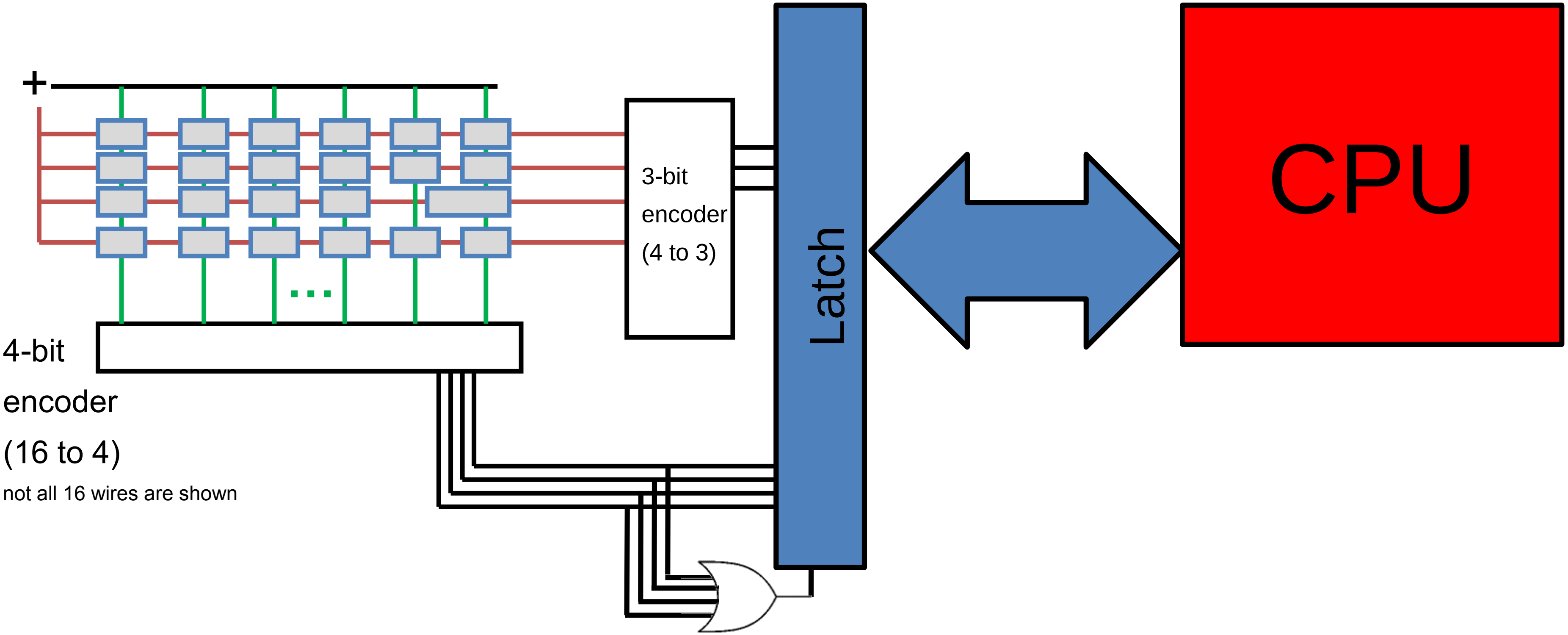
Architectural Support for Operating Systems

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]

Keyboard Design Again

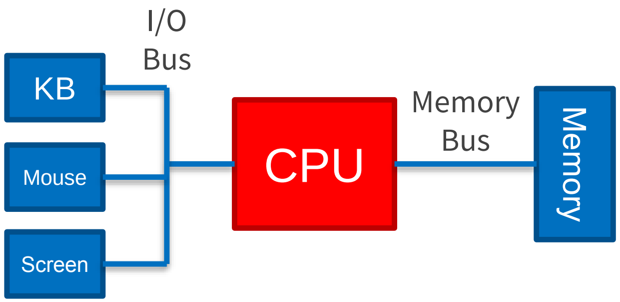


How and **when** does CPU read keycode?

