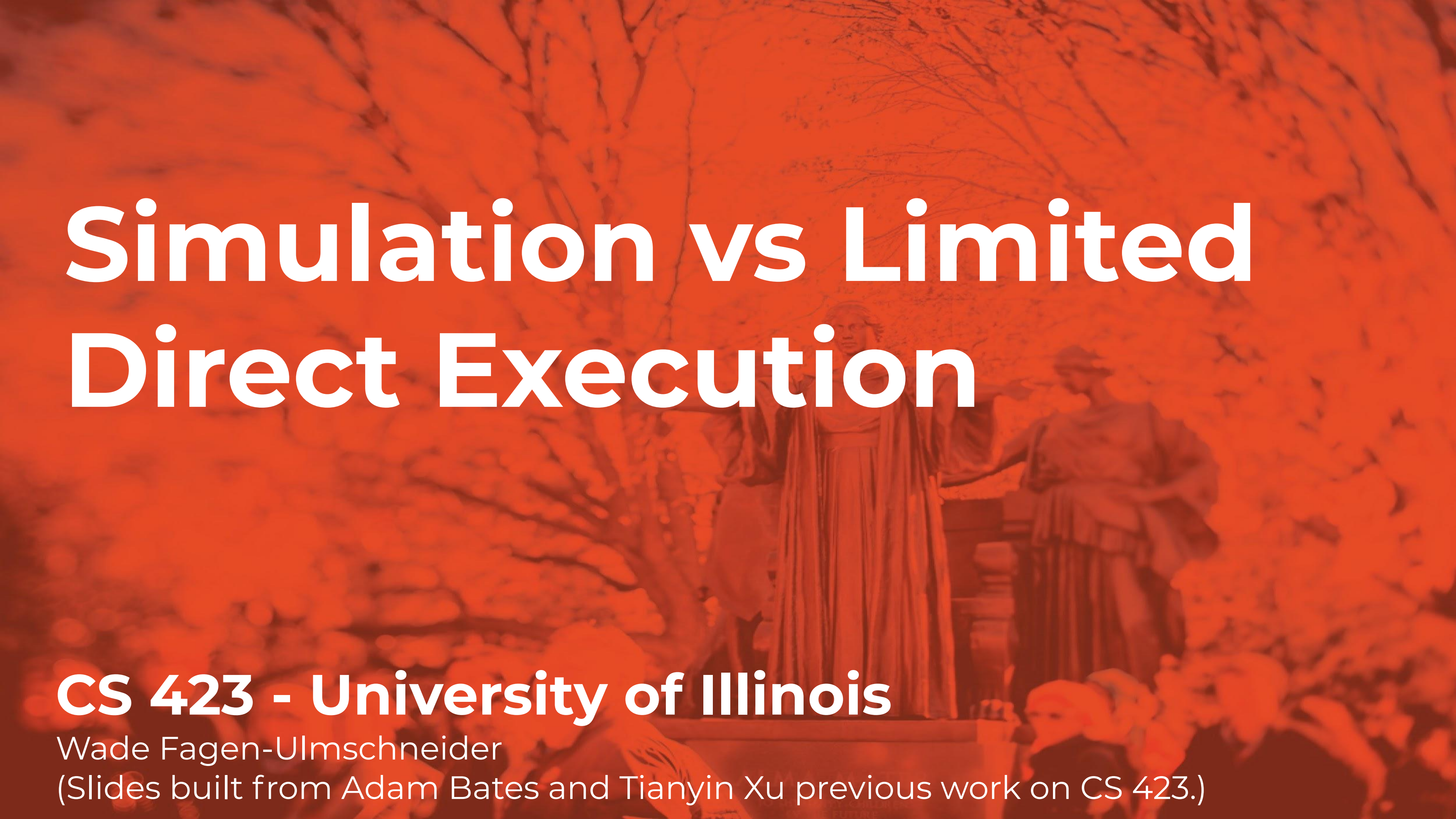
Welcome to Operating System Design (CS 423)

Wade Fagen-Ulmschneider
Spring 2021, University of Illinois

Slides built from Prof. Adam Bates and Prof. Tianyin Xu previous work on CS 423.



Simulation vs Limited Direct Execution

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

CPU Design

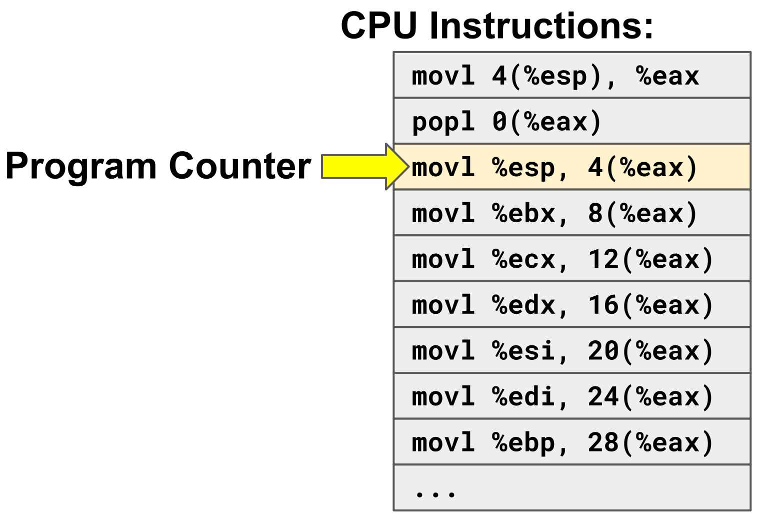
- CPUs, in the simplest terms, are efficient machines at doing the following:
  - Run the current operation at the current **program counter** address (“PC”).
  - Advance the PC.
  - Repeat.

Simple CPU Operation

CPU Instructions:

