Operating Systems

Prof. Yongkun Li
中国科大-计算机学院 教授
http://staff.ustc.edu.cn/~ykli

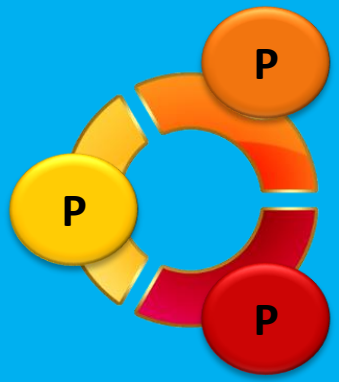
Ch6
Process Scheduling

Outline

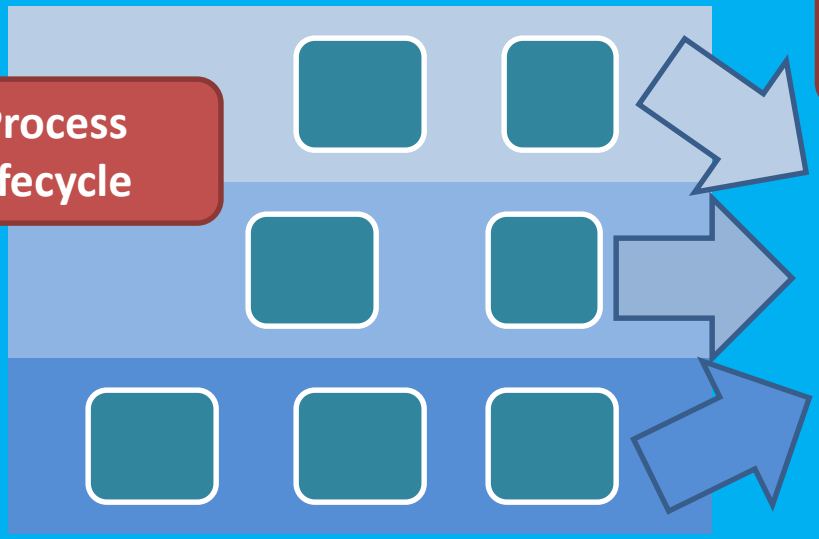
Process Process Process

User Space

Kernel Space

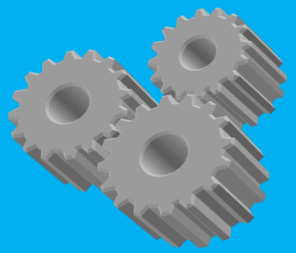


Process lifecycle



Scheduling Alg

Scheduler



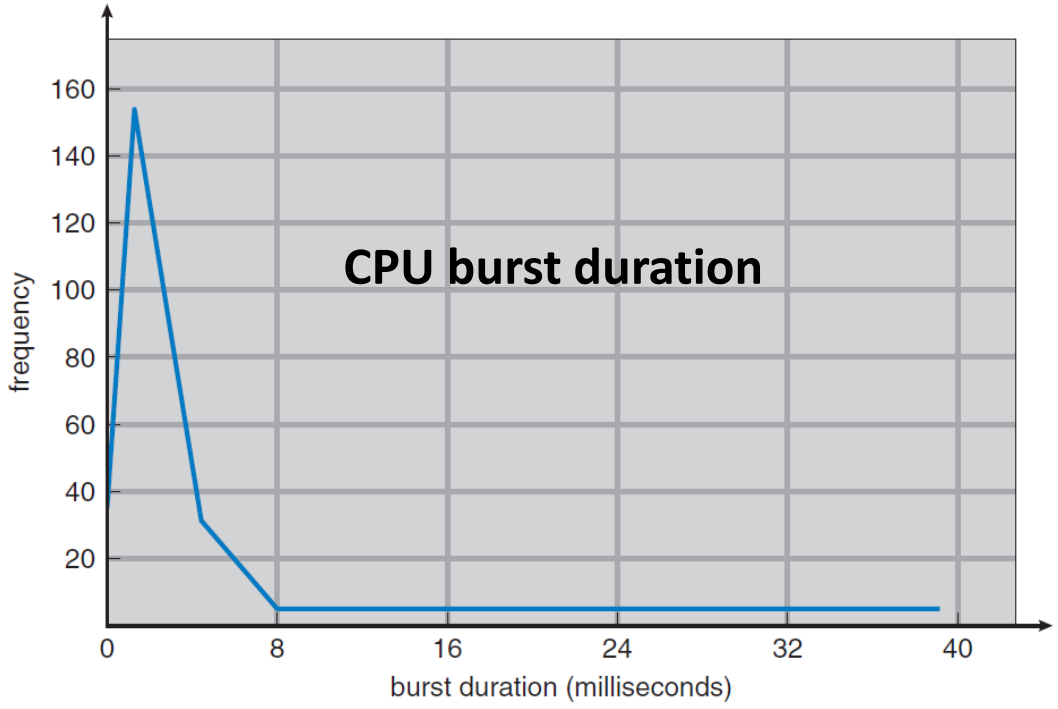
Process Communication & Synchronization

Process Scheduling

Context-switching

Why scheduling is needed

- Process execution
  - Consists of a cycle of CPU execution and I/O wait
  - CPU burst + I/O burst



Why scheduling is needed

Question. How to improve CPU utilization (CPU is much faster than I/O)?

Question. How to improve system responsiveness (interactive applications)?

Multiprogramming

Multitasking

A system may contain many processes which are at different states (ready for running, waiting for I/O)

Scheduling is required because the number of computing resource – the CPU – is **limited**.

Topics

- Process lifecycle
- Process scheduling
  - Context switching
  - Scheduling criteria
  - Scheduling algorithms
  - Applications/Scenarios

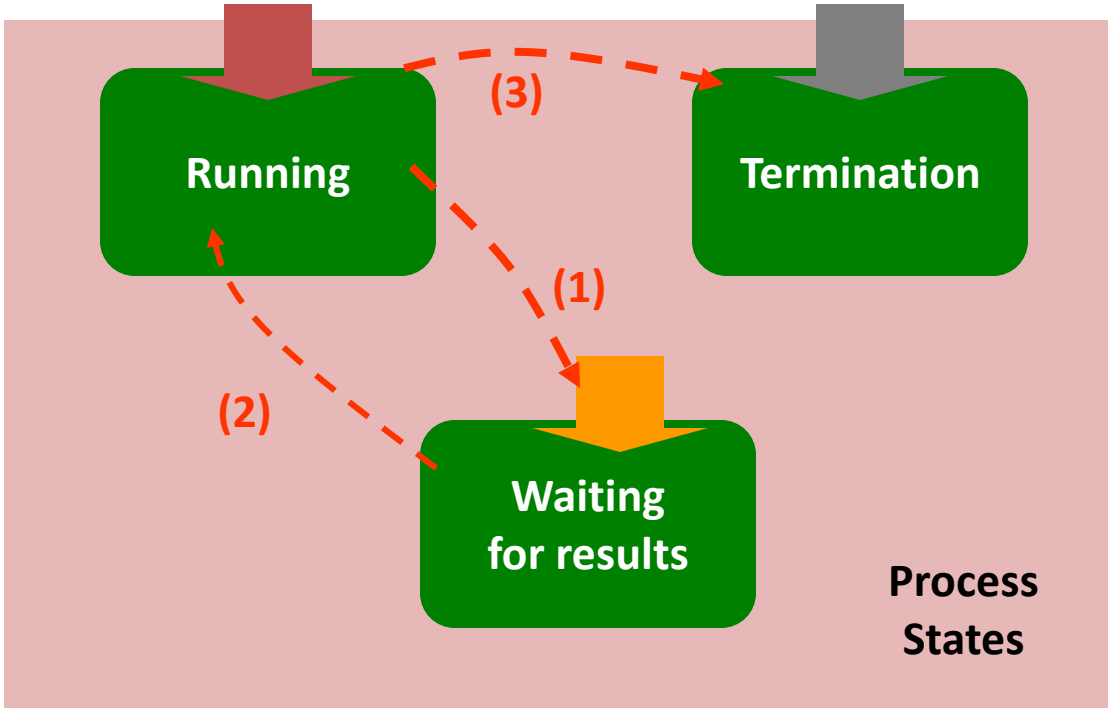
Topics

- **Process lifecycle**
- Process scheduling
  - Context switching
  - Scheduling criteria
  - Scheduling algorithms
  - Applications/Scenarios

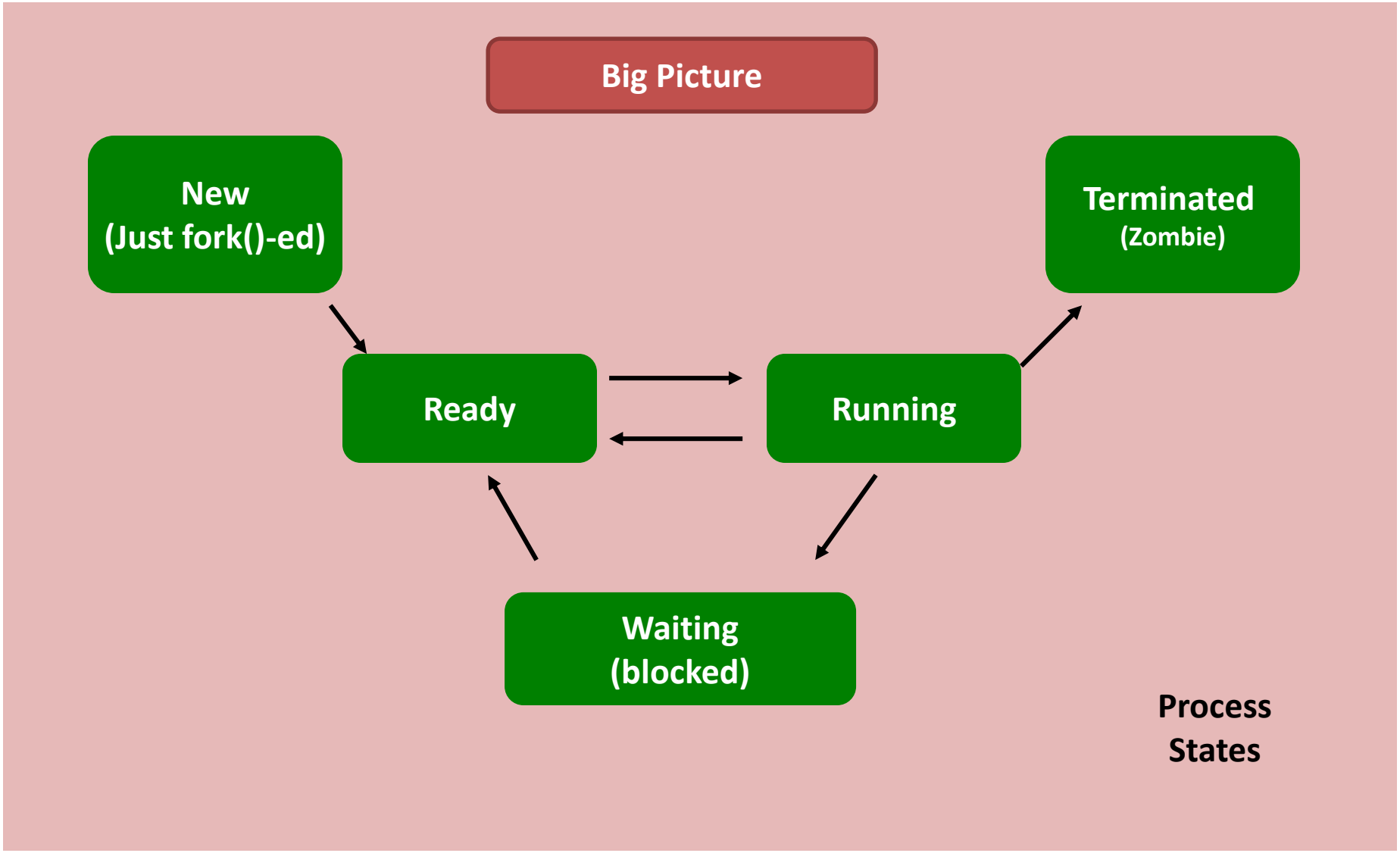
Programmer's point of view…

- This is how a fresh programmer looks at a process' life cycle.

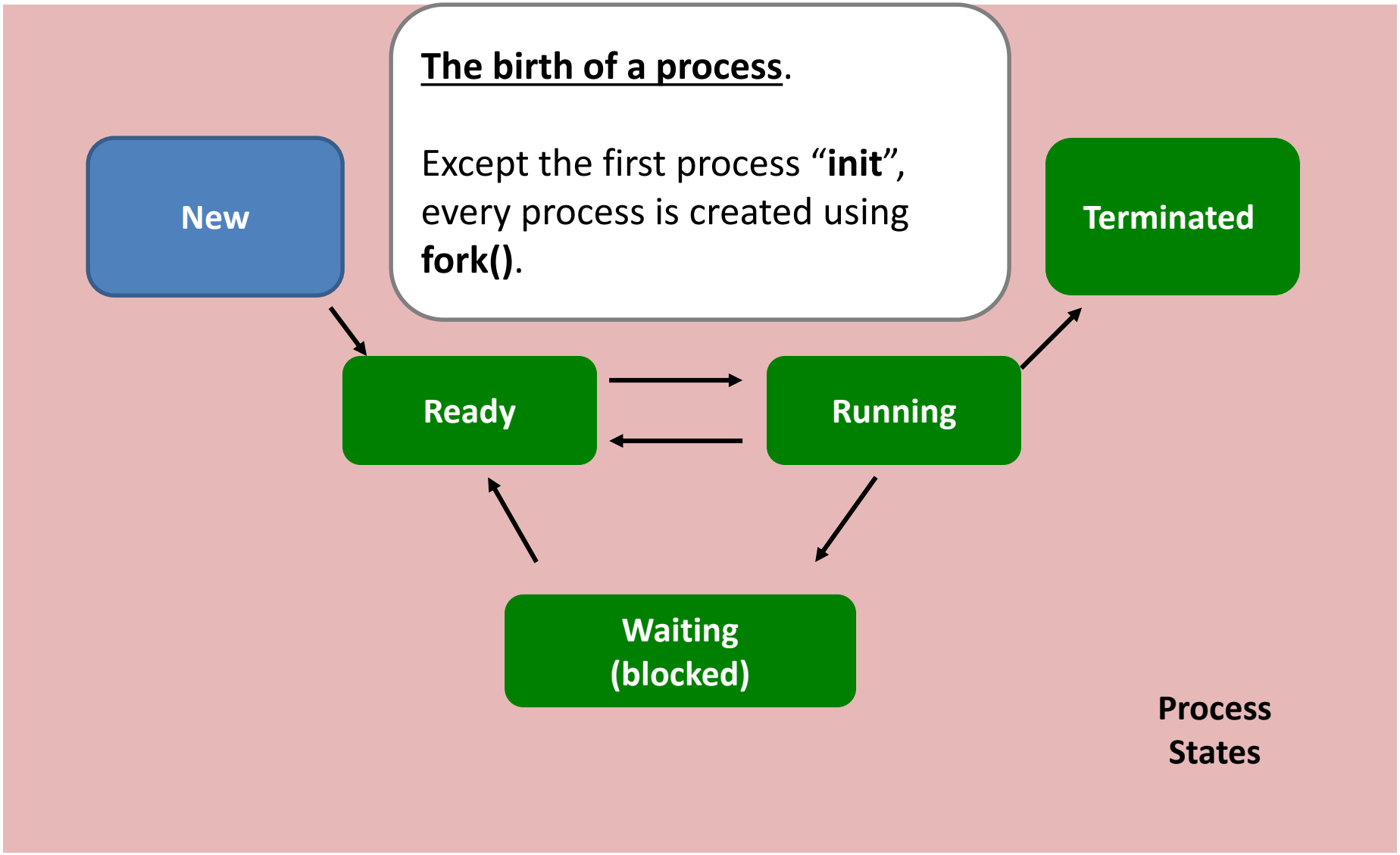
```
int main(void) {
    int x = 1;
    getchar();
    return x;
}
```



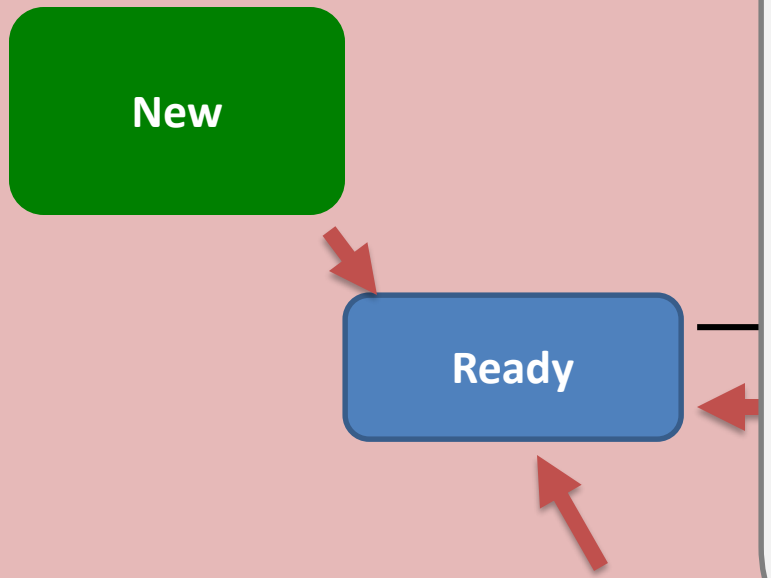
Kernel's point of view…



Kernel's point of view…



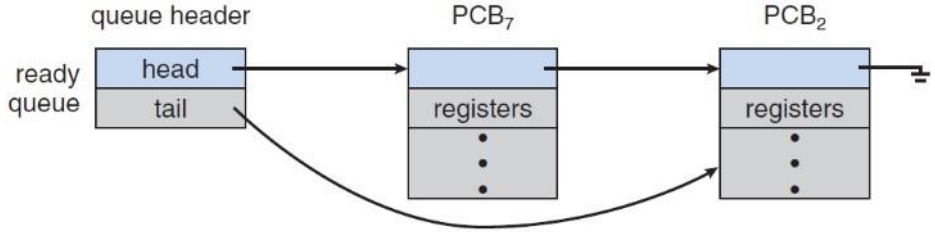
Kernel's point of view…



The process is ready.
It means it is ***ready to run*** **but is not running**.

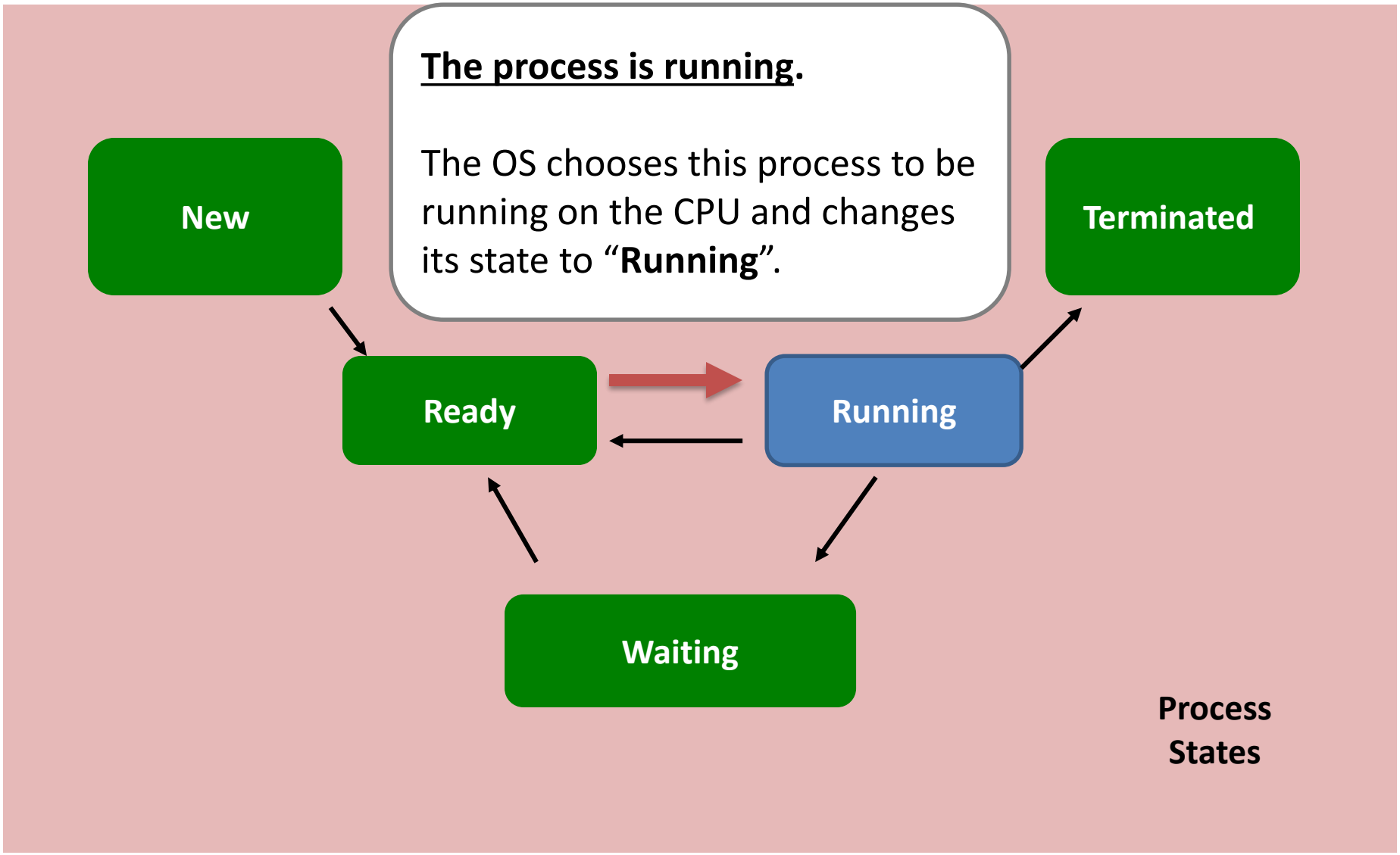
A process may become "**ready**" after...
- it is just created by `fork()`;
- it has been running on the CPU for some time and the OS chooses another process to run;
- returning from blocked states.

All ready processes are kept on a list called **ready queue**

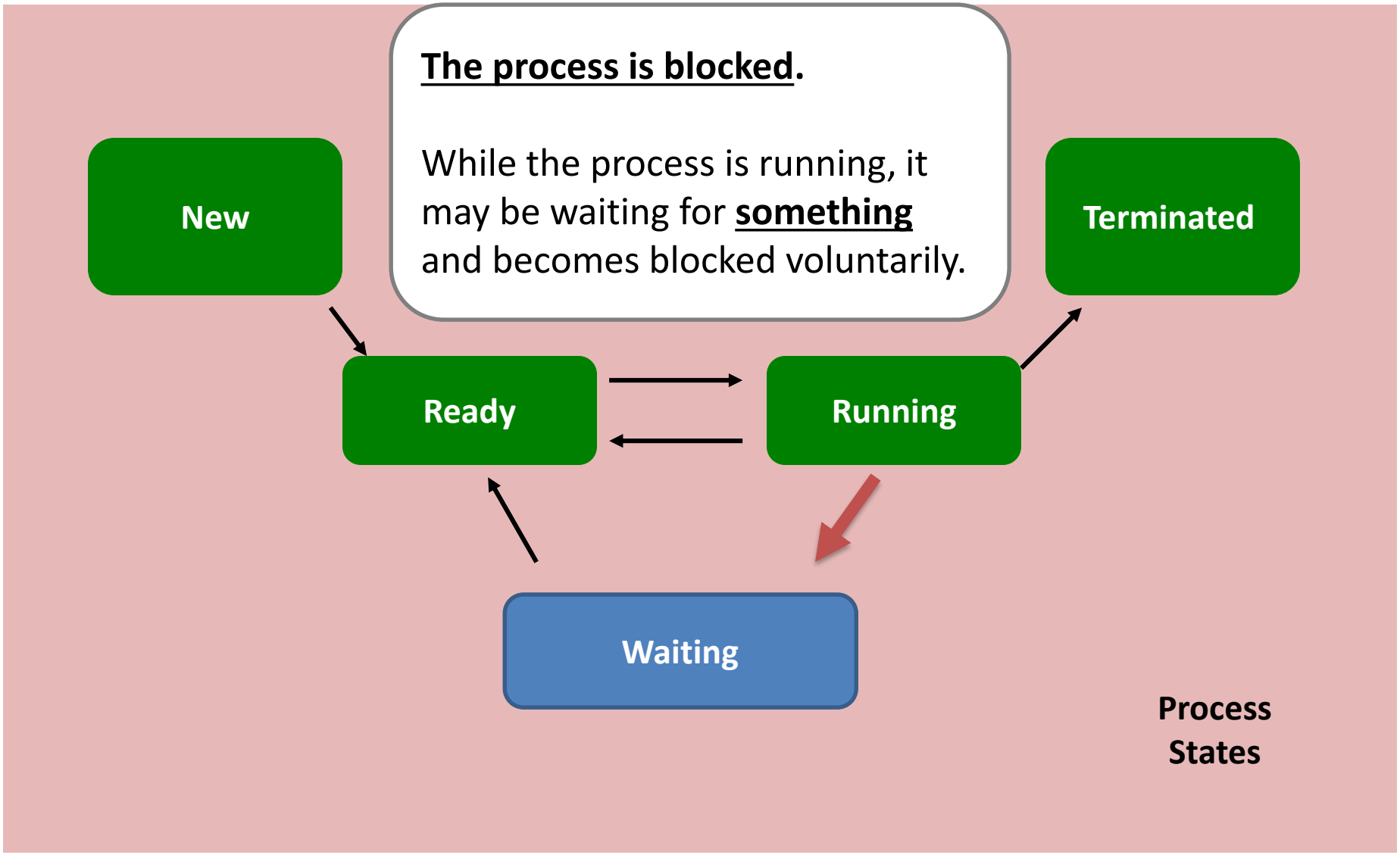


Process States

Kernel's point of view…



Kernel's point of view…

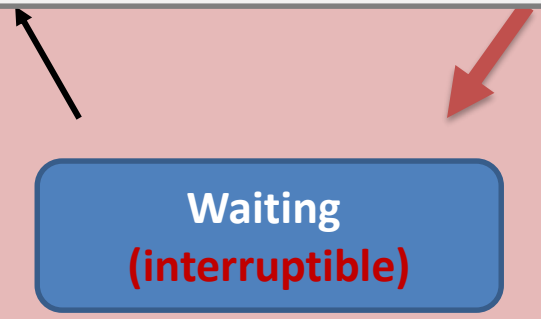


Kernel's point of view...