Device Interfacing Techniques

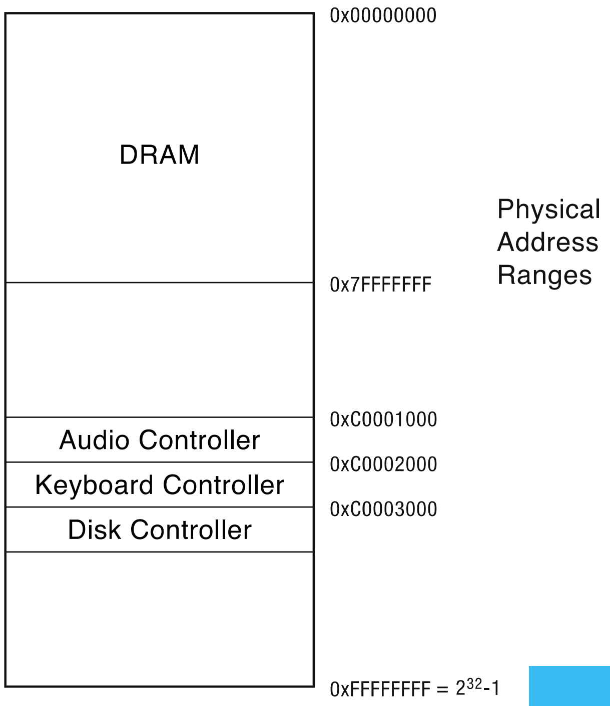
Programmed I/O

- CPU has dedicated, special instructions
- CPU has additional wires (I/O bus)
- Instruction specifies device and operation



Memory-mapped I/O

- Device communication goes over memory bus
- Reads/Writes to special addresses converted into I/O operations by dedicated device hardware
- Each device appears as if it is part of the memory address space
- **Predominant device interfacing technique**

