Managing and Scheduling Processes

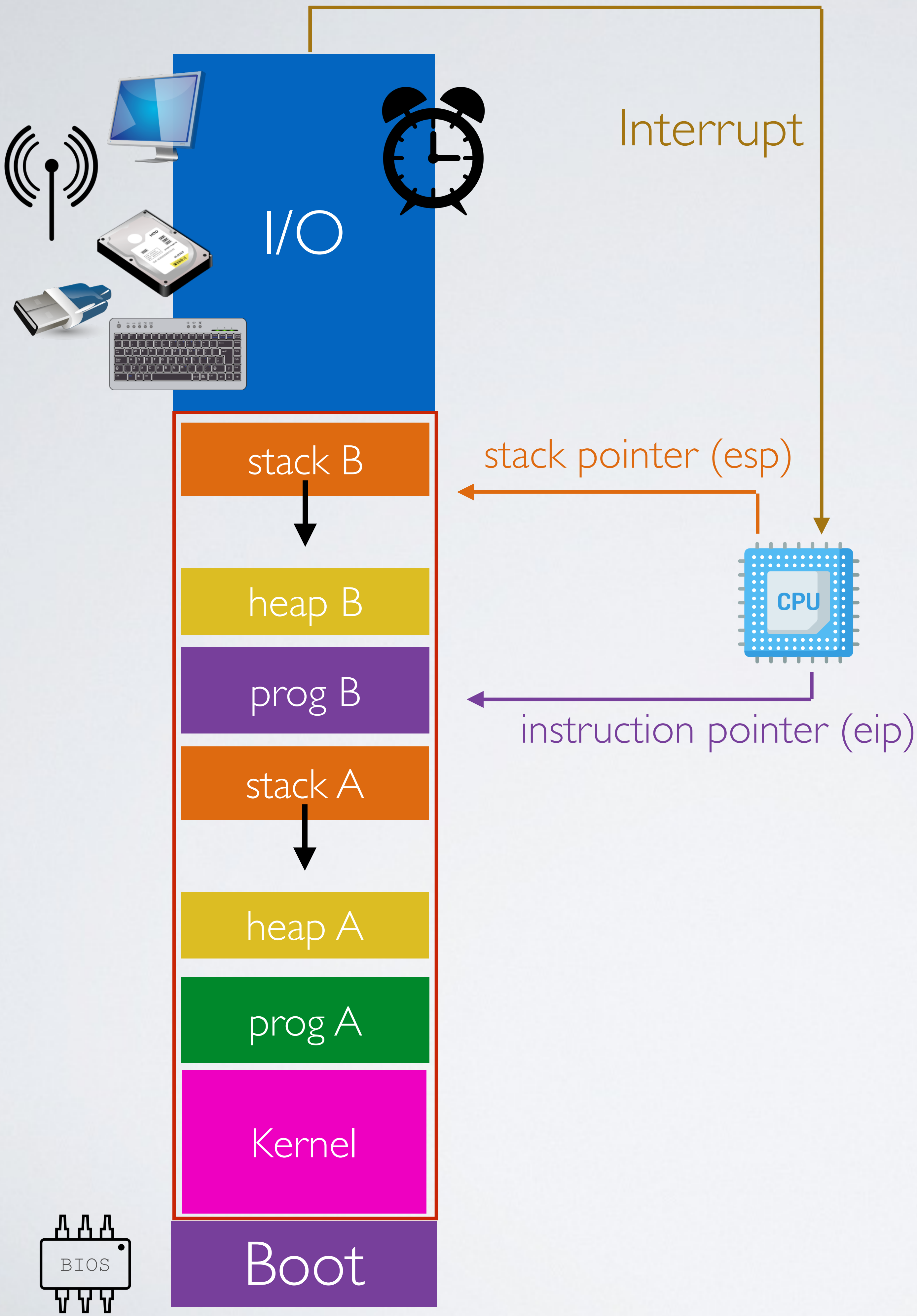
Thierry Sans

Program vs Process

- **Program** : static data on some storage
- **Process** : instance of a program execution

