Architectural Support for Operating Systems
(Chapter 2)

CS 4410
Operating Systems

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

Let’s start at the very beginning

A Short History of Operating Systems

History of Operating Systems

Phase 1: Hardware expensive, humans cheap

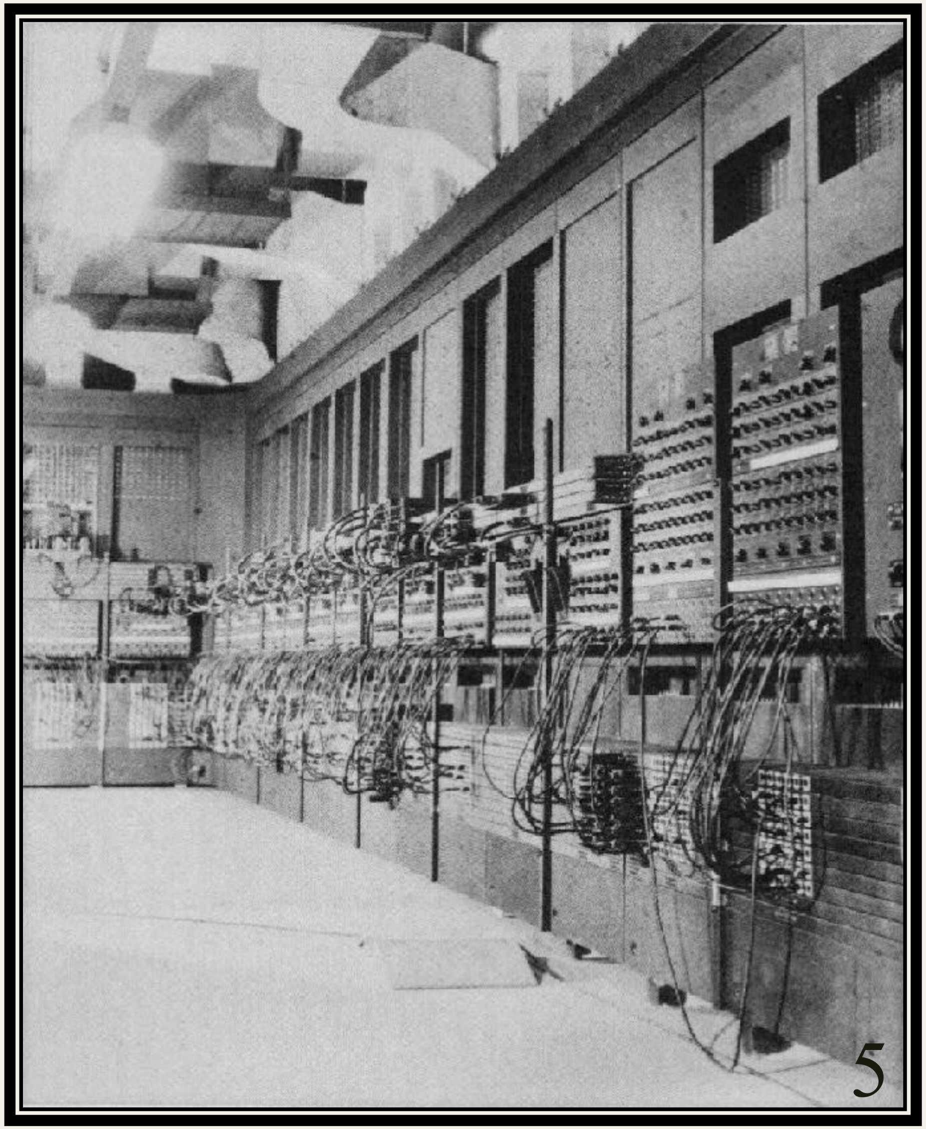
- *User at console: single-user systems*
- *Batching systems*
- *Multi-programming systems*

Hand programmed machines (1945-1955)

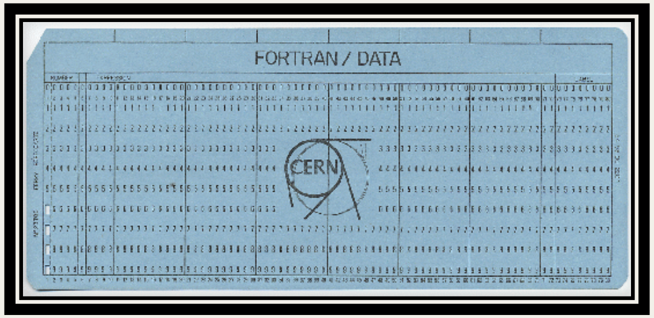
- Single user systems
- OS =

  loader + libraries
- Problem:

  low utilization of expensive components



Batch Processing (1955-1965)



- OS = loader +
  sequencer +
  output processor

Input

Card Reader

Tape

Compute

User Data

User Program

"System Software"