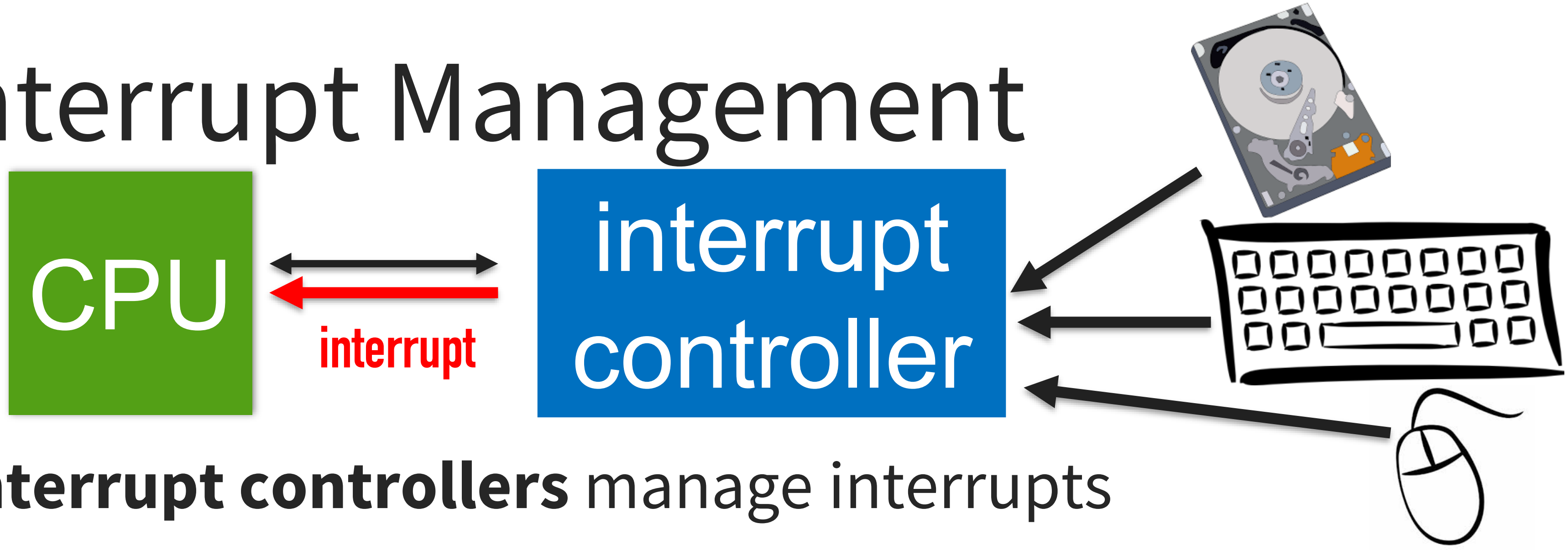


Polling vs. Interrupts

- First idea: CPU constantly reads the keyboard latch memory location to see if a key is pressed
  = Polling
  - Inefficient

- Alternative: add extra circuitry so keyboard can alert CPU when there is a keypress
  = interrupt driven I/O
  → CPU and devices can perform tasks concurrently, increasing throughput
  - Only need a bit of circuitry + a few extra wires to implement "alert" operation

Interrupt Management



CPU

interrupt

interrupt controller

Interrupt controllers manage interrupts

- Interrupts have descriptor of interrupting device
- Priority selector circuit examines all interrupting devices, reports highest level to the CPU
- Interrupt controller implements interrupt priorities

Interrupts can be **maskable** (can be turned off by the CPU for critical processing) or **nonmaskable** (signifies serious errors like power out warning, unrecoverable memory error, *etc.*)

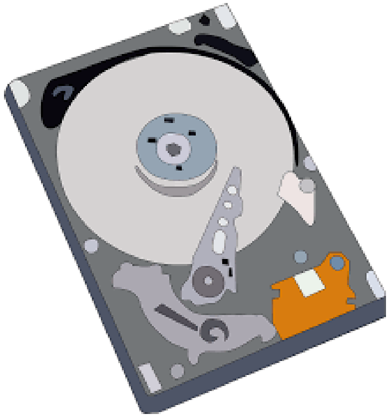
I/O Summary

Interrupt-driven operation with memory-mapped I/O:

- CPU initiates device operation (*e.g.*, read from disk): writes an operation descriptor to a designated memory location
- CPU continues its regular computation
- The device asynchronously performs the operation
- When the operation is complete, interrupts the CPU

What about bulk data transfers?

- One interrupt for each byte read!

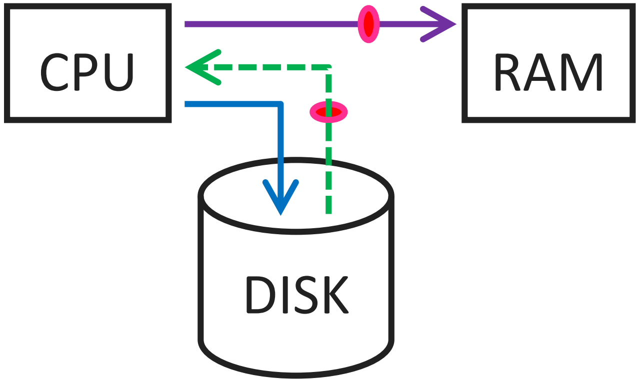


Direct Memory Access (DMA)

Interrupt-Driven I/O: Device ←→ CPU ←→ RAM

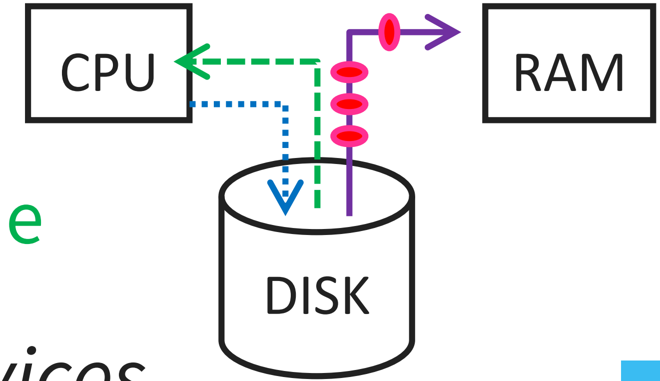
for (i = 1 .. n)

- CPU issues read request
- Device interrupts CPU with data
- CPU writes data to memory



+ Direct Memory Access: Device ←→ RAM

- CPU sets up DMA request
- for (i = 1 ... n)
  Device puts data on bus
  & RAM accepts it
- Device interrupts CPU after done



Critical for high-performance devices

Still More to Do

- CPU can talk to devices, now what?