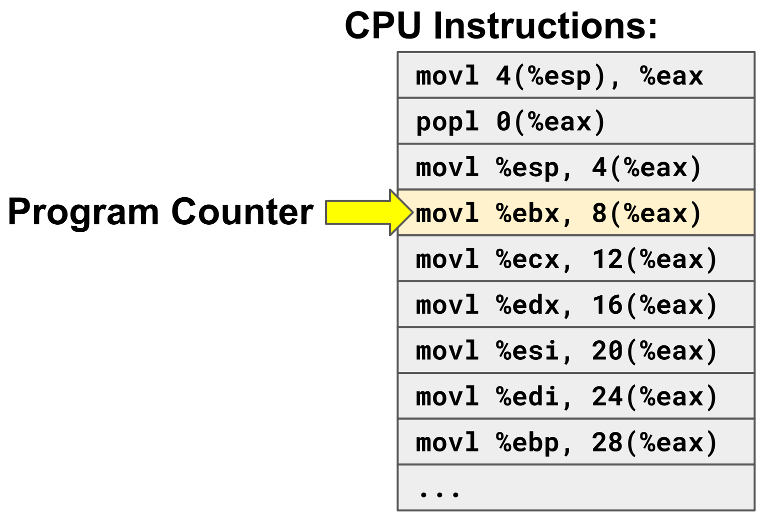
```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Simple CPU Operation



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Simple CPU Operation



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Processes

★ To organize what runs on your CPU, operating systems use a **process** to organize CPU instructions:

Processes

- ★ To organize what runs on your CPU, operating systems use a **process** to organize CPU instructions:
  - A process is an **instance of a program,**
  - Every process runs with a **limited set of rights** (memory address space, other permissions).

Limited Rights?

- Example:
  - Should a process be able to change its virtual page table?
  - Should a process be able to read any address in memory, or area of the disk (or any device)?
  - Should a process do other “privileged operations”?

Limited Rights?

★ Solution #1: **Simulation!**

Limited Rights?

- Solution #1: **Simulation!**
  - Our OS runs every process in a “CPU simulator”!
  - Check every instruction:
    - Permitted ⇒ Run it on the CPU
    - Illegal ⇒ Terminate the process

  - This is the basic model for many interpreted languages.

Limited Rights?

- Solution #1: **Simulation!**
  - Our OS runs every process in a “CPU simulator”!
  - Check every instruction:
    - Permitted ⇒ Run it on the CPU
    - Illegal ⇒ Terminate the process
  - This is the basic model for many interpreted languages.
- Very slow, lots of overhead.

Limited Rights?

★ Solution #2: **Run it Directly?!**
  - Allow arbitrary "user code" to run on the CPU.
  - Require the CPU to "user code" within a "protection ring".

Direct Execution

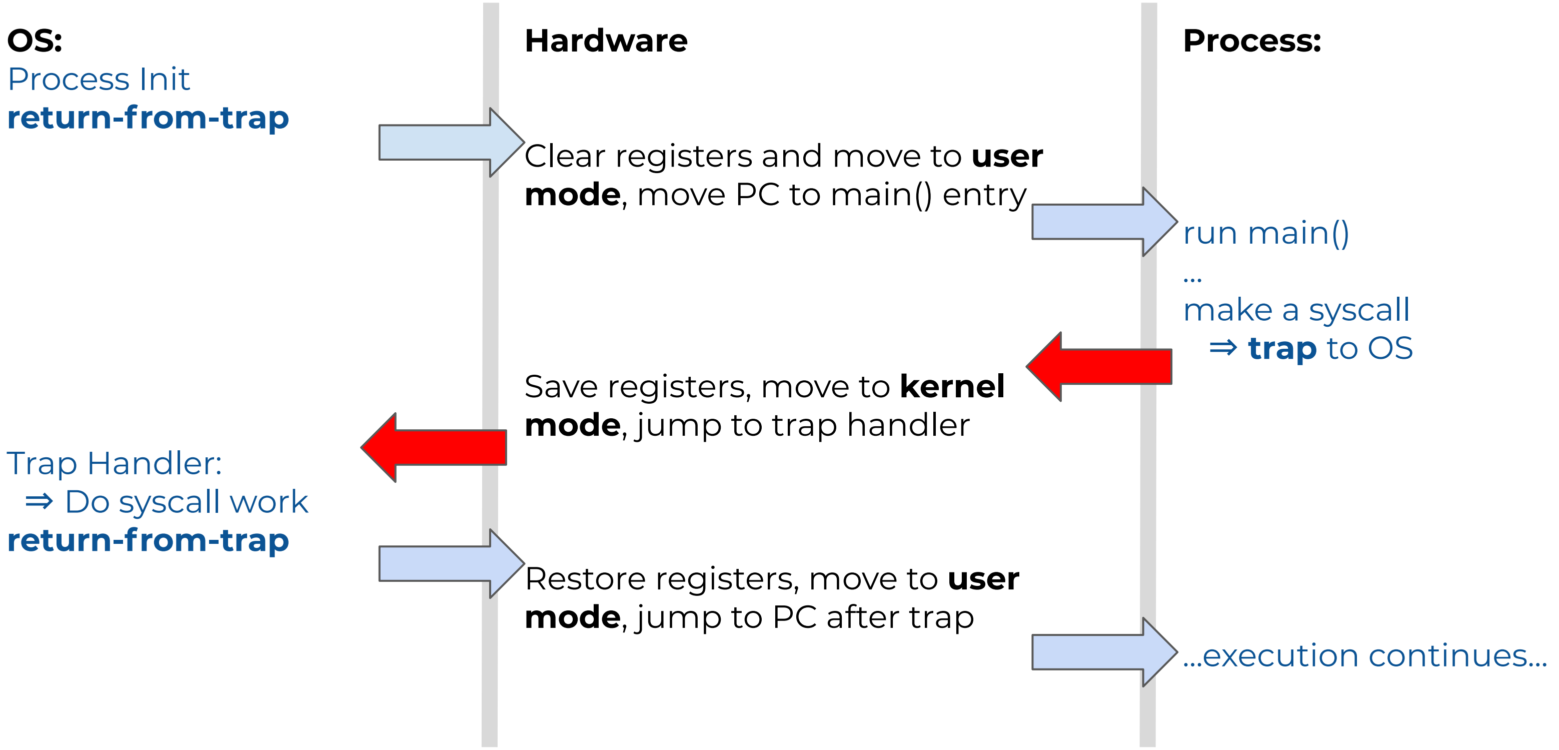
OS:
1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack (argv/argc)
5. Clear registers
6. **call** main()

Process:

7. Run main()
8. **return** from main()

9. Free memory for process
10. Remove process from process list

Limited Direct Execution



OS:
Process Init
return-from-trap

Hardware
Clear registers and move to **user mode**, move PC to main() entry

Process:
run main()
...
make a syscall
⇒ **trap** to OS

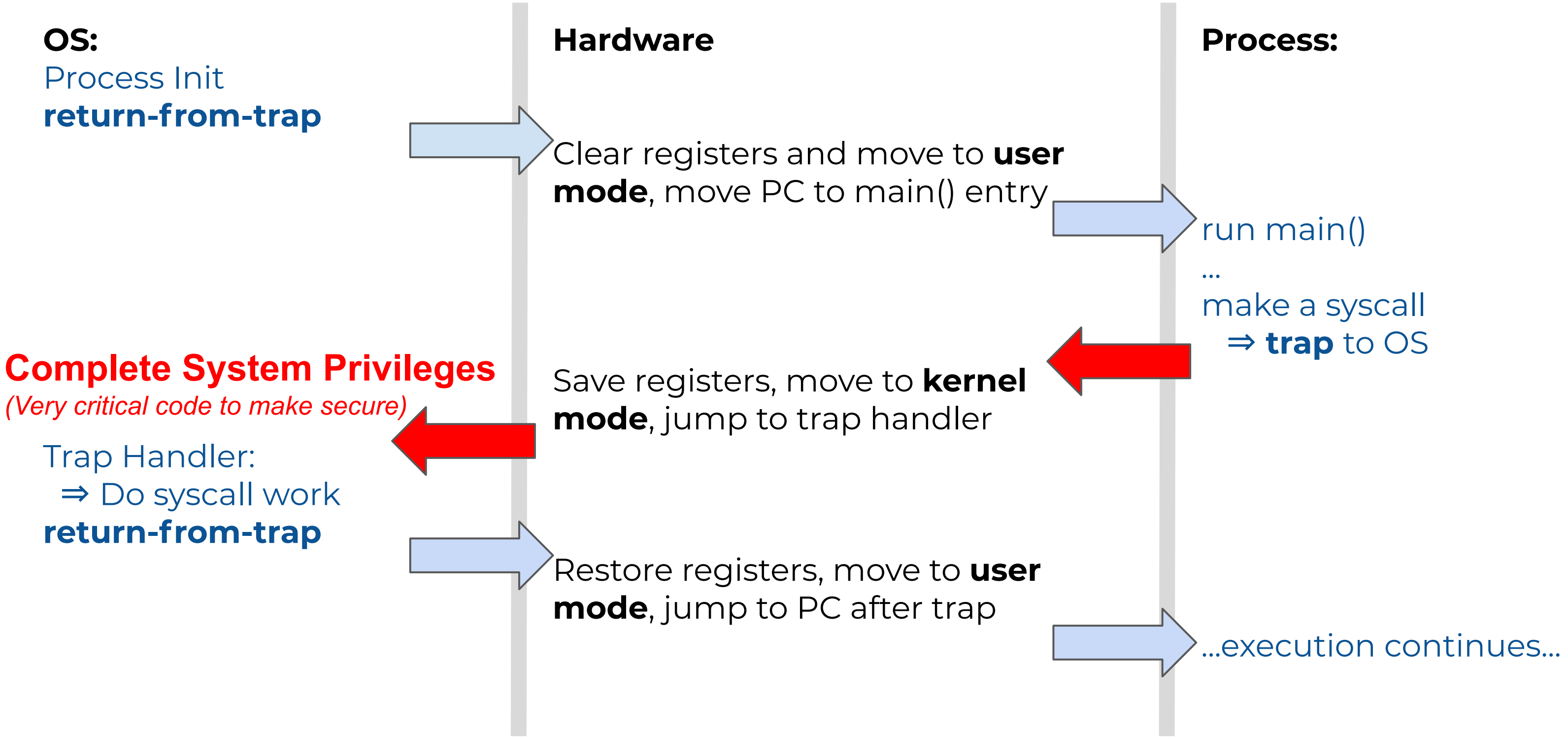
Hardware
Save registers, move to **kernel mode**, jump to trap handler

OS:
Trap Handler:
⇒ Do syscall work
return-from-trap

Hardware
Restore registers, move to **user mode**, jump to PC after trap

Process:
...execution continues...

Limited Direct Execution



OS:
Process Init
return-from-trap

Hardware
Clear registers and move to **user mode**, move PC to main() entry

Process:
run main()
…
make a syscall
⇒ **trap** to OS

Hardware
Save registers, move to **kernel mode**, jump to trap handler

OS:
Complete System Privileges
(Very critical code to make secure)
Trap Handler:
⇒ Do syscall work
return-from-trap

Hardware
Restore registers, move to **user mode**, jump to PC after trap

Process:
…execution continues…

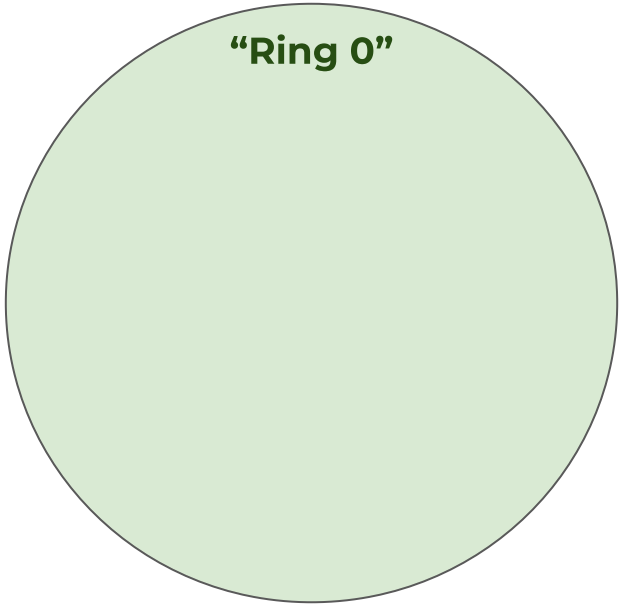