Example. **Reading a file**.

Sometimes, the process has to wait for the response from the device and, therefore, it is **blocked**.

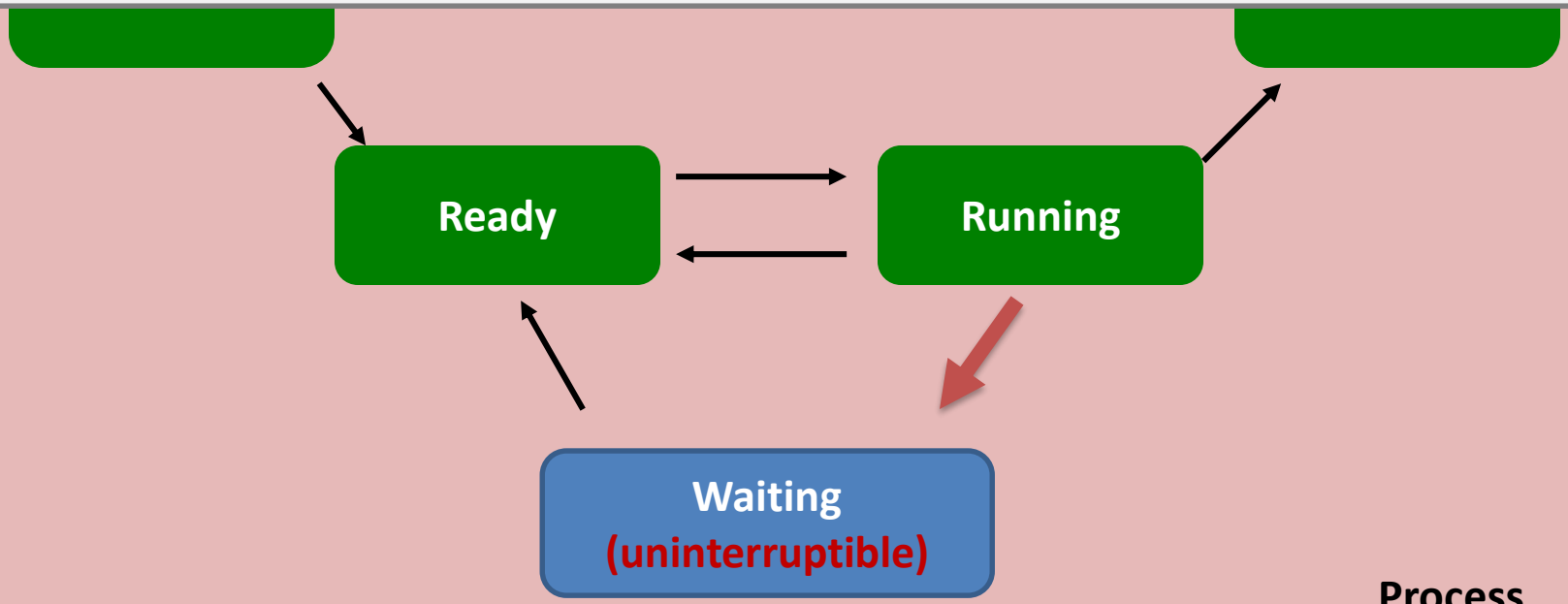
Nevertheless, this blocking state is **interruptible**. E.g., "**Ctrl + C**" can get the process out of the waiting state (but goes to termination state instead).



Process States

Kernel's point of view…

Sometimes, a process needs to wait for a resource but it doesn't want to be disturbed while it is waiting. In other words, **the process wants that resource very much**. Then, the process status is set to the **uninterruptible** status.

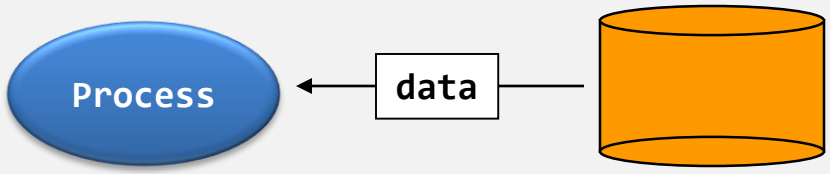


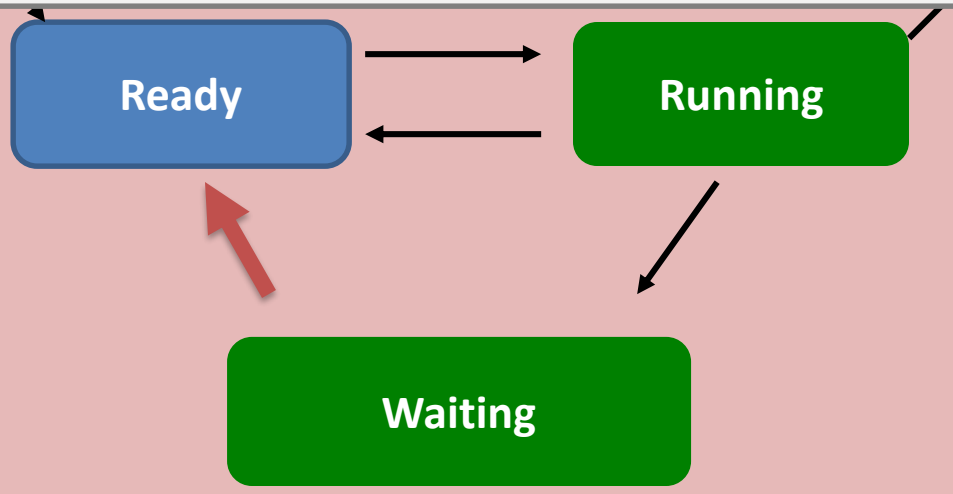
Process States

Kernel's point of view…

Return back to ready.

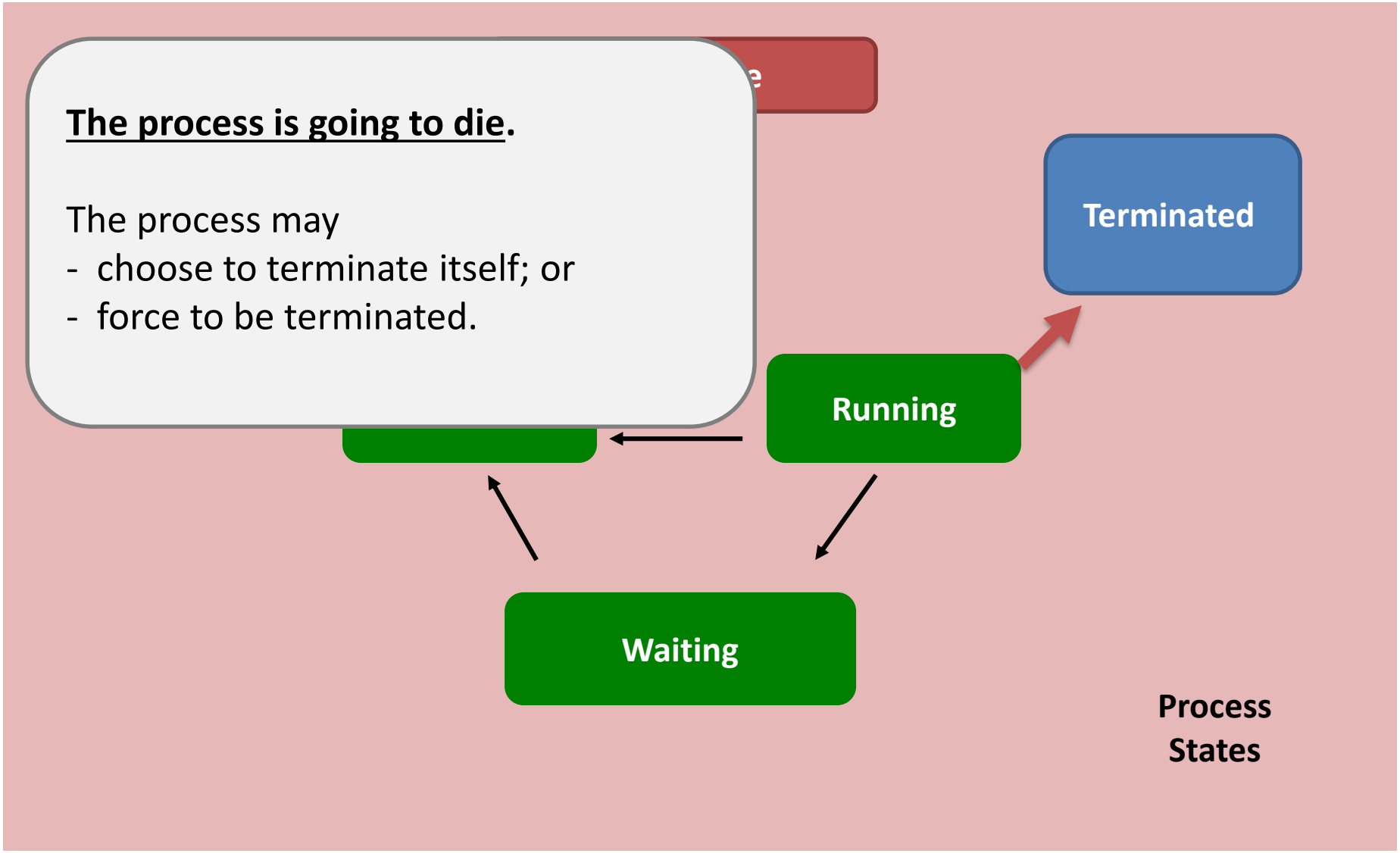
When response arrives, the status of the process changes back to **Ready**. from any one of the blocked states.





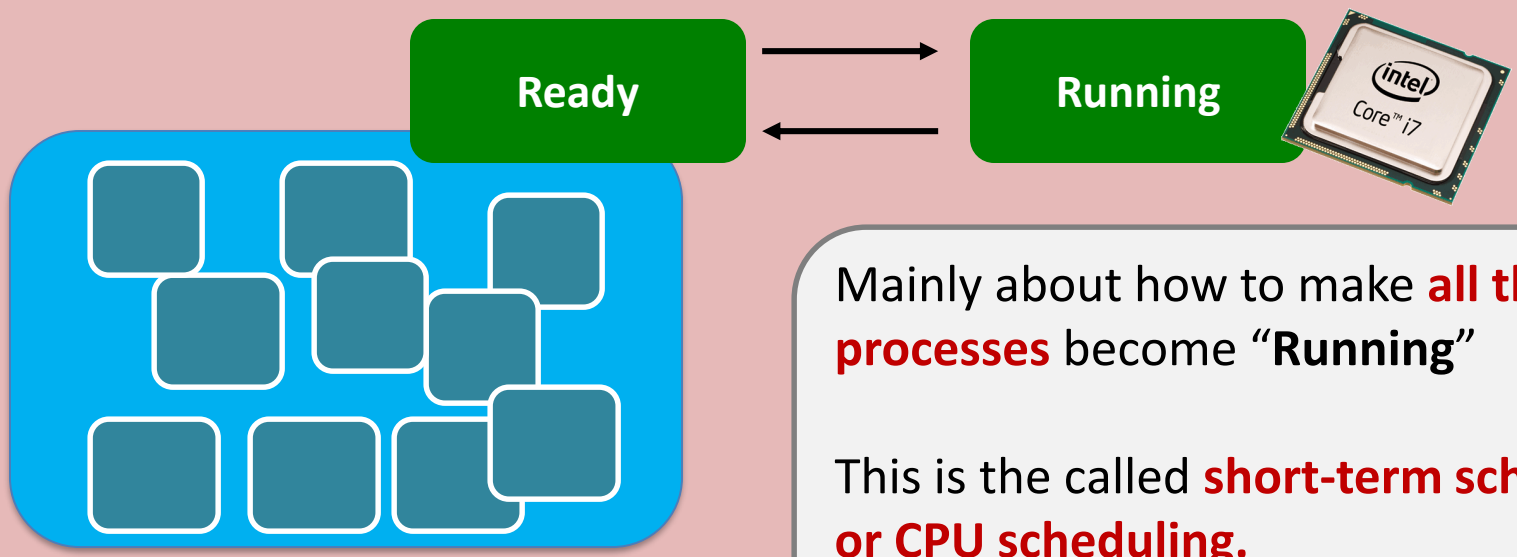
Process States

Kernel's point of view…



What is scheduling?

So, what is process scheduling?



Mainly about how to make **all the ready processes** become "**Running**"

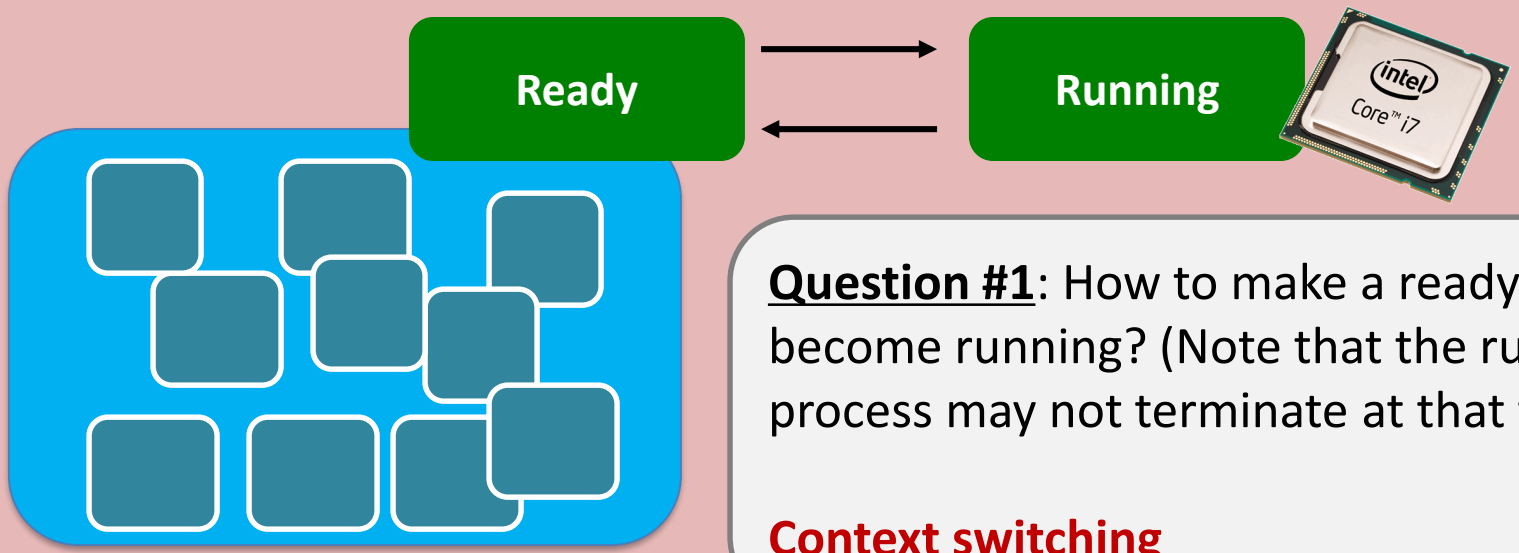
This is the called **short-term scheduling or CPU scheduling.**

Triggering Events

- When process scheduling happens:

A new process is created.	When "**fork()**" is invoked and returns successfully. Then, whether the parent or the child is scheduled is up to the scheduler's decision.
An existing process is terminated.	**The CPU is freed**. The scheduler should choose another process to run.
A process waits for I/O.	**The CPU is freed**. The scheduler should choose another process to run.
A process finishes waiting for I/O.	The interrupt handling routine **makes a scheduling request**, if necessary.

Key Issues



Question #1: How to make a ready process become running? (Note that the running process may not terminate at that time)

Context switching

Question #2: How to decide which process should be running?

Scheduling criteria & scheduling algorithms

Question #3: How to design scheduling in a real/specific system?

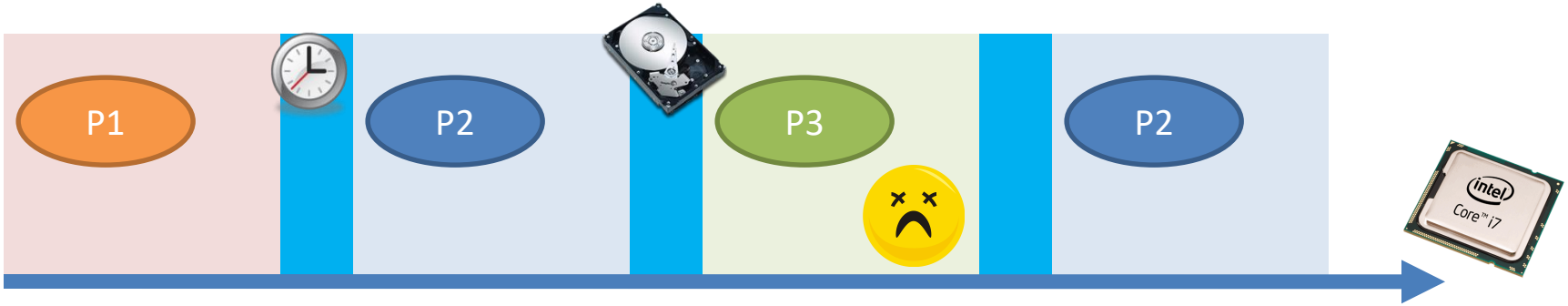
Multiprocessor system, real-time system, algorithm evaluation

Topics

- Process lifecycle
- **Process scheduling**
  - **Context switching**
  - Scheduling criteria
  - Scheduling algorithms
  - Applications/Scenarios

What is context switching?

- Before we can jump into the process scheduling topic, we have to understand what "**context switching**" is.



Scheduling is the procedure that decides which process to run next.

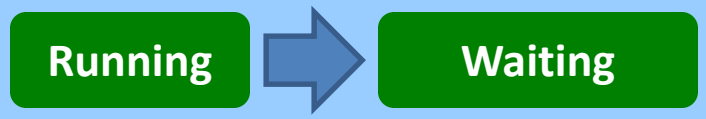
Context switching is the actual switching procedure, from one process to another.


Timer interrupt.

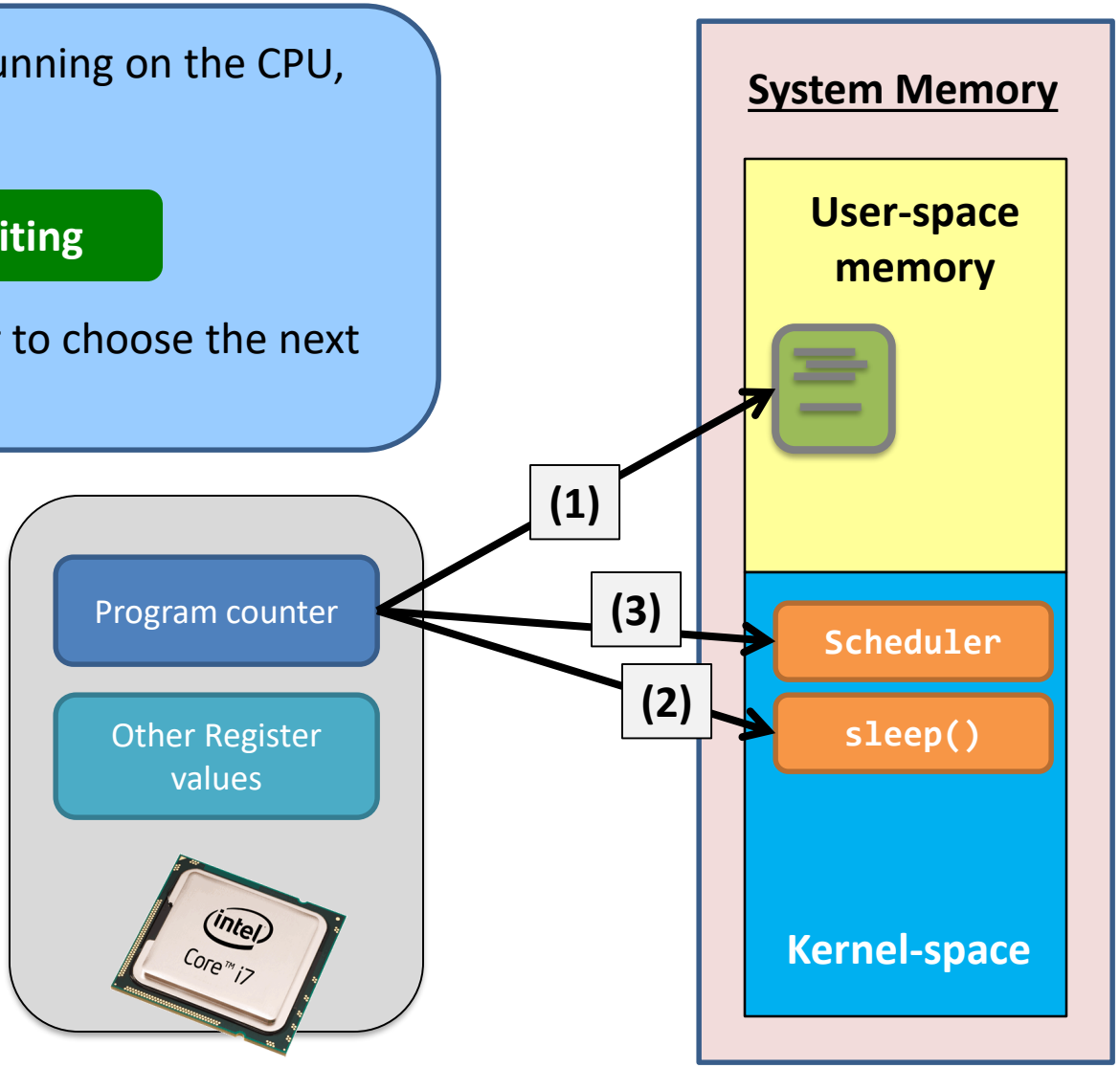

Hardware interrupt.

Switching from one process to another.

Suppose this process gives up running on the CPU, e.g., calling **`sleep()`**. Then:



Now, it is time for the scheduler to choose the next process to run.



Switching from one process to another.

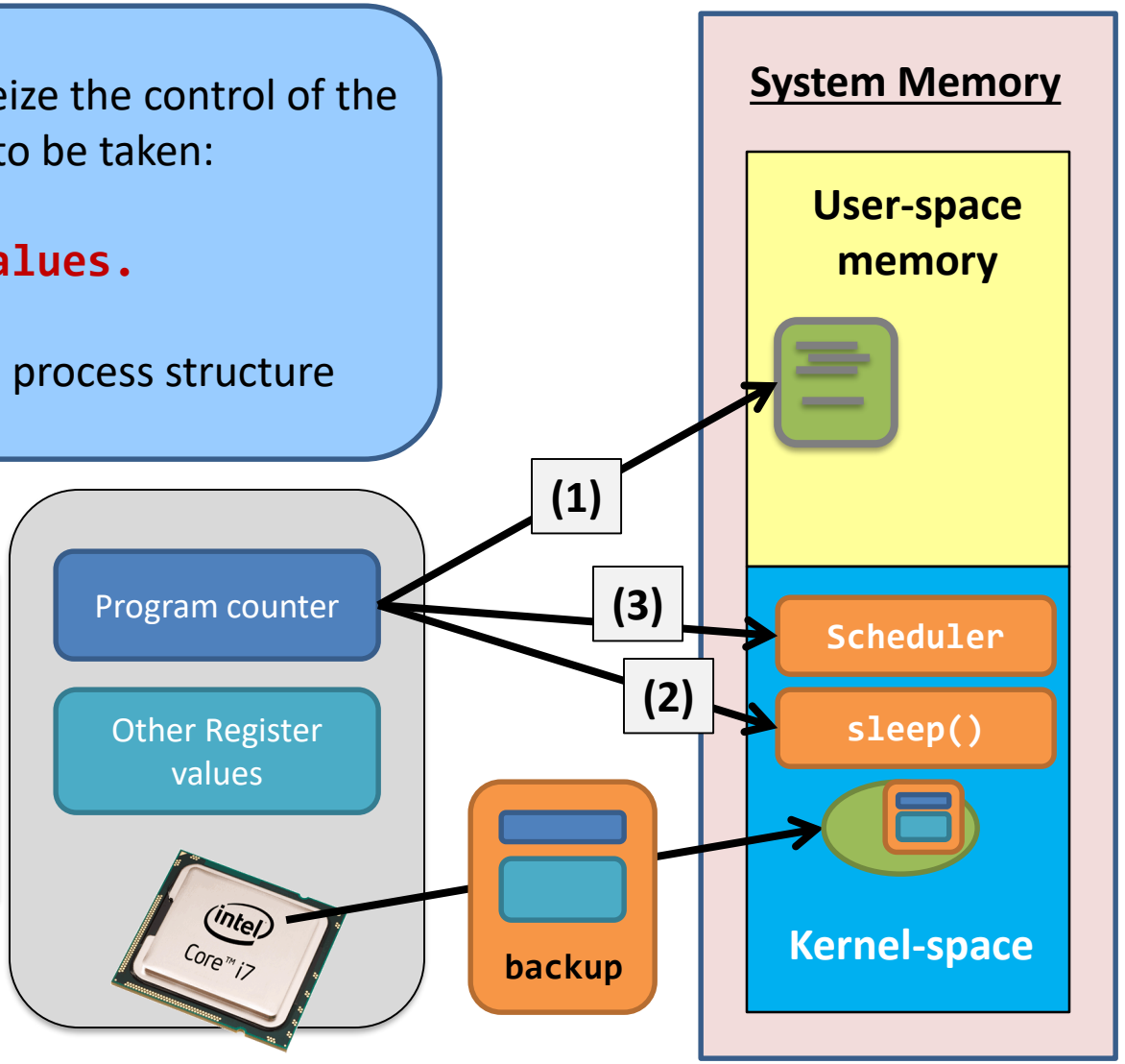
But, before the scheduler can seize the control of the CPU, a very important step has to be taken:

`Backup all registers' values.`

The backup will be stored in the process structure

The context of a process

The union of the user-space memory and the registers' values of the process



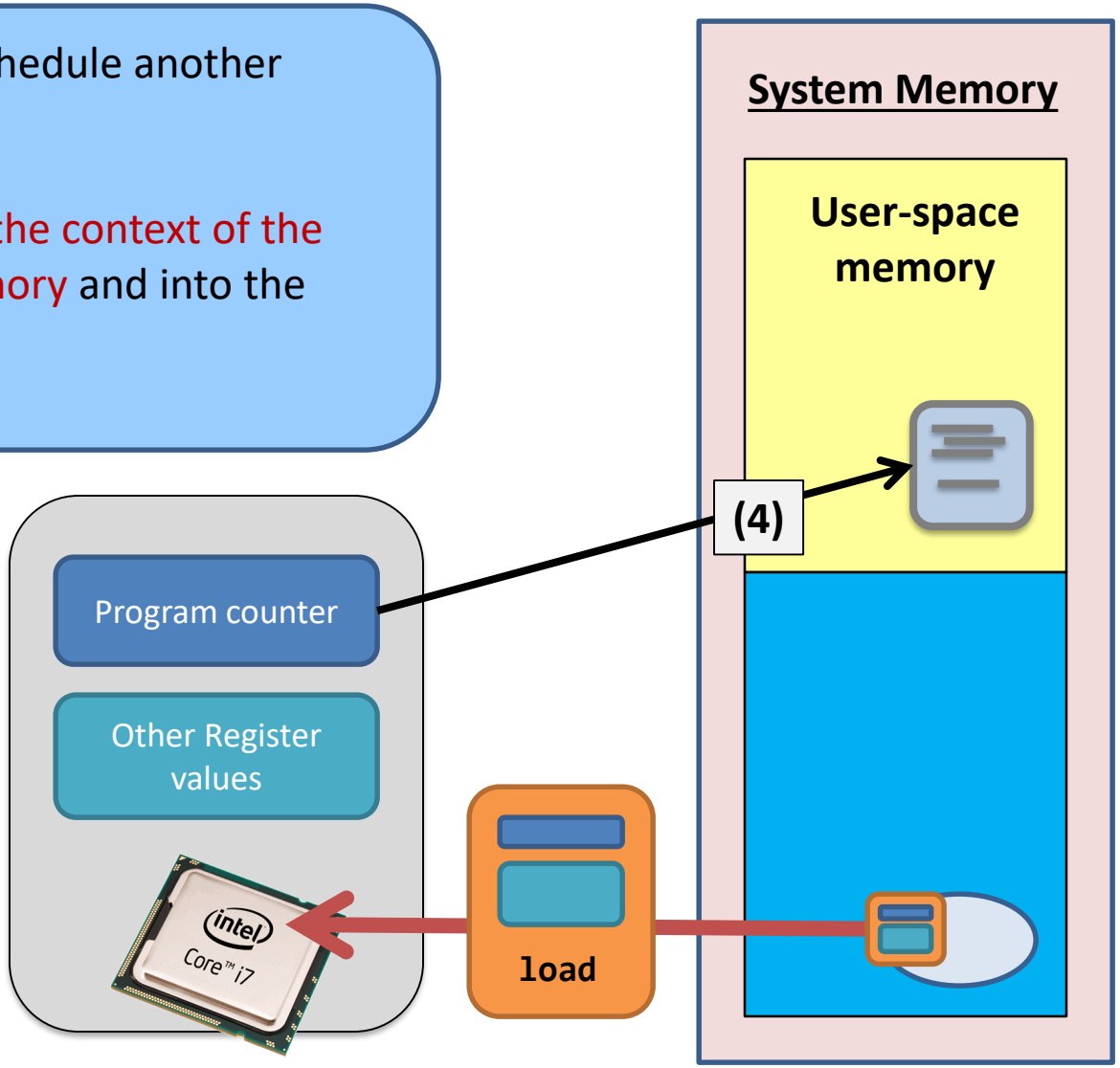
Switching from one process to another.