Remaining problems:

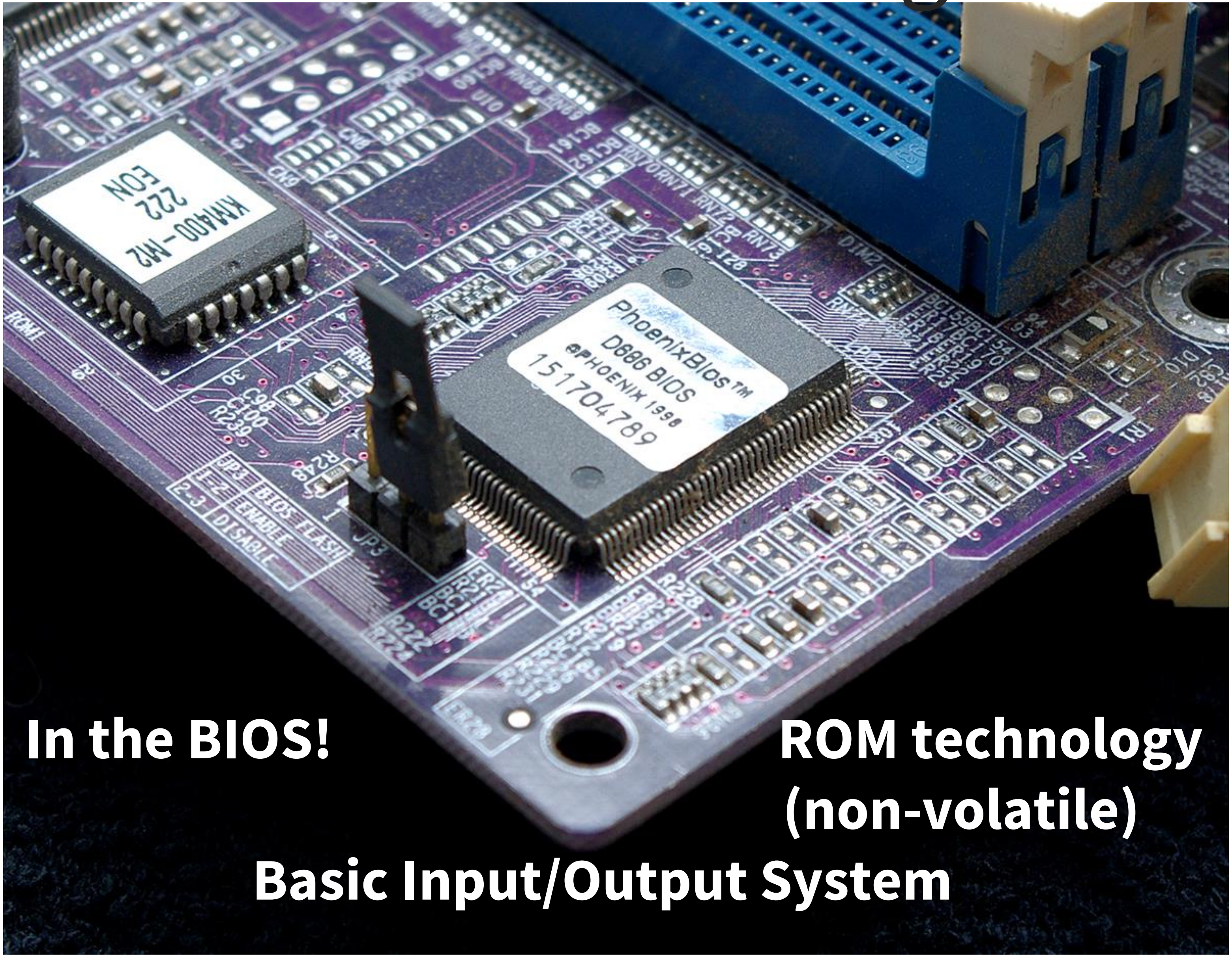
- What to do while waiting for I/O?
  → Run another program
- How to decide which program to run?
- How do multiple programs share devices?
  - Interrupt design assumes there's only one program

Enter the OS

- Manages shared hardware
- Isolates programs from each other

Let’s start at the very beginning

Where ~~When~~ does life begin?