Protection Levels / Protection Rings

CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

x86 Protection Modes

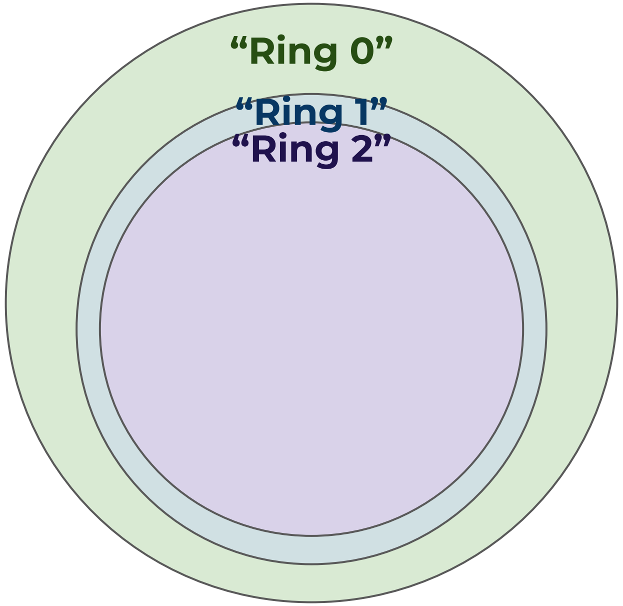
- The x86 architecture provides four protection modes:



- Complete access to all functions of the CPU. Absolutely no protections.
- This is the protection mode that kernel code runs in.

x86 Protection Modes

- ★ The x86 architecture provides four protection modes:

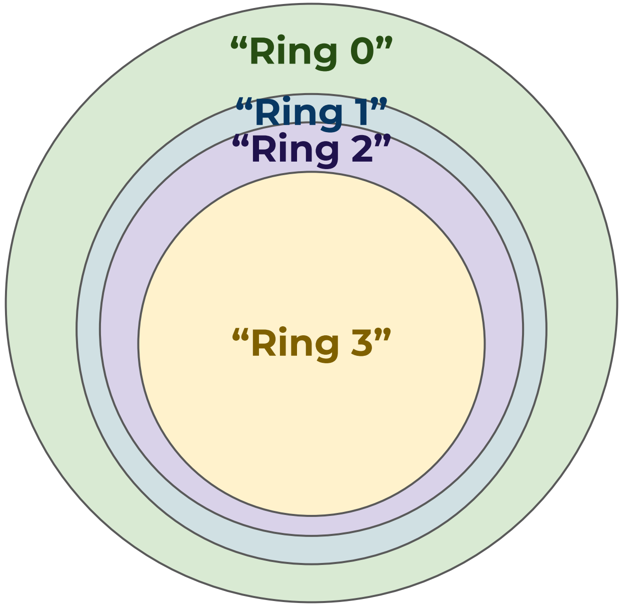


Ring 1 and **Ring 2**:

- ★ Customizable levels of protection, historically rarely used as ARM and other systems only had "two rings".
  - ○ *OSes are designed to be run across many CPUs, so designing such a critical piece of the OS just for x86 was not widely done.*
- ★ In recent years, "Ring 1" has been used for hosting virtualized kernels when full hardware virtualization is not available (ex: no VT-X, no AMD-V support).

x86 Protection Modes

★ The x86 architecture provides four protection modes:



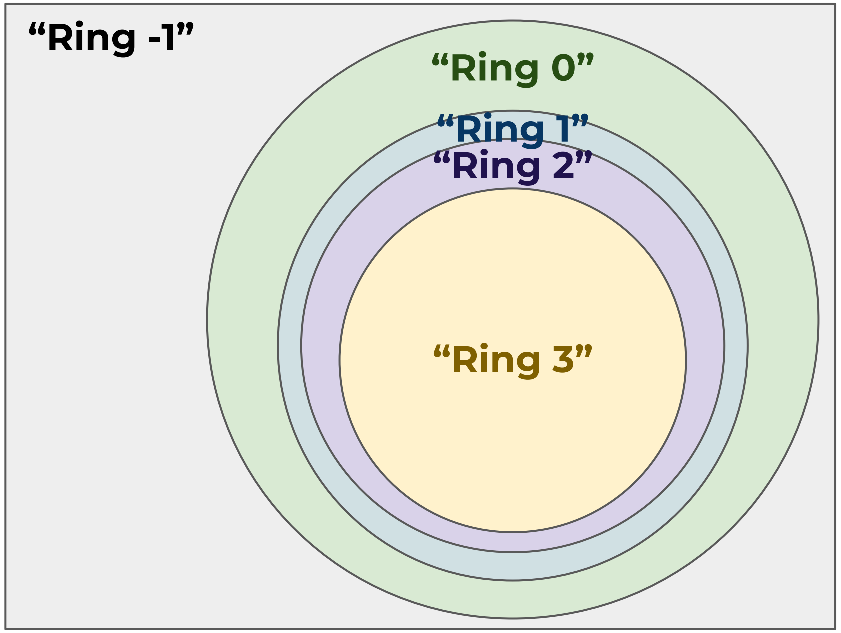
Ring 3:

★ Protection mode used for user-supplied code. All non-kernel software on an OS runs in this protection mode.

★ Often referred to as "user land" as opposed to "kernel land".

x86 Protection Modes

★ The x86 architecture provides four protection modes:



Ring -1 / "VMX root mode":

★ Hardware virtualization has introduced a new protection level to protect kernels from accessing resources controlled by the hypervisor.

★ *(We will ignore virtualization until later in the semester, but worth remembering we may be just an OS running as a virtual machine.)*

User Threads vs. Kernel Threads

Privileged Execution (“Ring 0”):

- ★ **Can do absolutely anything and everything.** No restrictions.

Userland Execution (“Ring 3”):

- ★ **Cannot change privilege level:**
  - ○ A user process should not be able to grant itself more privileges.
- ★ **Cannot modify page tables:**
  - ○ Page tables are a key aspect of process-level isolation.
- ★ **Cannot register interrupt handlers:**
- ★ **Limits on the I/O operations.**

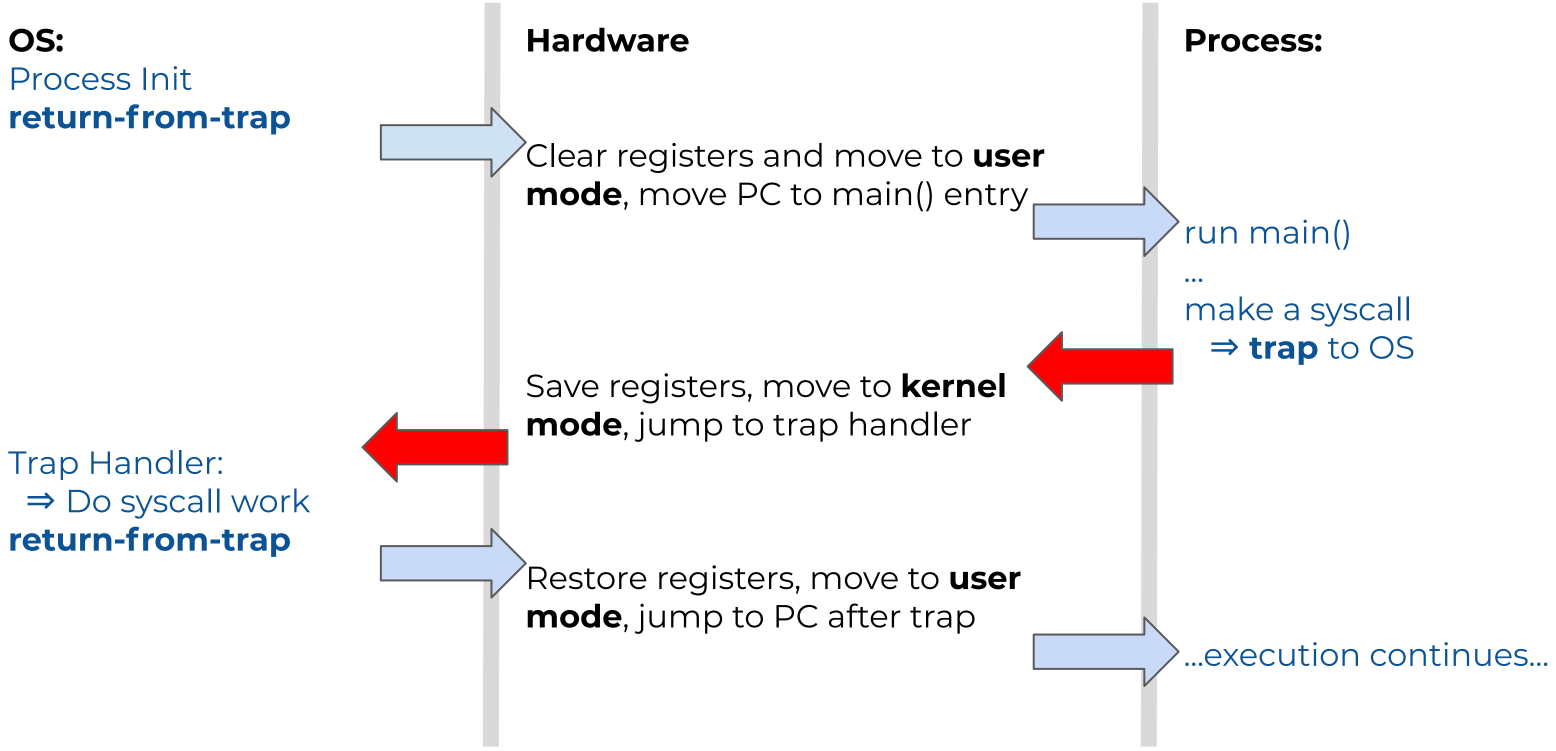
Paths between User and Kernel Code

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Limited Direct Execution



OS:	**Hardware**	**Process:**
Process Init **return-from-trap**		
	Clear registers and move to **user mode**, move PC to main() entry	
		run main() … make a syscall ⇒ **trap** to OS
	Save registers, move to **kernel mode**, jump to trap handler	
Trap Handler: ⇒ Do syscall work **return-from-trap**		
	Restore registers, move to **user mode**, jump to PC after trap	
		…execution continues…

How do we move to the kernel?

- ★ **Method #1**: syscall
  - All system calls will **trap** to the kernel, transferring execution from the user code to the kernel.

How do we move to the kernel?

★ **Method #1**: syscall

- ○ All system calls will **trap** to the kernel, transferring execution from the user code to the kernel.

★ What if the user makes no system calls?

