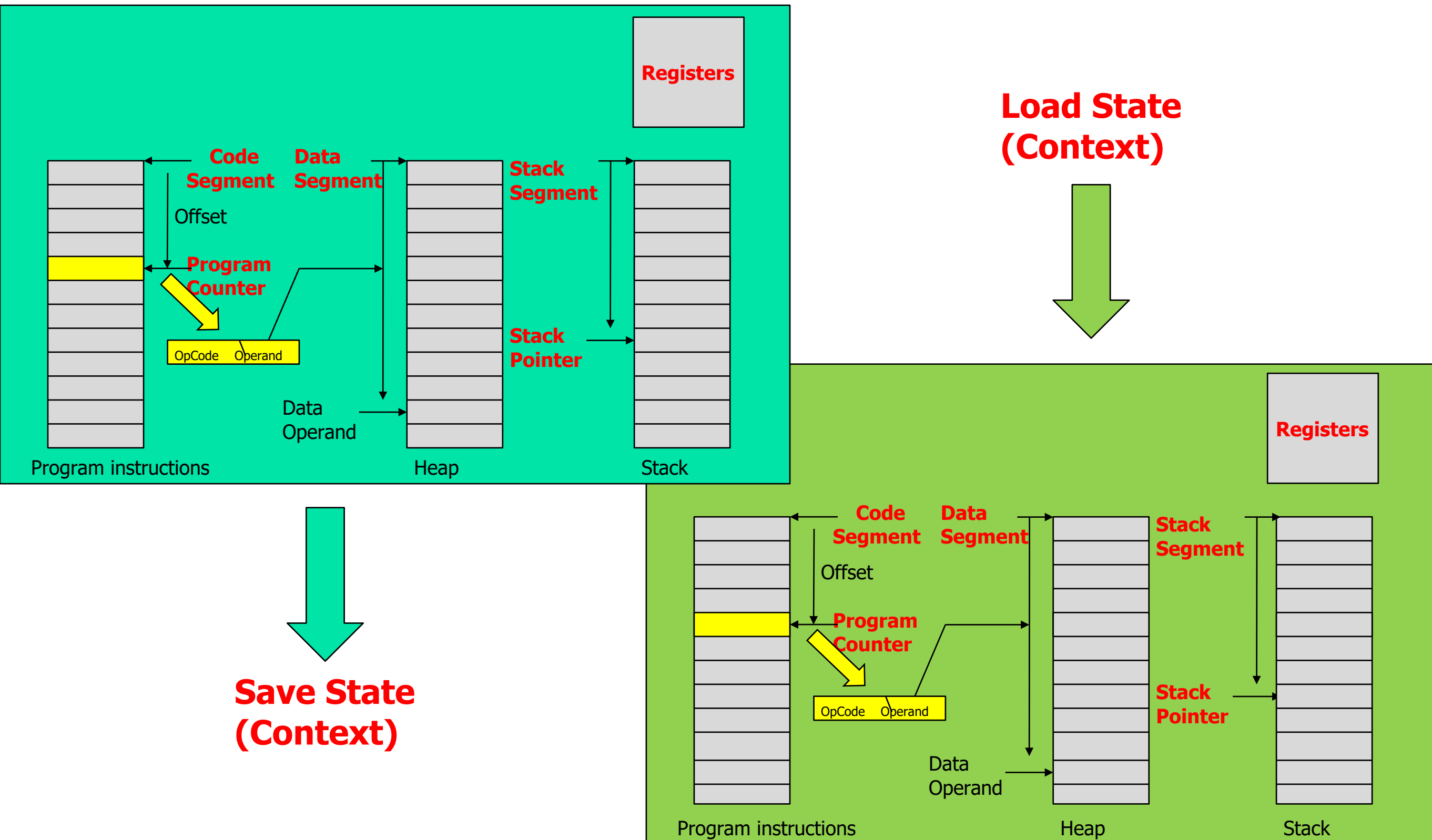
What's a 'real' CPU?