➡ Several process of the same program can run concurrently

The architecture



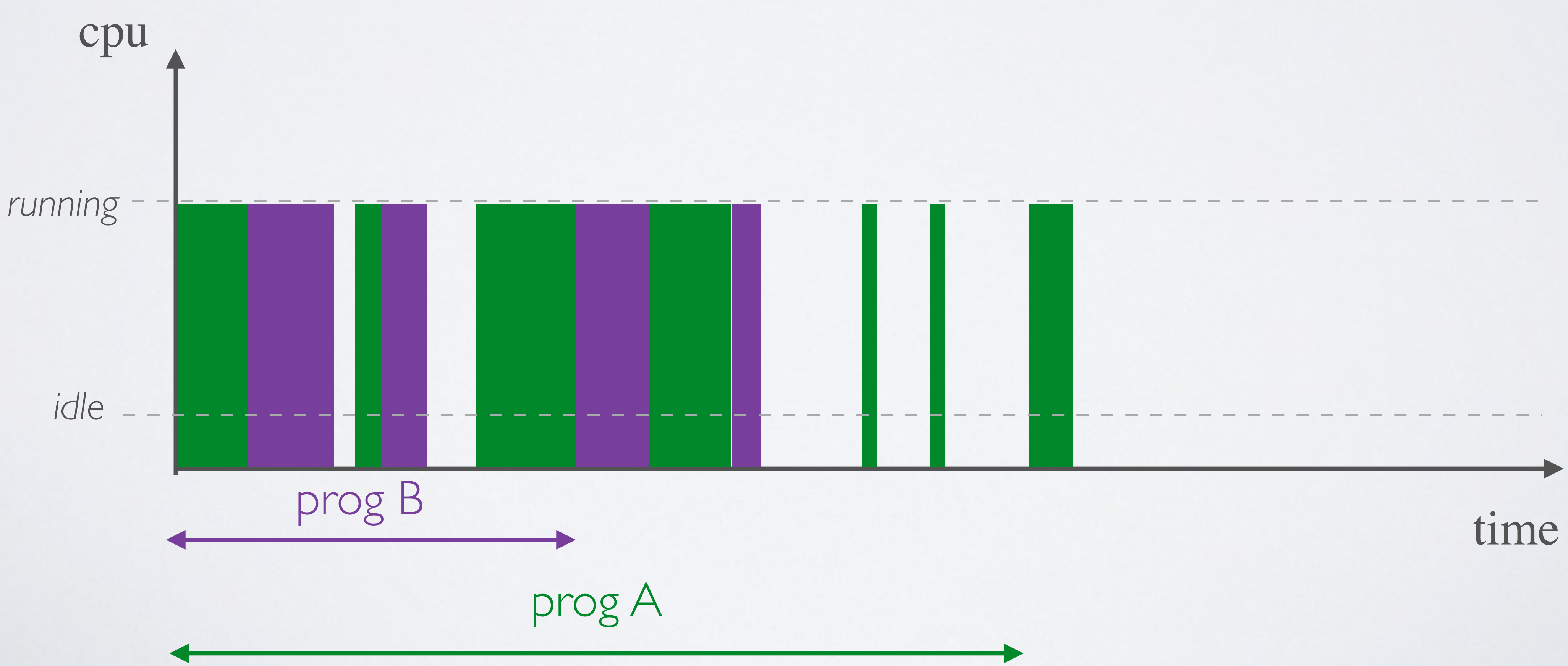
Running processes concurrently

A CPU core will run multiple processes concurrently by running each process for a little amount of time before switching to another one

➡ **Limited Direct Execution**

The CPU will switch to another process when either

- the running process runs out of time slice (system clock interrupt)
- or the running process initiates an I/O that will take some time



The advantages of concurrency

- ✓ **From the system perspective**
  better CPU usage resulting in a faster execution overall (but not individually)

- ✓ **From the user perspective**
  programs seem to be executed in parallel

- ➡ It requires **some mechanisms to manage and schedule** these concurrent processes

Today's lecture

1. **Interrupt Handling**
   How to handle events such as I/O and exceptions?

2. **Context switching**
   How to you switch the running process?

3. **The Process API**
   How do you create and terminate processes?

4. **Scheduling**
   How to choose which process to run among all the ready ones?