```
while (1) { }   /* Will never make a syscall. */
```

How do we move to the kernel?

- **Method #2**: **interrupts**
  - Interrupts will interrupt the current execution and run the kernel code defined for the specific interrupt.
    - *Interrupt handlers are **run in the kernel**, in “kernel mode”.*

How do we move to the kernel?

- **Method #2: interrupts**
  - Interrupts will interrupt the current execution and run the kernel code defined for the specific interrupt.
    - *Interrupt handlers are* ***run in the kernel****, in "kernel mode".*
  - Many types of interrupts:
    - Data from a hard drive read operation is available,
    - A network packet arrives,
    - A CPU timeout occurs,
    - ...etc...

How do we move to the kernel?

- **Method #2**: **interrupts**
  - Interrupts will interrupt the current execution and run the kernel code defined for the specific interrupt.
    - *Interrupt handlers are **run in the kernel**, in "kernel mode".*
  - Many types of interrupts:
    - Data from a hard drive read operation is available,
    - A network packet arrives,
    - A CPU timeout occurs,
    - ...etc...

How do we move to the kernel?

- ★ **Method #3**: **exceptions**
  - Triggered by unwanted program behavior.

  - Ex: A program attempts to run a CPU opcode that is illegal in the current protection mode (ex: **`LGDT`** sets the interrupt handler, which can only be done by the kernel).

  - Handled by kernel code when it occurs.

Timer Interrupt

OS:

Process Init

return-from-trap

Hardware

Clear registers and move to **user mode**, move PC to main() entry

Process A:

while (1) { }

Timer Interrupt!

Save registers, move to **kernel mode**, jump to interrupt handler

Interrupt Handler:

⇒ Run system scheduler

⇒ Context switch to a new process

return-from-trap

Restore registers, move to **user mode**, jump to PC after trap

Process B:

while (1) { }

Four paths from User ⇒ Kernel code:

- **New process/thread start**
  - Jumps to the first instruction in the program/thread.

Four paths from User ⇒ Kernel code:

- ★ **New process/thread start**
  - ○ Jumps to the first instruction in the program/thread.

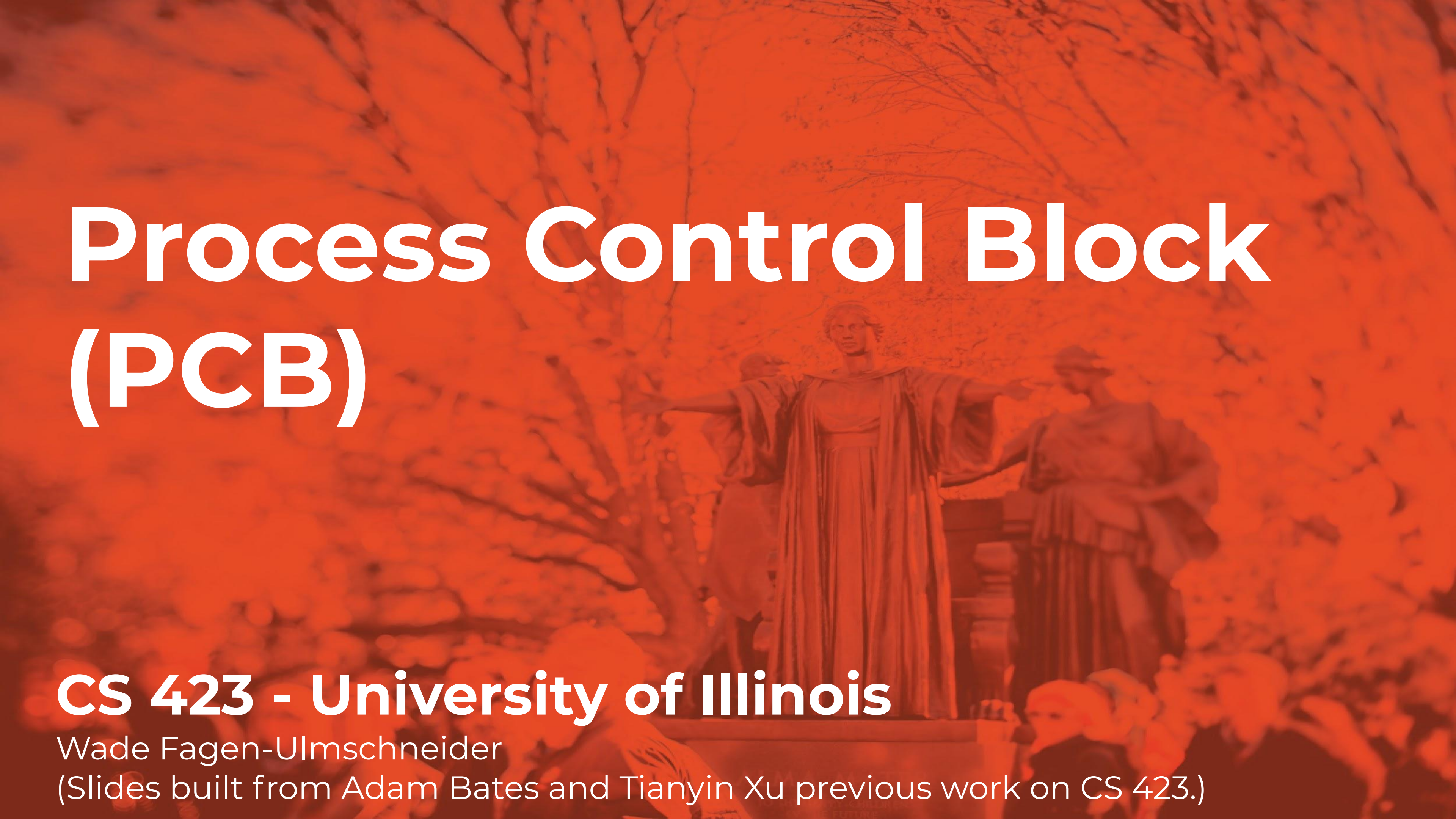
- ★ **Returns from a trap to kernel operation**
  - ○ Interrupt, exception, or system call.
  - ○ Jumps to the stored PC from previous execution of the process.

Four paths from User ⇒ Kernel code:

- ★ **New process/thread start**
  - ○ Jumps to the first instruction in the program/thread.

- ★ **Returns from a trap to kernel operation**
  - ○ Interrupt, exception, or system call.
  - ○ Jumps to the stored PC from previous execution of the process.

- ★ **User-level upcall (signal)**
  - ○ Jumps to the signal handler for the process.



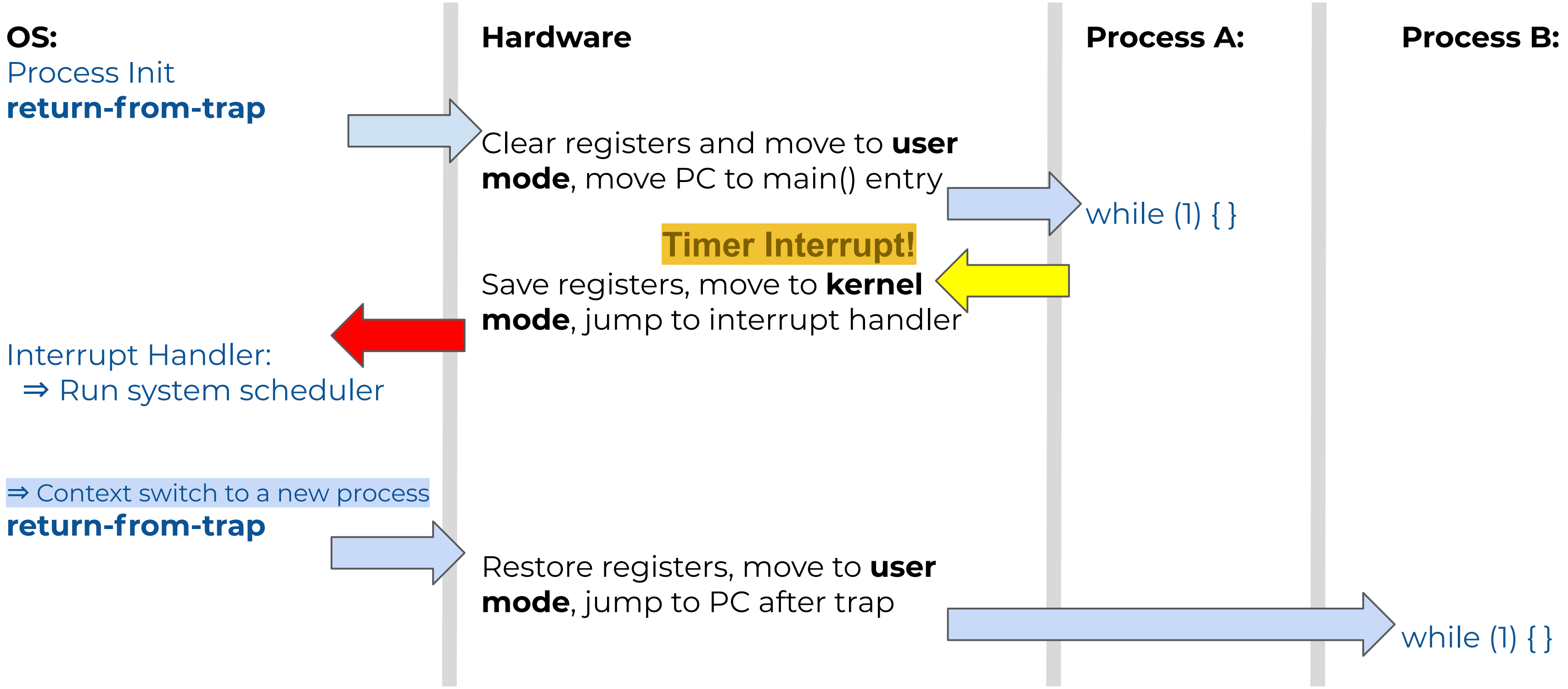
Process Control Block (PCB)

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Timer Interrupt



OS:

Process Init

return-from-trap

Hardware

Clear registers and move to **user mode**, move PC to main() entry

Process A:

while (1) { }

Timer Interrupt!

Save registers, move to **kernel mode**, jump to interrupt handler