Say, the scheduler decides to schedule another process.

Then, the schedule has to load the context of the new process into the main memory and into the CPU.

We call the entire operation:

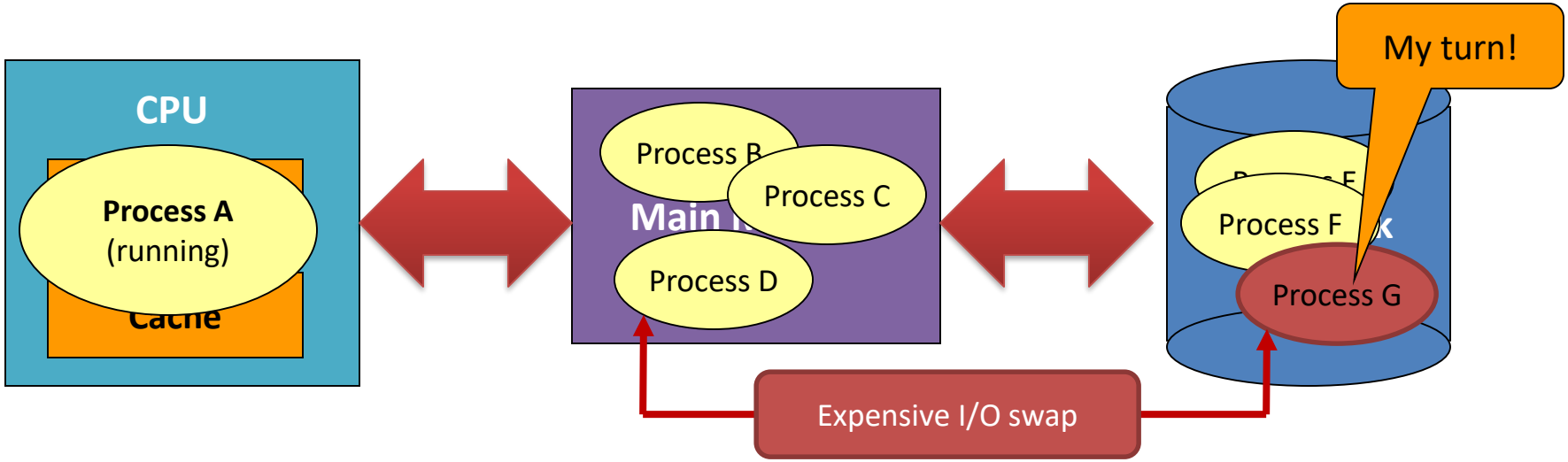
context switching



Context switching has a price to pay…

- However, context switching may be expensive…
  - Even worse, the target process may be currently stored in the hard disk.

- So, **minimizing the number of context switching** may help boost system performance.

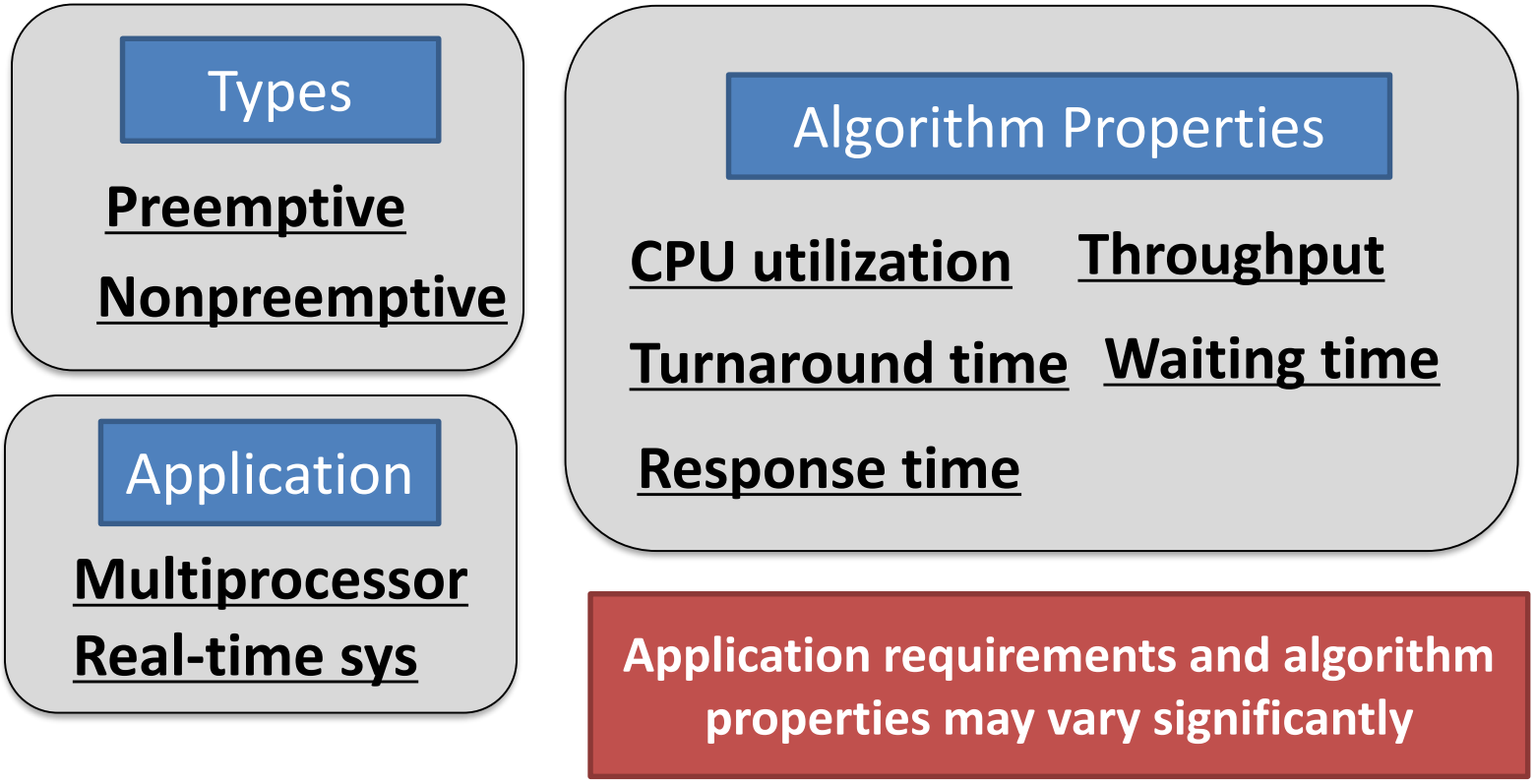


Topics

- Process lifecycle
- **Process scheduling**
  - Context switching
  - **Scheduling criteria**
  - Scheduling algorithms
  - Applications/Scenarios

Scheduling Criteria

- How to choose which algorithm to use in a particular situation?



Classes of process scheduling

- Non-preemptive scheduling.

What is it?	When a process is chosen by the scheduler, the process would never leave the scheduler until… -the process voluntarily waits for I/O, or -the process voluntarily releases the CPU, e.g., **`exit()`**.
What is the catch?	If the process is *purely CPU-bound*, it will seize the CPU from the time it is chosen until it terminates.
Pros	Good for systems that emphasize **the time in finishing tasks**. - Because the task is running without others' interruption.
Cons	Bad for nowadays systems in which **user experience** and **multi-tasking** are the primary goals.
Where can I find it?	Nowhere…but it could be found back in the mainframe computers in 1960s.

Classes of process scheduling

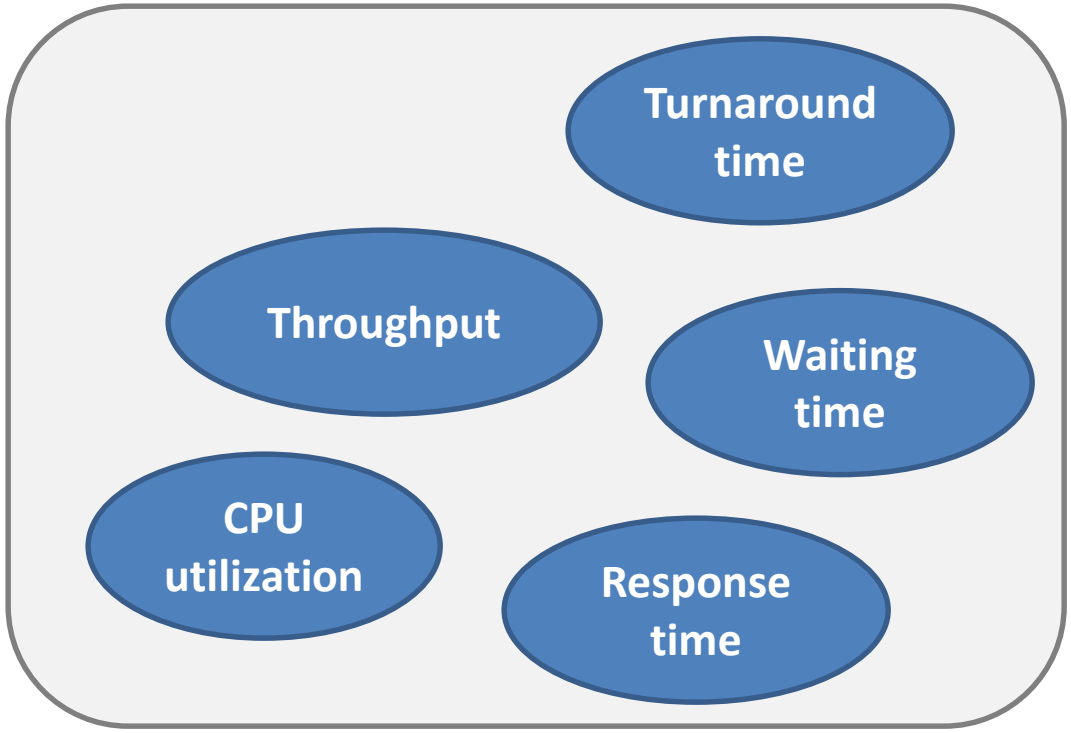
- Preemptive scheduling.

What is it?	When a process is chosen by the scheduler, the process would never leave the scheduler until… -the process voluntarily waits for I/O, or -the process voluntarily releases the CPU, e.g., **`exit()`**. -**particular kinds of interrupts and events are detected**.
What is the catch?	If that particular event is the ***periodic clock interrupt***, then you can have a **time-sharing system**.
Pros	Good for systems that emphasize **interactiveness**. - Because every task will receive attentions from the CPU.
Cons	Bad for systems that emphasize the time in finishing tasks.
Where can I find it?	Everywhere! This is the design of nowadays systems.

Performance measures

In algorithm design:

What factors/performance measures should be carefully considered?



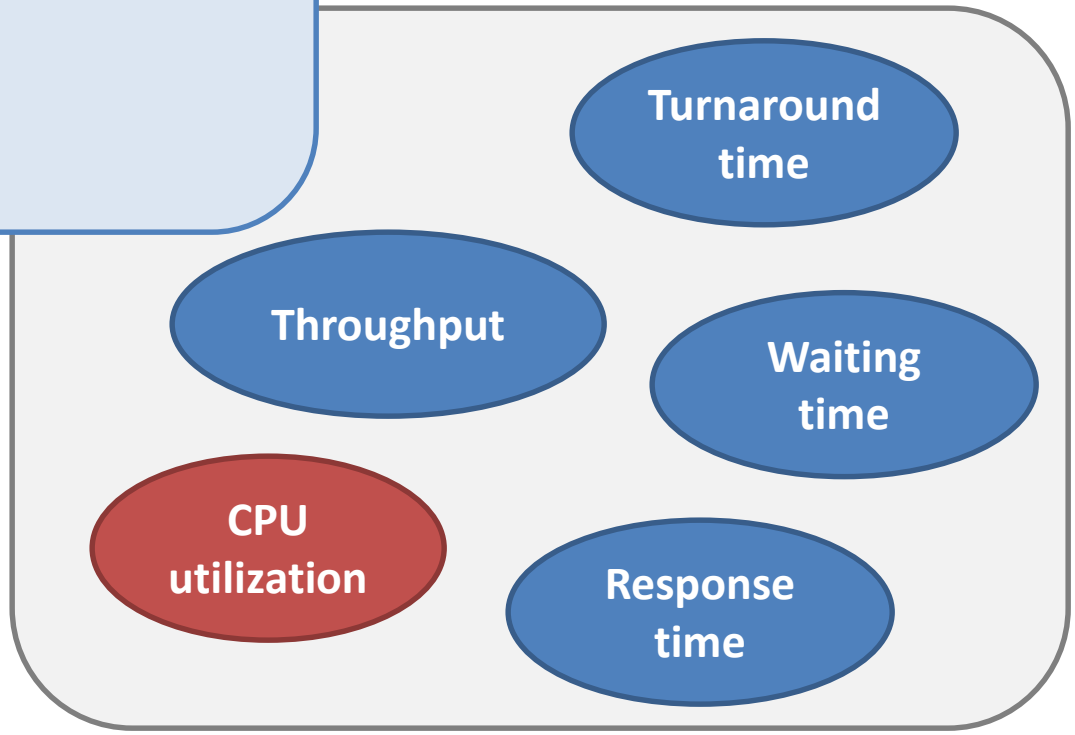
Performance measures

CPU utilization.

We want to keep CPU as busy as possible.

Theoretically, can range from 0-100%, but in real system, range from 40%-90%

The higher the better

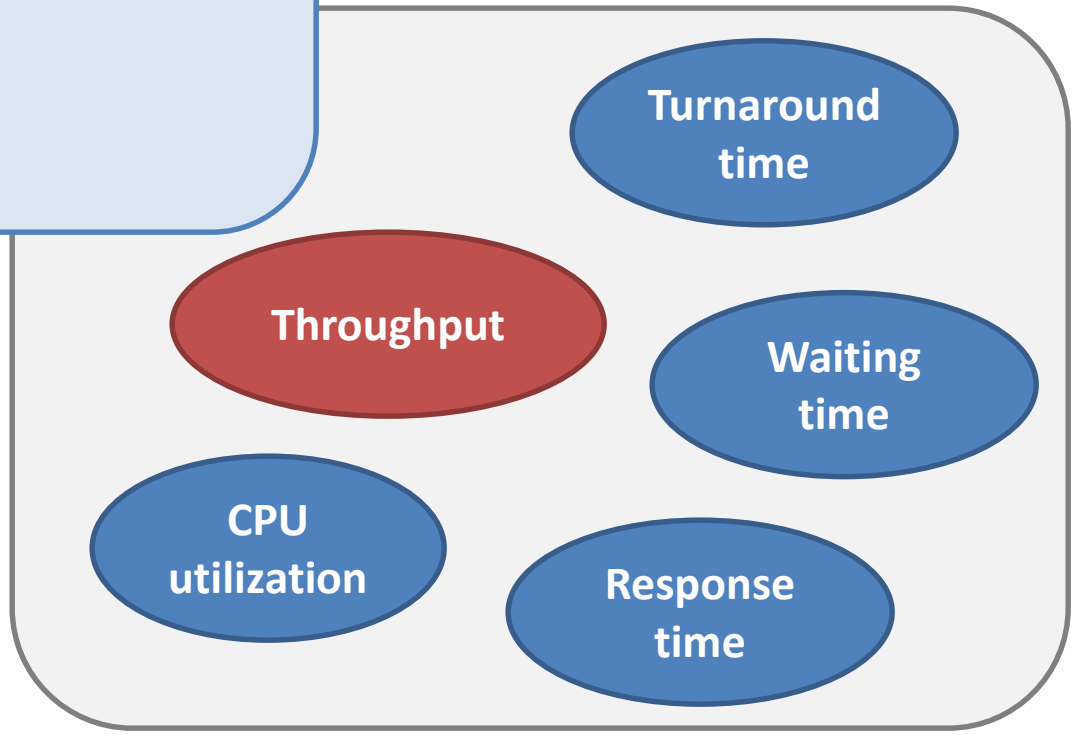


Performance measures

Throughput.

Number of processes that are completed per time unit

The higher the better

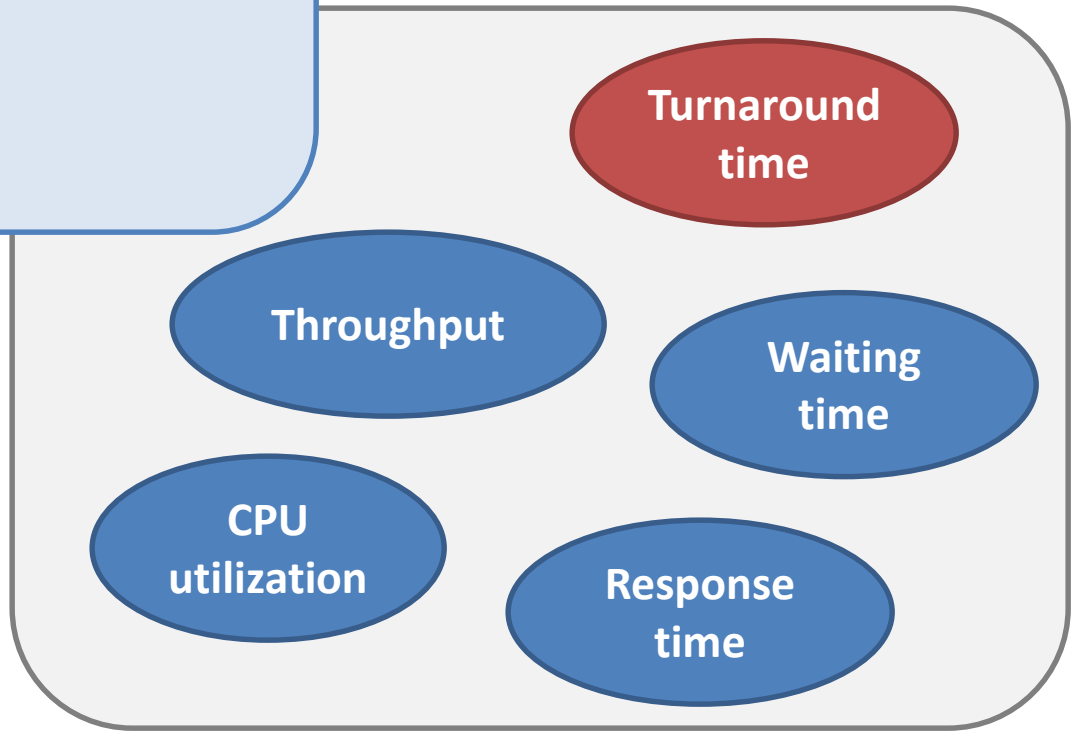


Performance measures

Turnaround time.

Time to execute the process: interval from the time of submission to the time of completion (total running time + waiting time+ doing I/O)

The lower the better

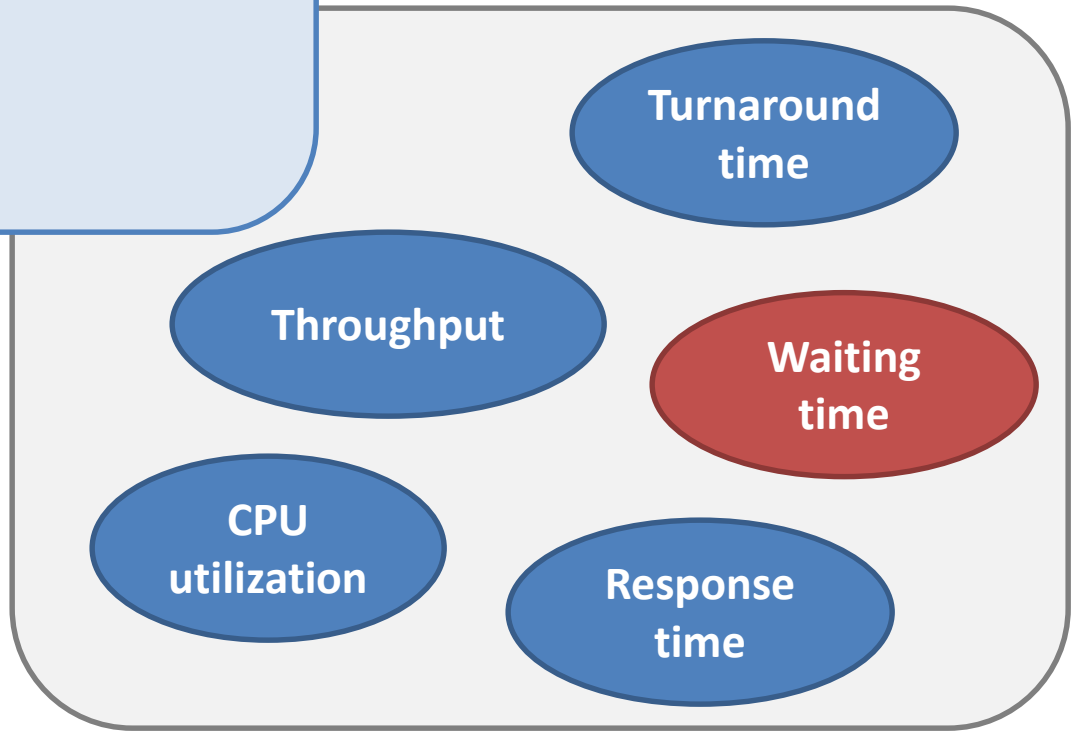


Performance measures

Waiting time.

The time spent waiting in the ready queue

The lower the better

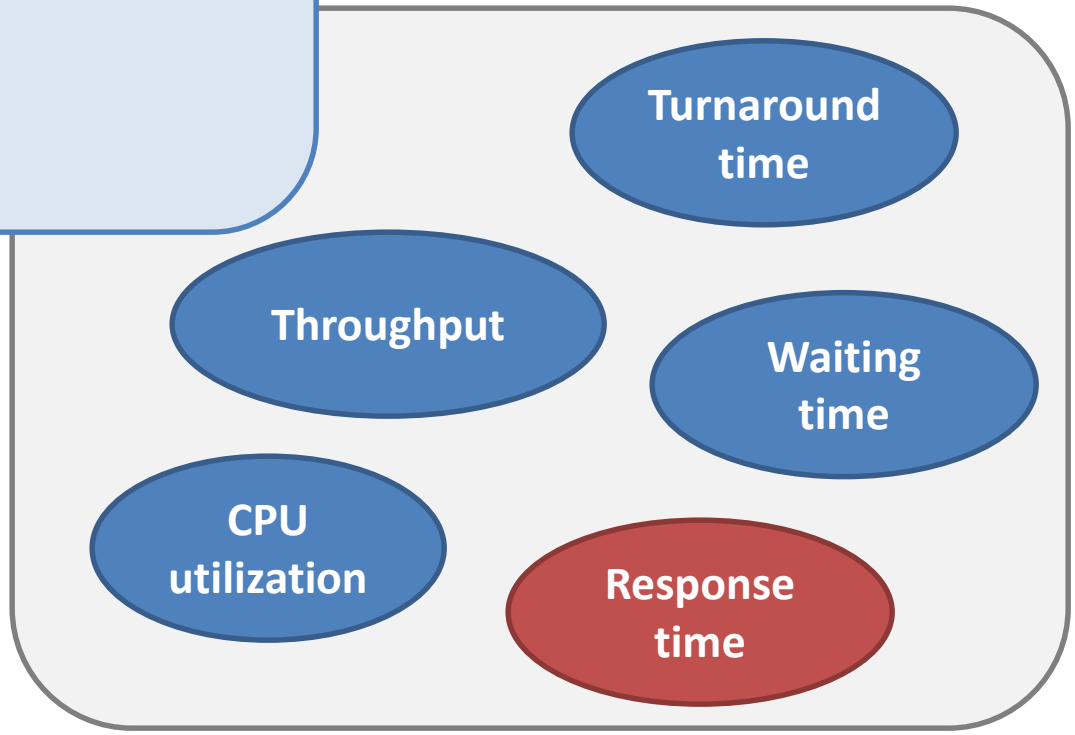


Performance measures

Response time.