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



The Context Switch



Discussion: Last Class

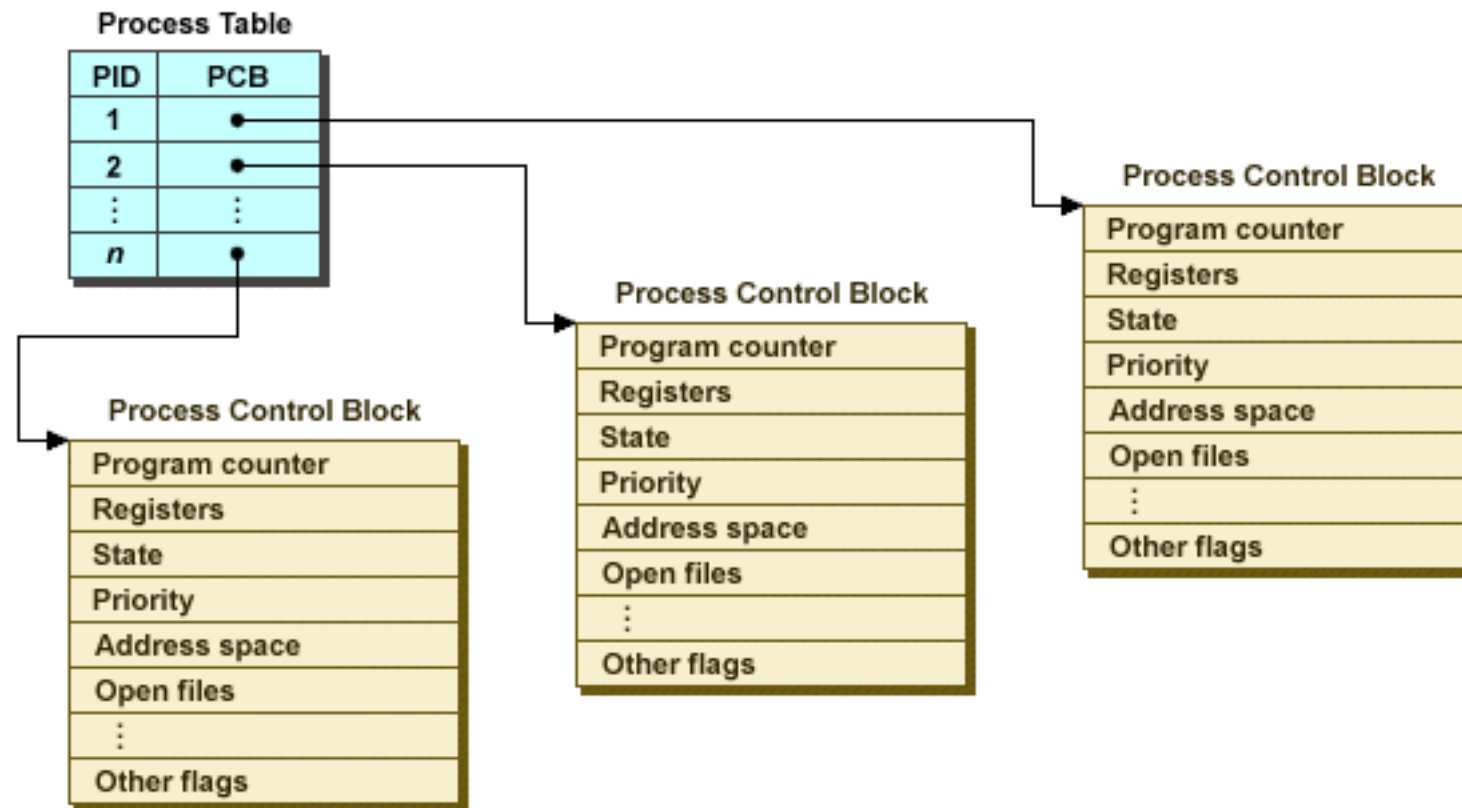


- Where is CPU State physically stored for active task?
 - Registers!
 - Program Counter is a register
 - Segment Registers
 - Code Segment
 - Data Segment
 - Stack Segment
- CPU has access to RAM and can save PC to stack before context switching.

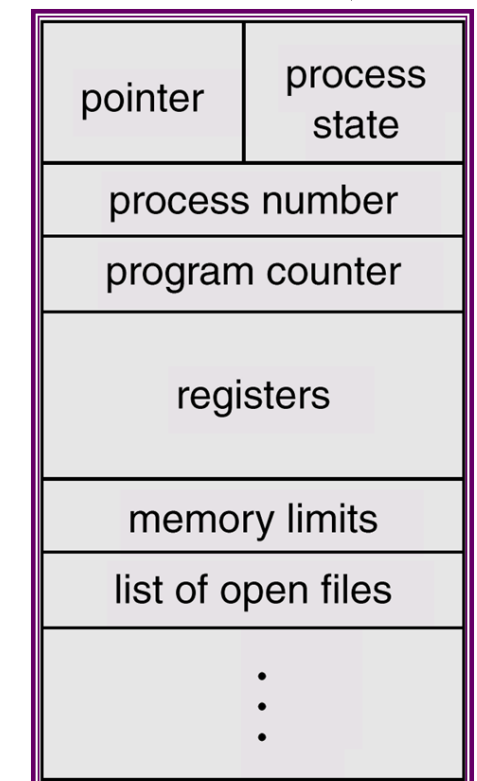
Process Control Block



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



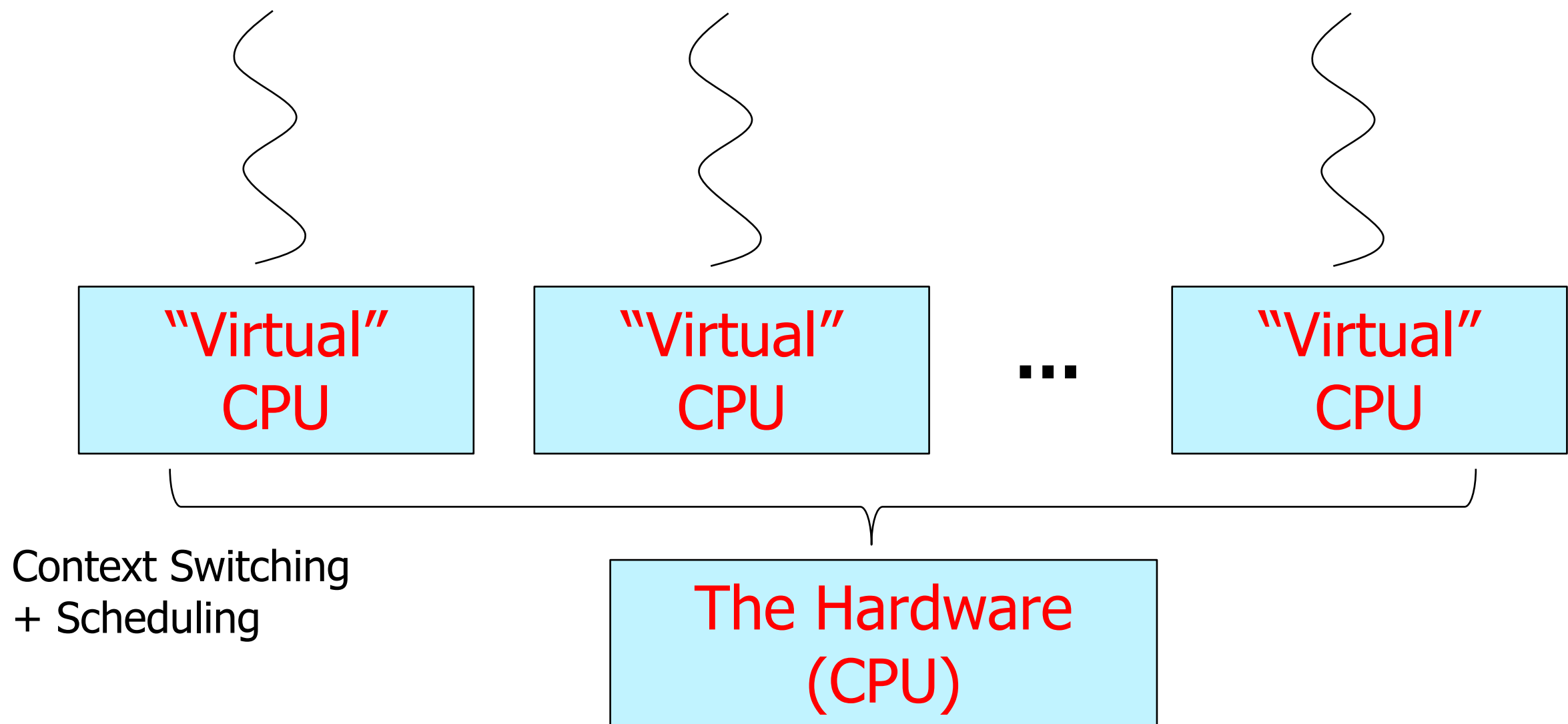
An alternate PCB diagram

Where We Are:



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

What's missing from this picture?