Operating System

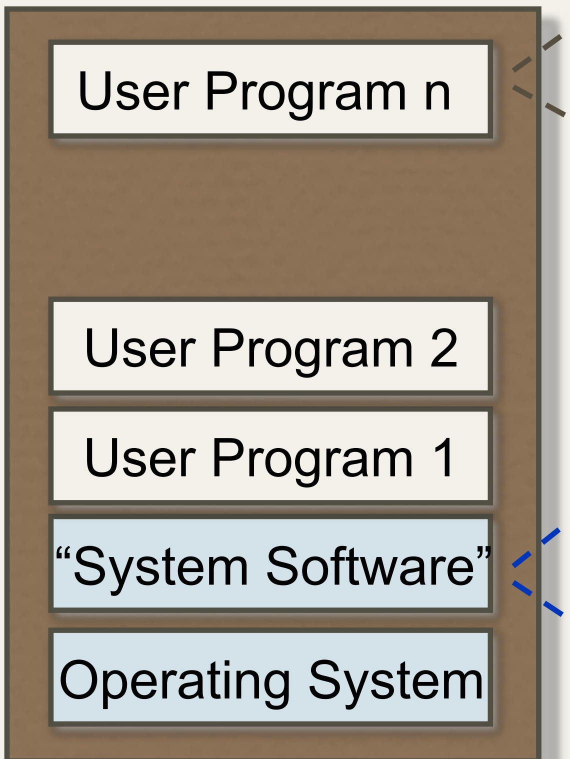
Tape

Printer

Output

Multiprogramming (1965-1980)

- Keep several jobs in memory
- Multiplex CPU between jobs.

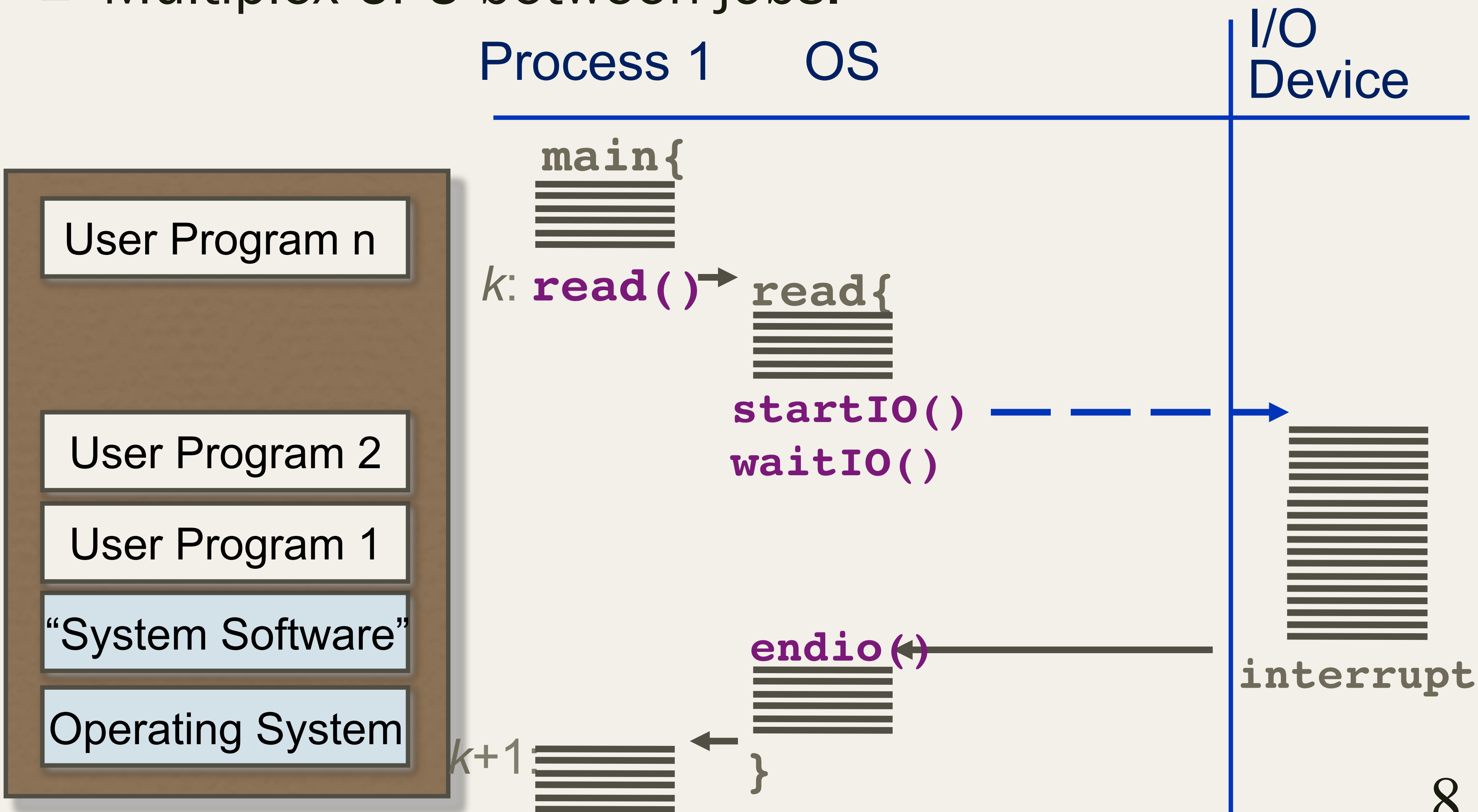



```
program P
begin
   ...
   Read(var)
   ...
end P
```

```
system call Read(var)
begin
   StartIO(input device)
   WaitIO(interrupt)
   EndIO(input device)
   ...
end Read
```