The time from the submission of a request until the first response is produced (useful measure for interactive systems)

The lower the better



Challenge

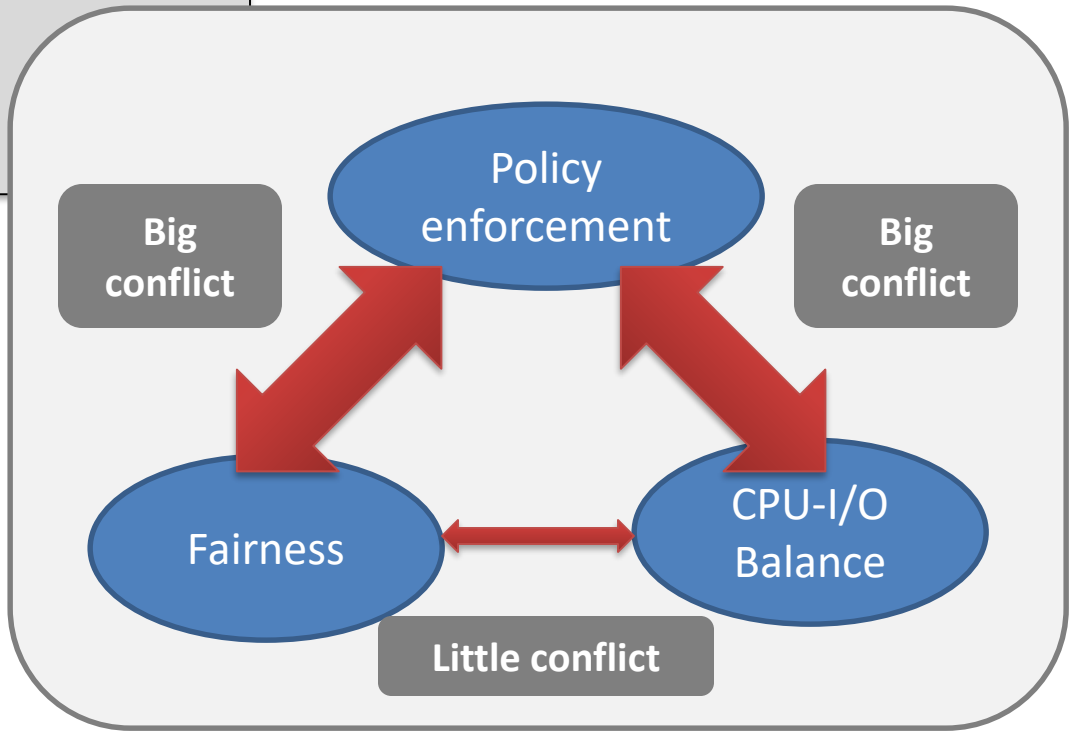
Question:

Can we optimize all the above measures simultaneously?

Usually can not!

Common goal

Design Tradeoff

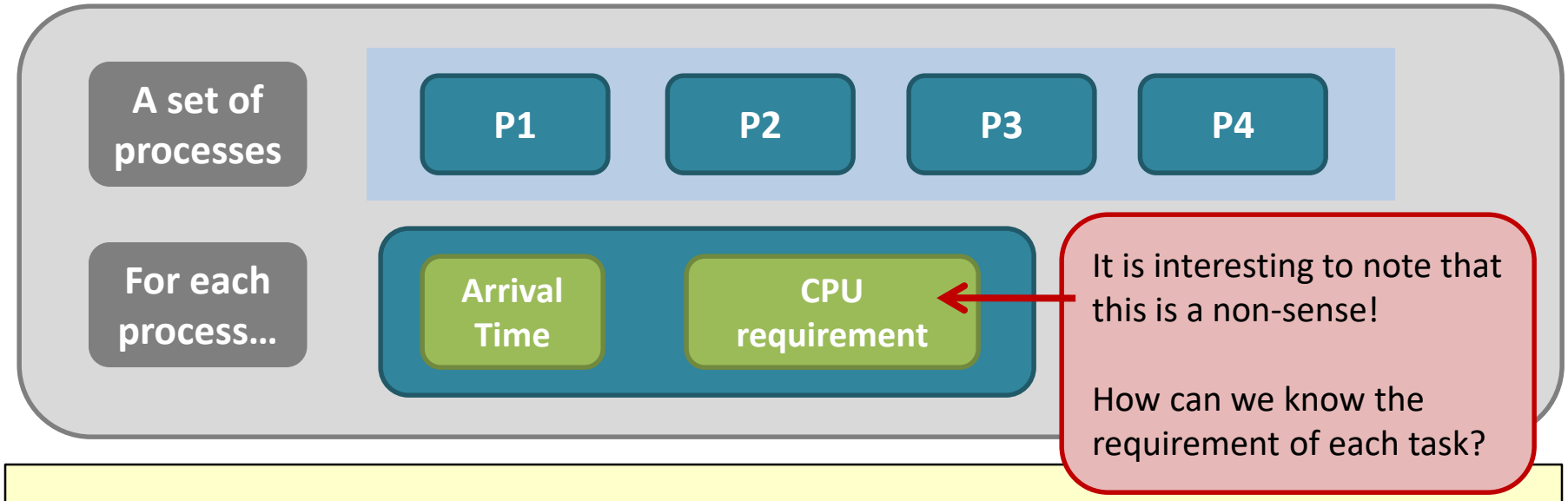


Topics

- Process lifecycle
- **Process scheduling**
  - Context switching
  - Scheduling criteria
  - **Scheduling algorithms**
  - Applications/Scenarios

Scheduling algorithms

- **Inputs to the algorithms.**



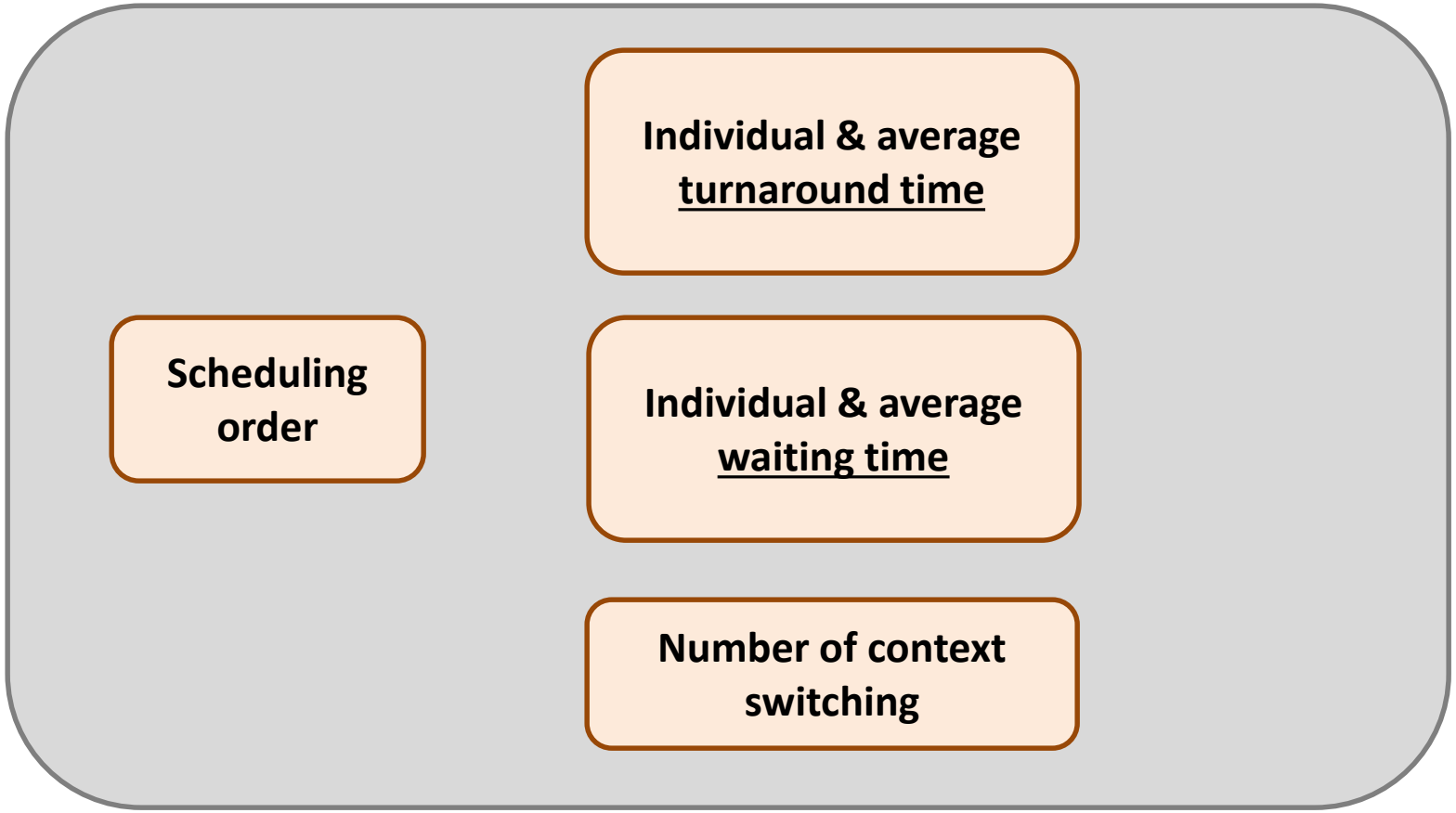
Online VS Offline

An **offline scheduling algorithm** assumes that you know all the processes submitted to the system before hand. But, an **online scheduling algorithm** does not have such an assumption.

Yet, every real scheduler has to work in an "**online scenario**". So, we have to think in an "online" way…

Scheduling algorithms

- **Outputs of the algorithms.**



Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

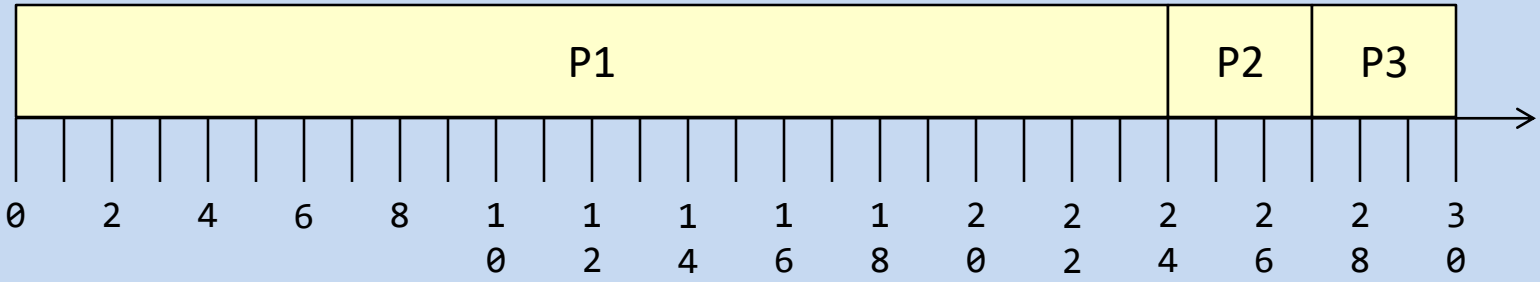
First-come, first-served scheduling

- Example 1.

Gantt Chart

No preemption

Output



Waiting time: P1 = 0; P2 = 23; P3 = 25;

Average waiting time = (0+23+25)/3 = 16;

Turnaround time: P1 = 24; P2 = 26; P3 = 28;

Average turnaround time = (24+26+28)/3 = 26;

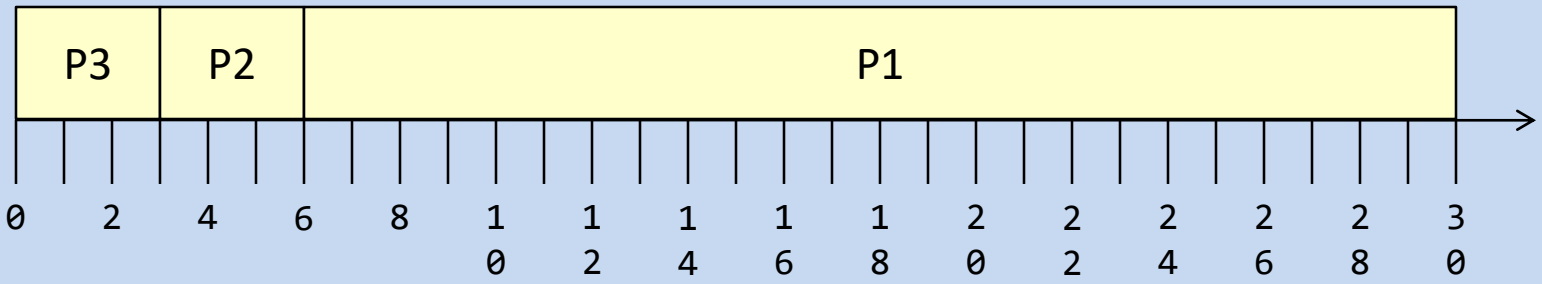
Task	Arrival Time	CPU Req.
P1	0	24
P2	1	3
P3	2	3

Input

First-come, first-served scheduling

- Example 2.

Gantt Chart



Output

Waiting time: P1 = 4; P2 = 2; P3 = 0;

Average waiting time = (4+2+0)/3 = 2;
(which is 16 in the previous case)

Turnaround time: P1 = 28; P2 = 5; P3 = 3;

Average turnaround time = (28+5+3)/3 = 12;
(which is 26 in the previous case)

Task	Arrival Time	CPU Req.
P3	0	3
P2	1	3
P1	2	24

Input order changed

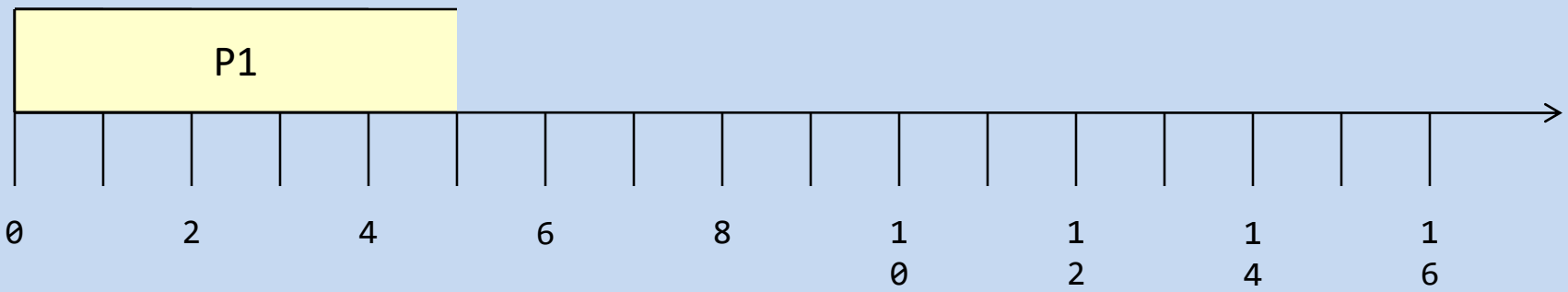
First-come, first-served scheduling

- A short summary:
  - FIFO scheduling is sensitive to the input.
  - The average waiting time is often long. Think about the scenario (convoy effect):
    - Someone is standing before you in the queue in KFC, and
    - you find that he/she is ordering the **bucket chicken meal** (P1 in example 1)!!!!
    - So, two people (P2 and P3) are unhappy while only P1 is happy.
  - Can we do something about this?

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

Non-preemptive SJF



Time = 5

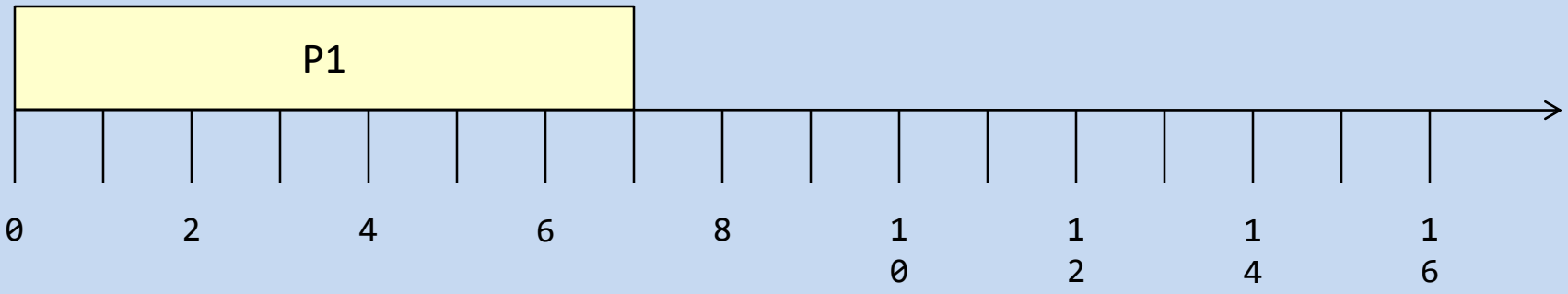
Not allow preemption

Set of processes

P1 **P2** **P3** **P4**

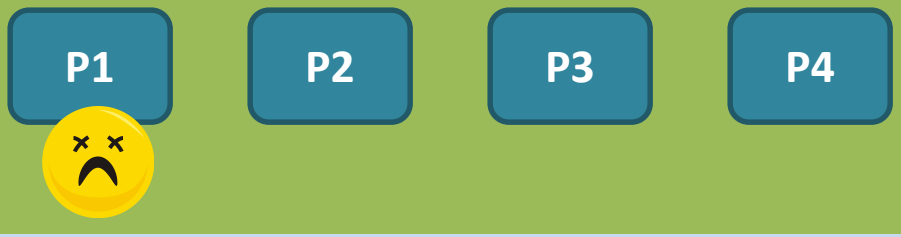
Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF



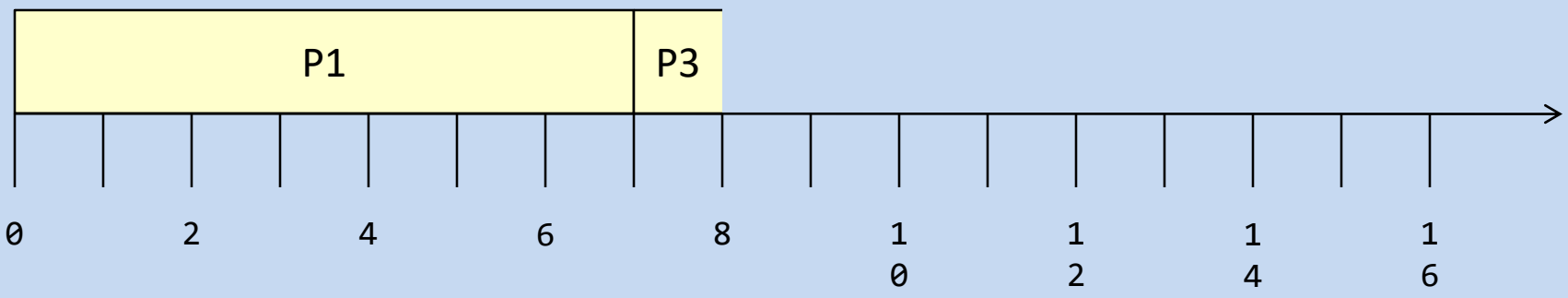
Time = 7

Set of processes



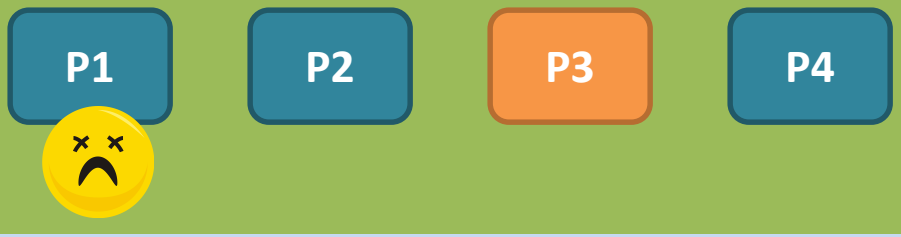
Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF



Time = 7

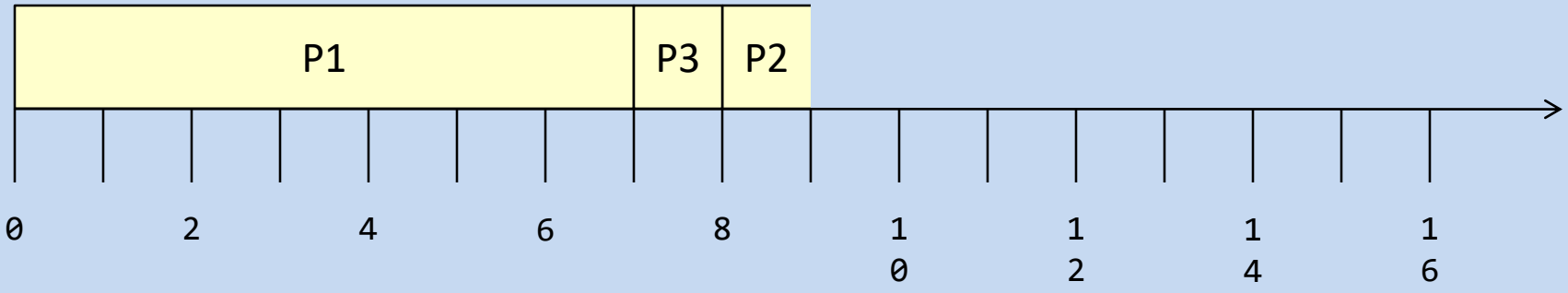
Set of processes



Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

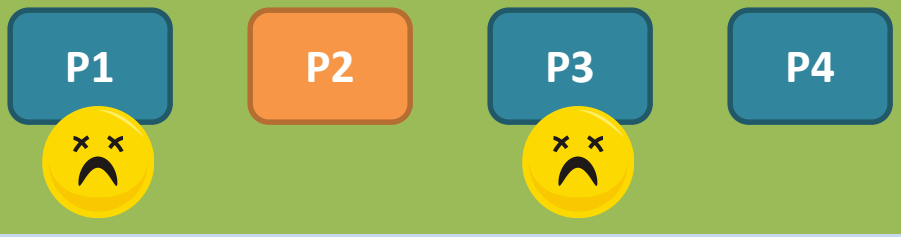
Non-preemptive SJF

In this example, we use **FIFO** to break the tie.



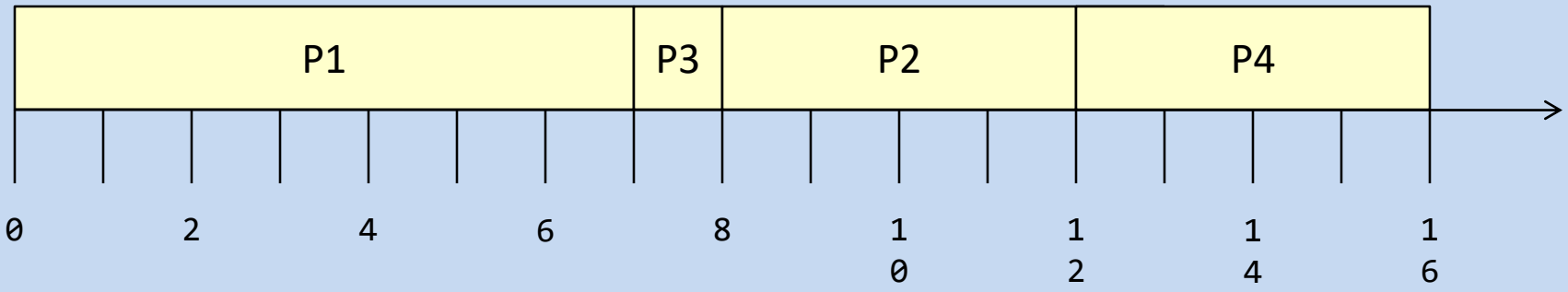
Time = 8

Set of processes



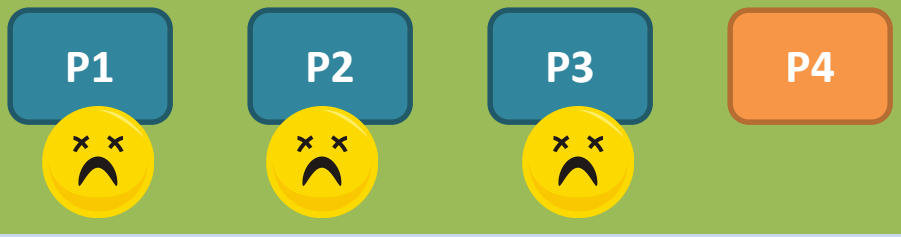
Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF



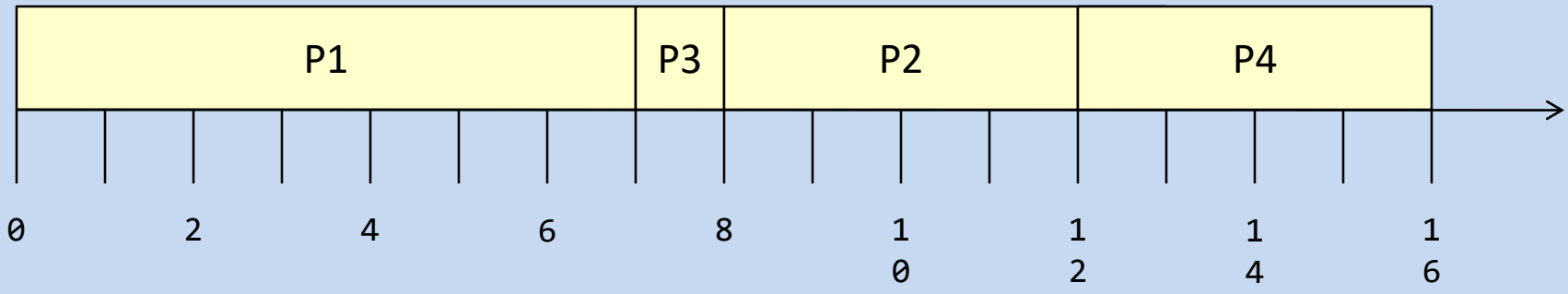
Time = 16

Set of processes



Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF



Waiting time:

P1 = 0; P2 = 6; P3 = 3; P4 = 7;

Average = (0 + 6 + 3 + 7) / 4 = 4.