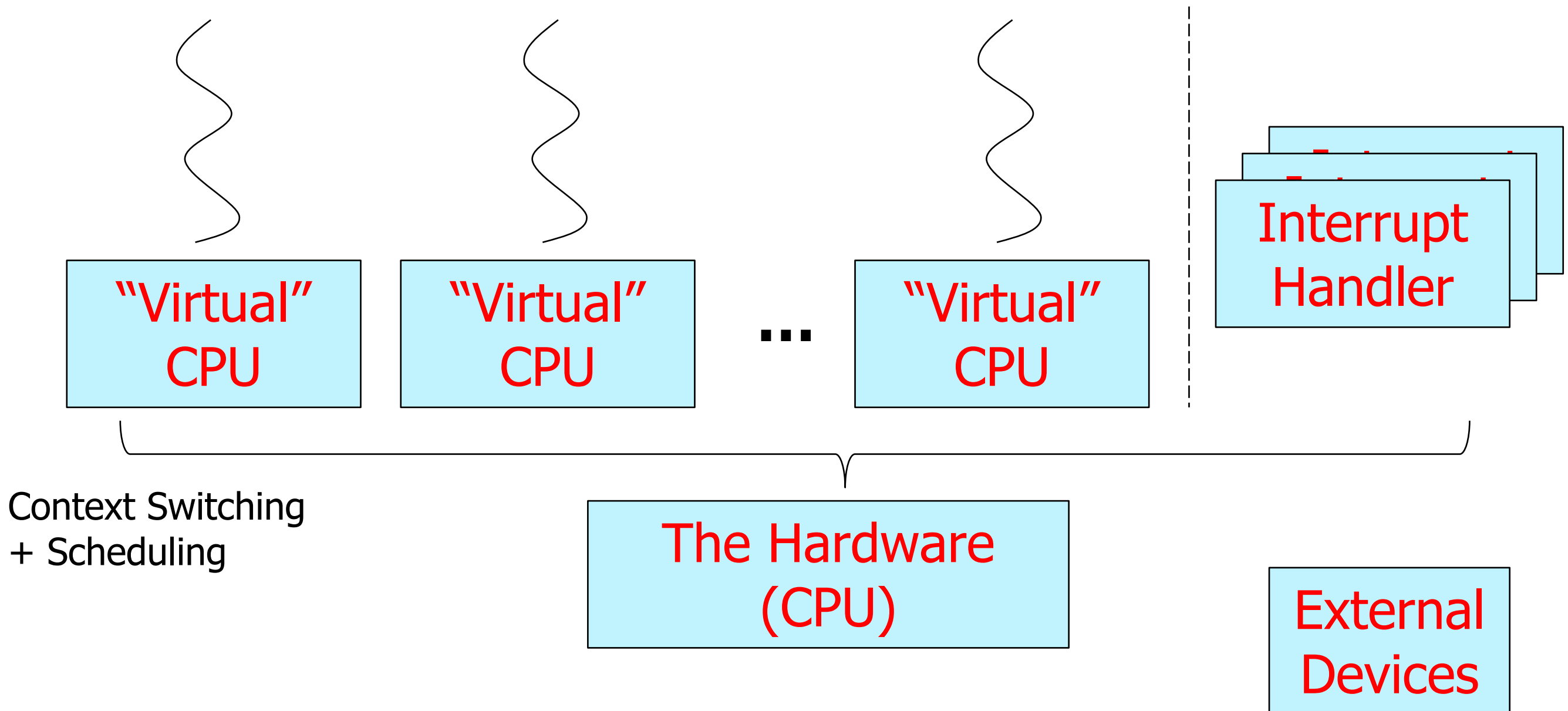


Where We Are:

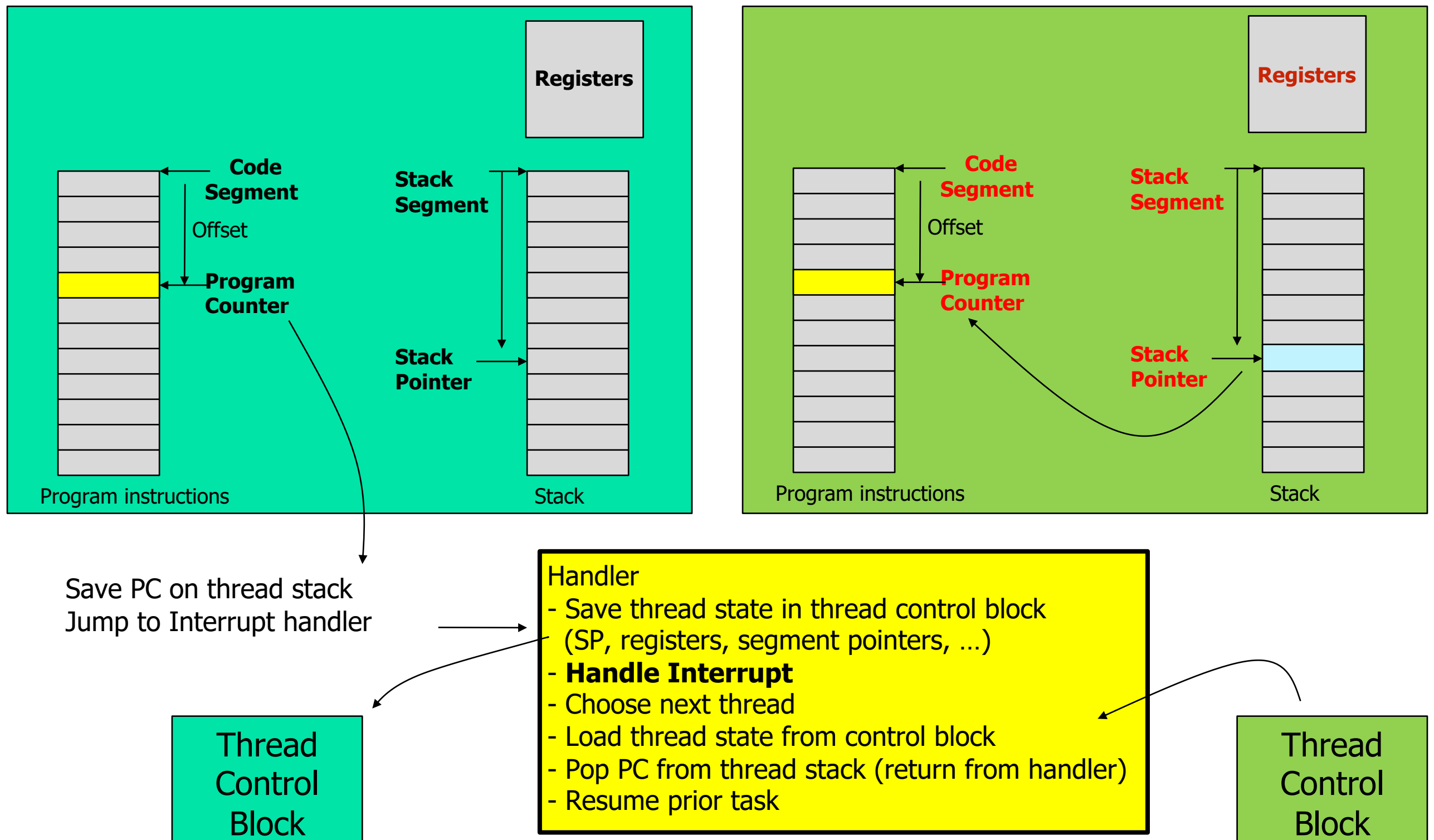


Interrupts to drive scheduling decisions!