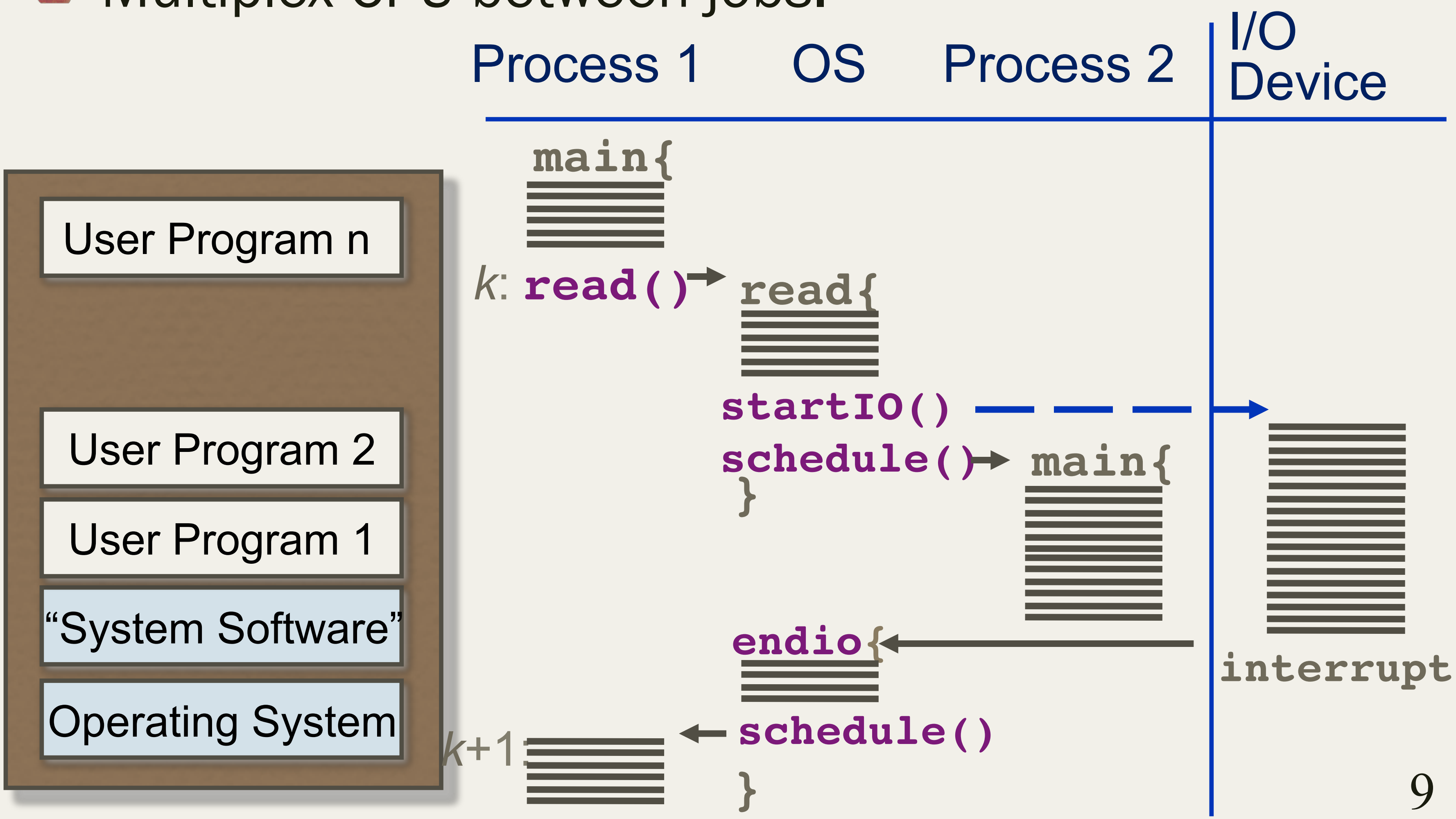
Multiprogramming (1965-1980)

- Keep several jobs in memory
- Multiplex CPU between jobs.



Multiprogramming (1965-1980)

- Keep several jobs in memory
- Multiplex CPU between jobs.