1. Managing Interrupts

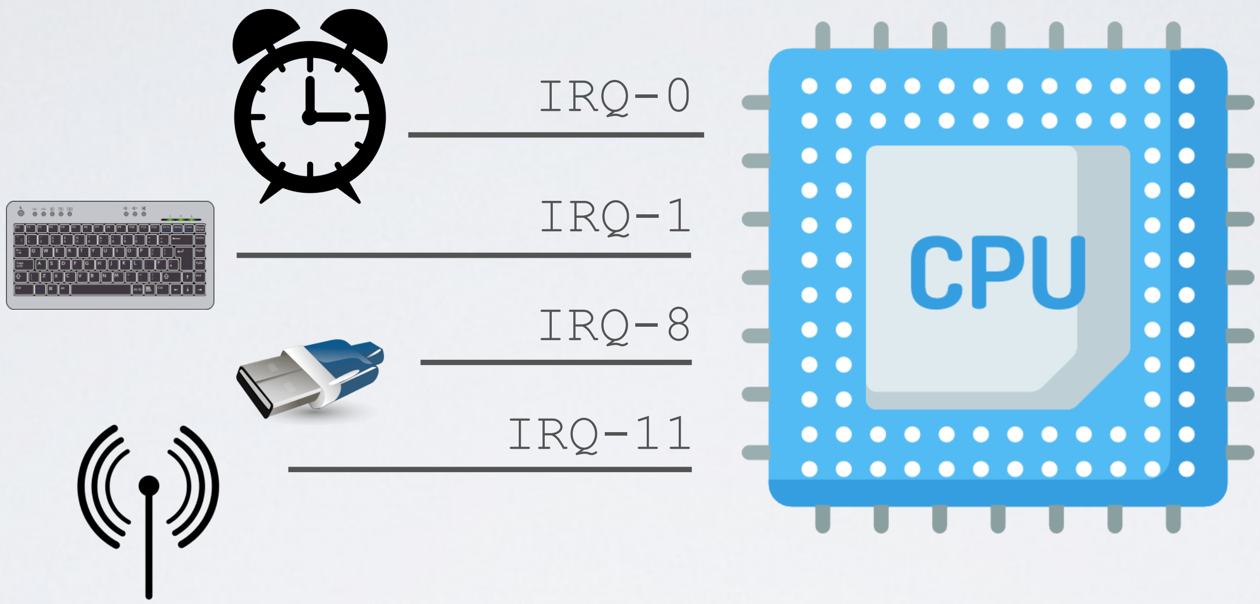
Two kinds of interrupts

Hardware interrupts (asynchronous)
caused by an I/O device that needs some attention

Software Interrupts a.k.a exceptions (synchronous)
caused by executing instructions

- fault
  e.g divide by zero
  e.g page fault (coming later with memory management)
- trap - x86 `int` instruction (intended by the programmer)
  e.g `int $0x80` for Linux system call trap
  e.g `int $0x30` for Pintos system call trap

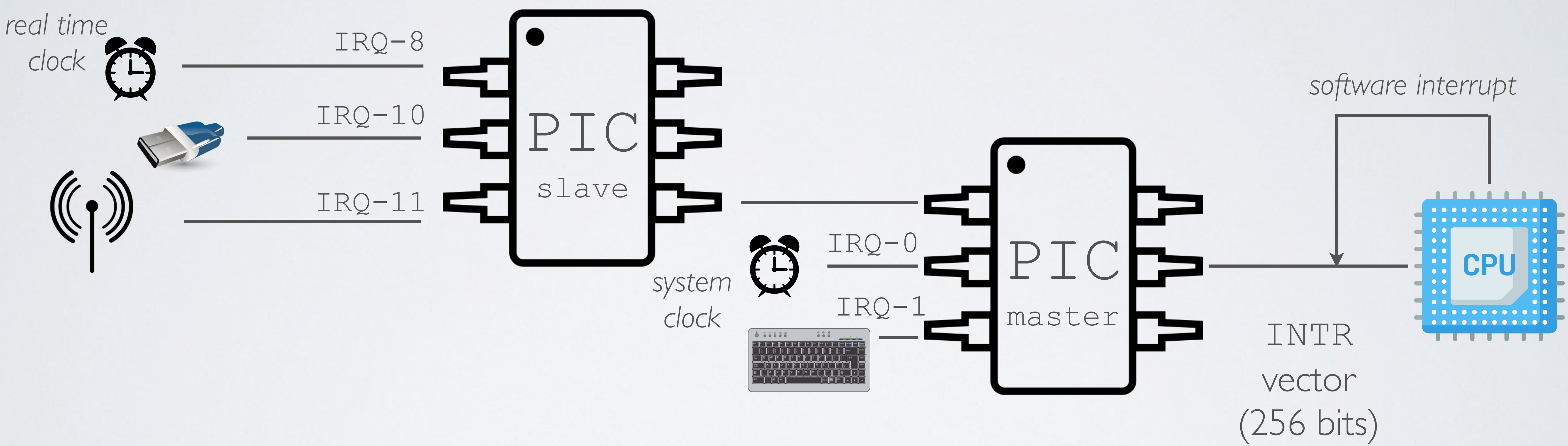
Hardware Interrupt - the naive implementation



➡ I/O devices are wired to **Interrupt Request lines** (IRQs)

- Not flexible (hardwired)
- CPU might get interrupted all the time
- How to handle interrupt priority

Hardware Interrupt and Software Interrupt - the real implementation



➡ I/O devices have unique or shared IRQs that are managed by two **Programmable Interrupt Controllers** (PIC)

Programmable Interrupt Controllers (PIC)