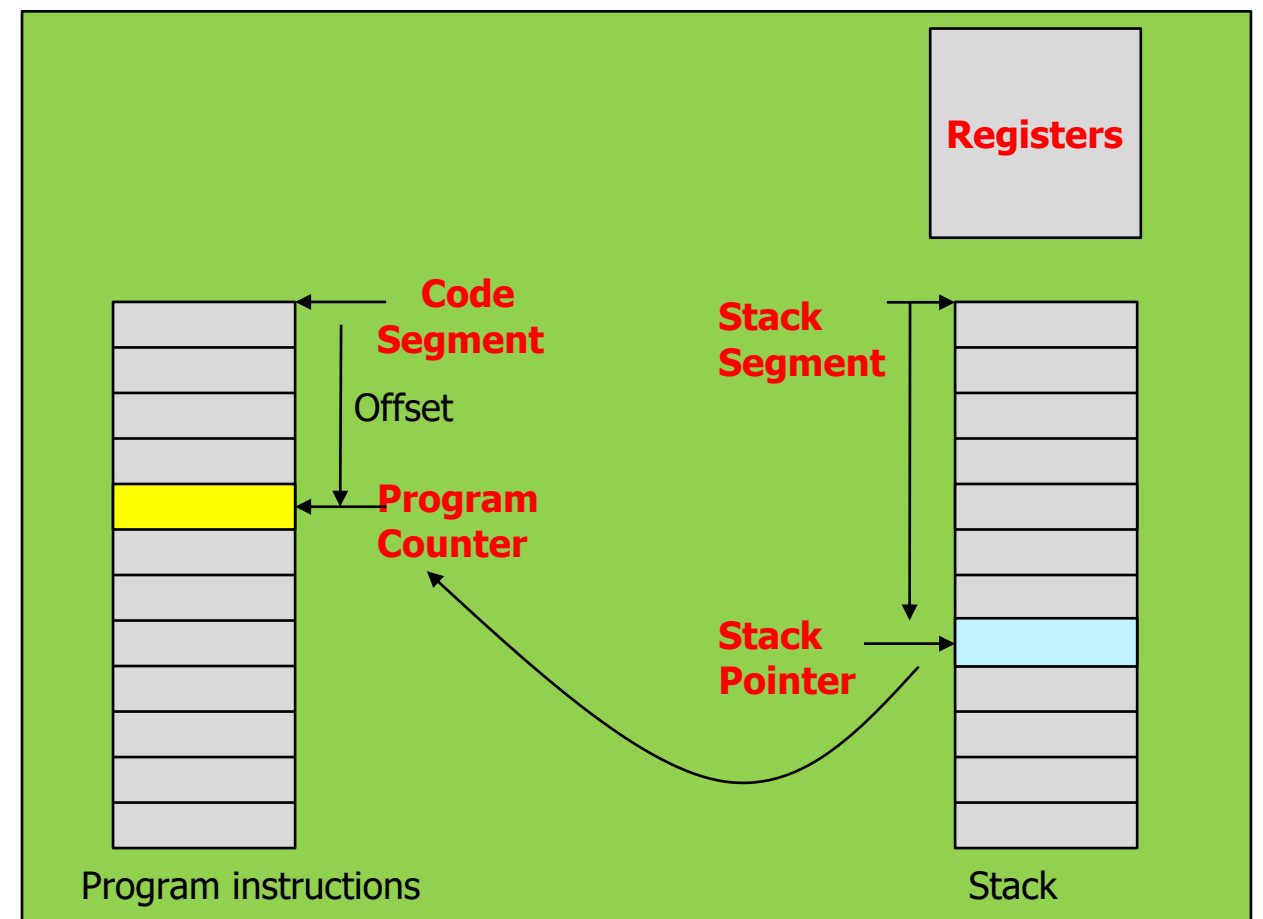
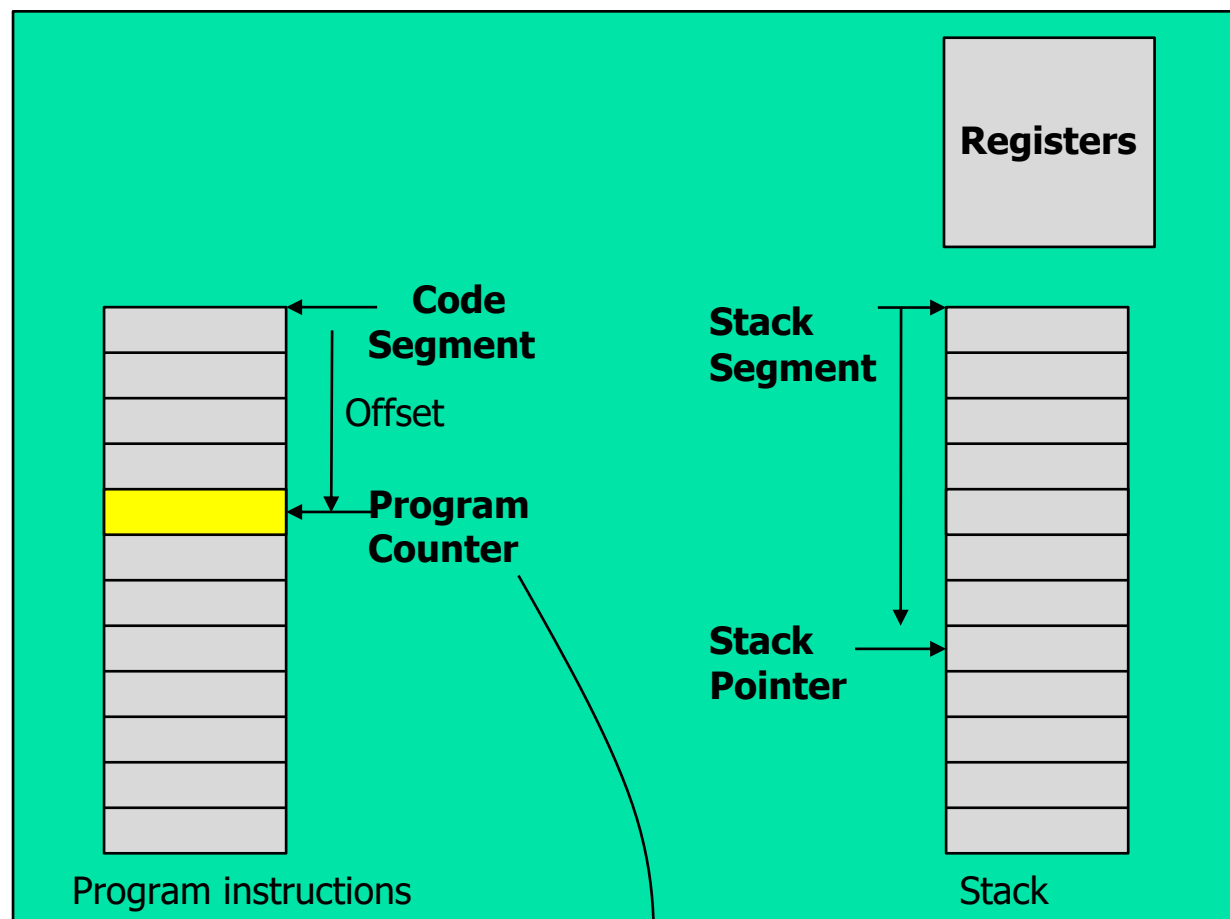
Interrupt handlers are also tasks that share the CPU.