History of Operating Systems

Phase 1: Hardware expensive, humans cheap

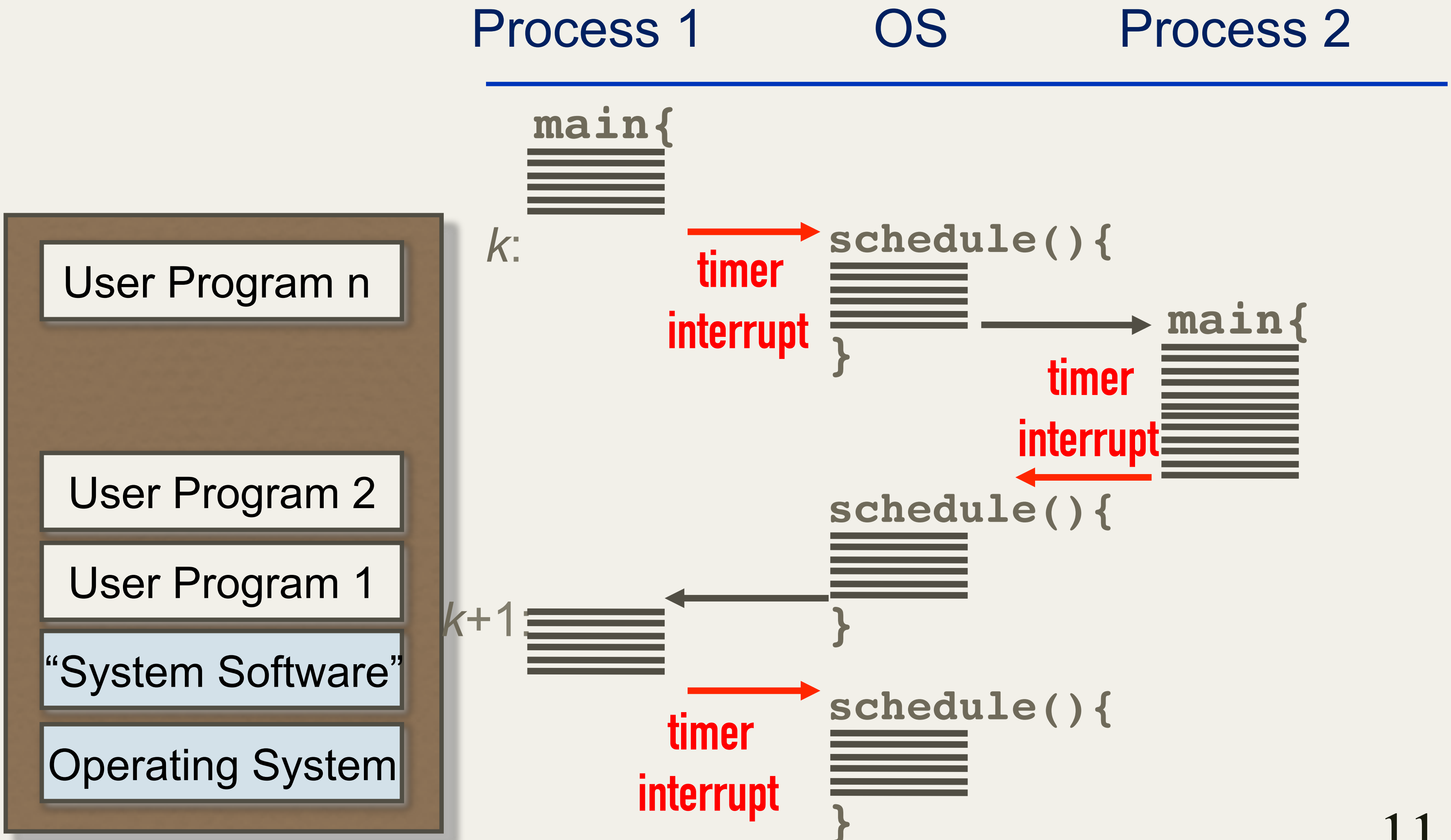
- *User at console: single-user systems*
- *Batching systems*
- *Multi-programming systems*

Phase 2: Hardware cheap humans expensive

- *User at console: single-user systems*

Timeshareing (1970-)

- Timer interrupt used to multiplex CPU between jobs



History of Operating Systems

Phase 1: Hardware expensive, humans cheap

- *User at console: single-user systems*
- *Batching systems*
- *Multi-programming systems*