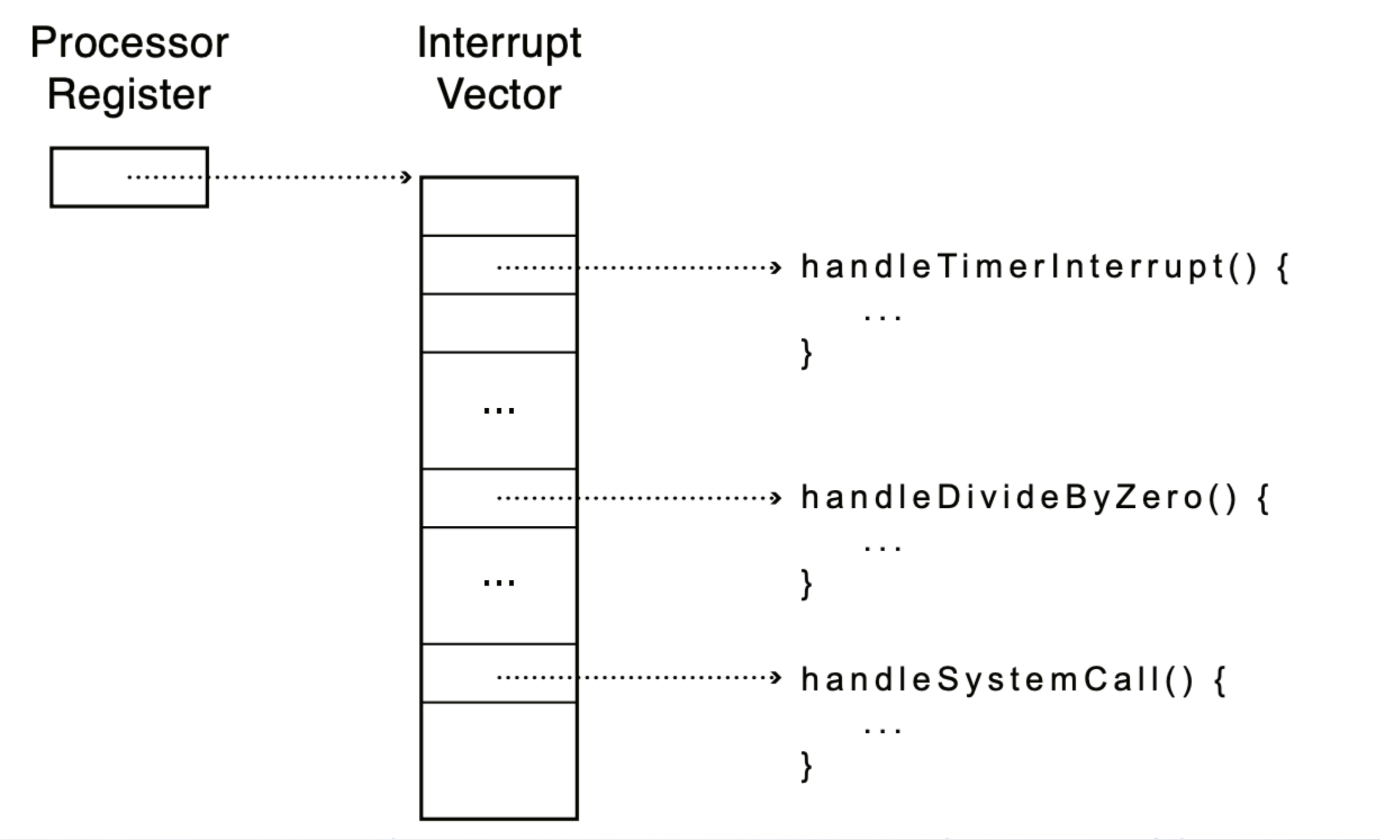
➡ Responsible to tell CPU when and which devices wishes to interrupt through the INTR vector

✓ 16 lines of interrupt (IRQ0 - IRQ15)

✓ Interrupts have different priority

✓ Interrupts can be masked

Handling an interrupt



1. The CPU receives an interrupt on the INTR vector
2. The CPU stops the running program and transfer control to the corresponding handler in the Interrupt Descriptor Table (IDT)
3. The handler saves the current running program state
4. The handler executes the functionality
5. The handler restores (or halt) the running program