CTX Switch: Interrupt



Can also CTX Switch from Yield



Save PC on thread stack
Jump to yield() function

Thread
Control
Block

```
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

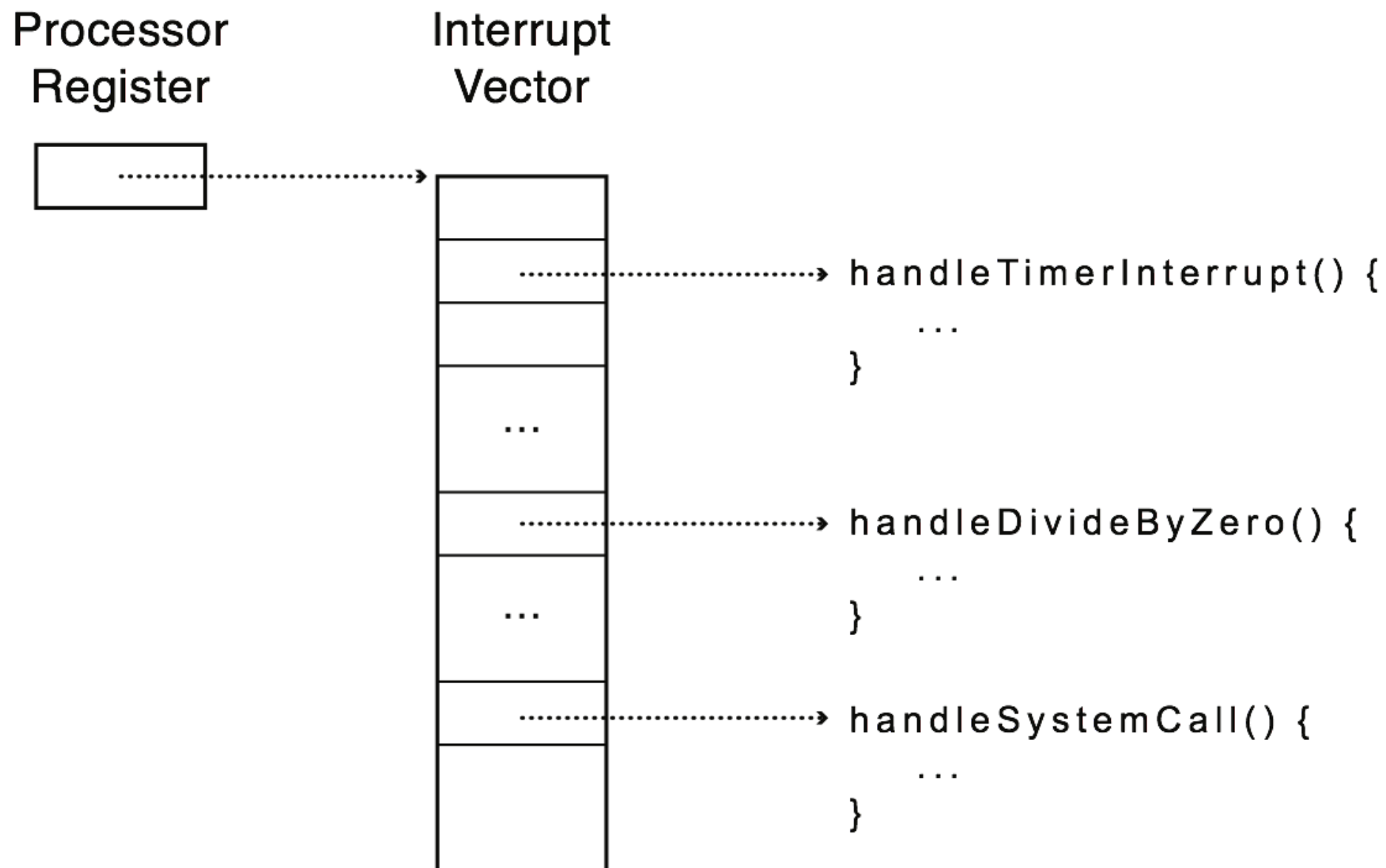


- **Interrupt Vector Table**
 - Where the processor looks for a handler
 - Limited number of entry points into kernel
 - Stored in RAM at a known address
- **Atomic transfer of control**
 - Single instruction to change:
 - Program counter
 - Stack pointer
 - Memory protection
 - Kernel/user mode
- **Transparent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

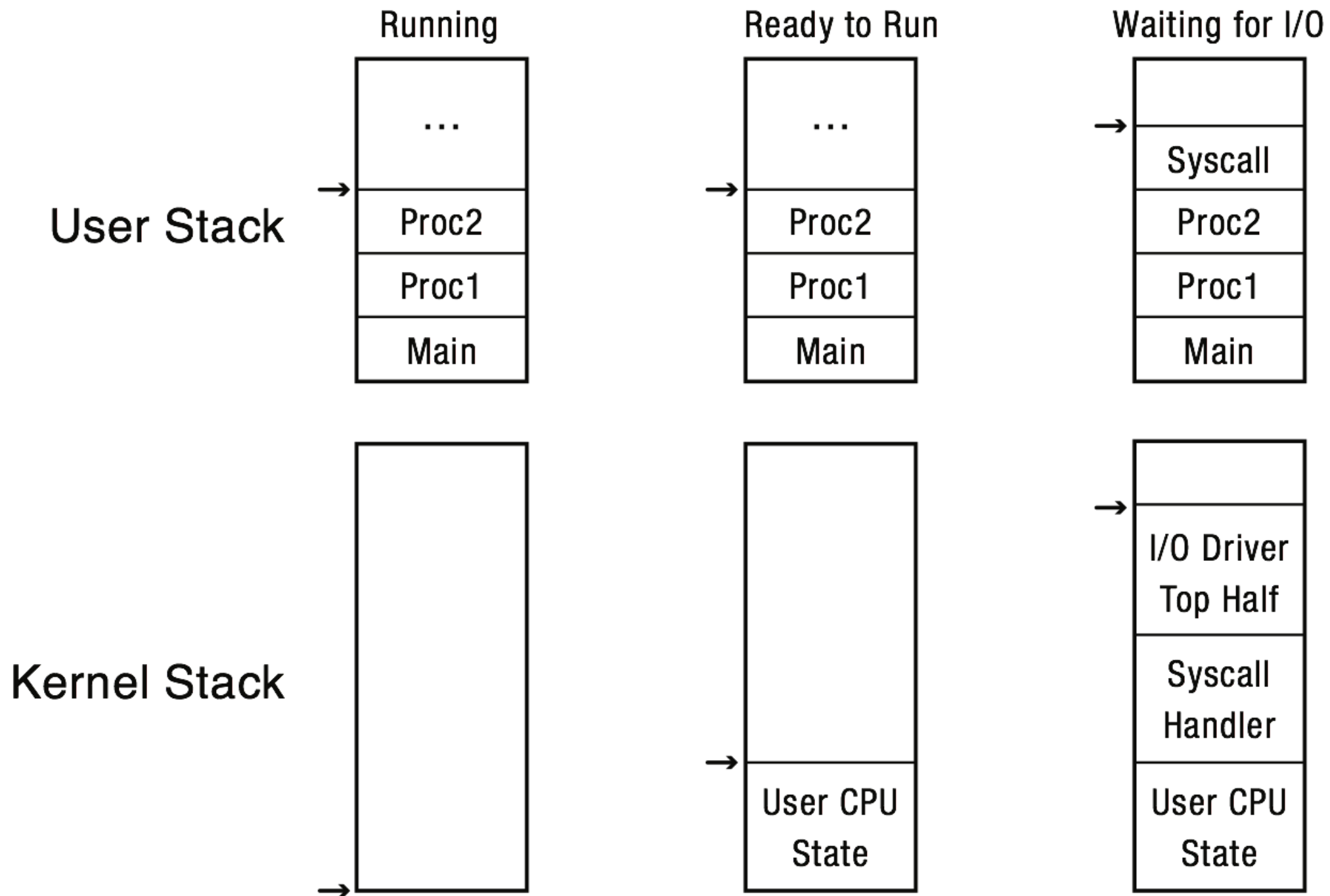


Interrupt Stack



- 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													VC															1
2														TERM															2
3																													3
4																													4
5																													5
6																													6
7																													7
8																													8
9																													9
10																													10
11																													11
12																													12
13																													13
14																													14
15																													15
16																													16
17																													17
18																													18
19																													19
20																													20
21																													21