Phase 2: Hardware cheap humans expensive

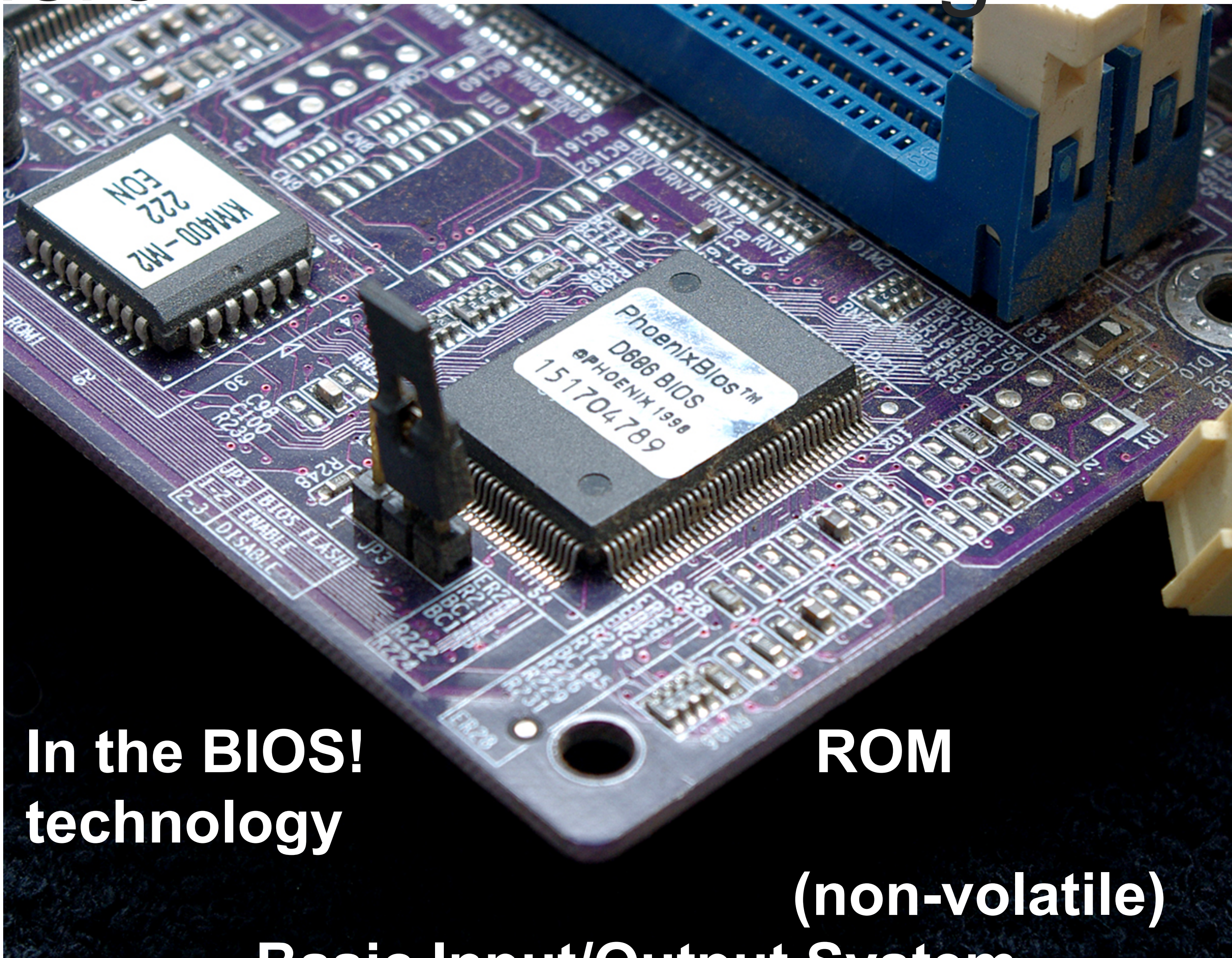
- *User at console: single-user systems*

Phase 3: H/W **very** cheap humans **very** expensive

- *Personal computing: One system per user*
- *Distributed computing: many systems per user*
- *Ubiquitous computing: LOTS of systems per users*

THE END

Where ~~When~~ does life begin?



In the BIOS! **ROM**

technology

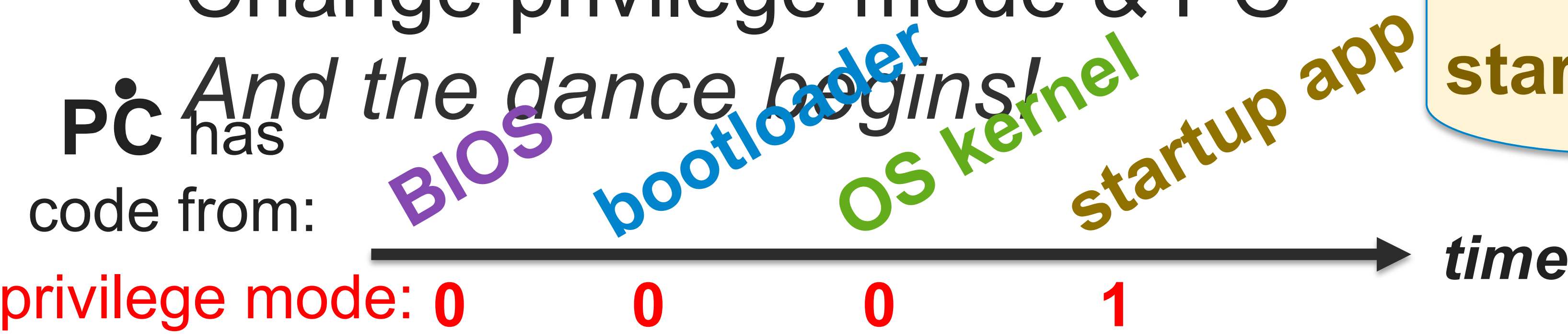
(non-volatile)