Interrupt Handler:

⇒ Run system scheduler

⇒ Context switch to a new process

return-from-trap

Restore registers, move to **user mode**, jump to PC after trap

Process B:

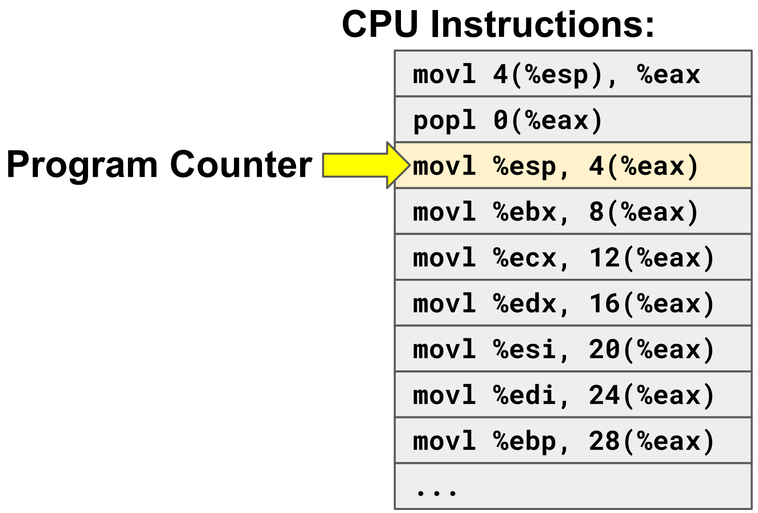
while (1) { }

Process Control Block

★ When we run a process, what is the state of the system that is specific to that process?

Process Control Block

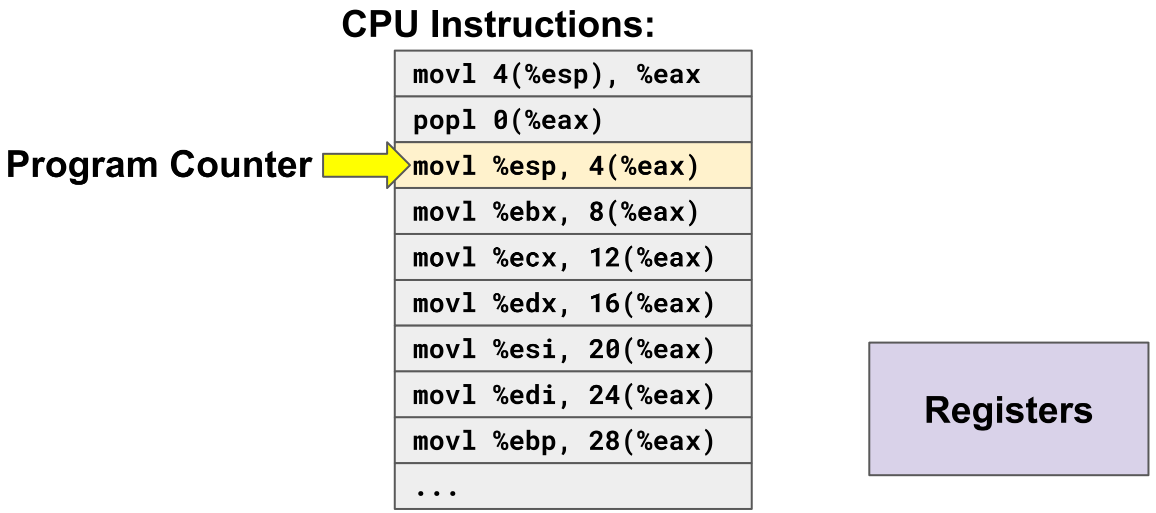
★ When we run a process, what is the state of the system that is specific to that process?



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Process Control Block

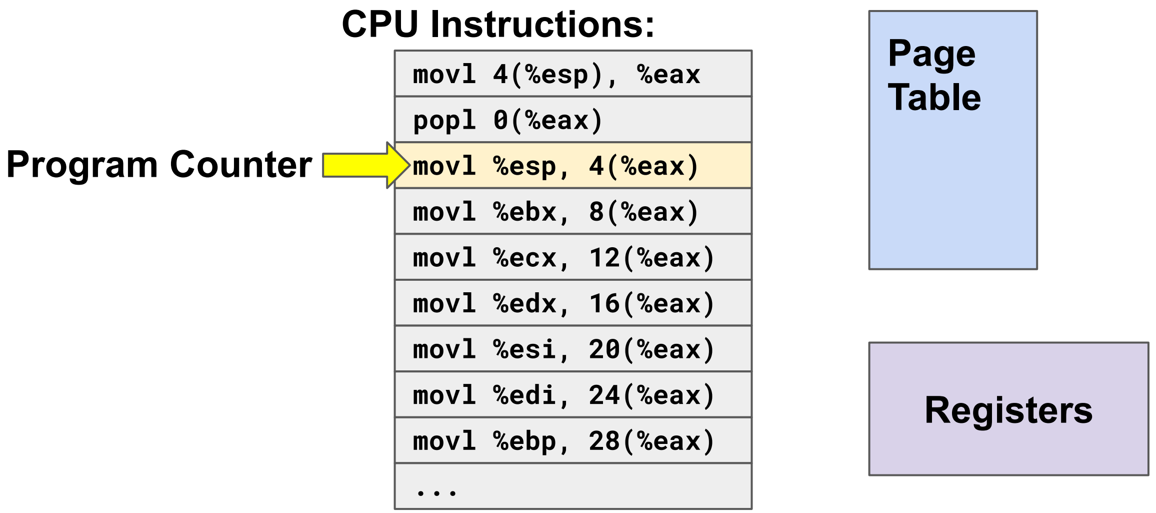
★ When we run a process, what is the state of the system that is specific to that process?



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Process Control Block

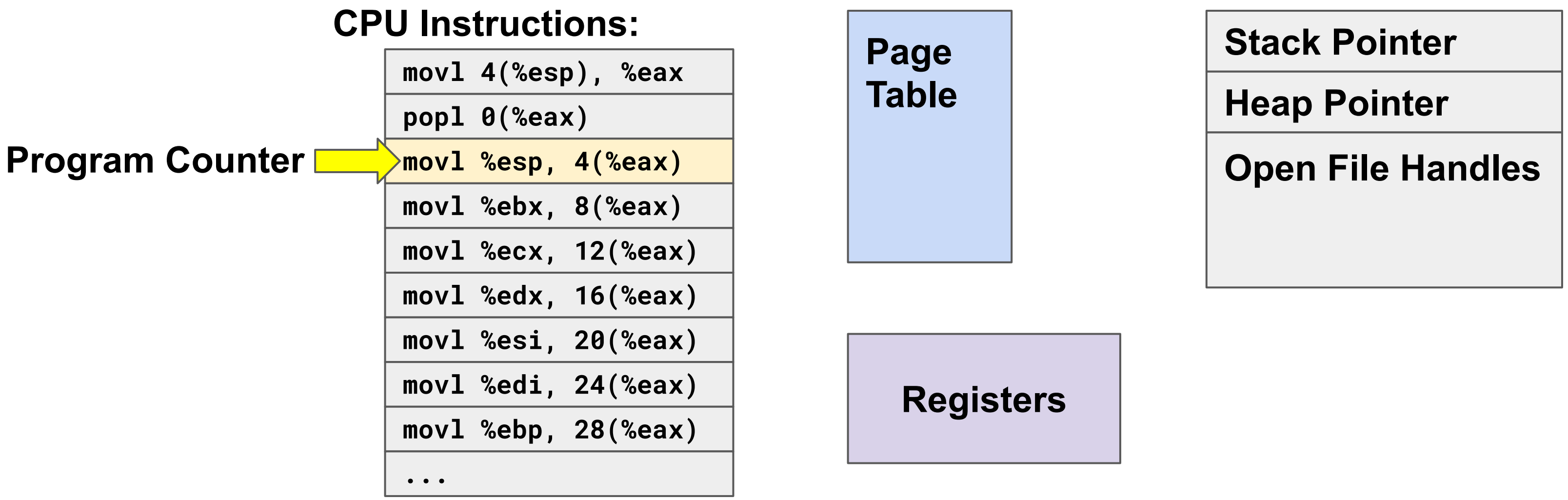
★ When we run a process, what is the state of the system that is specific to that process?



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Process Control Block

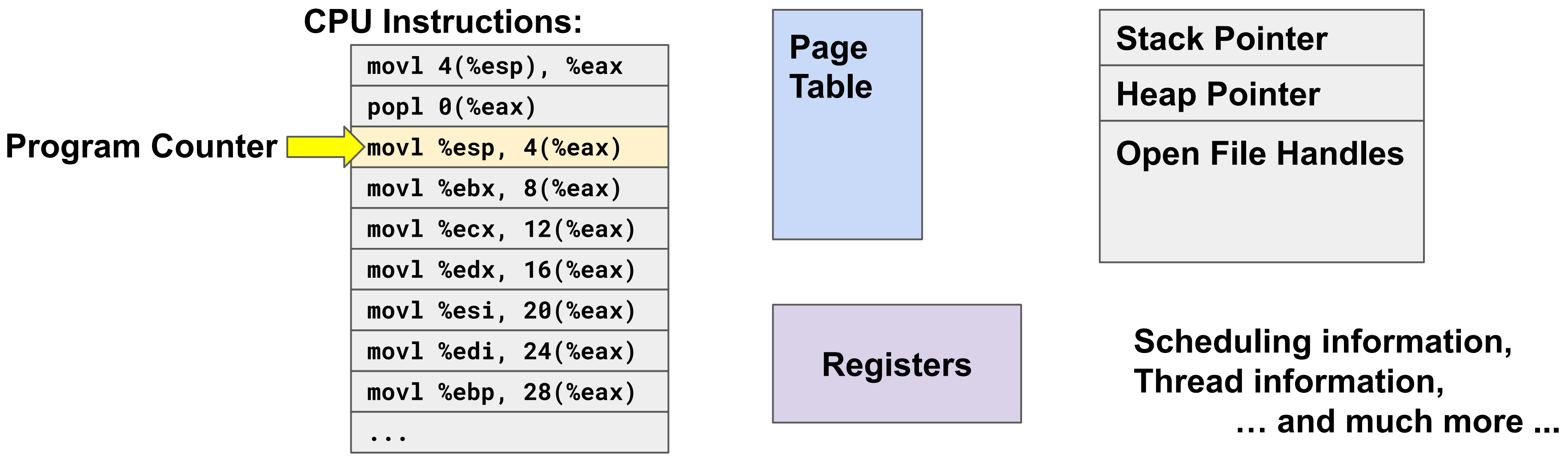
★ When we run a process, what is the state of the system that is specific to that process?



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Process Control Block

★ When we run a process, what is the state of the system that is specific to that process?



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Process Control Block

- ★ The **Process Control Block (PCB)** contains everything necessary to store the state of the process.

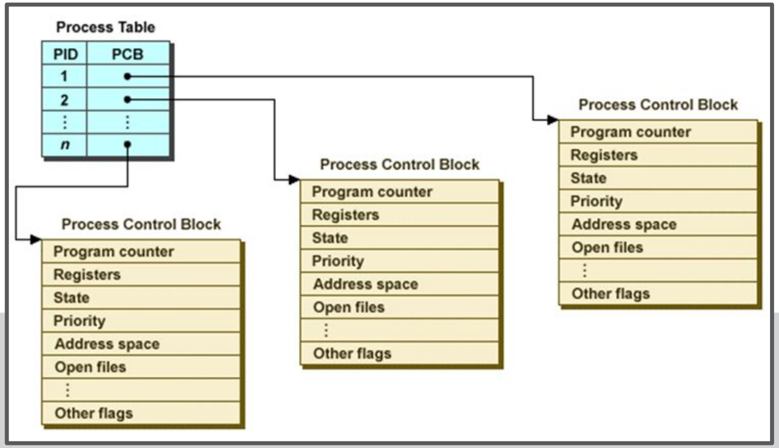
Process Control Block

- The **Process Control Block (PCB)** contains everything necessary to store the state of the process.

  - When the kernel scheduler switches the execution on a CPU to a new process:
    - The state of the current process is saved to its PCB, **and**
    - The state of the new process is loaded from its PCB.

Process Control Block

- ★ The **Process Control Block (PCB)** contains everything necessary to store the state of the process.
  - ○ When the kernel scheduler switches the execution on a CPU to a new process:
    - ■ The state of the current process is saved to its PCB, **and**
    - ■ The state of the new process is loaded from its PCB.
- ★ **One PCB /process:**



Thread Control Block

★ The **Thread Control Block (TCB)** contains thread-specific state for all threads in a process.

Thread Control Block

- The **Thread Control Block (TCB)** contains thread-specific state for all threads in a process.

  - Only needs to maintain data unique to the thread and not shared with the process:
    - Ex: Threads **share** the same page table.
    - Ex: Threads **do not share** the same PC or stack pointer.

Interrupt Vector Table

CS 423 - University of Illinois

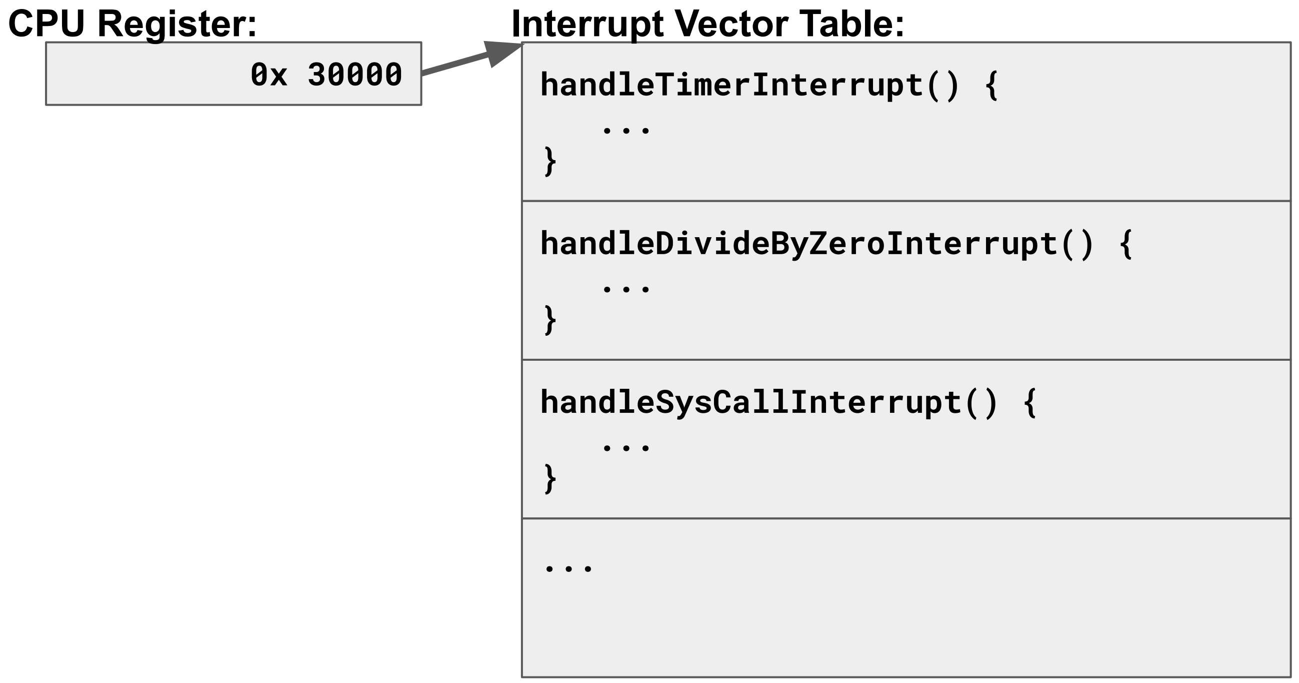
Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Interrupt Vector Table

- ★ The **Interrupt Vector Table** is where a processor finds the kernel code for all interrupts.
  - Stored in RAM at a known address,
  - Interrupts are always run as the kernel,
  - Limited number of interrupts possible for a CPU/architecture

Interrupt Vector Table