22																													22
23																													23
24																													24
25																													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

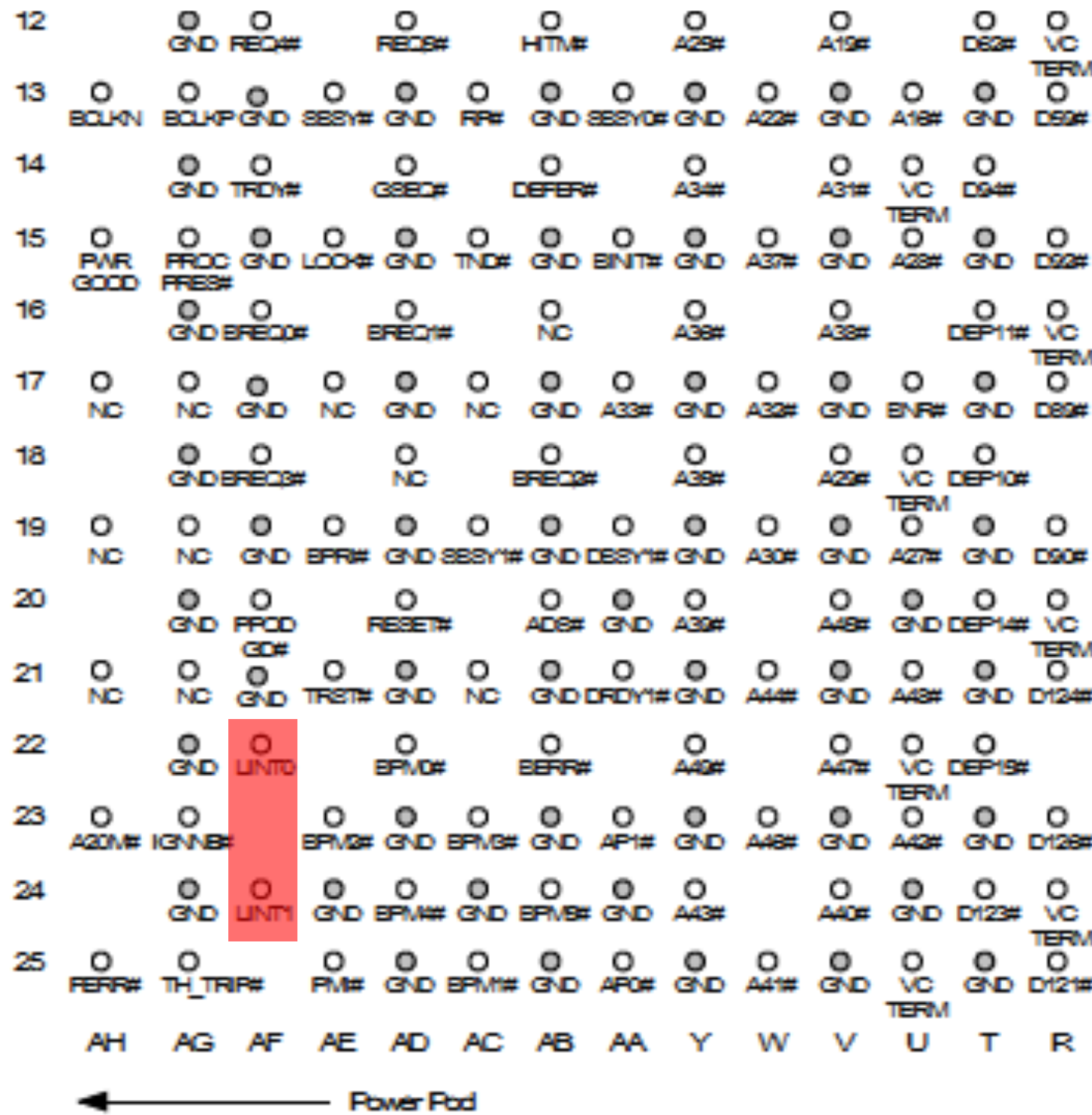
Ex: Itanium 2 Pinout



	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	VC TERM	GND		GND	VC TERM		GND		VC TERM	GND		GND	VC TERM		GND	1	
2		GND	TERMA	GND	ID0#	GND	ID1#	GND	A07#	GND	A04#	VC TERM	D30#	GND	D27#	VC TERM	D30#	GND	NC	VC TERM	D11#	GND	D07#	VC TERM	D04#	GND	3.3V	VC TERM	2	
3	TUNER(1)	TUNER(2)	TERMB	GND	ID2#	GND	ID3#	GND	A08#	GND	A05#	VC TERM	D31#	GND	D28#	VC TERM	D31#	GND	D12#	GND	D08#	VC TERM	D05#	GND	NC	GND	NC	GND	3	
4		GND	OUTEN		ID4#		ID5#		A13#		A10#	GND	DEP3#	VC TERM	D34#	GND	STEP1#	VC TERM	D15#	GND	D12#	VC TERM	STEN2#	GND	D03#	VC TERM	NC	NC	4	
5	NC	NC	GND	ID6#	GND	ID7#	GND	A11#	GND	A12#	GND	NC	GND	D32#	GND	STEN1#	GND	D19#	GND	DEP1#	GND	D08#	GND	STEP0#	GND	D02#	GND	GND	5	
6		GND	RSP#		ID8#		ID9#		A03#		A03#	VC TERM	DEP2#	GND	D35#	VC TERM	D35#	GND	D18#	VC TERM	D09#		D06#	VC TERM	D03#	GND	NC	VC TERM	6	
7	TDO	TDI	GND	RSP#	GND	ID10#	GND	DROV0#	GND	A14#	GND	A09#	VC TERM	D31#	GND	D32#	GND	D21#	GND	DEP0#	GND	D19#	GND	D10#	GND	D00#	GND	THRM	ALERT#	7
8		GND	INT#		RSP#		RSP#		A17#		A19#	VC TERM	DEP3#	VC TERM	D34#	GND	D35#	VC TERM	D48#	GND	D42#	VC TERM	D48#	GND	D37#	VC TERM	NC	GND	8	
9	TMS	TCK	GND	REC0#	GND	D35#	GND	D35#	GND	A21#	GND	A13#	GND	D33#	GND	D32#	GND	D33#	GND	DEP4#	GND	D38#	GND	D40#	GND	D39#	GND	VSSMON		9