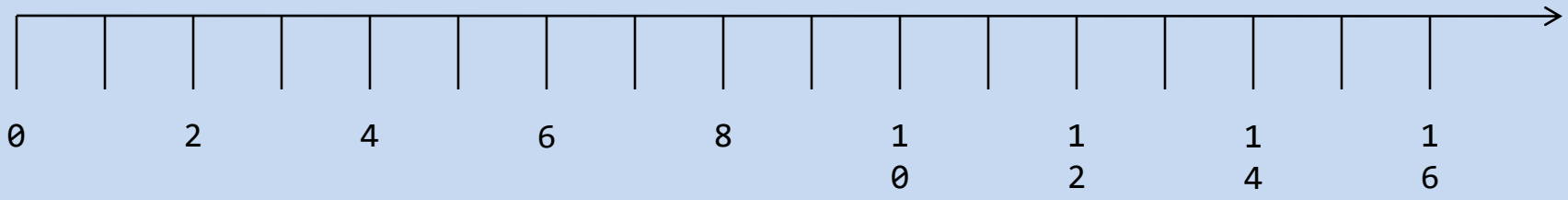
Turnaround time:

P1 = 7; P2 = 10; P3 = 4; P4 = 11;

Average = (7 + 10 + 4 + 11) / 4 = 8.

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Preemptive SJF



Rules for preemptive scheduling
(for this example only)

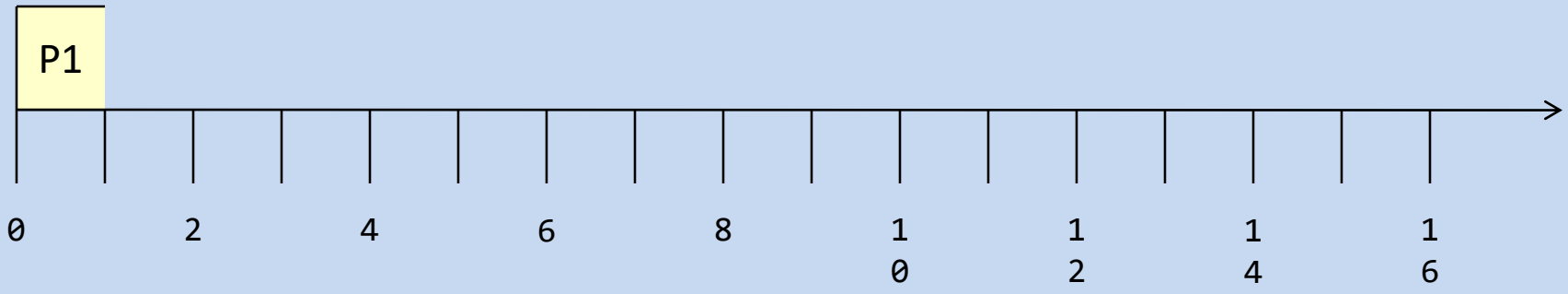
-Preemption happens when a new process arrives at the system.

-Then, the scheduler steps in and selects the next task based on **their remaining CPU requirements**.

Shortest-remaining-time-first

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Preemptive SJF



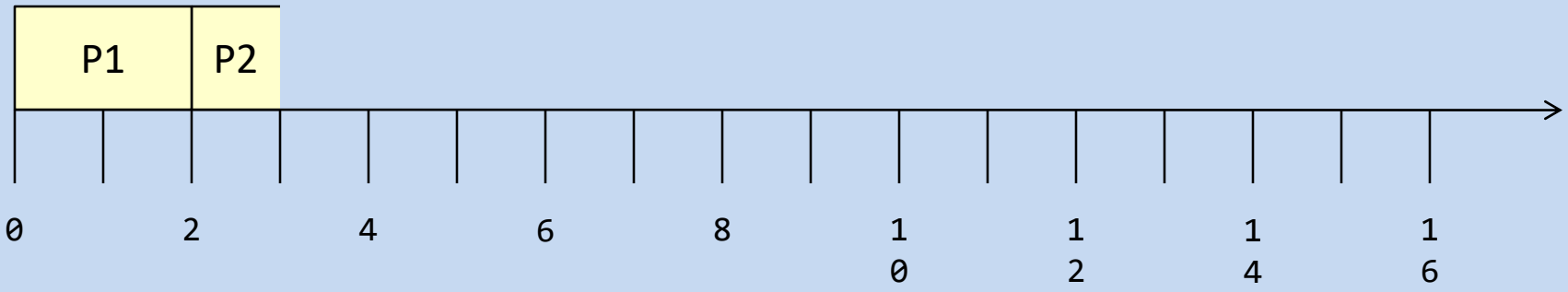
Time = 0

Set of processes

P1

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Preemptive SJF



Time = 2

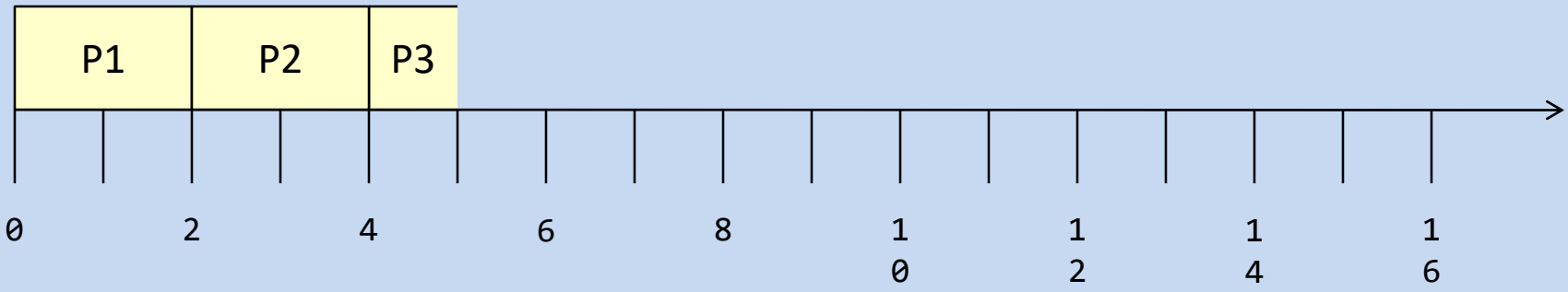
P2 is selected!

Set of processes

P1 **P2**

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	4
P3	4	1	1
P4	5	4	4

Preemptive SJF



Time = 4

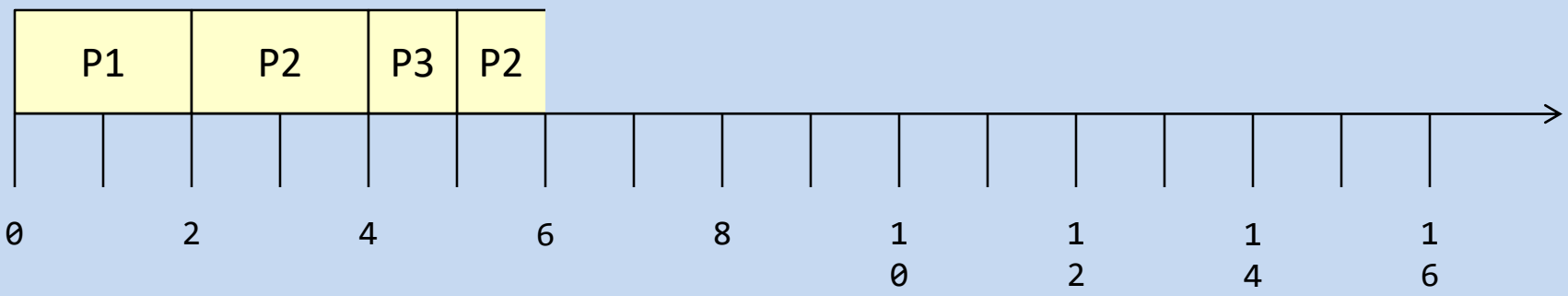
P3 is selected!

Set of processes

P1 **P2** **P3**

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	1
P4	5	4	4

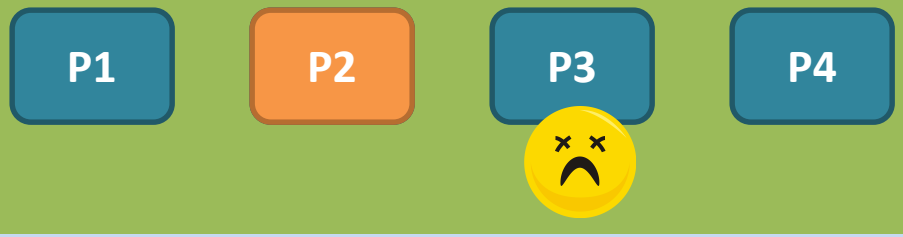
Preemptive SJF



Time = 5

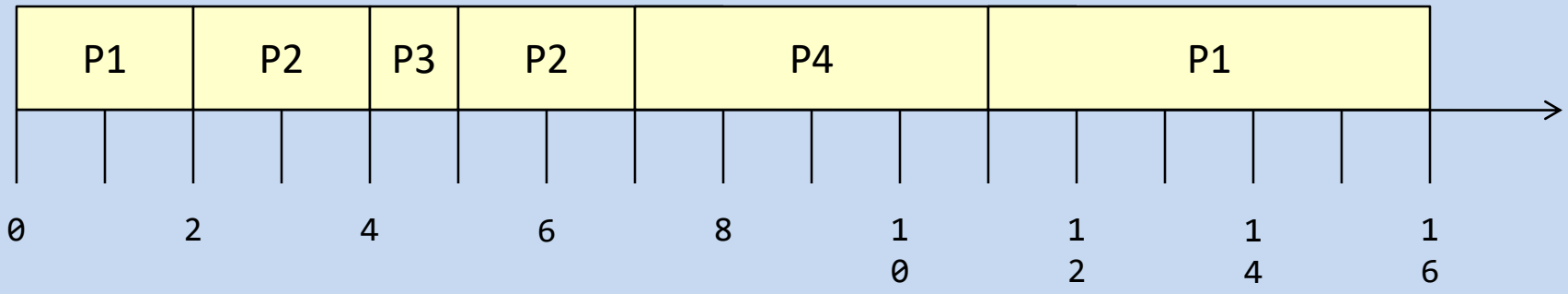
P2 is selected!

Set of processes



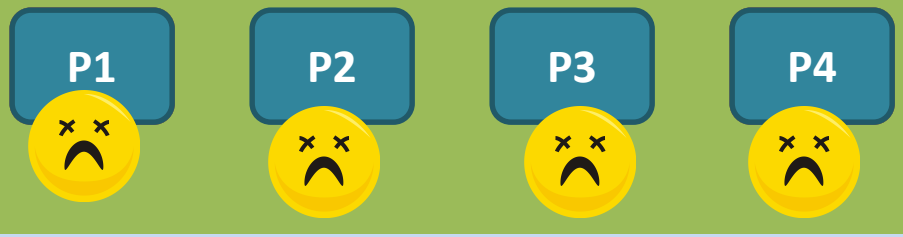
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	4

Preemptive SJF



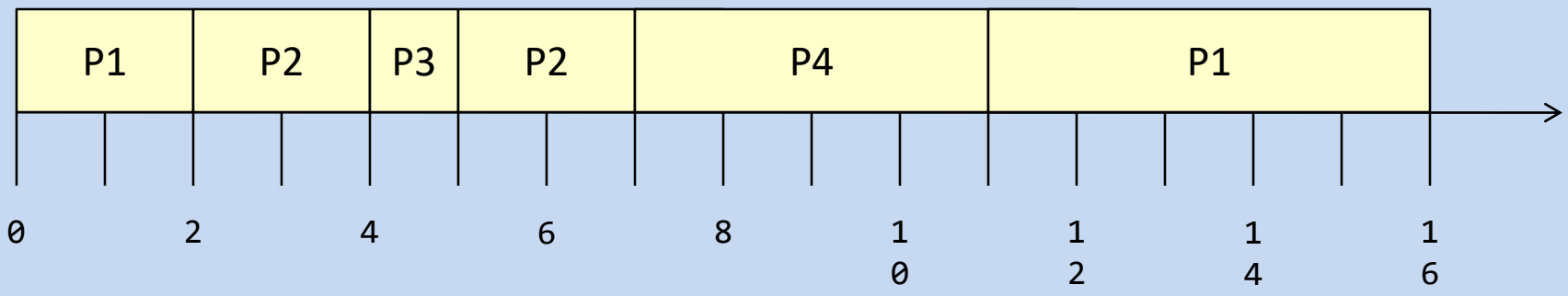
Time = 16

Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

Preemptive SJF



Waiting time:

P1 = 9; P2 = 1; P3 = 0; P4 = 2;

Average = (9 + 1 + 0 + 2) / 4 = 3.

Turnaround time:

P1 = 16; P2 = 5; P3 = 1; P4 = 6;

Average = (16 + 5 + 1 + 6) / 4 = 7.

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

SJF: Short summary

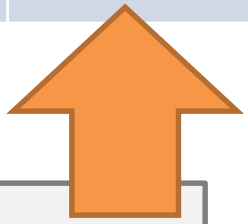
	Non-preemptive SJF	Preemptive SJF
Average waiting time	4	**3 (smallest)**
Average turnaround time	8	**7 (smallest)**
# of context switching	**3 (smallest)**	5

The waiting time and the turnaround time decrease at the expense of the **increased number of context switching**.

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

SJF: Short summary

	Non-preemptive SJF	Preemptive SJF
Average waiting time	4	**3 (smallest)**
Average turnaround time	8	**7 (smallest)**
# of context switching	**3 (smallest)**	5



SJF is provably optimal in that it gives the **minimum average waiting time**

Challenge: How to know the length of the next CPU request?

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

SJF: Short summary

Challenge: How to know the length of the next CPU request?

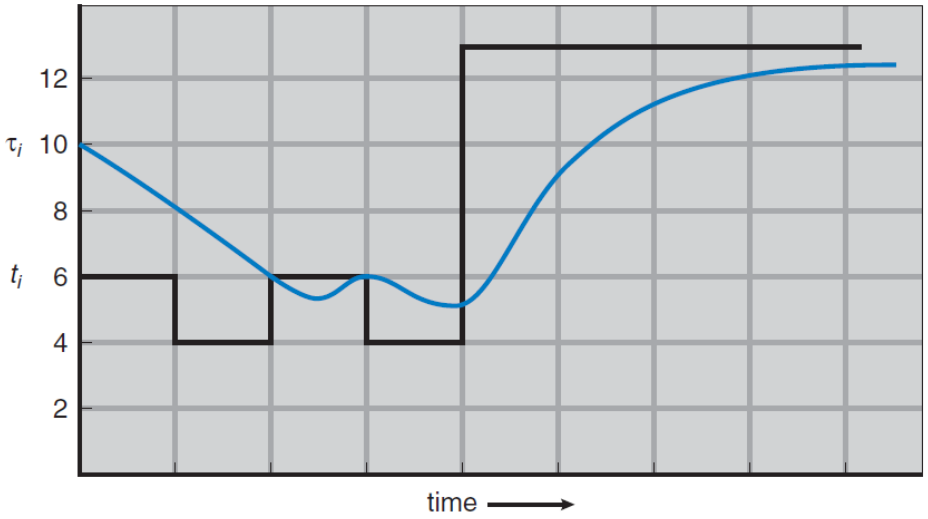
Solution: Prediction (by expecting that the next CPU burst will be similar in length to the previous ones)

General approach
exponential average

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

Predicted value

Most recent information



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

Round-robin

- Round-Robin (RR) scheduling is preemptive.
  - Every process is given a **quantum**, or the amount of time allowed to execute.
  - When the quantum of a process is **used up** (i.e., 0), the process releases the CPU and **this is the preemption**.
  - Then, the scheduler steps in and it chooses **the next process which has a non-zero quantum** to run.

- Processes are running one-by-one, like a circular queue.
  - Designed specially for time-sharing systems

Round-robin

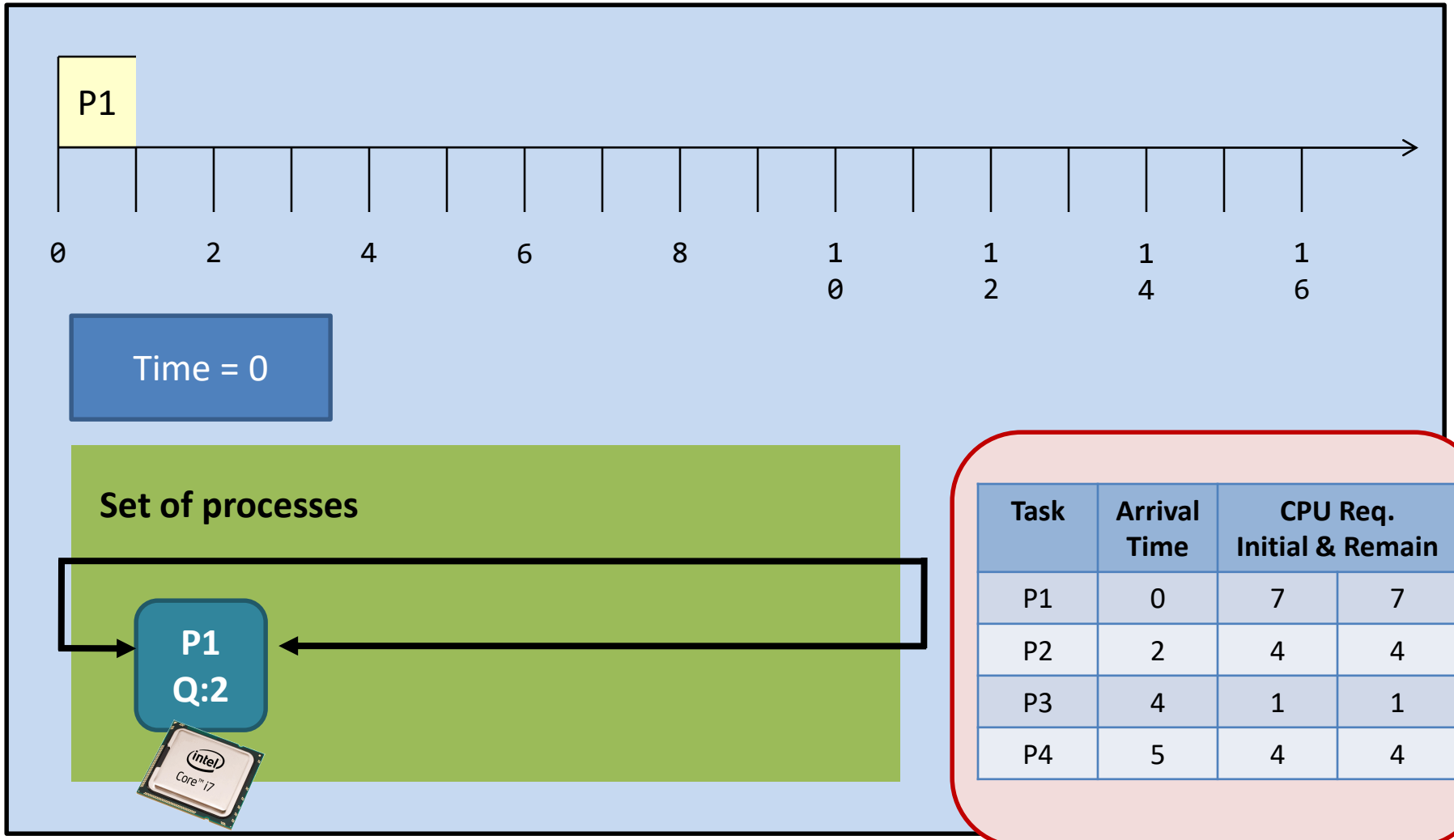
Rules for Round-Robin
(for this example only)

-The quantum of every process is fixed and is 2 units.

-The process queue is sorted according the processes' arrival time, in an ascending order.
(This rule allows us to break tie.)

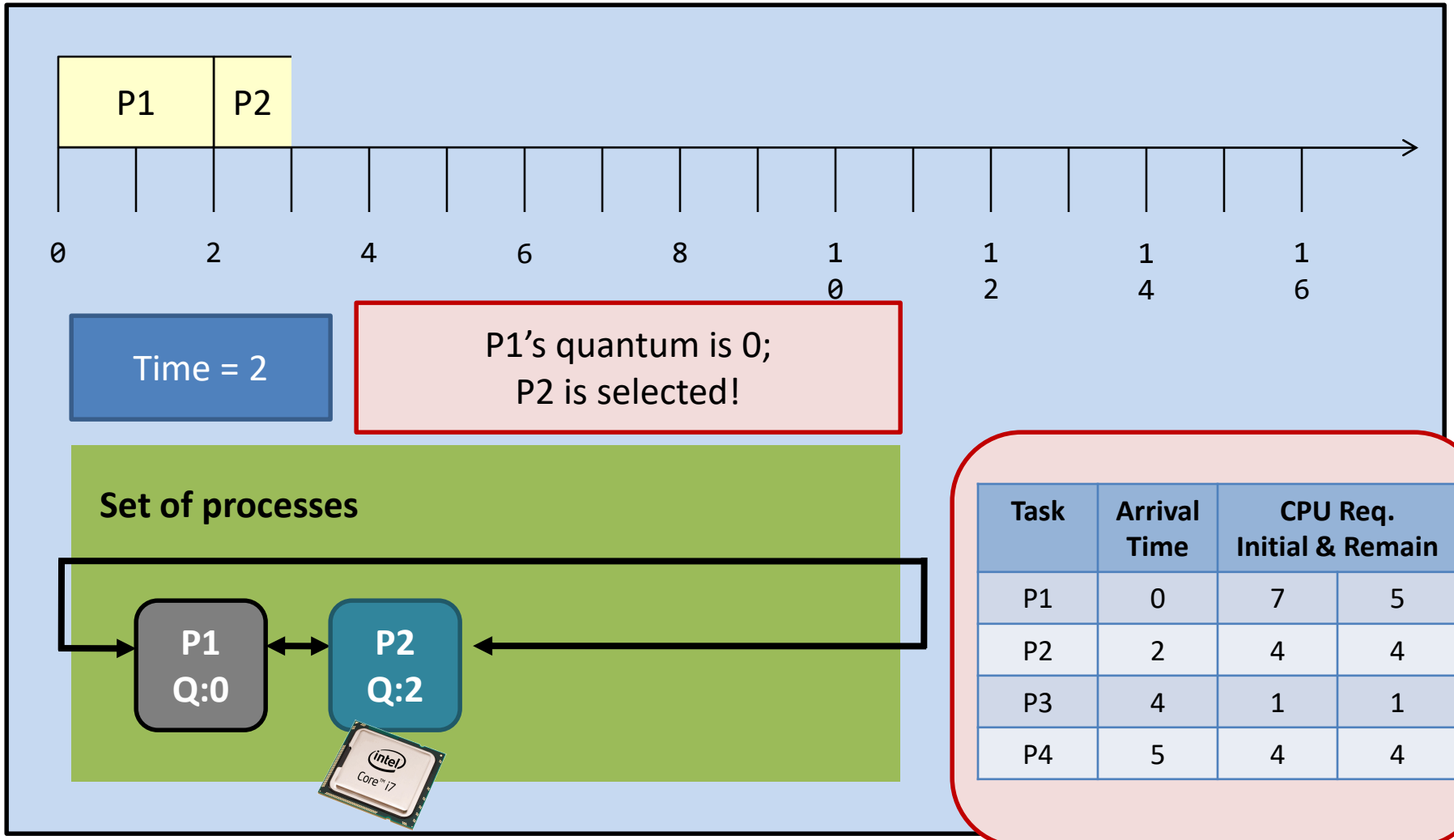
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round-robin