Where are these interrupt handlers defined

- **Linux**
  `cat /proc/interrupt`

- **Windows**
  `msinfo32.exe`

- **Pintos**
  `see src/threads/interrupt.c`

Example

When a key is pressed…

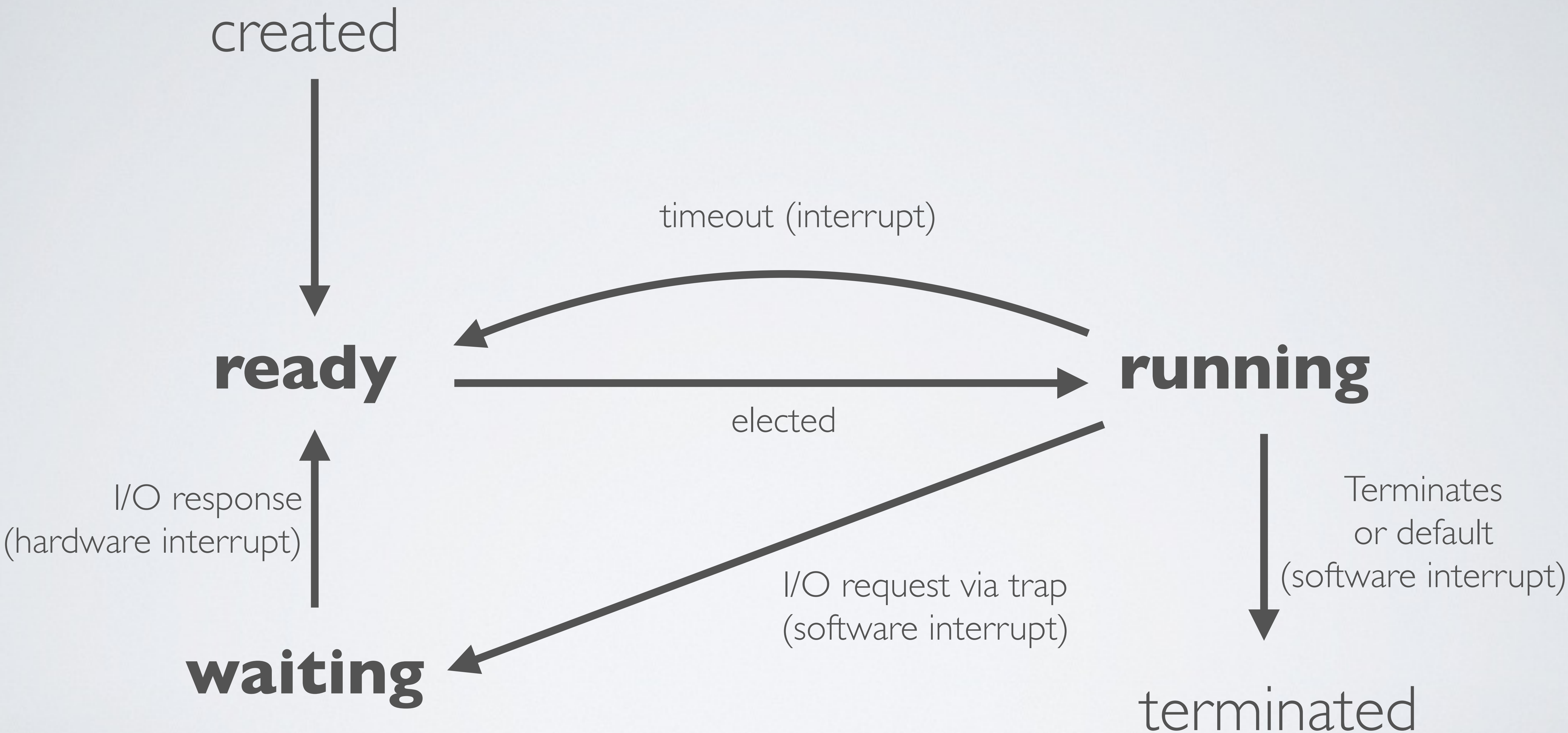
1. the keyboard controller tells PIC to cause an interrupt on IRQ #1
2. the PIC, which decides if CPU should be notified
3. If so, IRQ 1 is translated into a vector number to index into CPU's Interrupt Descriptor Table
4. The CPU stop the current running program
5. The CPU invoke the current handler
6. The handler talks to the keyboard controller via IN and OUT instructions to ask what key was pressed
7. The handler does something with the result (e.g write to a file in Linux)
8. The handler restores the running program

2. Context Switching

When the CPU runs processes concurrently

- Only one process at a time is **running** (on one core)
- Several processes might be **waiting** for an I/O response
- Several processes might be **ready** to be executed

The different states of a process



Context switching when

When the OS receives a fault

1. suspends the execution of the running process
2. terminate the process

When the OS receives a System Clock Interrupt or a System Call Trap (I/O request)

3. suspends the execution of the running process
4. saves its execution context
5. changes the process's state to ready (timeout) or waiting (I/O request)
6. elects a new process from the ones in the ready state
7. changes its state to running
8. restores its execution context
9. resumes its execution

When the OS receives any other I/O interrupt

1. executes the I/O operation
2. switches the process, that was waiting for that I/O operation, into the ready state
3. resumes the execution of the current program

➡ **For each process, the OS needs to keep track of its state (ready, running, waiting) and its execution context (registers, stack, heap and so on)**

Process Control Block

PCB (Process Control Block) - data structure to record process information

- Pid (process id) and ppid (parent process)
- State (as either running, ready, waiting)
- Registers (including eip and esp)
- User (forthcoming lecture on user space)
- Address space (forthcoming lecture on memory management)
- Open files (coming next with filesystem)
- Others

State Queues

- The OS maintains a collection of queues with the PCBs of all processes
  - One queue for the processes in the ready state
  - Multiple queues for the processes in the waiting state (one queue for each type of I/O request)

3. The Process API

From the system programmer's perspective

- Create and terminate
- Communicate
- Get information
- Control process (stop and resume)

Create a process

- ➡ A process is created by another process (concept of parent process and child process)

Process creation on Unix using `fork`

`int fork()`

1. Creates and initializes a new PCB
2. Creates a new address space
3. Initializes the address space with a copy of the entire contents of the address space of the parent (with one exception)
4. Initializes the kernel resources to point to the resources used by parent (e.g., open files)
5. Places the PCB on the ready queue

Why `fork` and `exec`?