10		GND	REC1#		REC2#		HT#		A24#		A20#	VC TERM	DEP7#	D51#	VC TERM	STEP3#	D50#	VC TERM	D32#		STEN3#	VC TERM	D34#	GND	GND	GND	VC TERM		10	
11	NC	NC	GND	REC3#	GND	DROV1#	GND	A22#	GND	A23#	GND	NC	GND	D30#	GND	STEN5#	GND	D38#	GND	DEP5#	GND	D41#	GND	STEP2#	GND	D33#	GND	VOCMON		11
12		GND	REC4#		REC5#		HT#		A25#		A19#	VC TERM	D57#	D51#	VC TERM	NC	GND	D48#	VC TERM	D48#	GND	D43#	GND	D39#	GND	NC	GND	VC TERM		12
13	BOLYN	BOLYP	GND	SSSY0#	GND	RFP#	GND	SSSY0#	GND	A22#	GND	A18#	GND	D39#	GND	D38#	GND	D32#	GND	D47#	GND	D43#	GND	D39#	GND	NC	GND	GND		13
14		GND	TRDY#		D35#		DEP8#		A34#		A31#	VC TERM	D34#	D37#	VC TERM	D34#		NC	VC TERM	D79#		D68#	VC TERM	D69#	GND	NC	VC TERM		14	
15	PAR	GOOD	FREQ	GND	LOOK#	GND	TND#	GND	BNIT#	GND	A37#	GND	A23#	GND	D32#	GND	D31#	GND	D31#	GND	D78#	GND	D71#	GND	D57#	GND	NC	GND	SMA2	15
16		GND	EPREC0#		EPREC1#		NC		A39#		A38#	VC TERM	DEP11#	VC TERM	D33#		STEP5#	VC TERM	D33#	GND	D78#	VC TERM	STEN4#	GND	D68#	VC TERM	NC	GND		16
17	NC	NC	GND	NC	GND	NC	GND	A33#	GND	A32#	GND	BNF#	GND	D39#	GND	STEN5#	GND	D38#	GND	DEP3#	GND	D72#	GND	STEP4#	GND	D73#	GND	SMA1		17
18		GND	EPREC3#		NC		EPREC3#		A39#		A23#	VC TERM	DEP10#	D39#	VC TERM	D38#		D30#	VC TERM	D77#		D69#	VC TERM	D64#	GND	SMA0	VC TERM		18	
19	NC	NC	GND	EPRE#	GND	SSSY1#	GND	D35#	GND	A30#	GND	A27#	GND	D30#	GND	D39#	GND	D32#	GND	DEP8#	GND	D75#	GND	D74#	GND	D70#	GND	GND		19