In the BIOS!

ROM technology (non-volatile)

Basic Input/Output System

http://www.partesdeunacomputadora.net/motherboard/que-es-la-bio

On System Start Up

- BIOS copies bootloader into memory
- Bootloader copies OS kernel into memory
- Kernel:
  - Initializes data structures (devices, core map, interrupt vector table, *etc.*)
  - Copies first process from disk
  - Change privilege mode & PC

PC has code from:

time

privilege mode: **0** **0** **0** **1**

One Brain, Many Personalities

https://www.theodysseyonline.com/are-our-apps-wasting-our-time

Supporting dual mode operation

1. Privilege mode bit (0=kernel, 1=user)
   Where? x86 → EFLAGS reg., MIPS →status reg.
2. Privileged instructions
   user mode → no way to execute unsafe insns
3. Memory protection
   user mode → memory accesses outside a process' memory region are prohibited
4. Timer interrupts
   kernel must be able to periodically regain control from running process
5. Efficient mechanism for switching modes
   must be fast because it happens a lot!

Privilege Mode Bit

- Some processor functionality cannot be made accessible to untrusted user apps
- Must differentiate user apps vs. OS code

Solution: **Privilege mode bit** indicates if current program can perform privileged operations

0 = Trusted = OS

1 = Untrusted = user

Privileged Instructions

Examples:

- changing the privilege mode
- writing to certain registers (page table base reg)
- enabling a co-processor
- changing memory access permissions
- signal other users' processes
- print character to screen
- send a packet on the network
- allocate a new page in memory

achieved via **system call**

CPU knows which instructions are privileged:

`insn==privileged && mode==1` → *Exception!*

Memory Protection

Step 1: **Virtualize Memory**

- Virtual address space: set of memory addresses that process can "touch" (CPU works with virtual addresses)
- Physical address space: set of memory addresses supported by hardware

Step 2: **Address Translation**

- function mapping *<pid, vAddr>* → *<pAddr>*

Sit tight. We'll talk all about this later.

Supporting dual mode operation

1. Privilege bit ✓
2. Privileged instructions ✓
3. Memory protection ✓
4. Timer interrupts
5. Efficient mechanism for switching modes

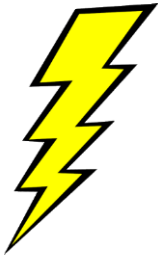
Interrupts

Timer Interrupts:

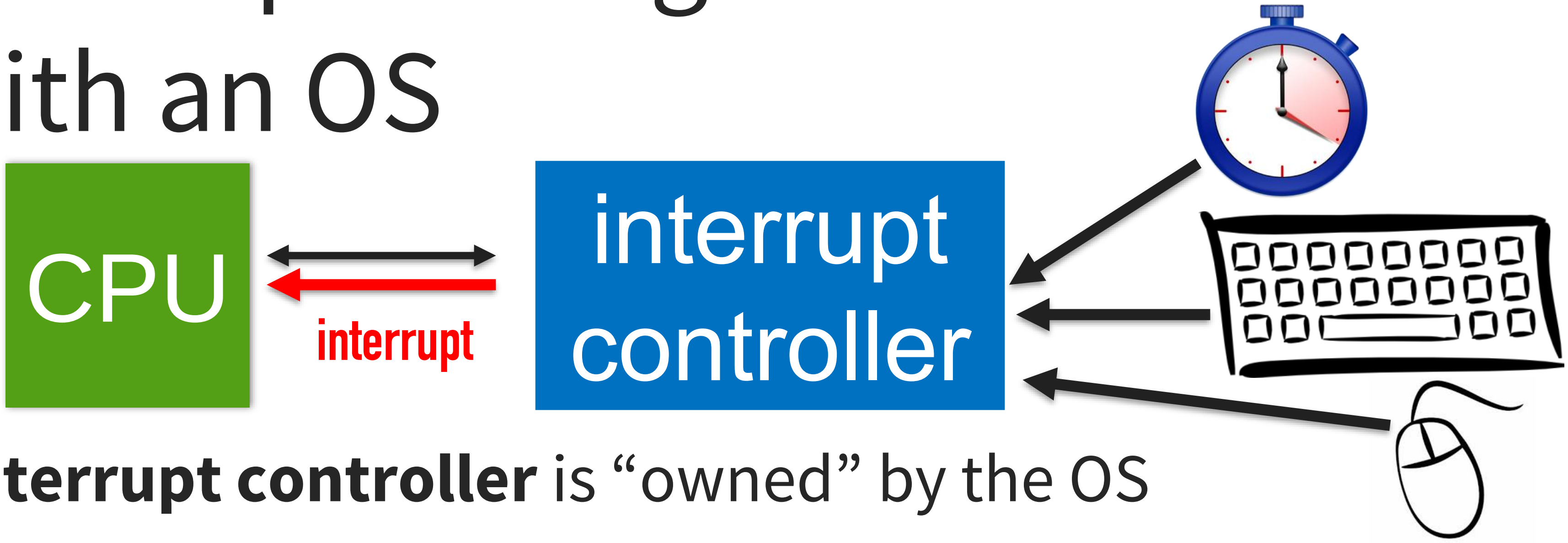
- Hardware timer set to expire after specified delay (time or instructions)
- Time's up? Control passes back to kernel.

More Generally: Hardware Interrupts

- External Event has happened – like device I/O
- OS needs to check it out.
- Process stops what it's doing, invokes OS, which handles the interrupt.



Interrupt Management with an OS



CPU

interrupt

interrupt controller

Interrupt controller is "owned" by the OS

- All interrupts are handled by kernel code
- Registering an interrupt handler is privileged instr

A **timer interrupt** is just another device, ensures OS gets control back at regular intervals via interrupt handler

Supporting dual mode operation

1. Privilege bit ✓
2. Privileged instructions ✓
3. Memory protection ✓
4. Timer interrupts ✓
5. Efficient mechanism for switching modes

From User to Kernel

Exceptions

- Synchronous
- User program mis-steps (*e.g.*, div-by-zero)
- Attempt to perform privileged insn
    - on purpose? breakpoints!

System Calls

- Synchronous
- User program requests OS service

Interrupts

- Asynchronous
- HW device requires OS service
    - timer, I/O device, interprocessor