`fork` is very useful when the child…

- is cooperating with the parent
- relies upon the parent's data to accomplish its task

➡ Simple interface

Example : a web server

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    // Handle client request
  } else {
    // Close socket
  }
}
```

Process creation on Unix using `exec`

```
int exec(char *prog, char *argv[])
```

1. Stops the current process
2. Loads the program "prog" into the process' address space
3. Initializes hardware context and args for the new program
4. Places the PCB onto the ready queue

➡ **Actually, `exec` does not create a new process**

Spawning

✓ Most calls to `fork` are followed by `exec` (a.k.a `spawn`)

- `minish.sh`
- `redirsh.c`
- `pipesh.c`

Argument against `fork`

"A fork() in the road"

Andrew Baumann (Microsoft Research), Jonathan Appavoo, Orran Krieger (Boston University), Timothy Roscoe (ETH Zurich) - *In Proceedings of HotOS 2019*

https://www.microsoft.com/en-us/research/uploads/prod/2019/04/fork-hotos19.pdf

➡ The main argument is security

Process creation on Windows

`CreateProcess: BOOL CreateProcess(char *prog, char *args)`

1. Creates and initializes a new PCB
2. Creates and initializes a new address space
3. Loads the program specified by "prog" into the address space
4. Copies "args" into memory allocated in address space
5. Initializes the saved hardware context to start execution at main (or wherever specified in the file)
6. Places the PCB on the ready queue

Wait for a process

Unix : `wait(int *wstatus)`

Windows : `WaitForSingleObject`

Terminate a process

Unix : `exit(int status)`

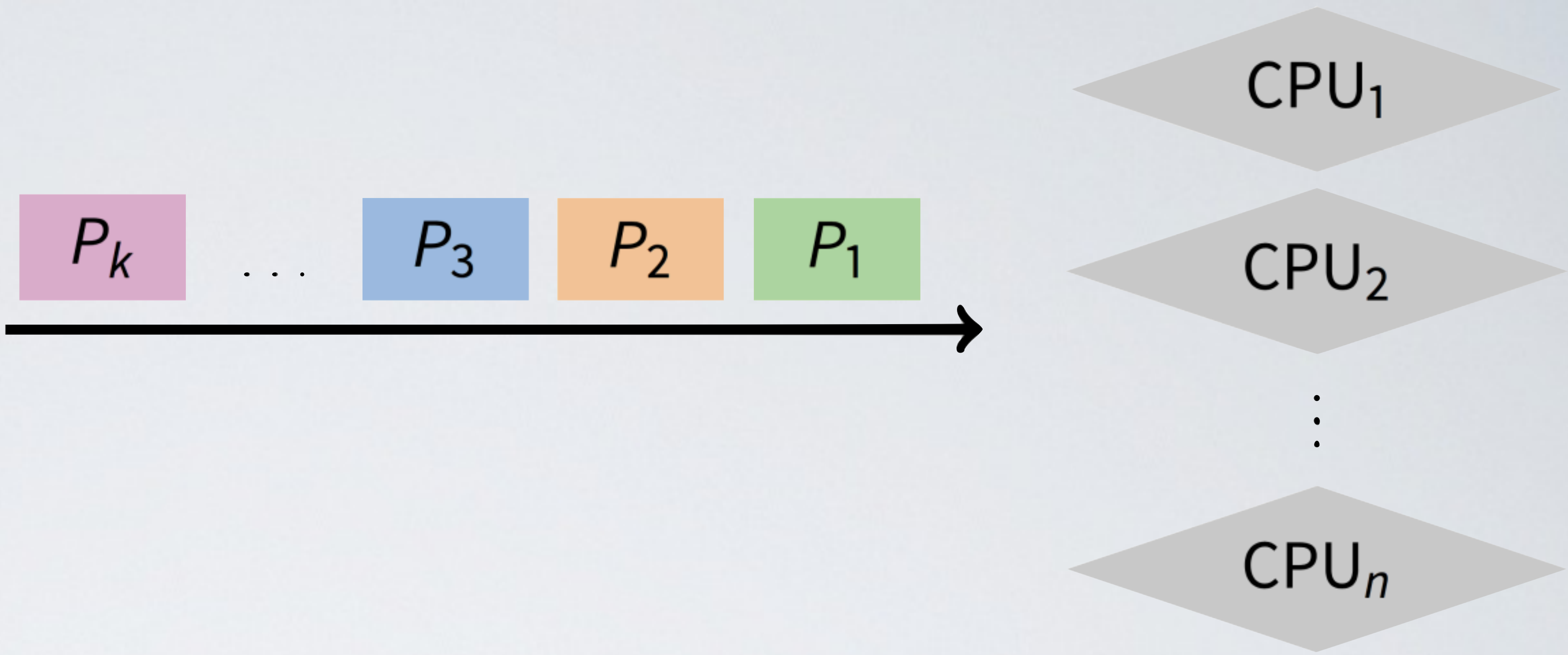
Windows : `ExitProcess(int status)`

➡ The OS will cleanup after the process:

- Terminates all threads (coming next)
- Closes open files, network connections
- Frees allocated memory (and VM pages out on disk)
- Removes PCB from kernel data structures

4. Scheduling

The scheduling problem



- n processes ready to run
- k ≥ 1 CPUs

➡ Scheduling Policy
which jobs should we assign to which CPU(s)?
and for how long?

Non Goals : Starvation

Starvation is when a process is prevented from making progress because some other process has the resource it requires (could be CPU or a lock)

➡ Starvation is usually a side effect of the scheduling algorithm