```
handleTimerInterrupt() {
   ...
}
```

```
handleDivideByZeroInterrupt() {
   ...
}
```

```
handleSysCallInterrupt() {
   ...
}
```

```
...
```

Interrupt Safety:

★ When an interrupt occurs, the CPU ensures:

1. **Atomic transfer of control** -- **all** of the following will be switched **atomicly** to the interrupt handler:
   - Program Counter
   - Stack Pointer
   - Memory Protection
   - User/Kernel Mode

Interrupt Safety:

★ When an interrupt occurs, the CPU ensures:

2. **Transparent, restartable execution** -- user program does not know the interrupt occurred!

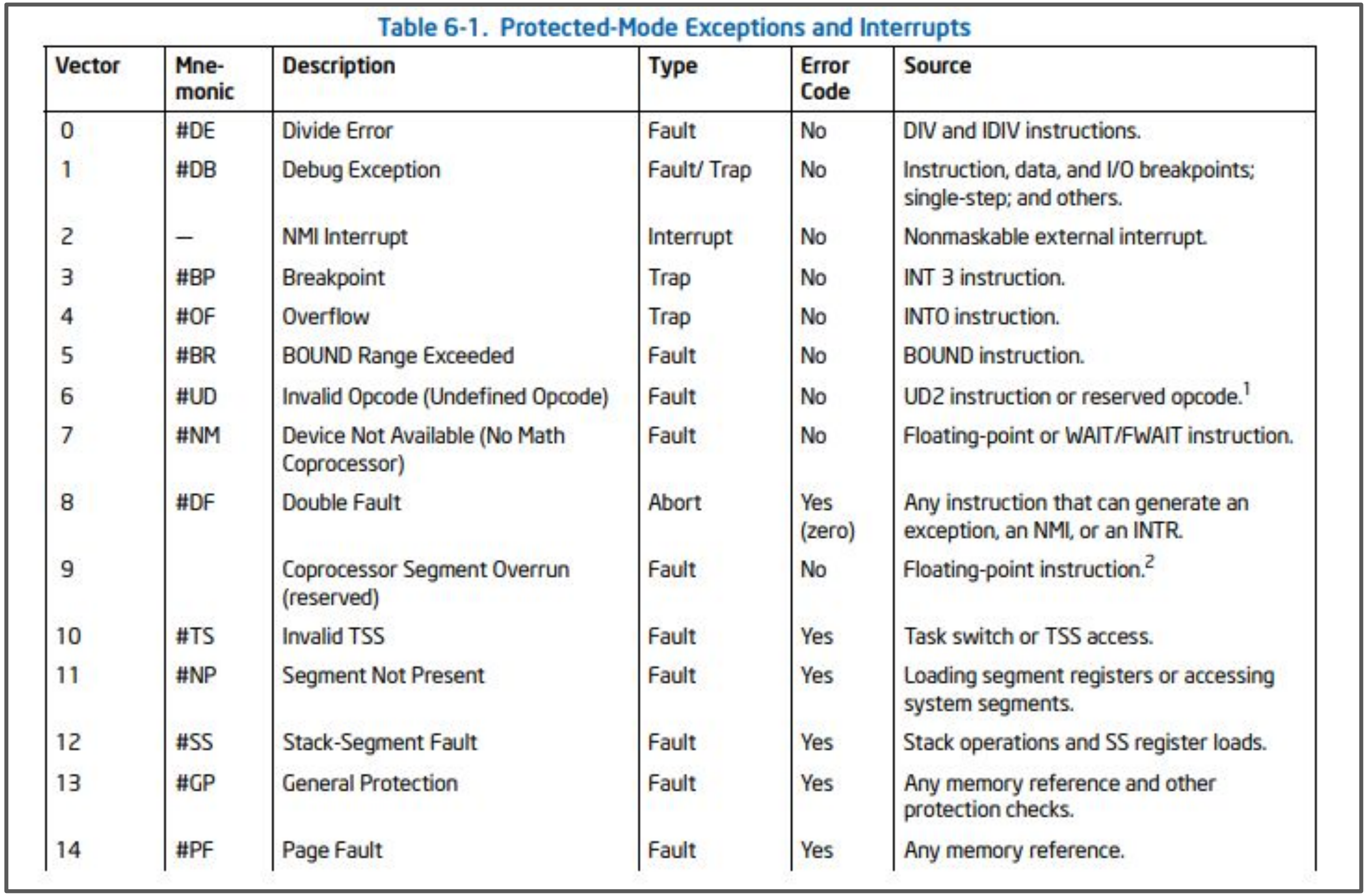


Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

Multiple Interrupts

- ★ **Fun Fact:** Most processes have both a kernel and a user stack!
    - ○ Multiple interrupts may be pending at one time.

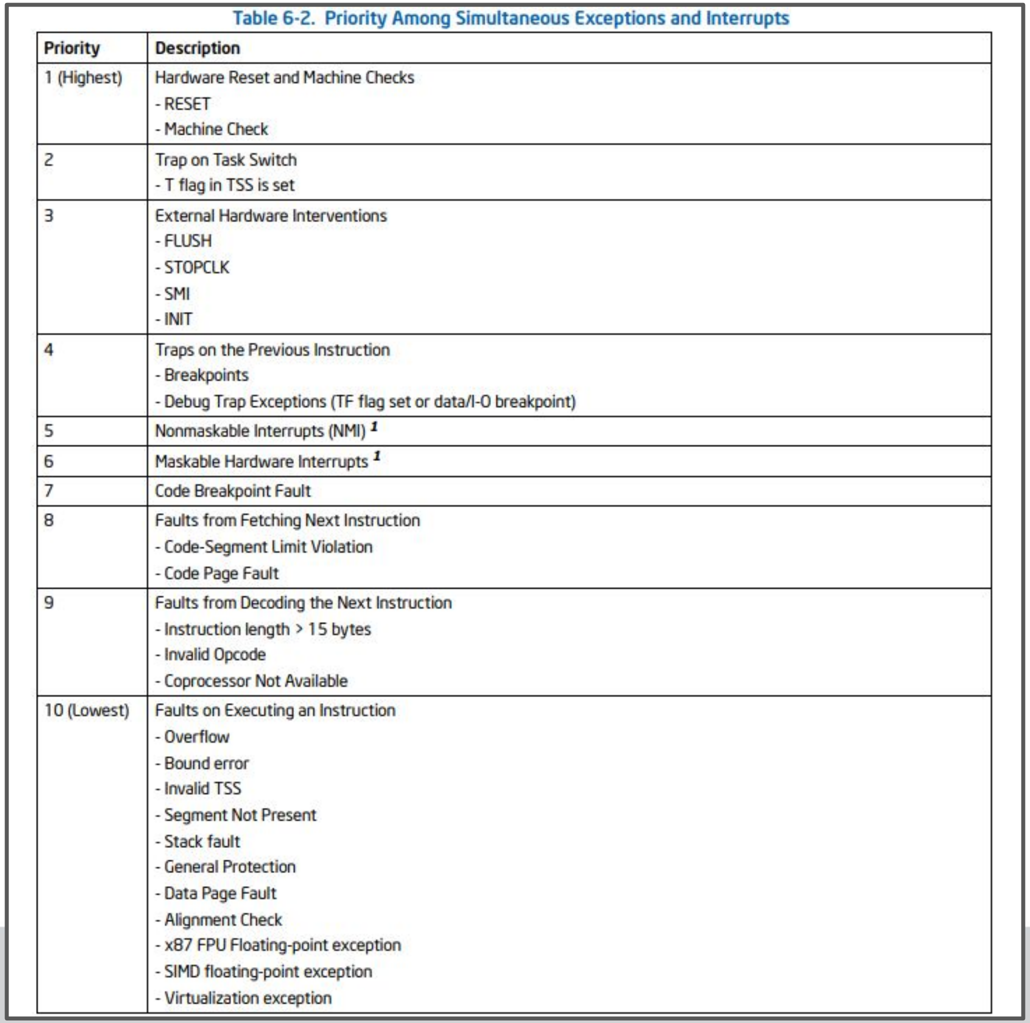


Table 6-2. Priority Among Simultaneous Exceptions and Interrupts

Priority	Description
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check
2	Trap on Task Switch - T flag in TSS is set
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Breakpoints - Debug Trap Exceptions (TF flag set or data/I-O breakpoint)
5	Nonmaskable Interrupts (NMI) [1]
6	Maskable Hardware Interrupts [1]
7	Code Breakpoint Fault