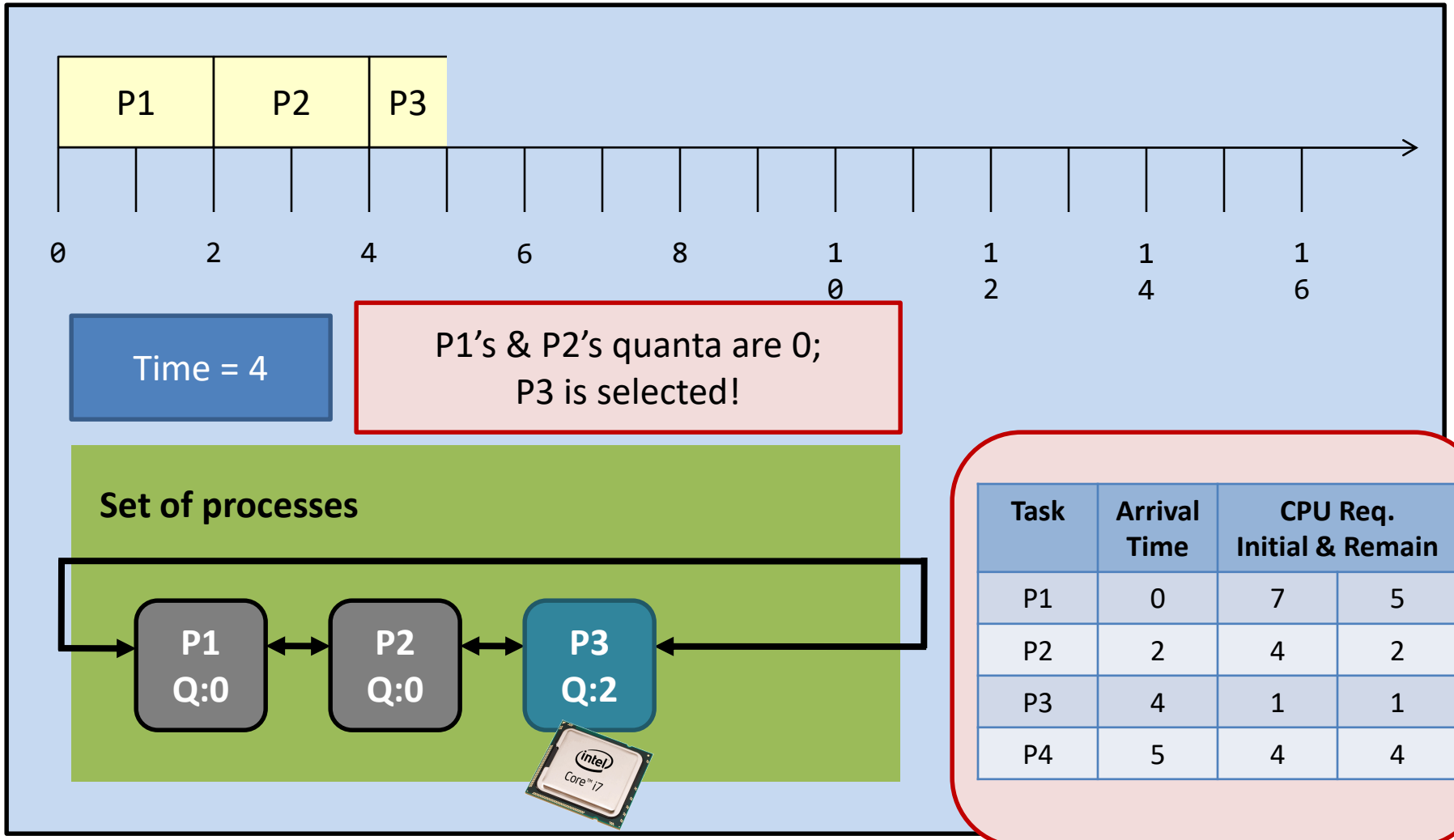
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round-robin



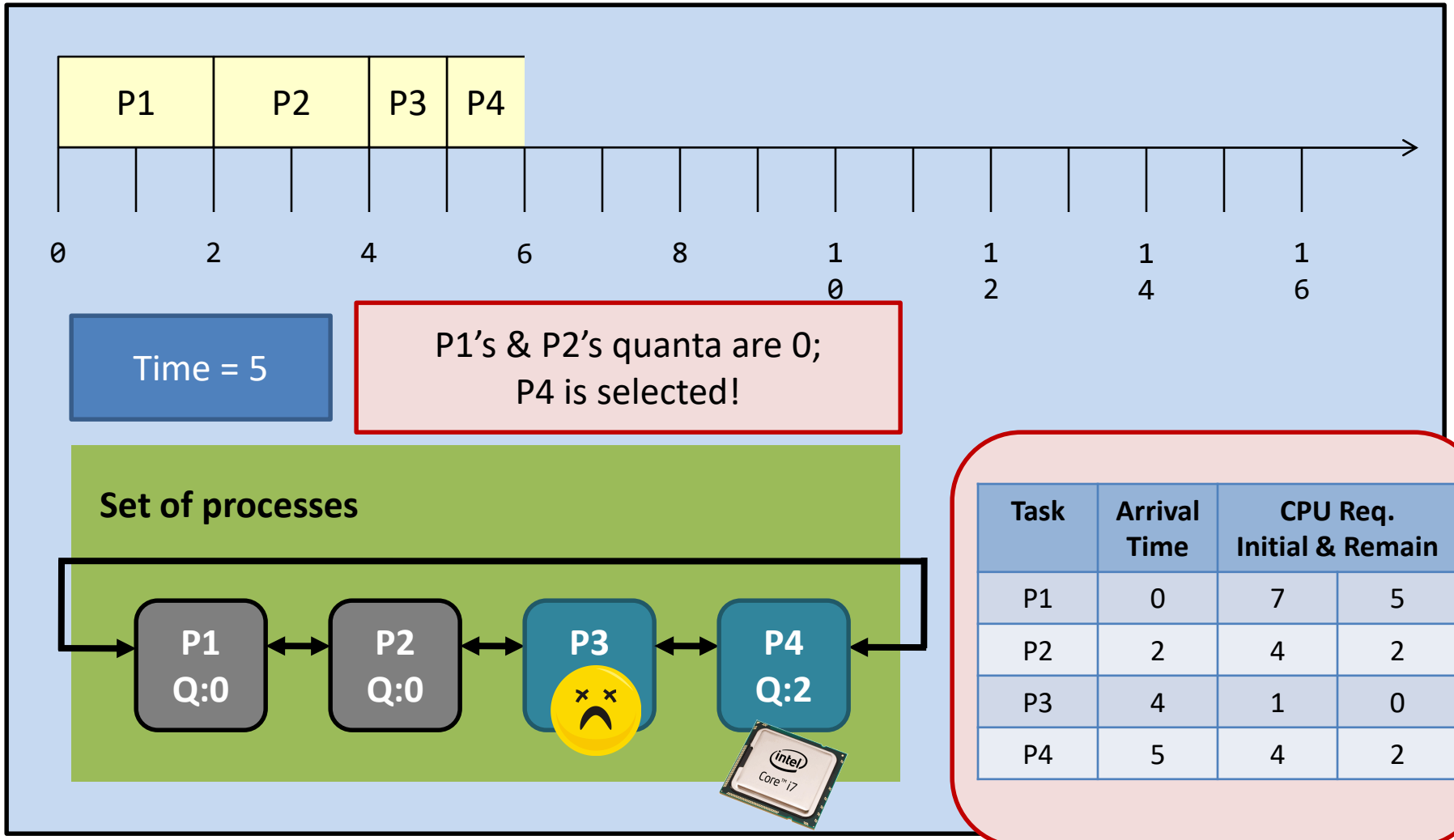
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round-robin



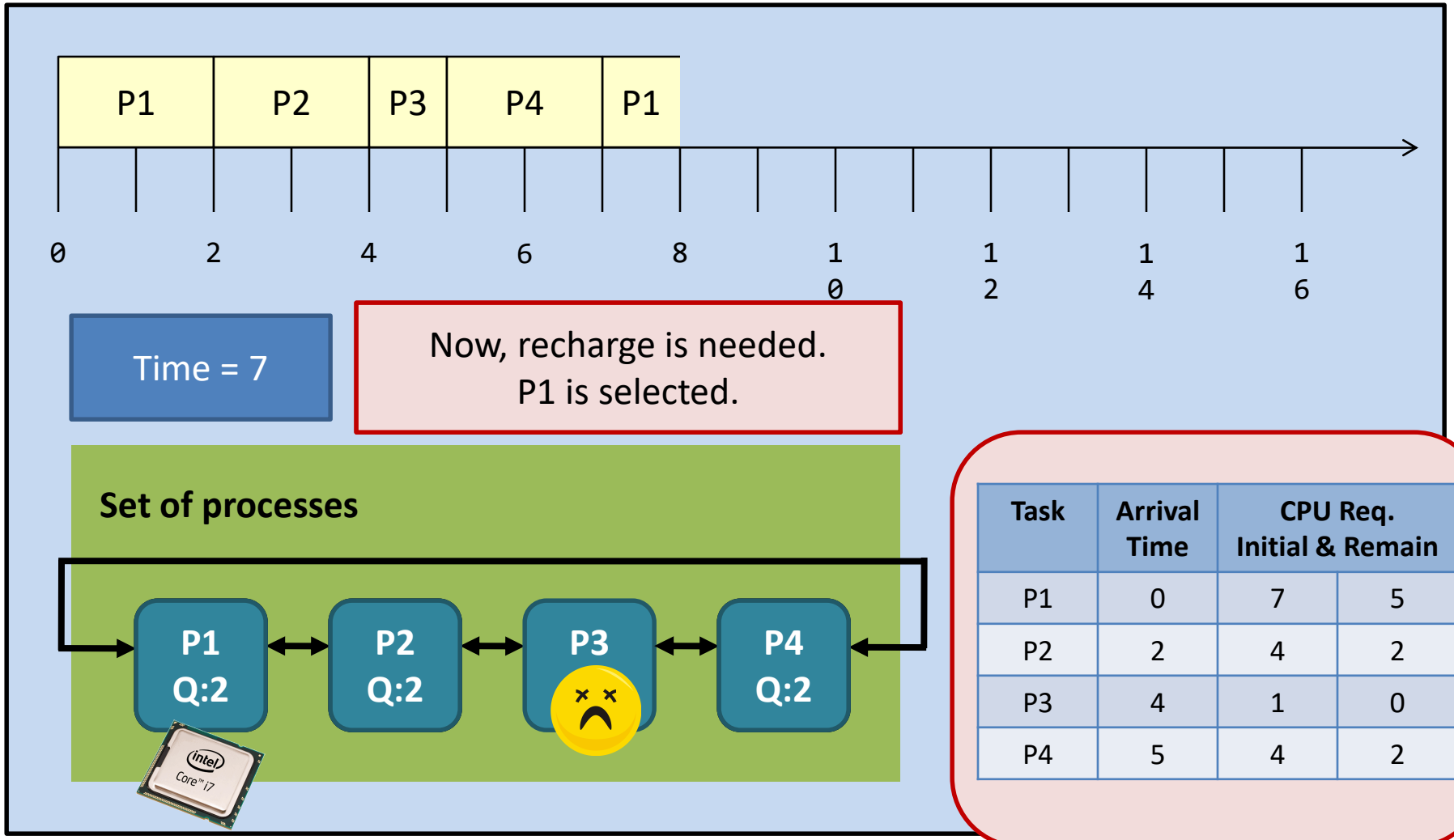
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	1
P4	5	4	4

Round-robin



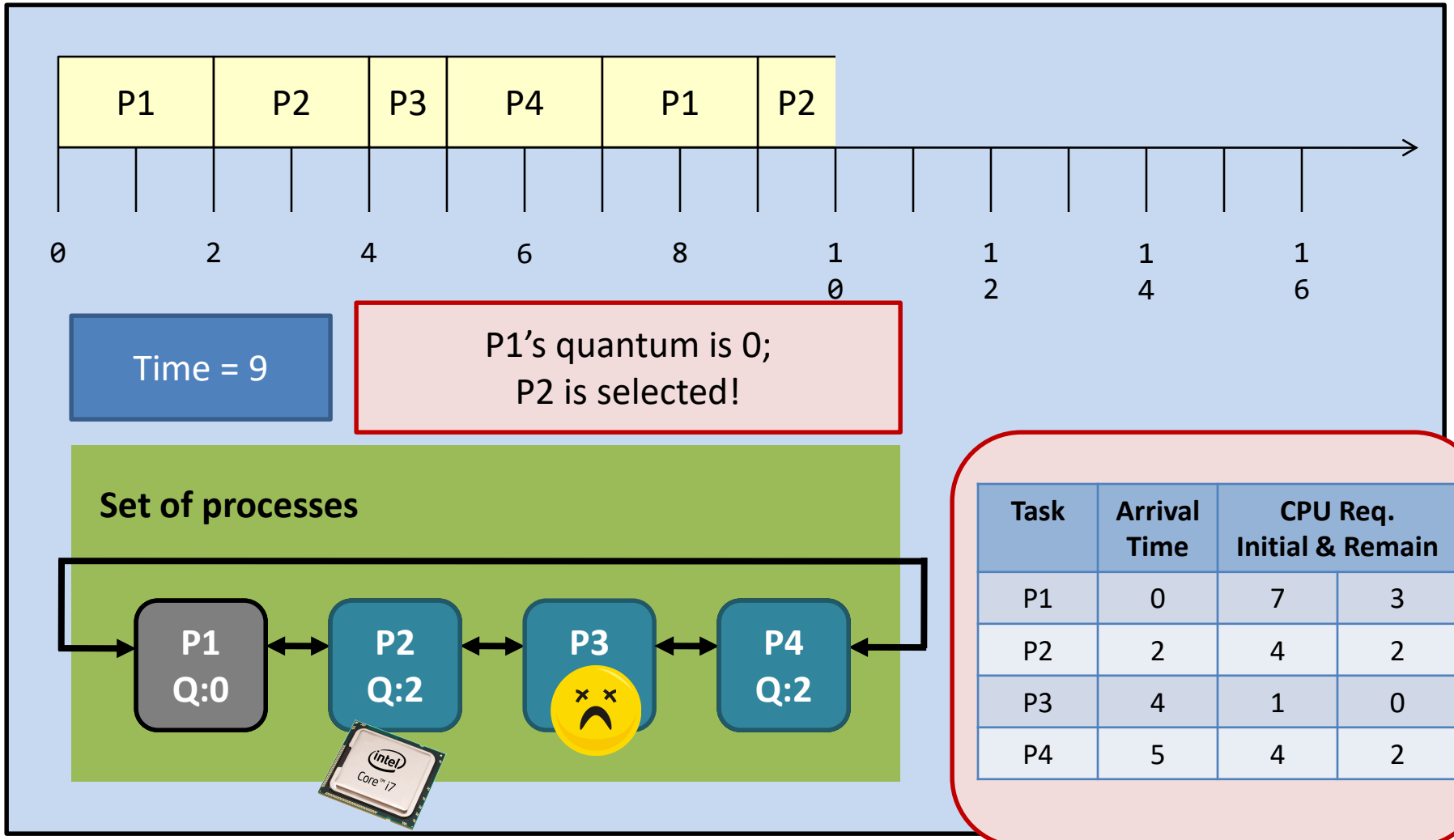
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	2

Round-robin



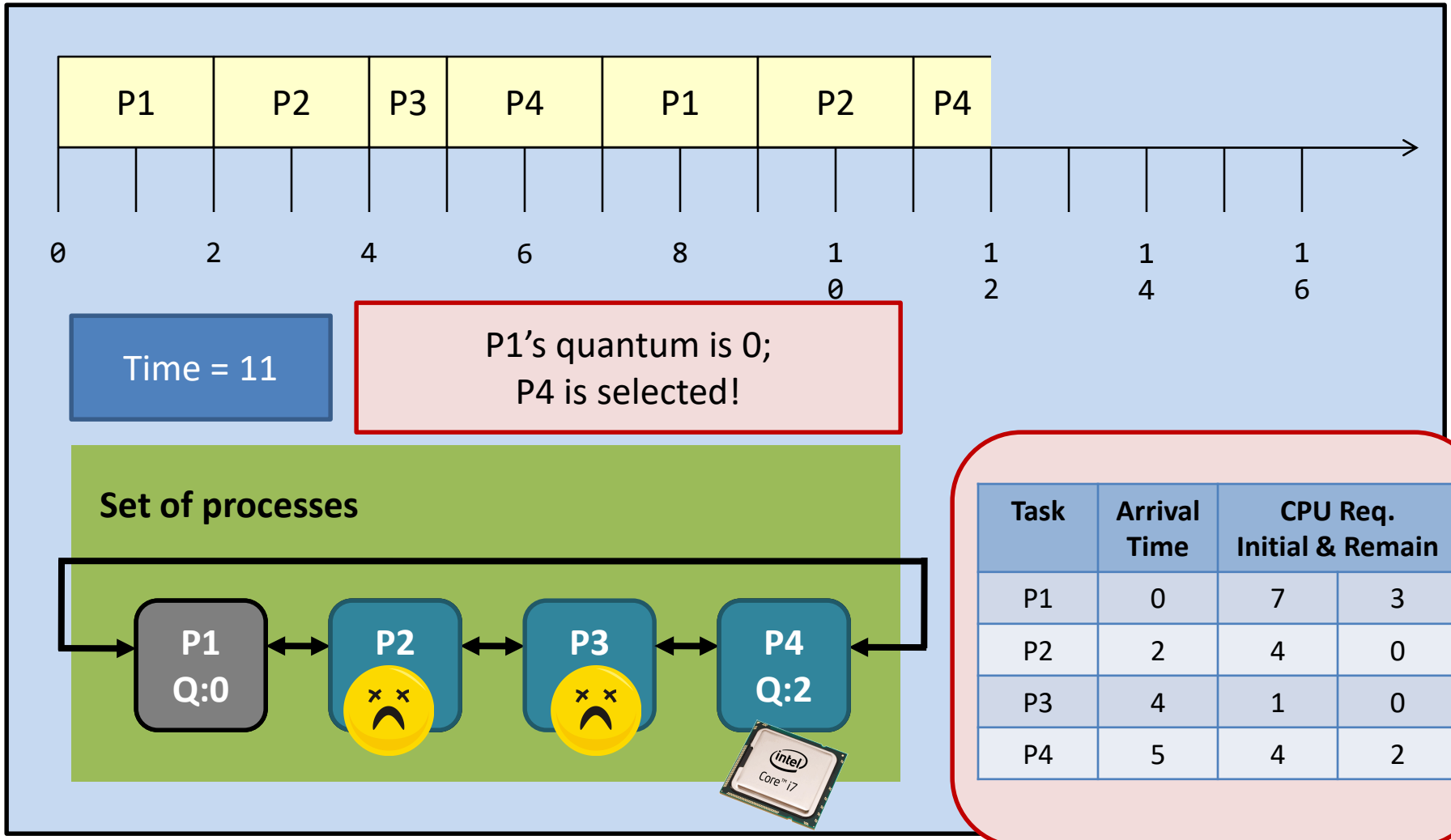
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	2

Round-robin



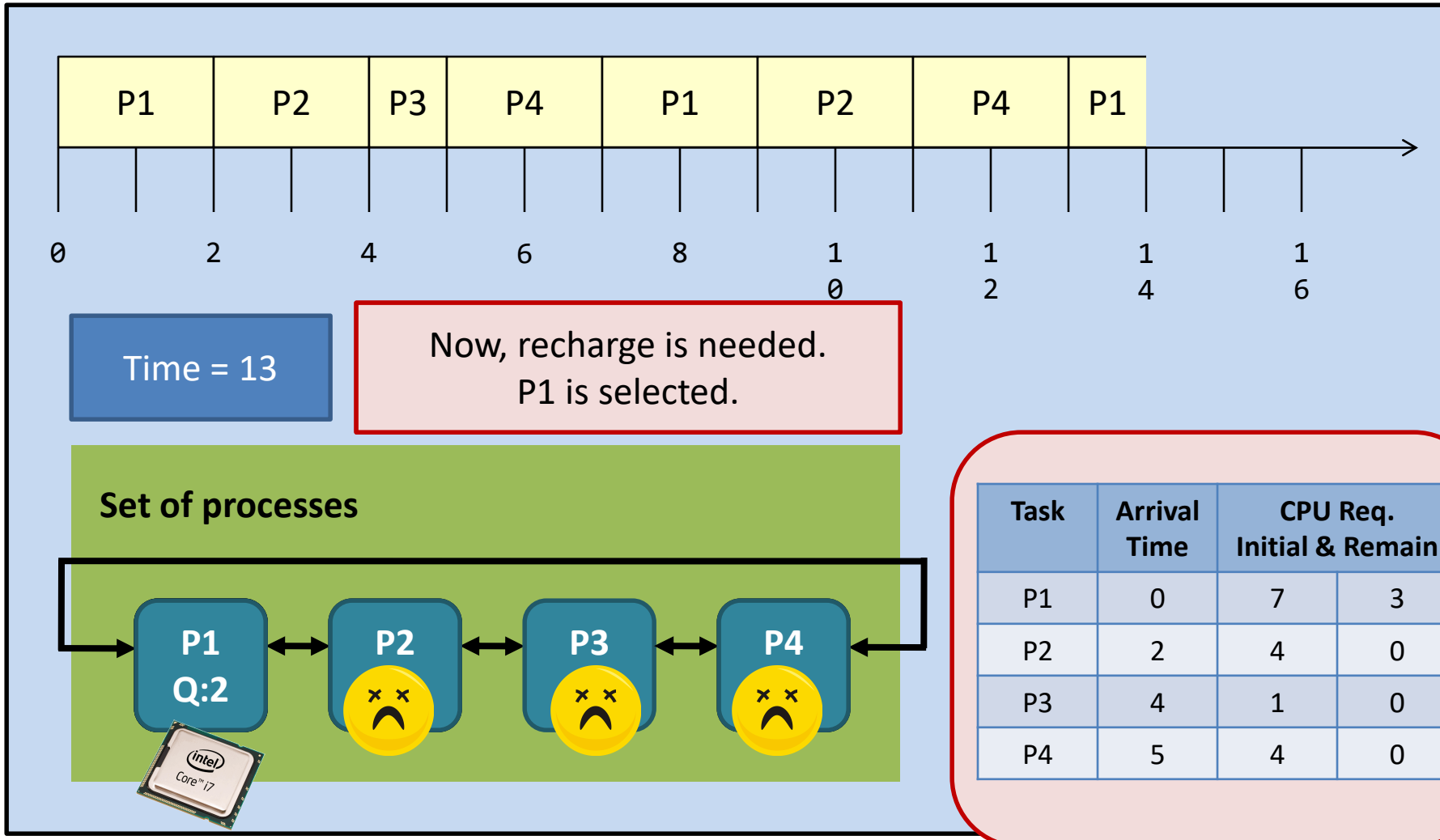
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	2
P3	4	1	0
P4	5	4	2

Round-robin



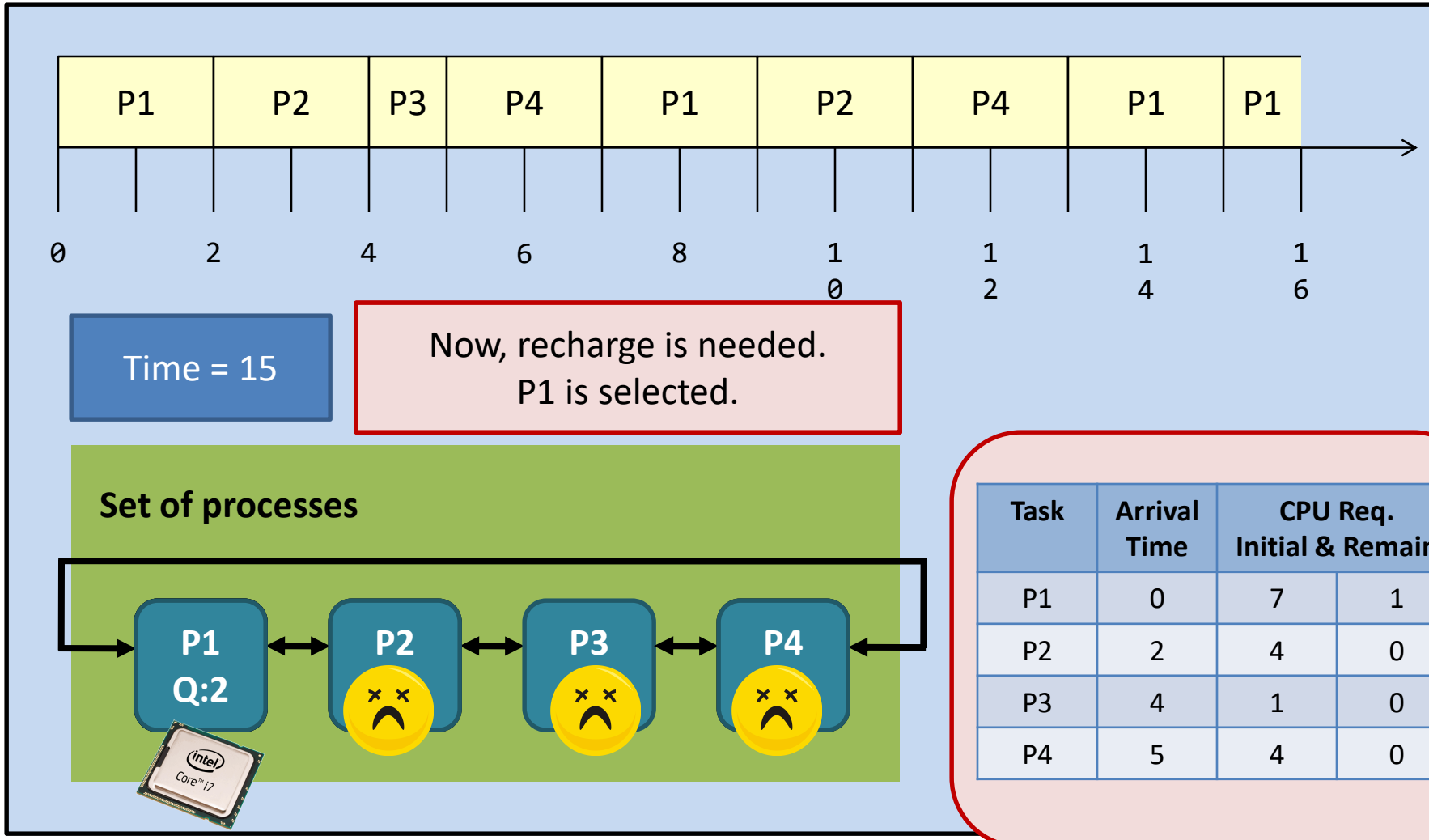
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	0
P3	4	1	0
P4	5	4	2

Round-robin



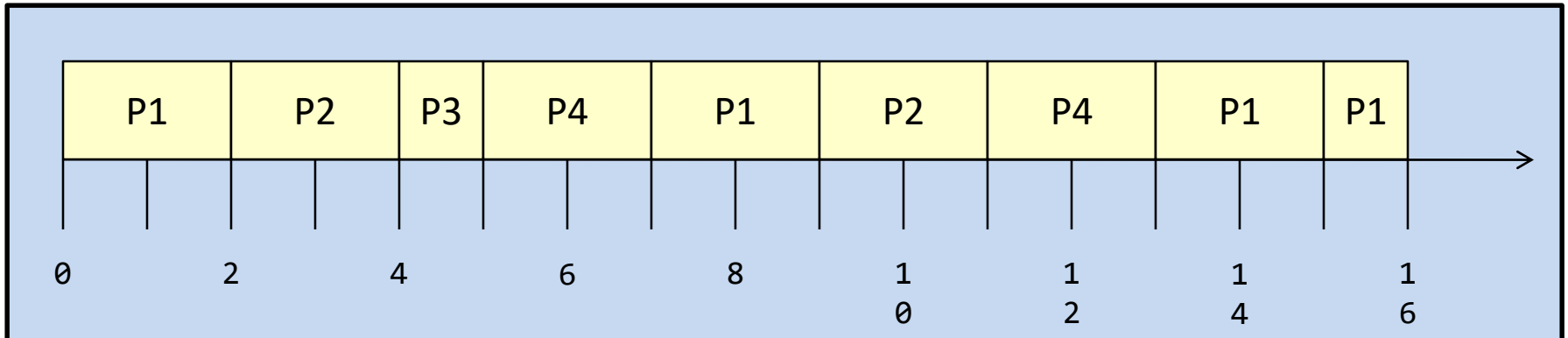
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	3
P2	2	4	0
P3	4	1	0
P4	5	4	0

Round-robin



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	1
P2	2	4	0
P3	4	1	0
P4	5	4	0

Round-robin



Waiting time:

P1 = 9; P2 = 5; P3 = 0; P4 = 4;

Average = (9 + 5 + 0 + 4) / 4 = 4.5

Turnaround time:

P1 = 16; P2 = 9; P3 = 1; P4 = 8;

Average = (16 + 9 + 1 + 8) / 4 = 8.5

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

RR VS SJF

	Non-preemptive SJF	Preemptive SJF	RR
Average waiting time	4	**3**	**4.5 (largest)**
Average turnaround time	8	**7**	**8.5 (largest)**
# of context switching	**3**	5	**8 (largest)**

So, the RR algorithm gets all the bad! Why do we still need it?