- e.g a high priority process always prevents a low priority process from running

➡ Starvation can be a side effect of synchronization (forthcoming lecture)

- e.g constant supply of readers always blocks out writers

Scheduling Criteria

- **Throughput** – # of processes that complete per unit time
  *# jobs/time (*Higher is better)

- **Turnaround time** – time for each process to complete
  Tfinish – Tstart (Lower is better)

- **Response time** – time from request to first response ()
  i.e. time between waiting to ready transition and ready to running transition
  Tresponse – Trequest (Lower is better)

➡ Above criteria are affected by secondary criteria

  - CPU utilization – %CPU fraction of time CPU doing productive work
  - Waiting time – Avg(Twait) time each process waits in the ready queue

How to balance criteria?

- **Batch systems** (supercomputers)
  strive for job throughput and turnaround time
- **Interactive systems** (personal computers)
  strive to minimize response time for interactive jobs

However, in practice, users prefer predictable response time over faster but highly variable response time

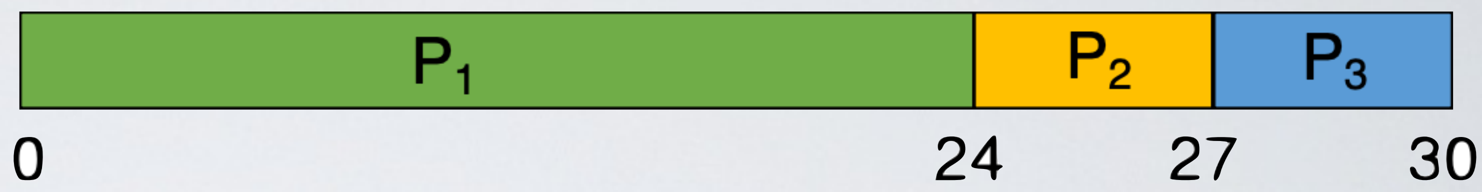
Often optimized for an average response time

Two kinds of scheduling algorithm

- **Non-preemptive scheduling** (good for batch systems) once the CPU has been allocated to a process, it keeps the CPU until it terminates

- **Preemptive scheduling** (good for interactive systems) CPU can be taken from a running process and allocated to another

FCFS - First Come First Serve (non-preemptive)



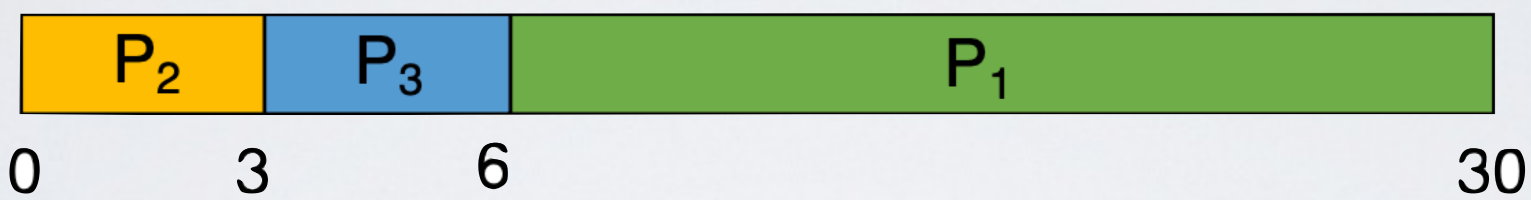
➡ Run jobs in order that they arrive (no interrupt)

Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(24 + 27 + 30) / 3 = 27 sec in average
Waiting Time	(0 + 24 + 27) / 3 = 17 sec in average

- **Problem : convoy effect**
  all other processes wait for the one big process to release the CPU

SJF - Shortest-Job-First (non-preemptive)

➡ Choose the process with the shortest processing time



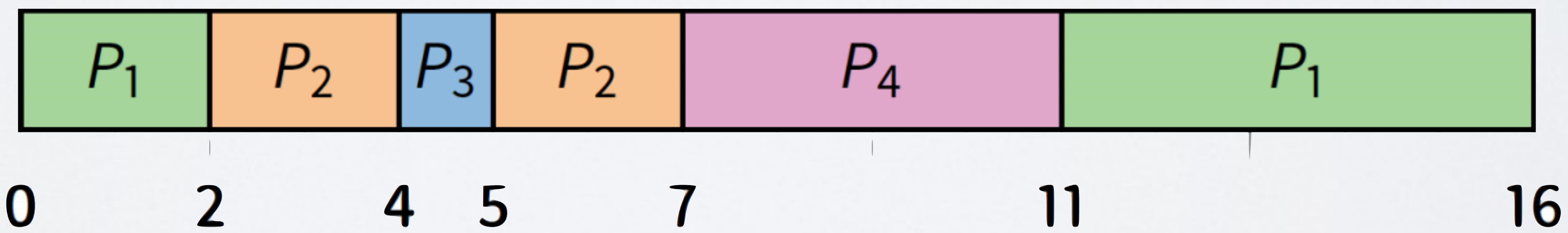
Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(30 + 3 + 6) / 3 = 13 sec in average
Waiting Time	(0 + 3 + 6) / 3 = 3 sec in average

◉ **Problem :** we need to know processing time in advance