8	Faults from Fetching Next Instruction - Code-Segment Limit Violation - Code Page Fault
9	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Invalid Opcode - Coprocessor Not Available
10 (Lowest)	Faults on Executing an Instruction - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - x87 FPU Floating-point exception - SIMD floating-point exception - Virtualization exception

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

Hardware-Generated Interrupts

- ★ **Hardware-generated** interrupts are physical I/O devices connected to different pins on the CPU's "interrupt controller":
  - Device hardware will signal on their corresponding pin,
  - Lots of different devices: network packets, mouse clicks, keyboard input, and the timer interrupt!

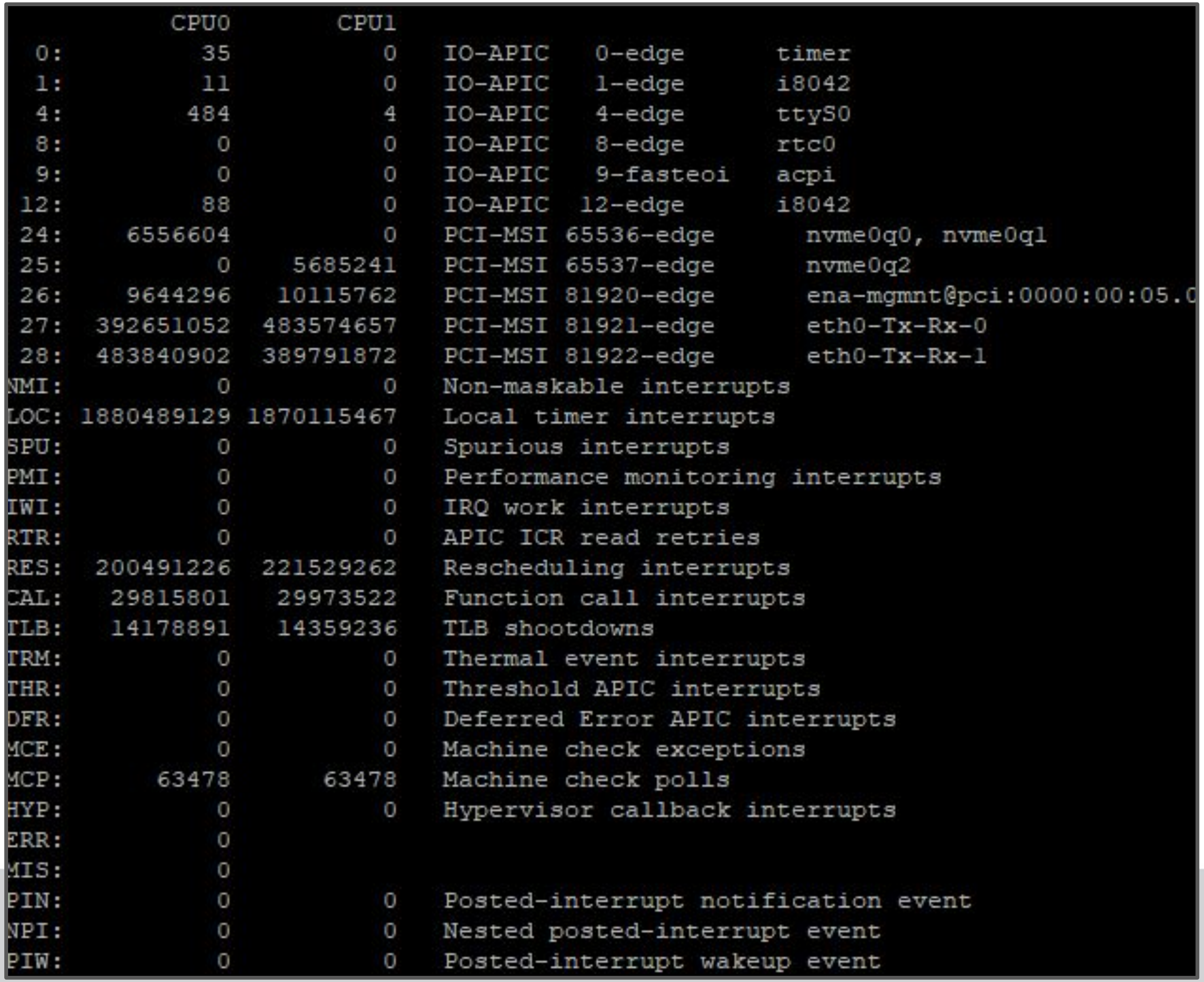
System-Generated Interrupts

- ★ Many system events generate interrupts, including page faults and other “need-to-load-data” events.

- ★ **User code** can also generate directly interrupts via the interrupt syscall.

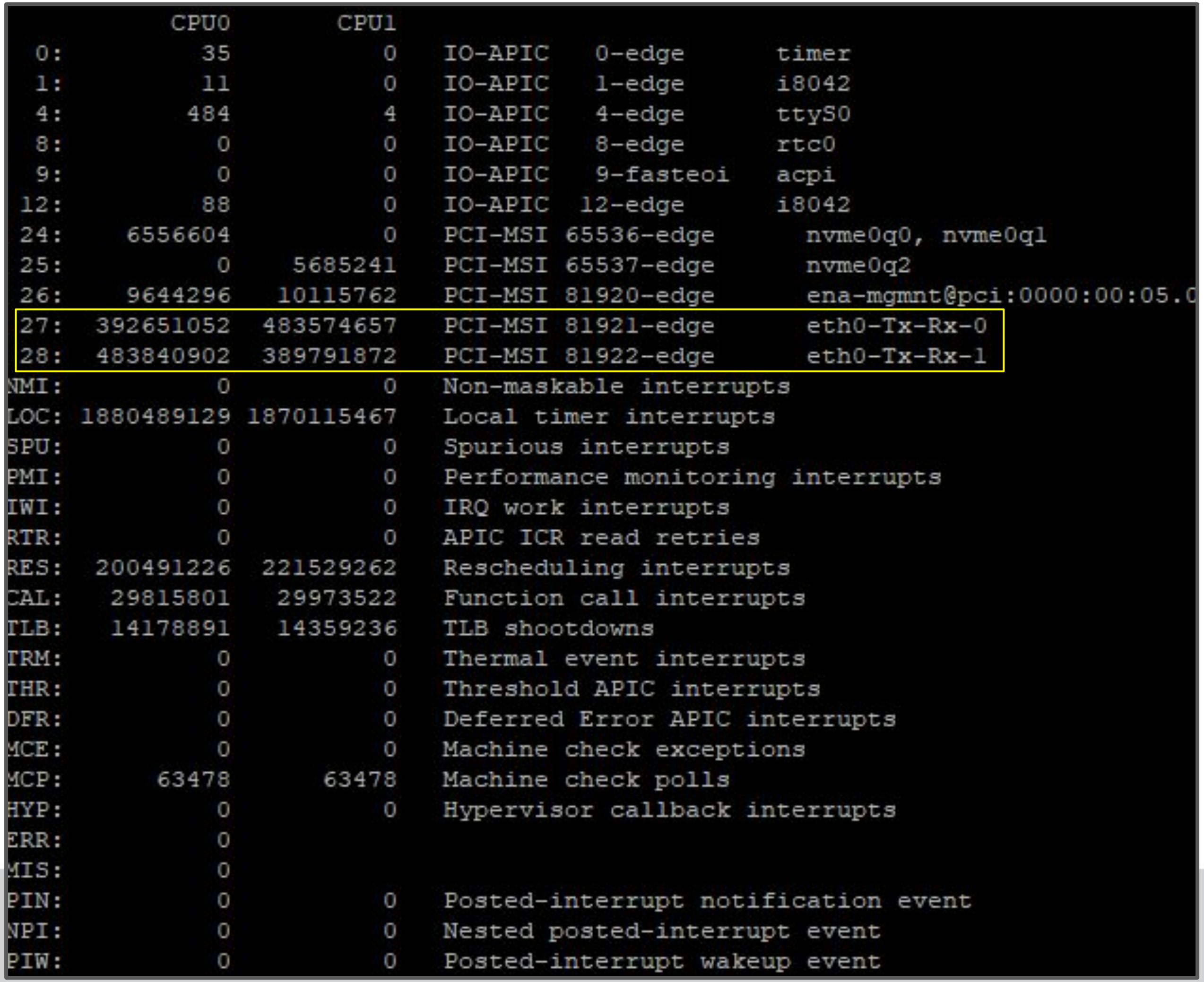
Multi-Core Interrupt Handling

- ★ 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 APIC to route specific interrupts to specific CPUs.
  - On Linux, **`/proc/interrupts`** shows the handling per CPU of each interrupt.