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
  - *And the dance begins!*



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

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

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**

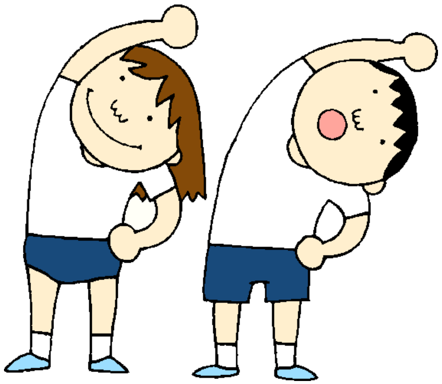
- Virtual address space: set of memory addresses that process can "touch" (CPU works with virtual addresses)
- Physical address space: set of memory addresses supported by hardware

Step 2: **Address Translation**

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

Supporting dual mode operation

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



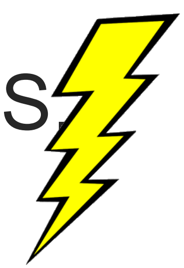
Interrupts

Timer Interrupts:

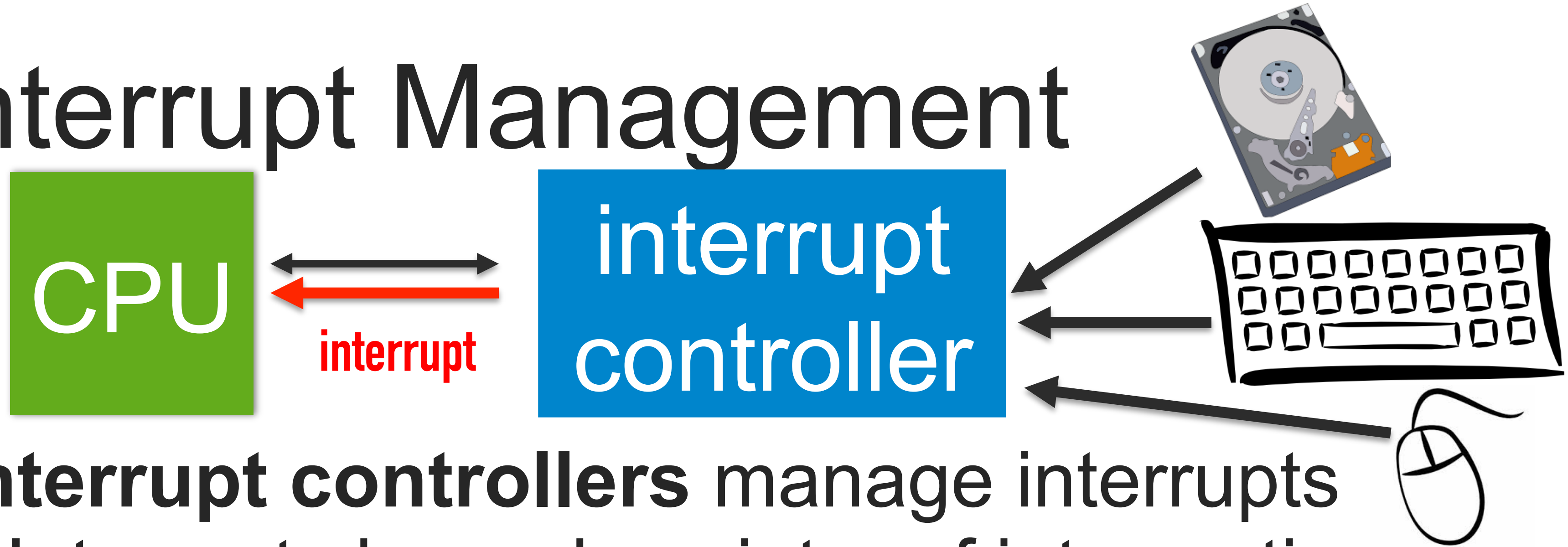
- Hardware timer set to expire after specified delay (time or instructions)
- Time’s up? Control passes back to kernel.

More Generally: Hardware Interrupts

- External Event has happened.
- OS needs to check it out.
- Process stops what it’s doing, invokes OS, which handles the interrupt.



Interrupt Management

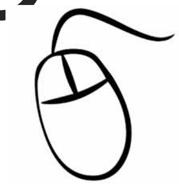


Interrupt controllers manage interrupts

- 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

Aside 1: Interrupt Driven I/O

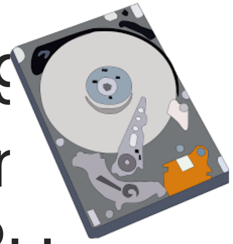


Memory-mapped I/O

- Device communication goes over the memory bus
- I/O operations by dedicated device hardware correspond to reads/writes to special addresses
- Devices appear as if part of the memory address space

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 (see slide 9
- The device asynchronously performs the operation
- When the operation is complete, interrupts the CPU
- Could happen for each byte read!