The responsiveness of the processes is great under the RR algorithm. E.g., you won't feel a job is "frozen" because every job is on the CPU from time to time!

Round-robin

Issue for Round-Robin

-How to set the size of the time quantum?

-Too large: FCFS

-Too small: frequent context switch

-In practice: 10-100ms

-A rule of thumb: 80% CPU burst should be shorter than the time quantum

Observations on RR

- Modified versions of round-robin are implemented in (nearly) every modern OS.
  - Users run a lot of **interactive jobs** on modern OS-es.
  - Users' priority list:
    - Number one - Responsiveness;
    - Number two - Efficiency;
    - In other words, "ordinary users" expect a fast GUI response than an efficient scheduler running behind.

- With the round-robin deployed, the scheduling **looks like random**.
  - It also looks like "***fair to all processes***".

Different algorithms

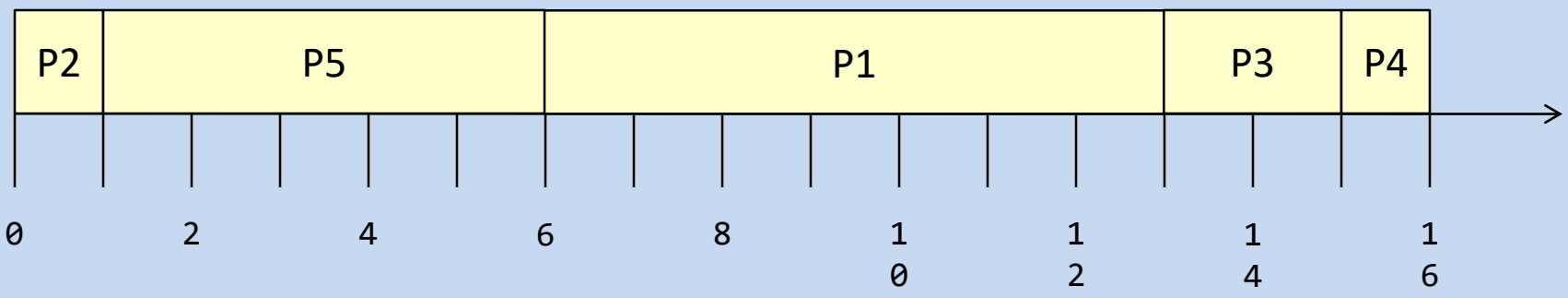
Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

Priority Scheduling

- Some basics:
  - A task is given a priority (and is usually an integer).
  - A scheduler selects the next process based on the priority.
    - ***A typical practice***: the highest priority is always chosen.
  - Special case: SJF, FCFS (equal priority)

- How to define priority
  - Internally: time limits, memory requirements, number of open files, CPU burst and I/O burst…
  - Externally: process importance, paid funds…

Priority Scheduling



Assumption:

-All arrive at time 0
-Low numbers represent high priority

Problem: Indefinite blocking or starvation

Solution: Aging (gradually increase the priority of waiting processes)

Task	CPU Burst	Priority
P1	7	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

Multilevel queue scheduling

- **Definitions.**
  - It is still a priority scheduler.
  - But, at each priority class, **different schedulers** may be deployed.
  - Eg: Foreground processes and background processes

Priority class 5	Non-preemptive, FIFO
Priority class 4	Non-preemptive, SJF
Priority class 3	RR with quantum = 10 units.
Priority class 2	RR with quantum = 20 units.
Priority class 1	RR with quantum = 40 units.

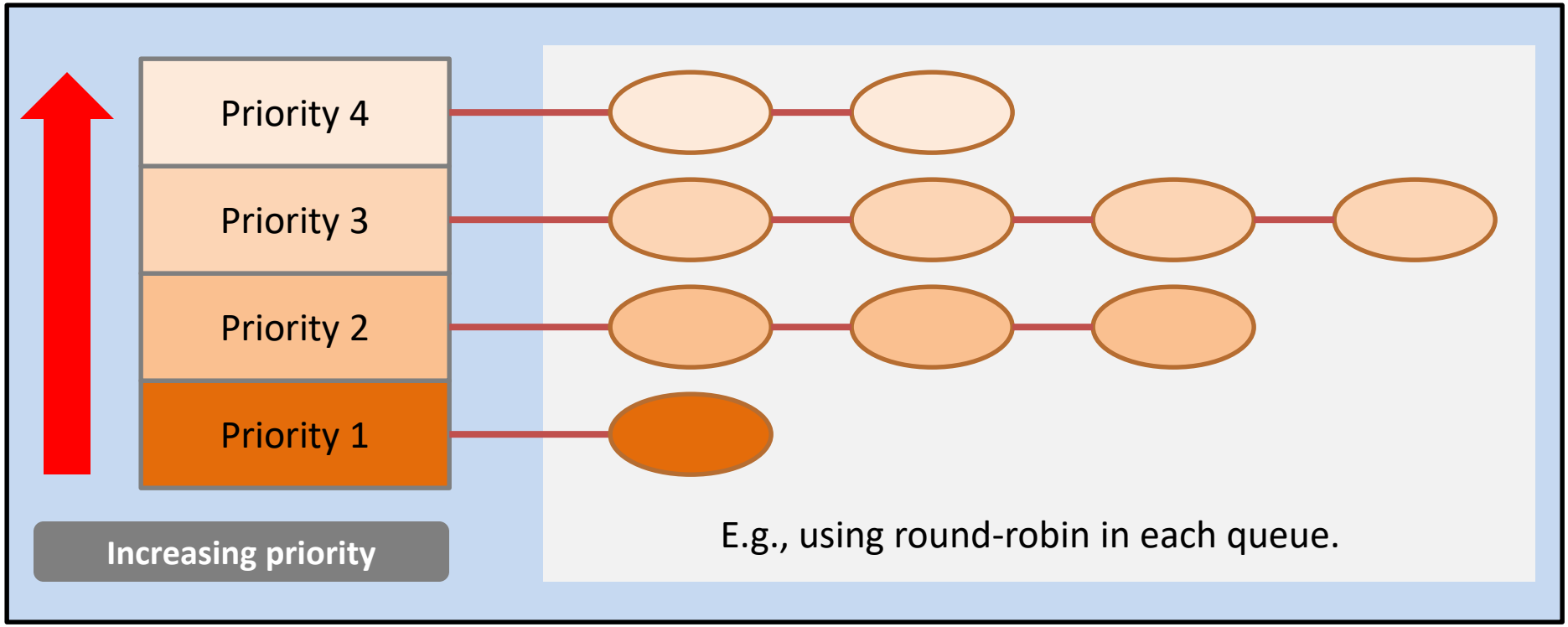
Just an example.

The processes are **permanently assigned** to one queue

Fixed-priority preemptive scheduling among queues

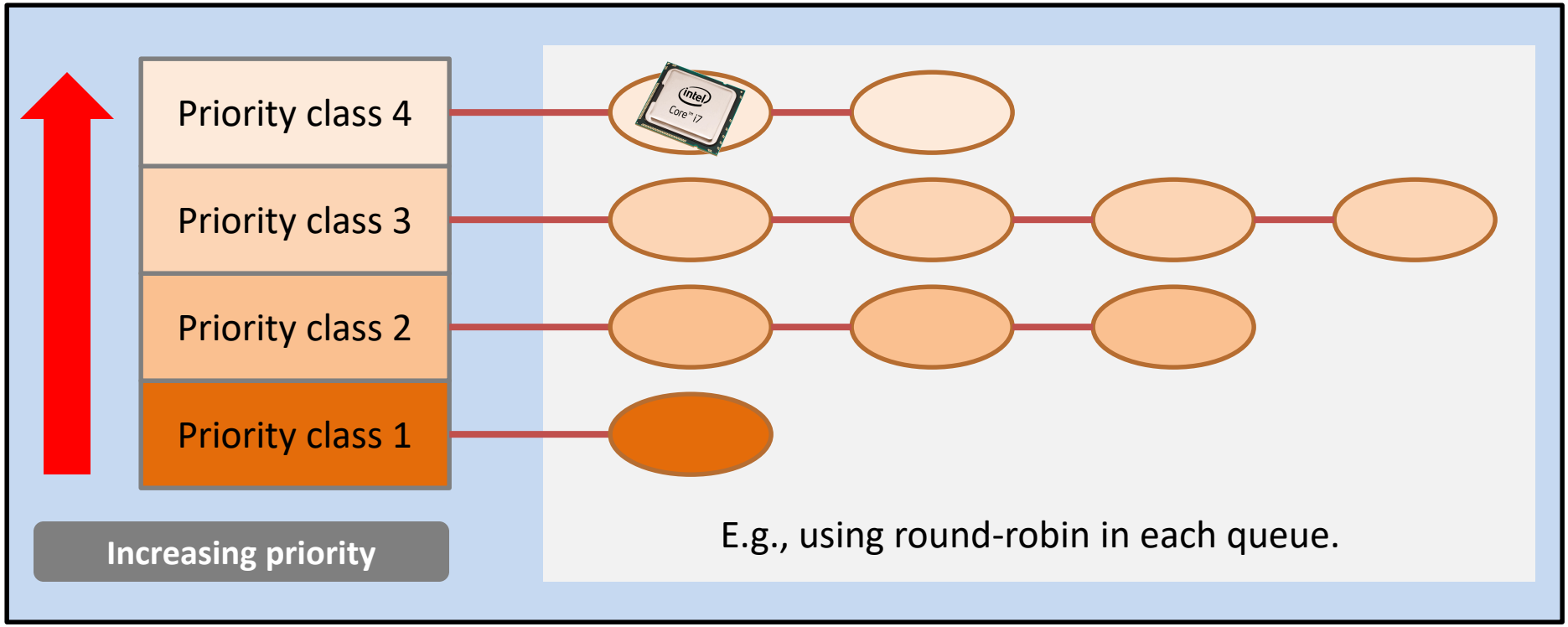
Multilevel queue scheduling– an example

- **Properties**: process is assigned a fix priority when they are submitted to the system.



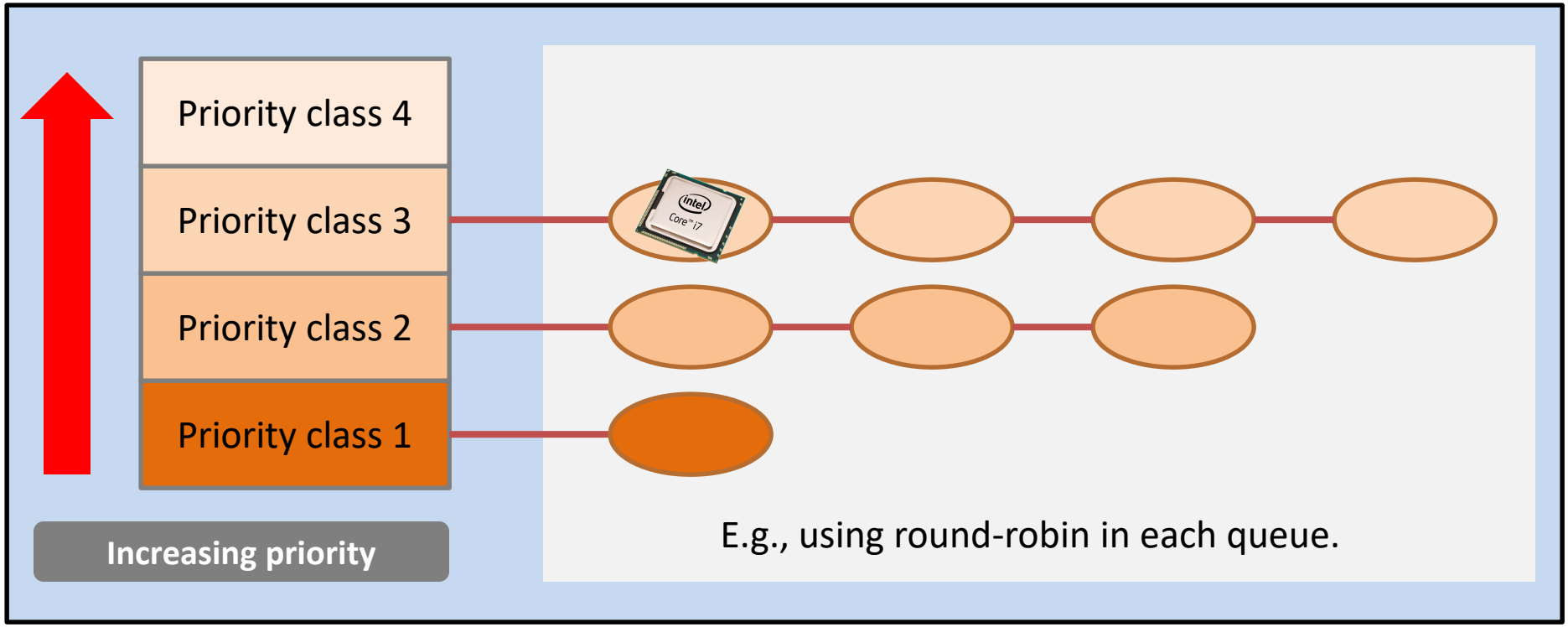
Multilevel queue scheduling– an example

- The highest priority class will be selected.
  - To prevent high-priority tasks from running indefinitely.
  - The tasks with a higher priority should be short-lived, but important;



Multilevel queue scheduling– an example

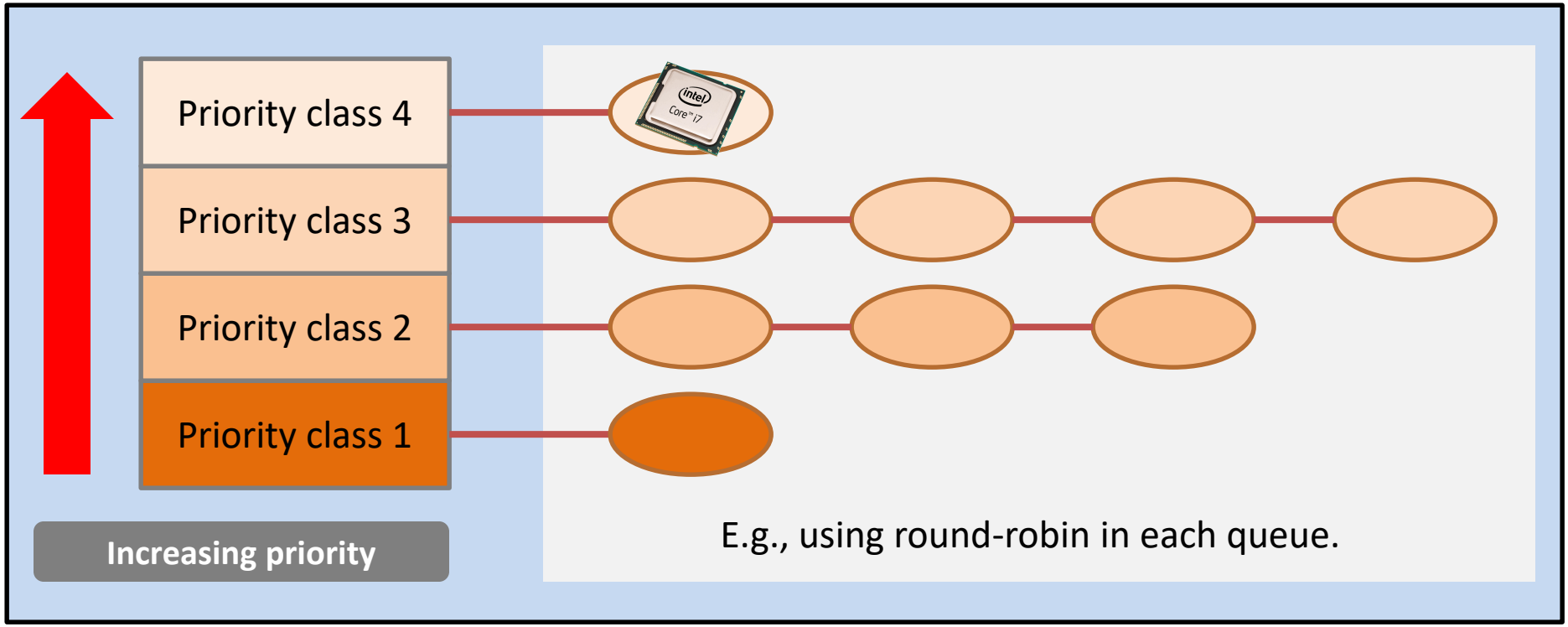
- Lower priority classes will be scheduled only when the upper priority classes has no tasks.



Multilevel queue scheduling– an example

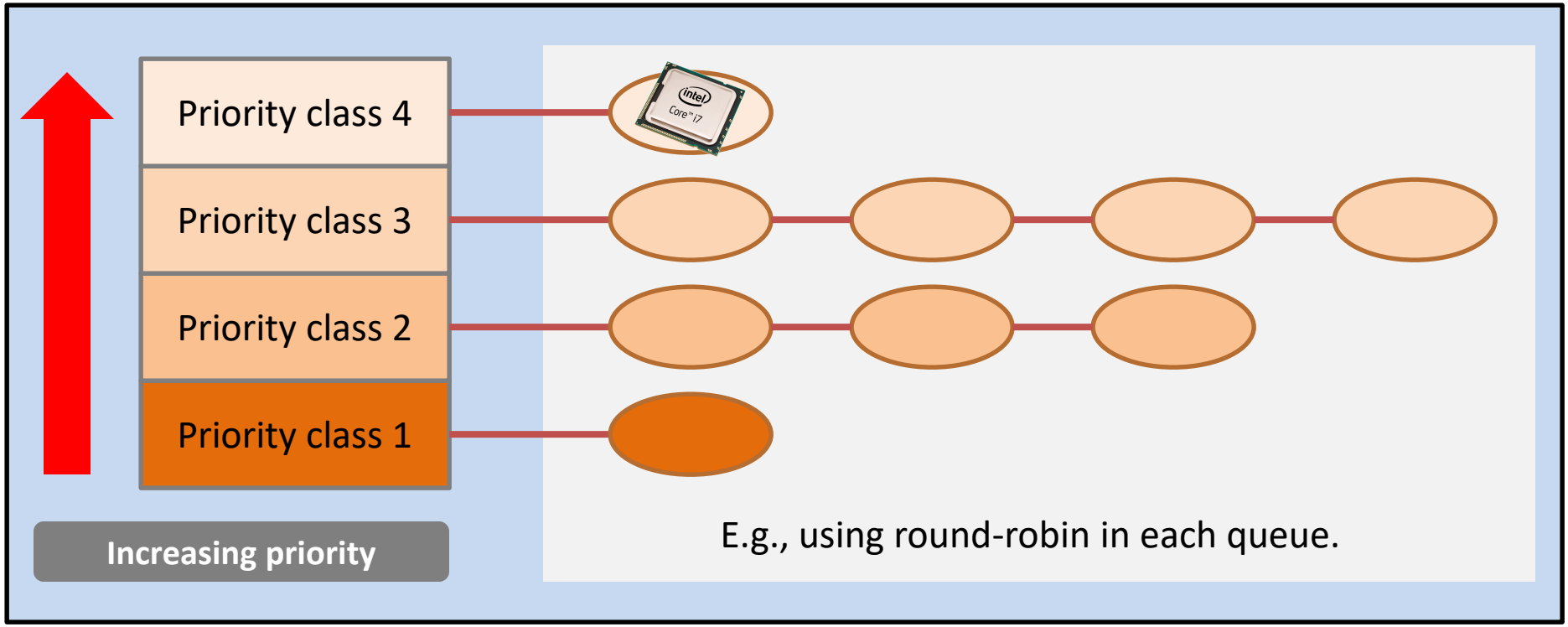
- Of course, it is a good design to have a high-priority task preempting a low-priority task.

  (conditioned that the high-priority task is short-lived.)



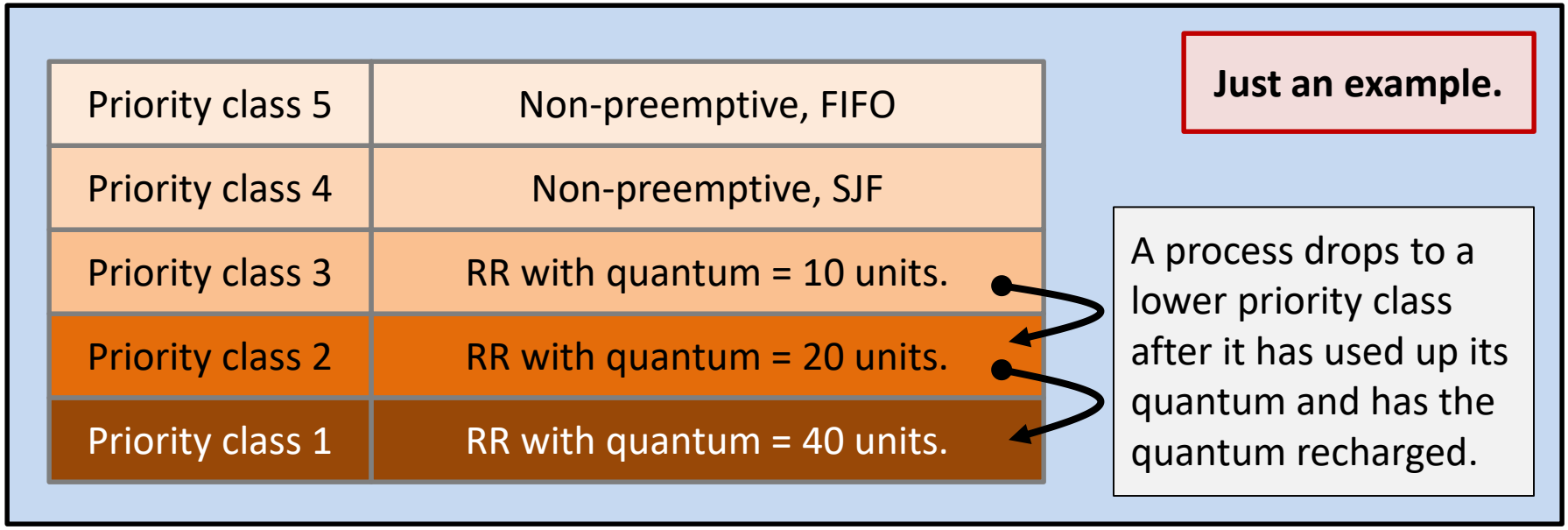
Multilevel queue scheduling– an example

- Any problem?
  - Fixed priority
  - Indefinite blocking or starvation



Multilevel feedback queue scheduling

- **How to improve the previous scheme?**
  - Allows a process to move between queues (dynamic priority).
  - Why needed?
    - Eg.: Separate processes according to their CPU bursts.

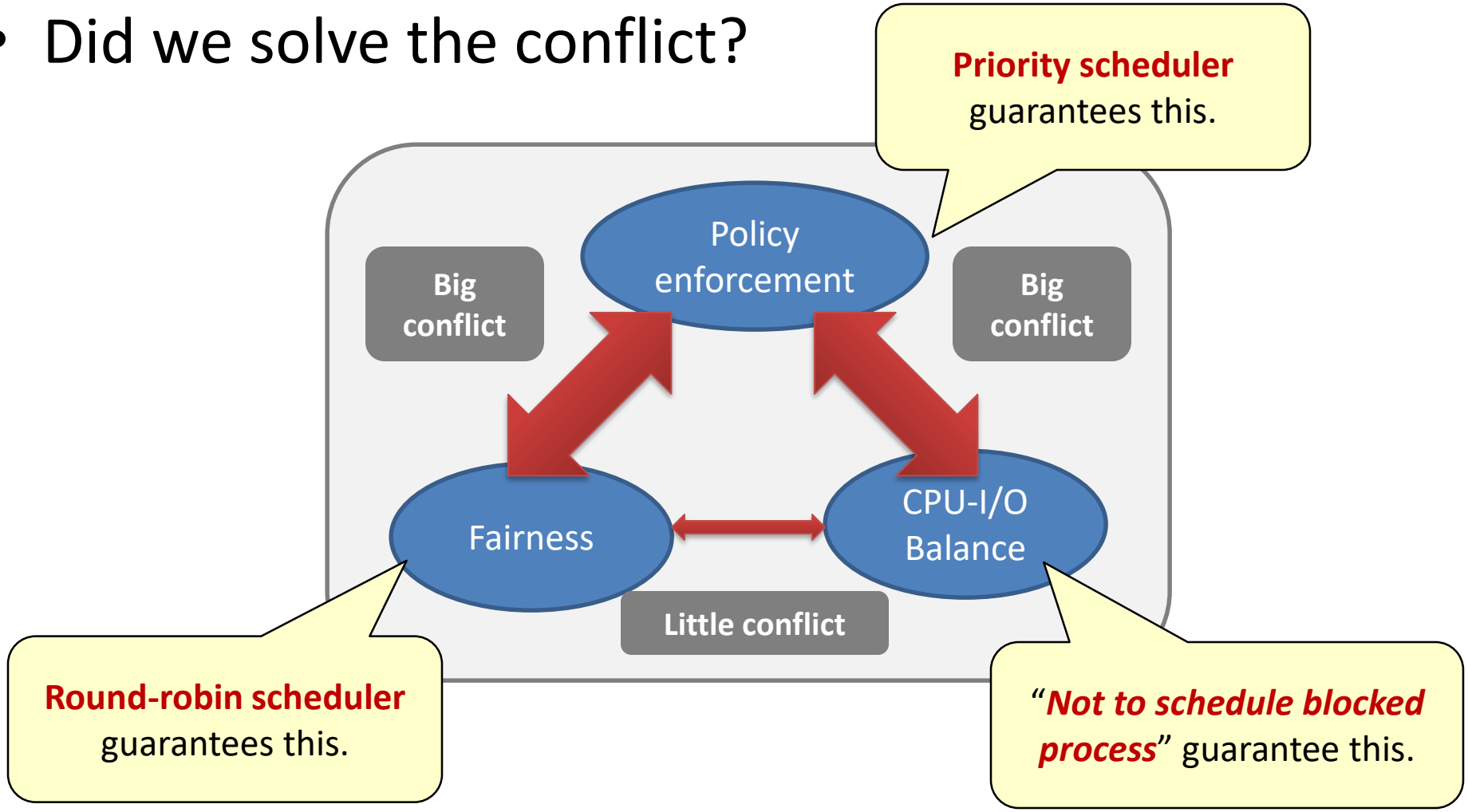


Multilevel feedback queue scheduling

- How to design (factors)?
  - Number of queues
  - Scheduling algorithm for each queue
  - Method for determining when to upgrade/downgrade a process
  - Method for determining which queue a process will enter
- Most general, but also most complex
  - Can be configured to match a specific system

Summary

- Did we solve the conflict?