```
            CPU0       CPU1
  0:          35          0   IO-APIC   0-edge      timer
  1:          11          0   IO-APIC   1-edge      i8042
  4:         484          4   IO-APIC   4-edge      ttyS0
  8:           0          0   IO-APIC   8-edge      rtc0
  9:           0          0   IO-APIC   9-fasteoi   acpi
 12:          88          0   IO-APIC  12-edge      i8042
 24:     6556604          0   PCI-MSI 65536-edge      nvme0q0, nvme0q1
 25:           0    5685241   PCI-MSI 65537-edge      nvme0q2
 26:     9644296   10115762   PCI-MSI 81920-edge      ena-mgmnt@pci:0000:00:05.0
 27:   392651052  483574657   PCI-MSI 81921-edge      eth0-Tx-Rx-0
 28:   483840902  389791872   PCI-MSI 81922-edge      eth0-Tx-Rx-1
NMI:           0          0   Non-maskable interrupts
LOC:  1880489129 1870115467   Local timer interrupts
SPU:           0          0   Spurious interrupts
PMI:           0          0   Performance monitoring interrupts
IWI:           0          0   IRQ work interrupts
RTR:           0          0   APIC ICR read retries
RES:   200491226  221529262   Rescheduling interrupts
CAL:    29815801   29973522   Function call interrupts
TLB:    14178891   14359236   TLB shootdowns
TRM:           0          0   Thermal event interrupts
THR:           0          0   Threshold APIC interrupts
DFR:           0          0   Deferred Error APIC interrupts
MCE:           0          0   Machine check exceptions
MCP:       63478      63478   Machine check polls
HYP:           0          0   Hypervisor callback interrupts
ERR:           0
MIS:           0
PIN:           0          0   Posted-interrupt notification event
NPI:           0          0   Nested posted-interrupt event
PIW:           0          0   Posted-interrupt wakeup event
```

`cat /proc/interrupts`
on my 91-divoc.com server
after an uptime of 238 days.

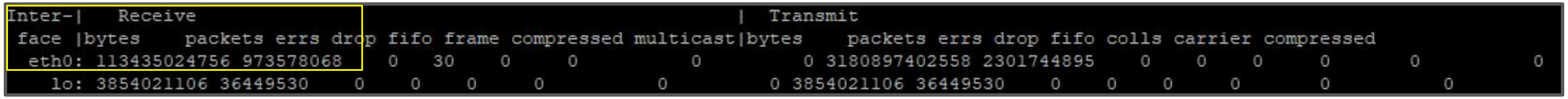


```
           CPU0       CPU1
  0:         35          0   IO-APIC   0-edge      timer
  1:         11          0   IO-APIC   1-edge      i8042
  4:        484          4   IO-APIC   4-edge      ttyS0
  8:          0          0   IO-APIC   8-edge      rtc0
  9:          0          0   IO-APIC   9-fasteoi   acpi
 12:         88          0   IO-APIC  12-edge      i8042
 24:    6556604          0   PCI-MSI 65536-edge      nvme0q0, nvme0q1
 25:          0    5685241   PCI-MSI 65537-edge      nvme0q2
 26:    9644296   10115762   PCI-MSI 81920-edge      ena-mgmnt@pci:0000:00:05.0
 27:  392651052  483574657   PCI-MSI 81921-edge      eth0-Tx-Rx-0
 28:  483840902  389791872   PCI-MSI 81922-edge      eth0-Tx-Rx-1
NMI:          0          0   Non-maskable interrupts
LOC: 1880489129 1870115467   Local timer interrupts
SPU:          0          0   Spurious interrupts
PMI:          0          0   Performance monitoring interrupts
IWI:          0          0   IRQ work interrupts
RTR:          0          0   APIC ICR read retries
RES:  200491226  221529262   Rescheduling interrupts
CAL:   29815801   29973522   Function call interrupts
TLB:   14178891   14359236   TLB shootdowns
TRM:          0          0   Thermal event interrupts
THR:          0          0   Threshold APIC interrupts
DFR:          0          0   Deferred Error APIC interrupts
MCE:          0          0   Machine check exceptions
MCP:      63478      63478   Machine check polls
HYP:          0          0   Hypervisor callback interrupts
ERR:          0
MIS:          0
PIN:          0          0   Posted-interrupt notification event
NPI:          0          0   Nested posted-interrupt event
PIW:          0          0   Posted-interrupt wakeup event
```

`cat /proc/interrupts`
on my 91-divoc.com server after an uptime of 238 days.

Total of **1.769b** interrupts on eth0 -- that's **7.35m interrupts /day** or **>85 interrupts /second** just on eth0!

`cat /proc/net/dev:`



```
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
  eth0: 113435024756 973578068    0   30    0     0          0         0 3180897402558 2301744895    0    0    0     0       0          0
    lo: 3854021106 36449530    0    0    0     0          0         0 3854021106 36449530    0    0    0     0       0          0
```

★ Received 113,435,024,756 (106 GiB) bytes from 973,578,068 packets.
- ~2 interrupts /packet
- ~64 B /interrupt

Interrupt Handler Design

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Interrupt Handling Design

★ Interrupts happen often and interrupt all other tasks on the CPU.

Interrupt Handling Design

- ★ Interrupts happen often and interrupt all other tasks on the CPU.

  - Code should be **as minimal as possible**,
  - Code should be **as fast as possible**,
  - **Run only absolutely necessary tasks** and defer all other work until later.

Interrupt Handling Design

- Interrupts happen often and interrupt all other tasks on the CPU.

    - Code should be **as minimal as possible**,
    - Code should be **as fast as possible**,
    - **Run only absolutely necessary tasks** and defer all other work until later.
        - The interrupt handler is the **“top half”** and then the deferred work is the **“bottom half”** in a **Two-Halves design** for handling interrupts.

Example: Network Packet

- ★ **Interrupt**: Data available on network device.
- ★ **Top Half** (interrupt handler):
  - Copy the data from network I/O device into memory
- ★ **Bottom Half** (deferred work):
  - Process the network packet,
  - Deliver it to the appropriate process based on TCP/UDP port

Two-Halves Design

- The two-halves design is the simplest approach to interrupt handling and can take on various forms:
  - All must be **statically defined**/allocated **at compile time**.
  - In Linux, there are three common mechanism for deferred work for interrupt handlers:
    - softirqs,
    - tasklets, or
    - workqueues

softirq

- ★ **softirq**s (**soft**ware **i**nterrupt **req**uests) uses a `raise_softirq()` to mark a given softirq must execute deferred work.

- ★ **softirqs** are later scheduled when scheduling permits

- ★ Linux has several different types of softirqs:
  - `HI_SOFTIRQ`
  - `TIMER_SOFTIRQ`
  - `NET_TX_SOFTRQ`
  - `NET_RX_SOFTIRQ`
  - `BLOCK_SOFTIRQ`
  - `TASKLET_SOFTIRQ`
  - `SCHED_SOFTIRQ`

softirq

- **softirq**s (**soft**ware **i**nterrupt **req**uests) uses a `raise_softirq()` to mark a given softirq must execute deferred work.

- **softirqs** are later scheduled when scheduling permits

- Linux has several different types of softirqs:
    - `HI_SOFTIRQ`
    - `TIMER_SOFTIRQ`
    - `NET_TX_SOFTRQ`
    - `NET_RX_SOFTIRQ`
    - `BLOCK_SOFTIRQ`
    - `TASKLET_SOFTIRQ`
    - `SCHED_SOFTIRQ`

tasklets

- ★ **Tasklets** are special types of softirqs that are easiest to use with some additional constraints:
  - tasklets are non-reentrant and have no internal state,
  - various tasklets can run simultaneously across many CPUs

- ★ Tasklets can also be dynamically created and/or removed.

workqueues

- ★ **Workqueues** provide a complete different mechanism for deferred work:
  - Workqueues run in their own thread, not as an interrupt handler (not softirq)
  - Can be scheduled by the scheduler with other threads (better management)
  - Associated with a Thread Control Block (TCB) and can save state/context.

MP1

★ For MP1, you'll work with designing your first kernel module and will use a Two-Halves design for handling a timer interrupt!

★ Released with Week #3 (Feb. 8, 2021)
  - TA Overview Session on Thursday, Feb. 11.