From Kernel to User

Resume *P* after exception, interrupt or syscall

- Restore PC SP, registers
- Restore mode

Switch to different process *Q*

- Load PC, SP, and registers from *Q*’s PCB
- Toggle mode

If new process

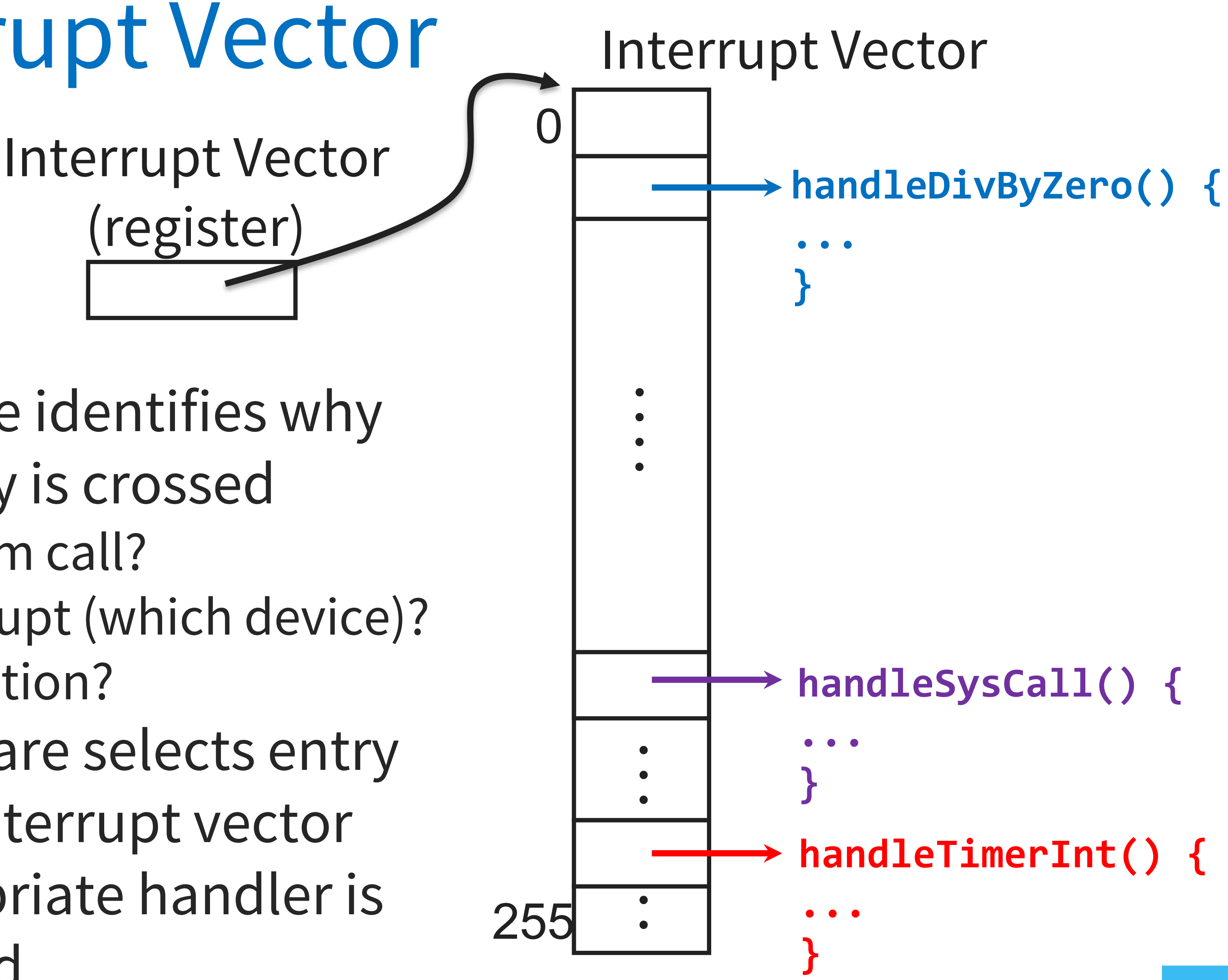
- Copy in program memory
- Set PC & SP
- Toggle mode

Safely switching modes

Common sequences of instructions to cross boundary, which provide:

- **Limited entry**
    - entry point in the kernel set up by kernel
- **Atomic changes to process state**
    - PC, SP, memory protection, mode
- **Transparent restartable execution**
    - user program must be restarted exactly as it was before kernel got control

Interrupt Vector



```
handleDivByZero() {
...
}
```

```
handleSysCall() {
...
}
```

```
handleTimerInt() {
...
}
```

Hardware identifies why boundary is crossed

- System call?
- interrupt (which device)?
- exception?