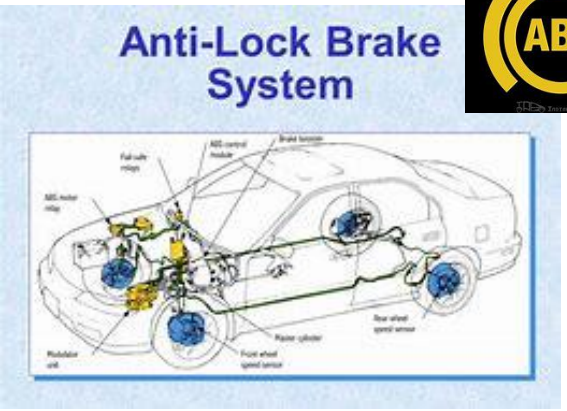


Multilevel feedback queue scheduling

- Applications/Scenarios
  - Real-time systems
  - Multiple processors
  - Example: Linux scheduler
  - Algorithm evaluation

- Applications/Scenarios
  - Real-time systems
  - Multiple processors
  - Example: Linux scheduler
  - Algorithm evaluation

Real-time CPU Scheduling



Antilock brake system: Latency requirement: 3-5 ms

Hard real-time systems: A task must be served by its deadline (otherwise, expired as no service at all)

Soft real-time systems: Critical processes will be given preference over noncritical processes (no guarantee)

Responsiveness: Respond immediately to a real-time process as soon as it requires the CPU

Support priority-based alg. with preemption

Interrupt latency (minimize or bounded):
- ✓ Determining interrupt type and save the state of the current process
- ✓ Minimize the time interrupts may be disabled

Dispatch latency:
- ✓ Time required by dispatcher (preemption running process and release resources of low-priority proc).
- ✓ Most effective way is to use preemptive kernel

Real-time CPU Scheduling Algorithms

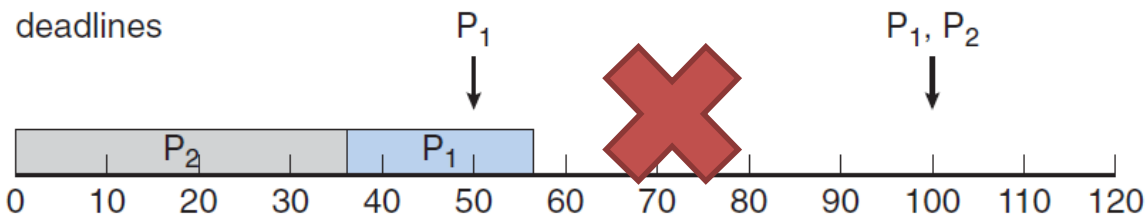
Rate monotonic scheduling

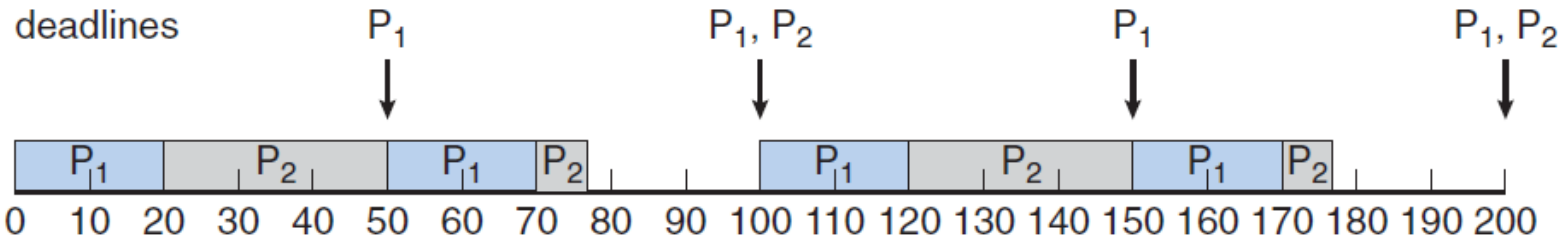
Assumption: Processes require CPU at constant periods: processing time t and period p (rate 1/p)

Each process is assigned a priority proportional to its rate, and schedule processes with a static priority policy with preemption (**fixed priority**)

Example

P1: p1=50, t1=20
P2: p2=100, t2=35





Real-time CPU Scheduling Algorithms

Rate monotonic scheduling

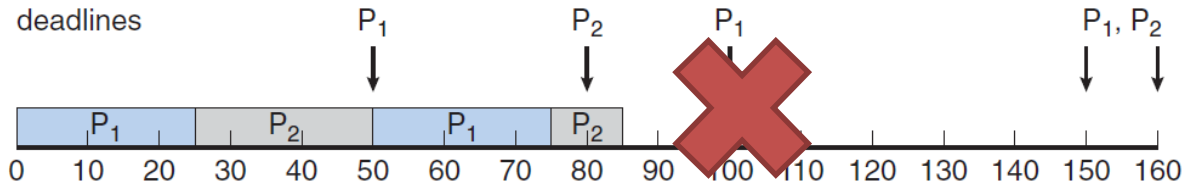
Any problem?

Processes require CPU at constant periods: processing time t and period p (rate 1/p)

Each process is assigned a priority proportional to its rate, and schedule processes with a static priority policy with preemption (**fixed priority**)

Example

P1: p1=50, **t1=25**
P2: **p2=80**, t2=35



Can not guarantee that a set of processes can be scheduled

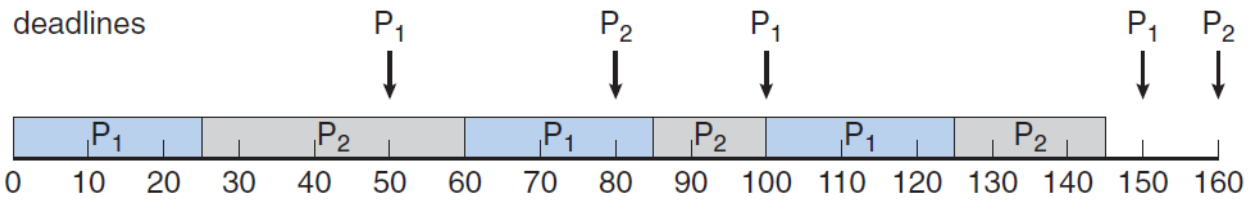
Real-time CPU Scheduling Algorithms

Earliest-deadline-first scheduling (EDF)

Dynamically assigns priorities according to deadline (the earlier the deadline, the higher the priority)

Example

P1: p1=50, **t1=25**
P2: **p2=80**, t2=35

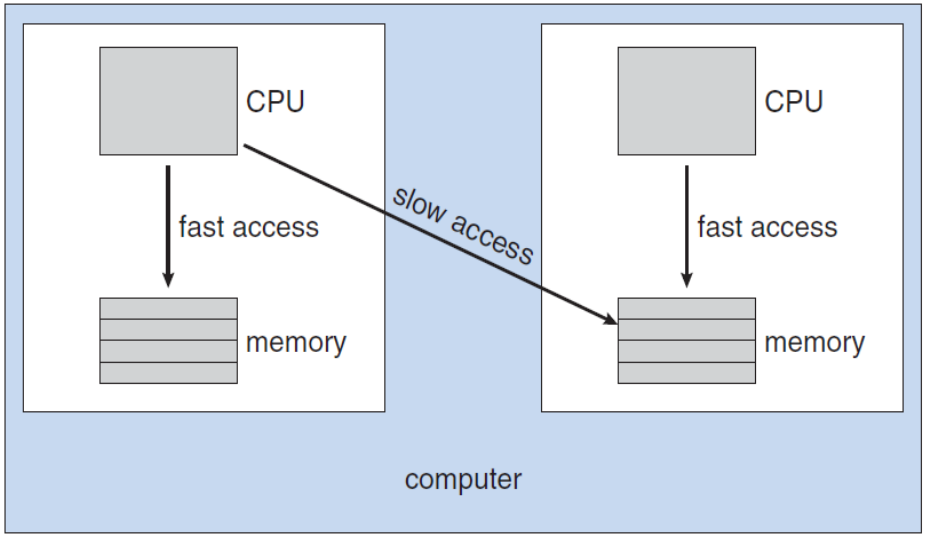


EDF does not require the processes to be periodic, nor require a constant CPU time per burst

EDF requires the announcement of deadlines

- **Applications/Scenarios**
  - Real-time systems
  - **Multiple processors**
  - Example: Linux scheduler
  - Algorithm evaluation

Scheduling Issues with SMP



SMP: Each processor may have its private queue of ready processes

Scheduling between processors

Process migration: Invalidating the cache of the first processor and repopulating the cache of the second processor)

Process migration is costly

Processor Affinity

Attempt to keep a process running on the same processor

Soft/hard affinity

NUMA

CPU scheduler and memory-placement algorithms work together

Load balancing

Push migration: a specific task periodically check the status & rebalance

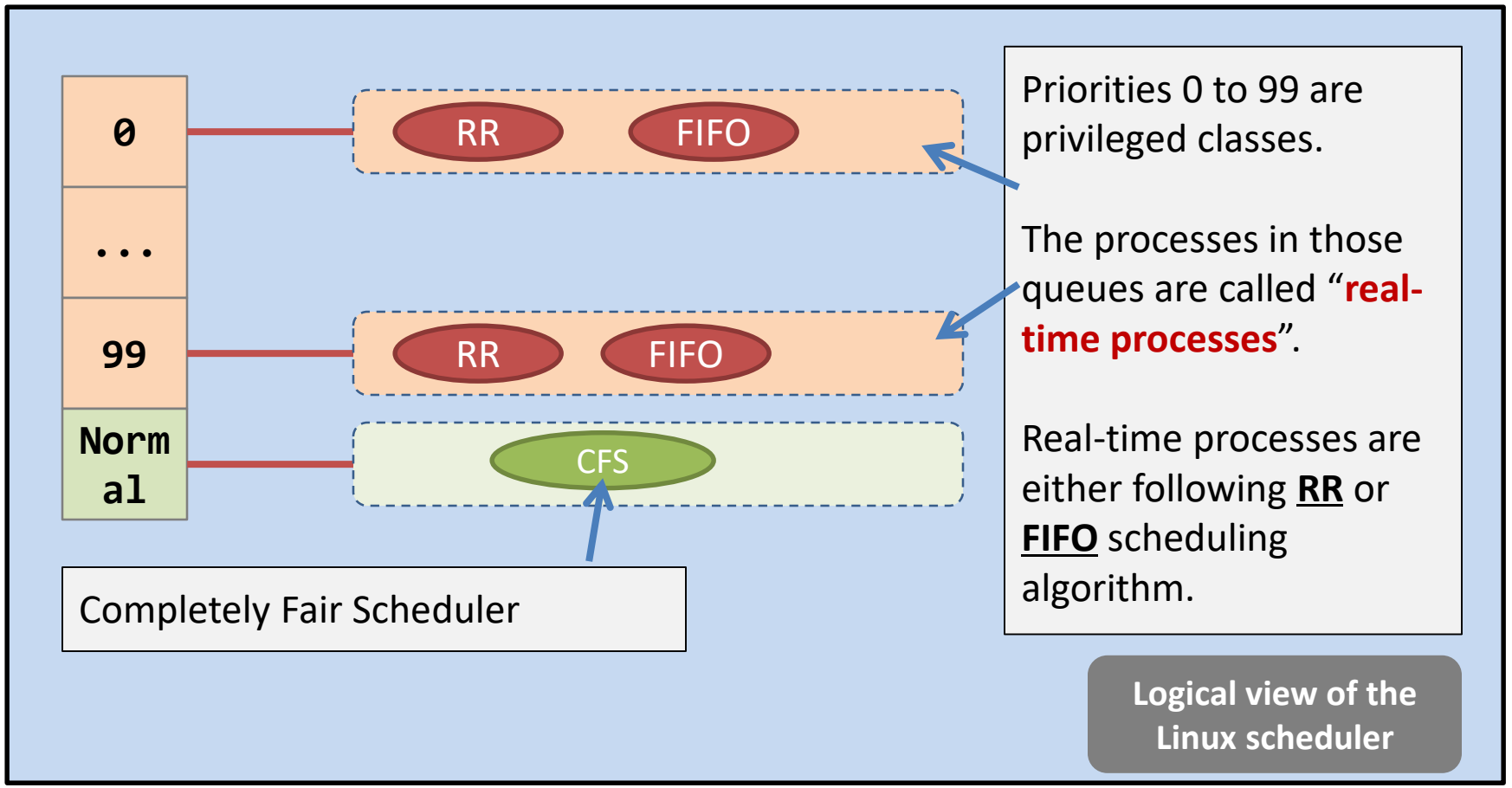
Pull migration: an idle processor pulls a waiting task from busy processor

No absolute rule concerning what policy is best

- **Applications/Scenarios**
  - Multiple processors
  - Real-time systems
  - **Example: Linux scheduler**
  - Algorithm evaluation

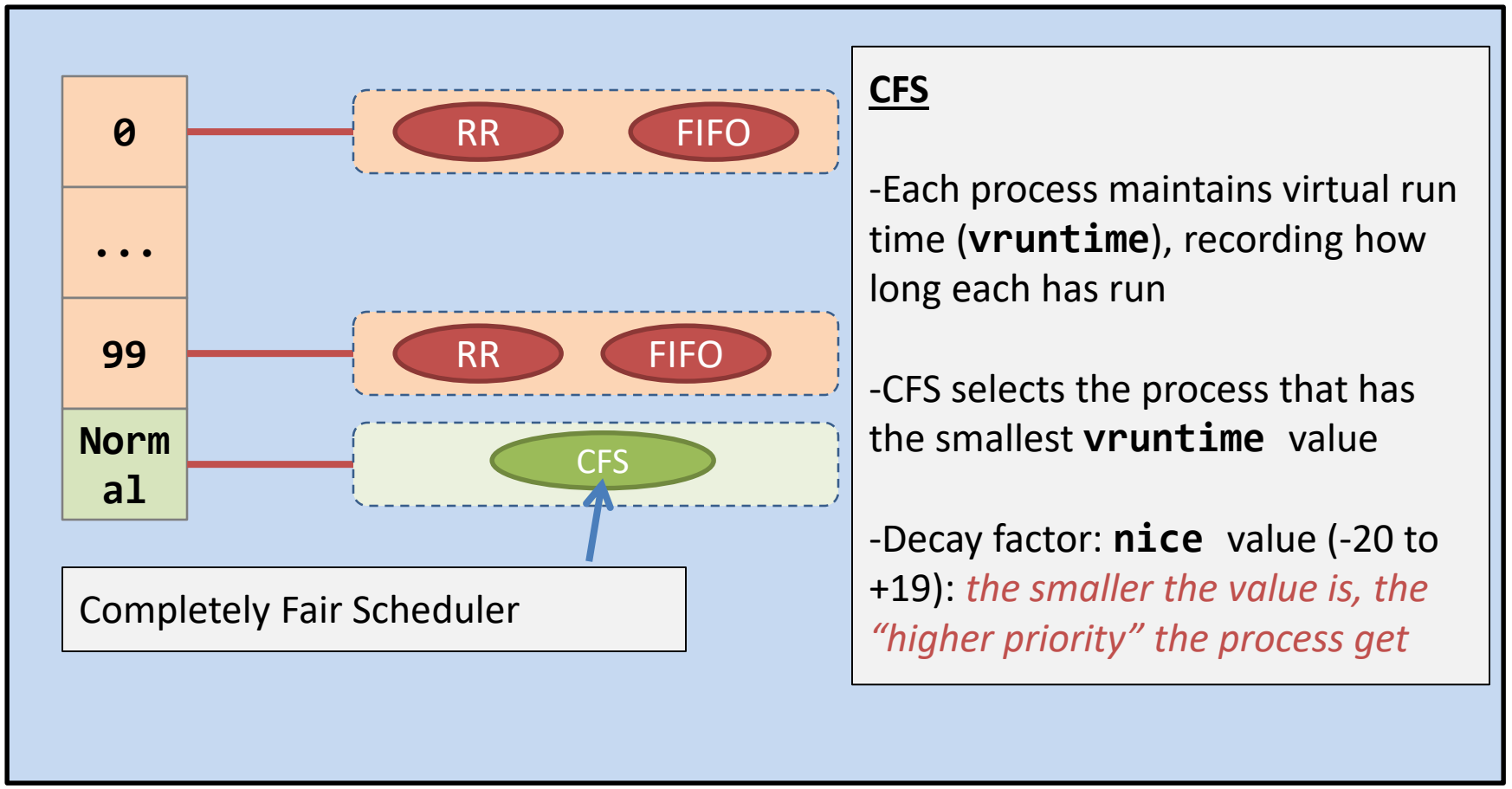
Linux Scheduler

- A multiple queue, (kind of) static priority scheduler.



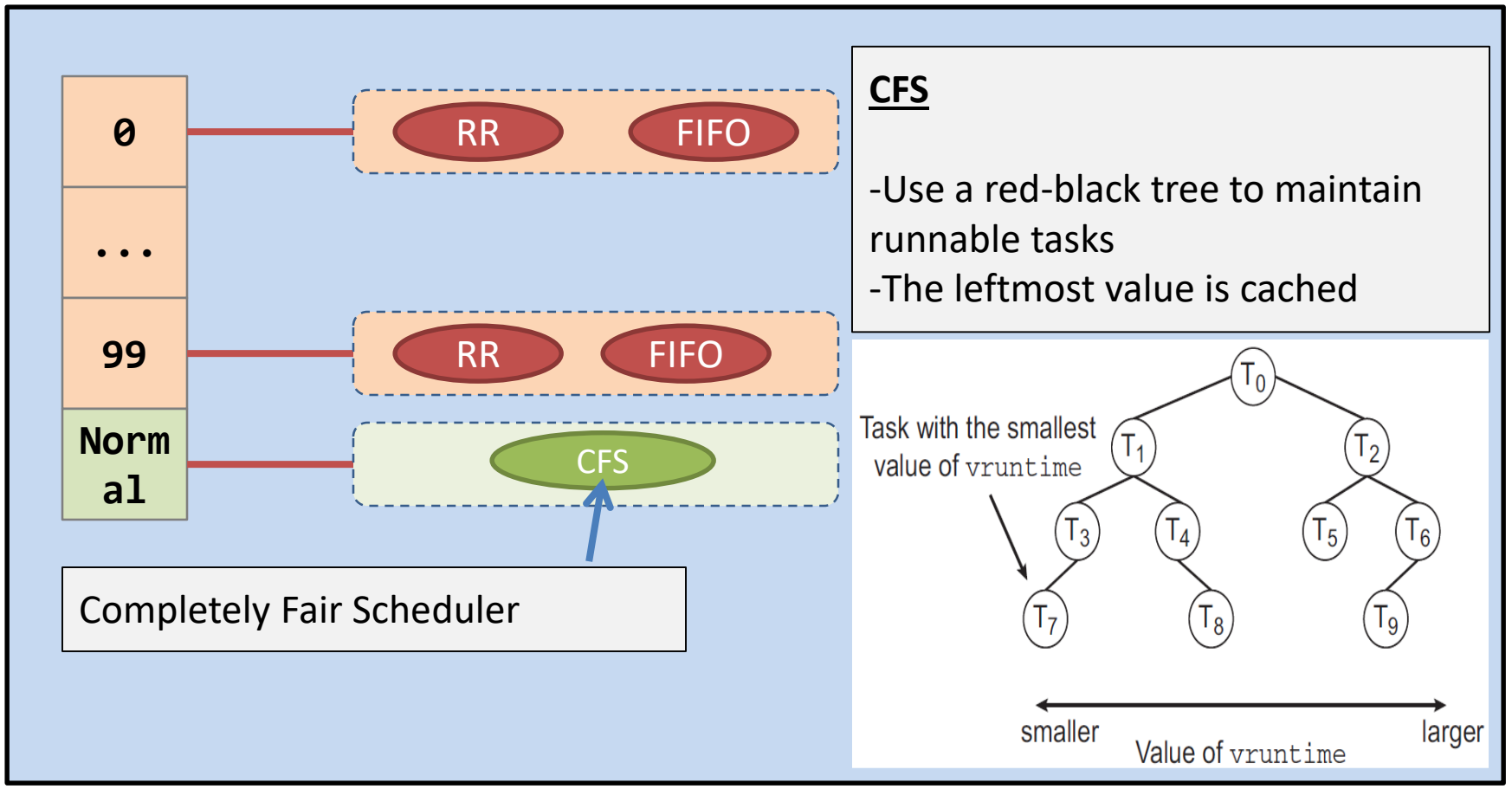
Linux Scheduler

- A multiple queue, (kind of) static priority scheduler.



Linux Scheduler

- A multiple queue, (kind of) static priority scheduler.



- **Applications/Scenarios**
  - Multiple processors
  - Real-time systems
  - Example: Linux scheduler
  - **Algorithm evaluation**

How to select/evaluate a scheduling algorithm?

How to select a scheduling alg? (many algorithms with different parameters and properties)

Step 1: Define a criteria or the importance of various measures (application dependent)

Step 2: Design/Select an algorithm to satisfy the requirements. How to guarantee?

Evaluate Algorithms

Deterministic modeling

Simple and fast

Demonstration examples

Queueing modeling

Queueing network analysis

Distribution of CPU and I/O burst (Poisson arrival)

Little's law: $n = \lambda \times W$

Simulation & Implementation

Trace driven

High cost (coding/debugging…)

Hard to understand the full design space

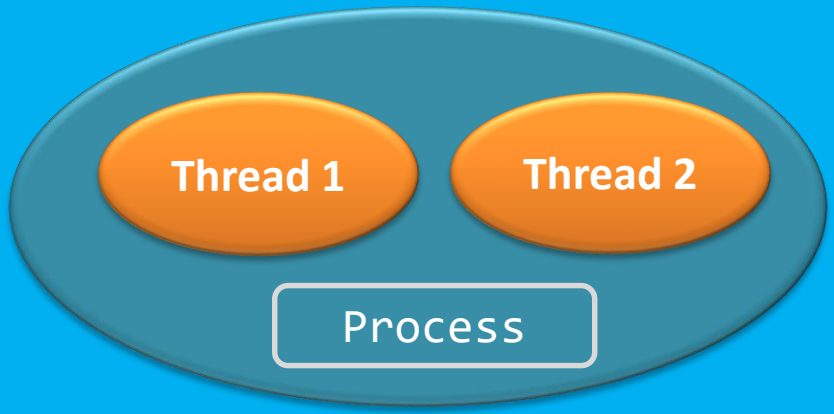
Summary on scheduling

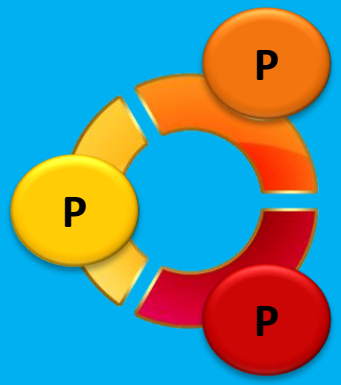
- So, you may ask:
  - "What is the **best** scheduling algorithm?"
  - "What is the **standard** scheduling algorithm?"

- There is **no best or standard** algorithm because of, *at least*, the following reasons:
  - No one could predict how many clock ticks does a process requires.
  - On modern OS-es, processes are submitted online.
  - Conflicting criterias

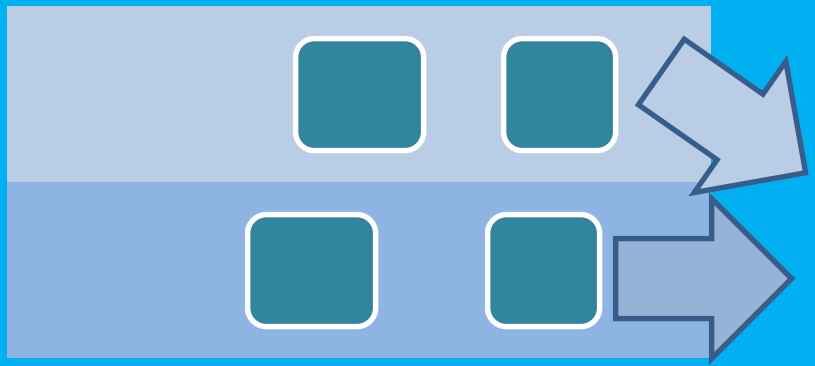
Summary on part 2

Process Operations (fork(),exec*(),wait())





Process Communication & Synchronization



Process Scheduling