20		GND	FFOD	GND	RESET#		A06#	GND	A39#		A48#	GND	DEP14#	VC TERM	D122#	GND	D113#	VC TERM	D117#	GND	D111#	VC TERM	D108#	GND	D102#	VC TERM	NC	GND	20	
21	NC	NC	GND	TRST#	GND	NC	GND	DROV1#	GND	A44#	GND	A48#	GND	D124#	GND	D127#	GND	D112#	GND	DEP12#	GND	D101#	GND	D39#	GND	D32#	GND	SWAP		21
22		GND	UNTO		EPV0#		EPRE#		A40#		A47#	VC TERM	DEP15#	D125#	VC TERM	STEP7#	D114#	VC TERM	D109#		STEN6#	VC TERM	D38#	GND	SMD	VC TERM		22		
23	A20V#	IGNNB		EPV6#	GND	EPV6#	GND	AP1#	GND	A48#	GND	A42#	GND	D126#	GND	STEN7#	GND	D118#	GND	DEP13#	GND	D103#	GND	STEP6#	GND	D37#	GND	GND		23
24		GND	UNTI		GND	EPV4#	GND	EPV5#	GND	A43#		A40#	GND	D123#	VC TERM	D120#	GND	D119#	VC TERM	NC	GND	D109#	VC TERM	D103#	GND	D104#	VC TERM	SMD	GND	24
25	PERF#	TH_TRIP#		PV#	GND	EPV1#	GND	AP0#	GND	A41#	GND	VC TERM	GND	D121#	GND	D119#	GND	D113#	GND	D110#	GND	D107#	GND	D104#	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

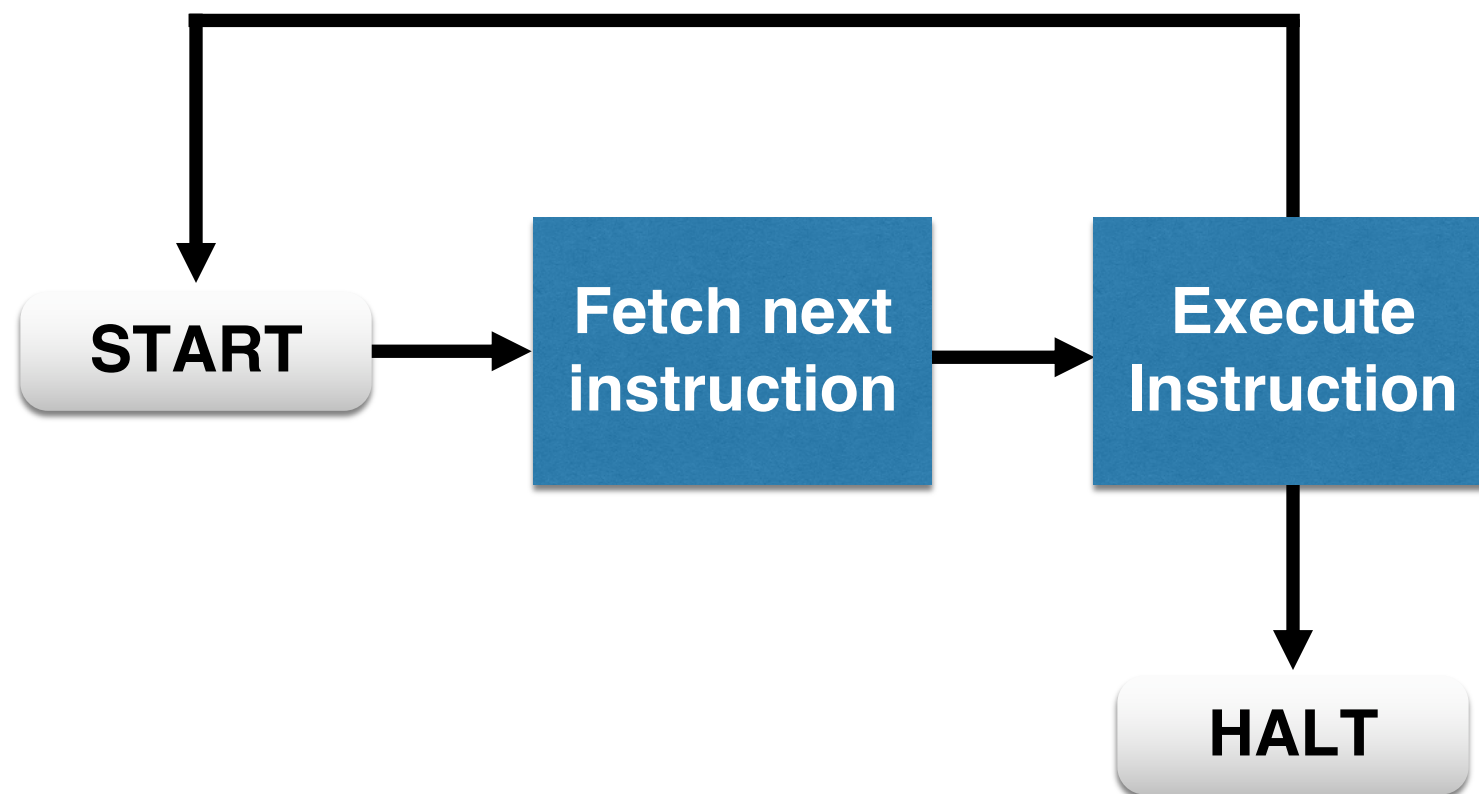


- How are interrupts handled on multicore machines?
 - On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
 - The OS can program the APICs to determine which interrupts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

Instruction Cycle



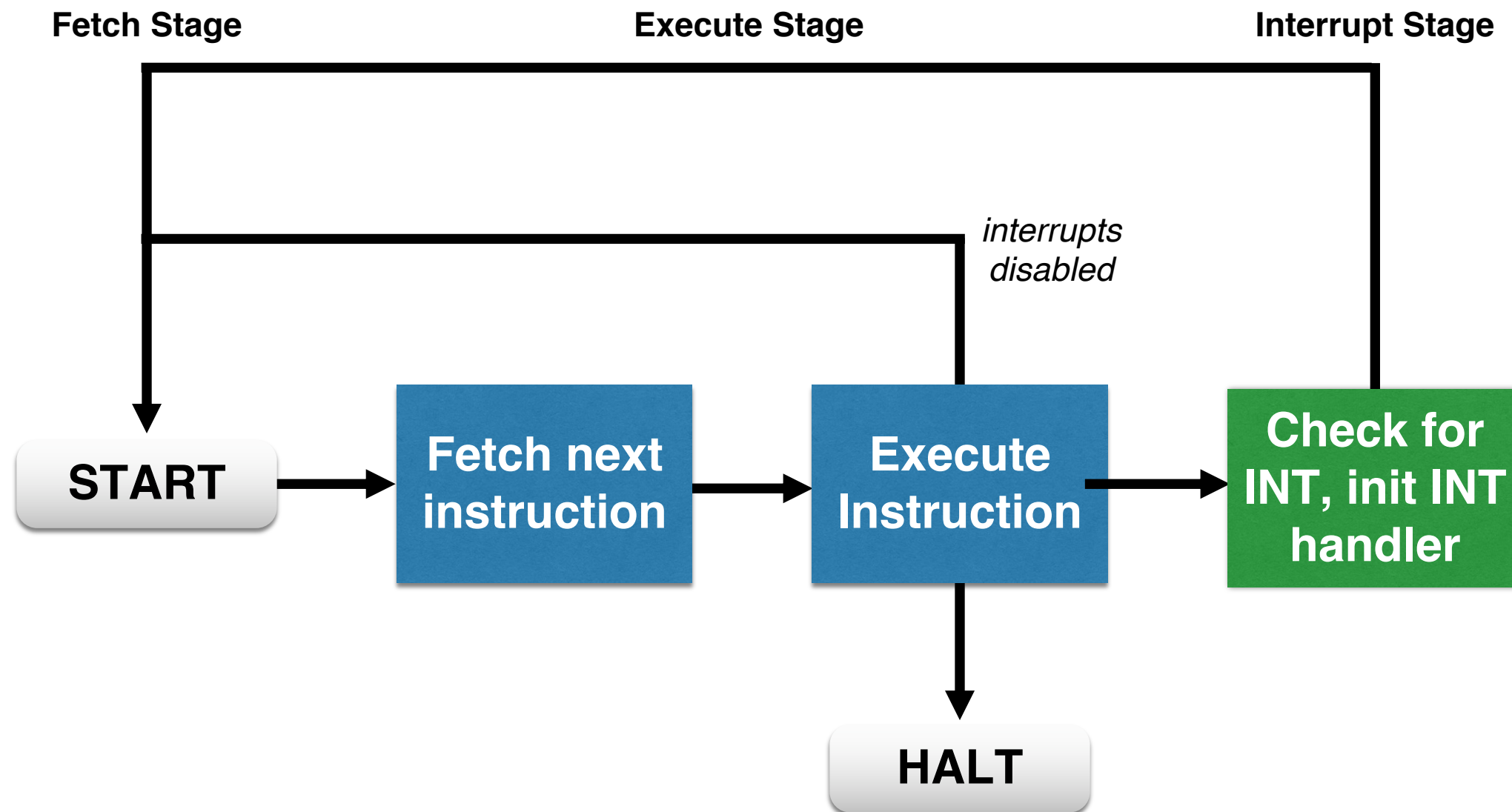
How does interrupt handling change the instruction cycle?



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.



- 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.



- 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);`