SRTF - Shortest-Remaining-Time-First (preemptive)

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

➡ if a new process arrives with CPU burst length less than remaining time of current executing process, preempt current process



✓ **Good :** optimize waiting time

◉ **Problem :** can lead to starvation

RR - Round Robin (preemptive)

➡ Each job is given a time slice called a quantum, preempt job after duration of quantum, move to back of FIFO queue

✓ **Good :** fair allocation of CPU, low waiting time (interactive)

◉ **Problem :** no priority between processes

Time Quantum

➡ Context switches are frequent and need to be very fast

- How to pick quantum?
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum - But not so large system reverts to FCFS

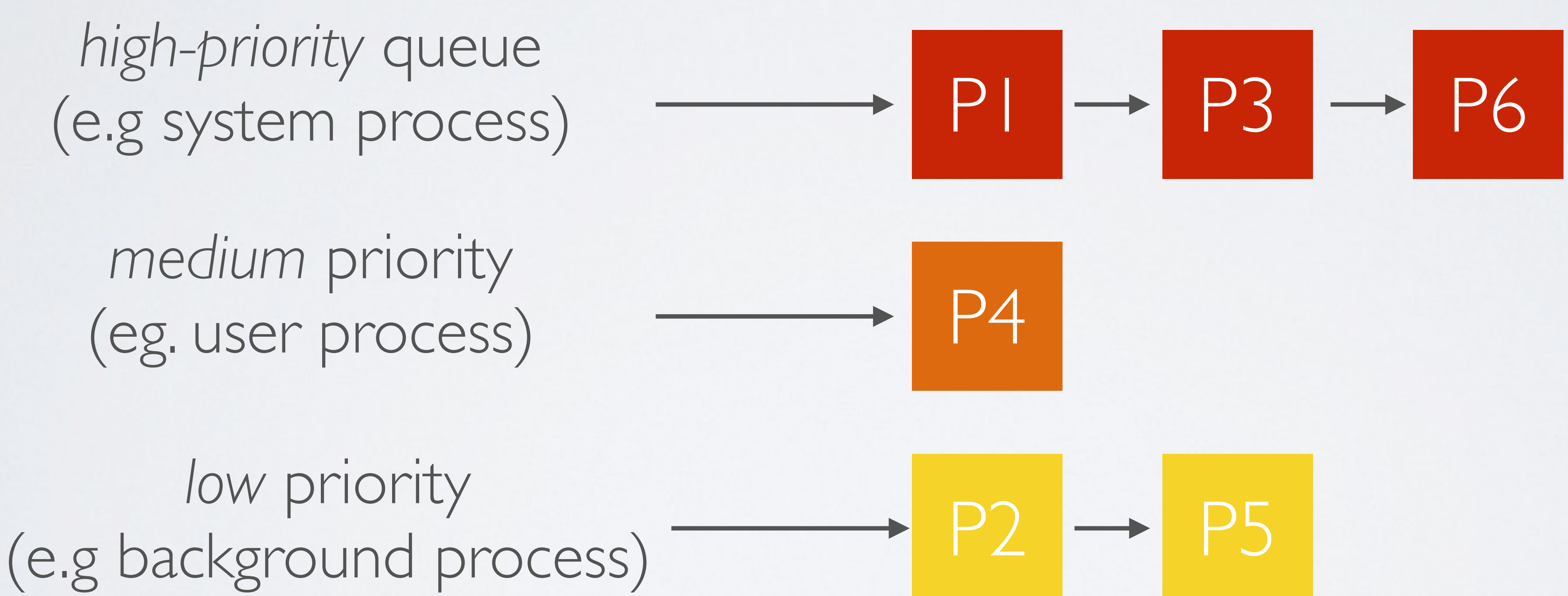
✓ Typical values: 1–100 ms

Why having priorities?

- ✓ Optimize job turnaround time for “batch” jobs
- ✓ Minimize response time for “interactive” jobs

MLQ - Multilevel Queue Scheduling (preemptive)

➡ Associate a priority with each process and execute highest priority process first. If same priority, do round-robin.



- **Problem 1 :** starvation of low priority processes
- **Problem 2**: how to decide on the priority?

Some solutions

- ➡ **To prevent starvation**
  change the priority over time by either
  - increase priority as a function of waiting time
  - or decrease priority as a function of CPU consumption
- ➡ **To decide on the priority**
  by observing and keeping track of the process
  e.g past executions, I/O

MLFQ - Multilevel **Feedback** Queue Scheduling (preemptive)

➡ Same as MLQ but change the priority of the process based on *observations*

Rule 1	If Priority(A) > Priority(B), A runs
Rule 2	If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue
Rule 3	When a job enters the system, it is placed at the highest priority (the topmost queue)
Rule 4	Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)
Rule 4	After some time period S, move all the jobs in the system to the topmost queue

✓ **Good :** Turing-award winner algorithm

Coming next

Multi-tasking based on process is expensive

- Context switching is expensive
- Inter-process communication is expensive

➡ **Solution :** Unix Threads

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*