- Hardware selects entry from interrupt vector
- Appropriate handler is invoked

Interrupt Stack

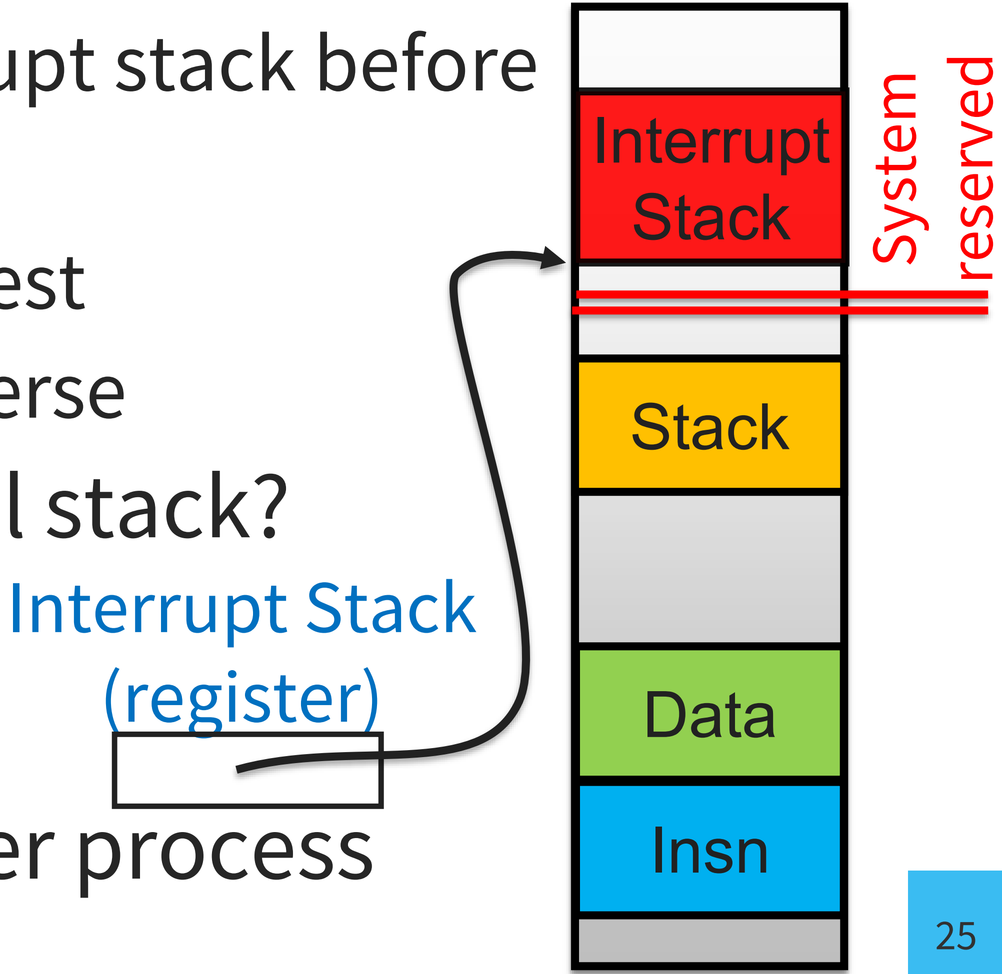
Privileged hw reg. points to Interrupt Stack

- on switch, hw pushes some process registers (SP, PC, …) on interrupt stack before handler runs. (Why?)
- handler pushes the rest
- on return, do the reverse

Why not use user-level stack?

- reliability
- Security

One interrupt stack per process



Complete Mode Transfer

Hardware transfer to kernel:

1. save privilege mode, set mode to 0
2. mask interrupts
3. save: SP, PC
4. switches SP to the kernel stack
5. save values from #3 onto kernel stack
6. save error code
7. set PC to the interrupt vector table

Interrupt handler

1. saves all registers
2. examines the cause
3. performs operation required
4. restores all registers

Performs "Return from Interrupt" insn (maybe)

- restores the privilege mode, SP and PC

Kernel Operation (conceptual, simplified)

1. Initialize devices
2. Initialize “first process”
3. `while (TRUE) {`
   - while device interrupts pending
     - handle device interrupts
   - while system calls pending
     - handle system calls
   - if run queue is non-empty
     - select a runnable process and switch to it
   - otherwise
     - wait for device interrupt

`}`

CPU Scheduling Lecture

Supporting dual mode operation

1. Privilege bit ✓
2. Privileged instructions ✓
3. Memory protection ✓
4. Timer interrupts ✓
5. Efficient mechanism for switching modes ✓

Made possible (and fast) by hardware!