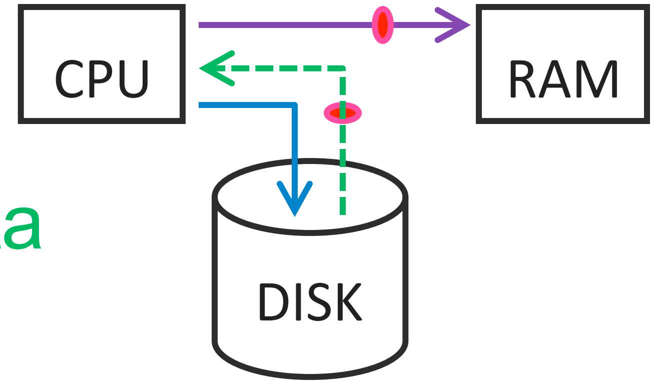


Aside 2: Direct Memory Access (DMA)

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

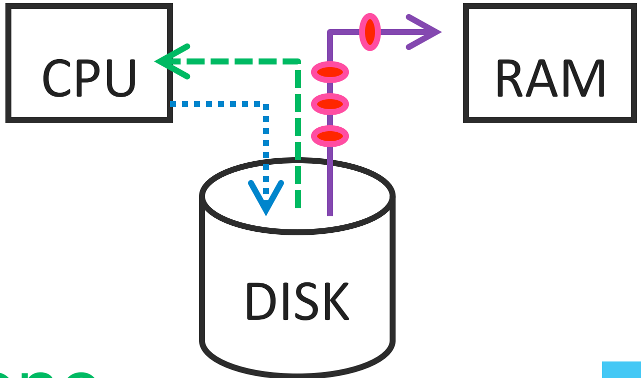
for (i = 1 .. n)

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



+ Direct Memory Access (DMA): Device ←→ RAM

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



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

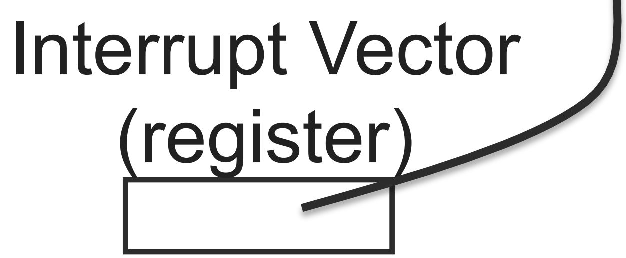
- Copy in program memory
- Set PC & SP
- Toggle mode

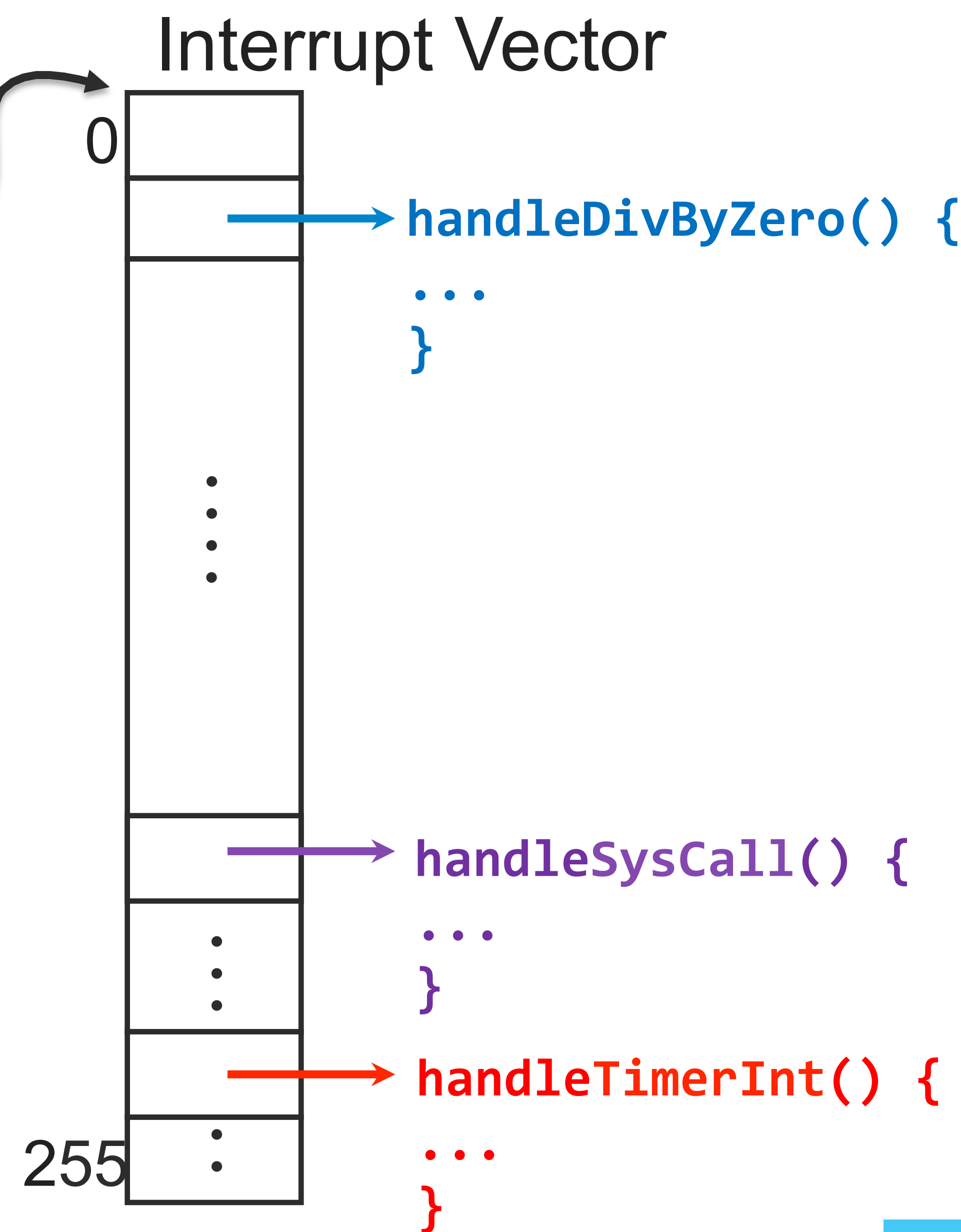
Safely switching modes

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





Hardware identifies why boundary is crossed

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

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

Interrupt Stack

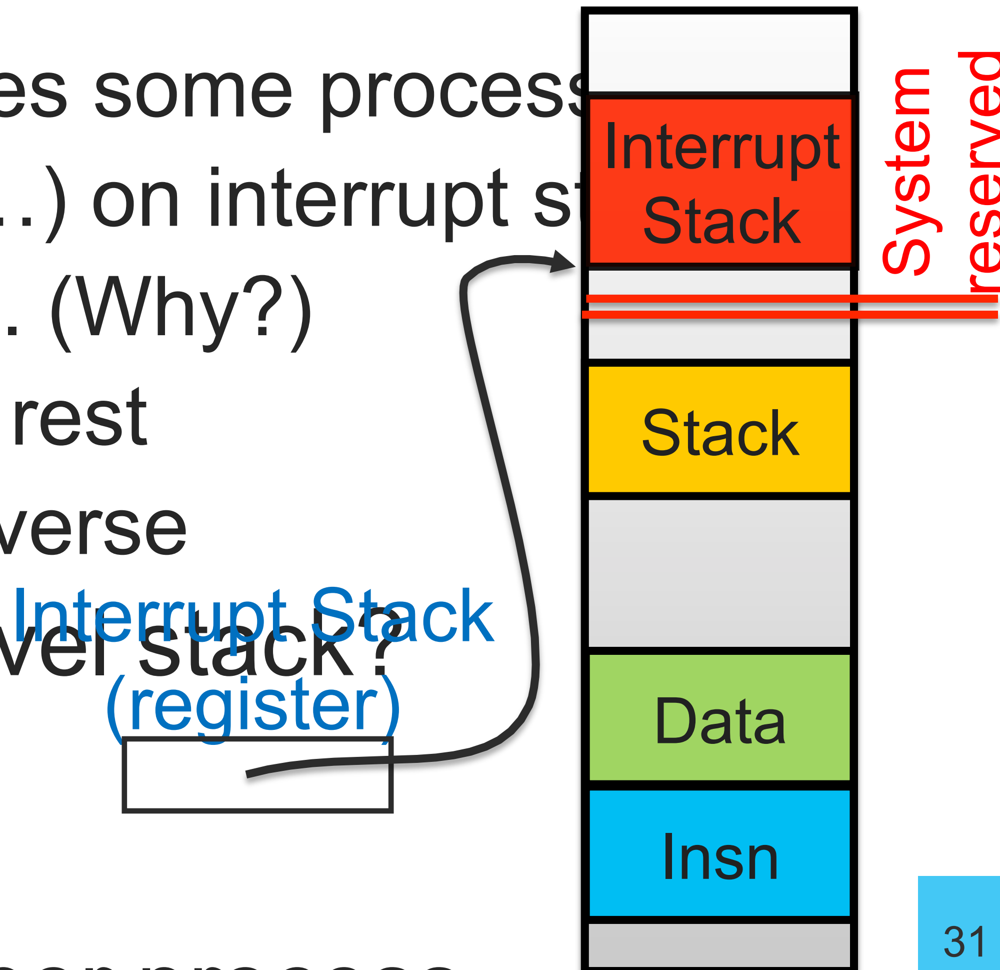
Privileged hw reg. points to Interrupt Stack

- 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!