Operating Systems

Eric Lo

8 - Scheduling

What is scheduling?

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

CPU-bound Process	I/O-bound process
Spends most of its running time on the CPU, i.e., **user-time > sys-time**	Spends most of its running time on I/O, i.e., **sys-time > user-time**
Examples - CSCI2100 assignments, AI programs.	**Examples** - **/bin/ls**, networking programs.

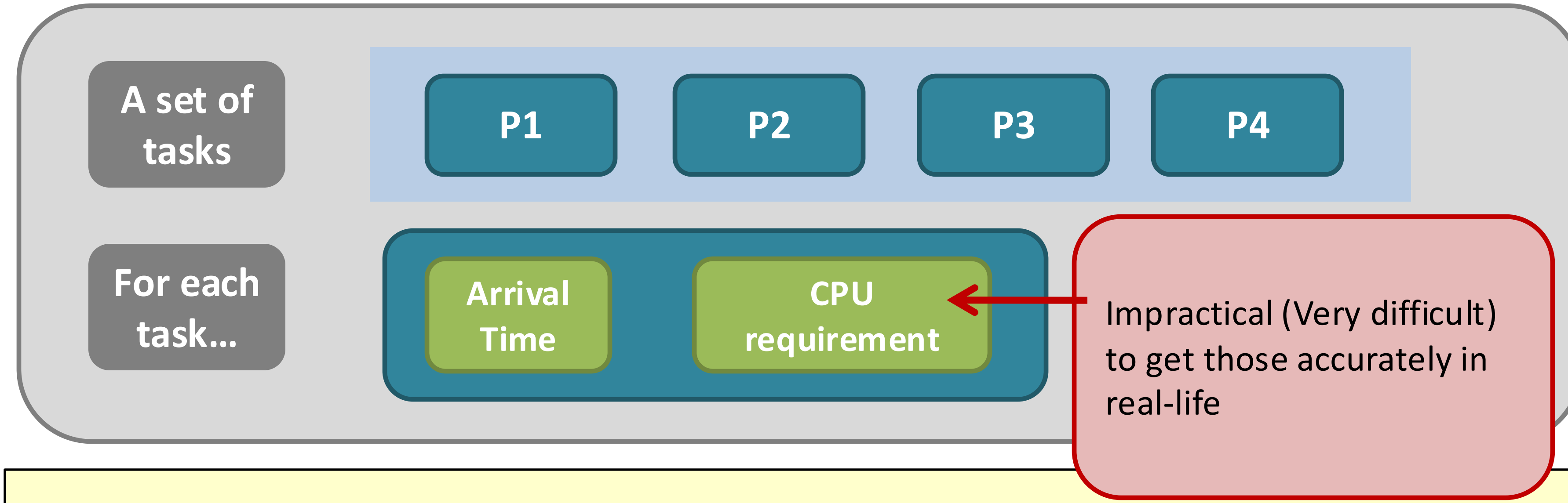
Classes of scheduling

- Preemptive scheduling (Non-preemptive is out)

What is it?	When a process/thread is chosen by the scheduler, the process would have the CPU until… -the process voluntarily waits for I/O, or -the process voluntarily releases the CPU, e.g., **`exit(), yield()`**. -**particular kinds of interrupts (e.g., periodic clock interrupt, a new process steps in) are detected**.
History	In old days, it was called “time-sharing” Nowadays, all systems are time-sharing
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.

Scheduling algorithms

- **Inputs to the algorithms.**



Online VS Offline

An **offline scheduling algorithm** assumes that you know the sequence of processes that a scheduler will face
- Theoretical baseline

An **online scheduling algorithm** does not have such an assumption.
- Practical use

Algorithm evaluation

Individual & average <u>turnaround time</u>

Individual & average <u>waiting time</u>

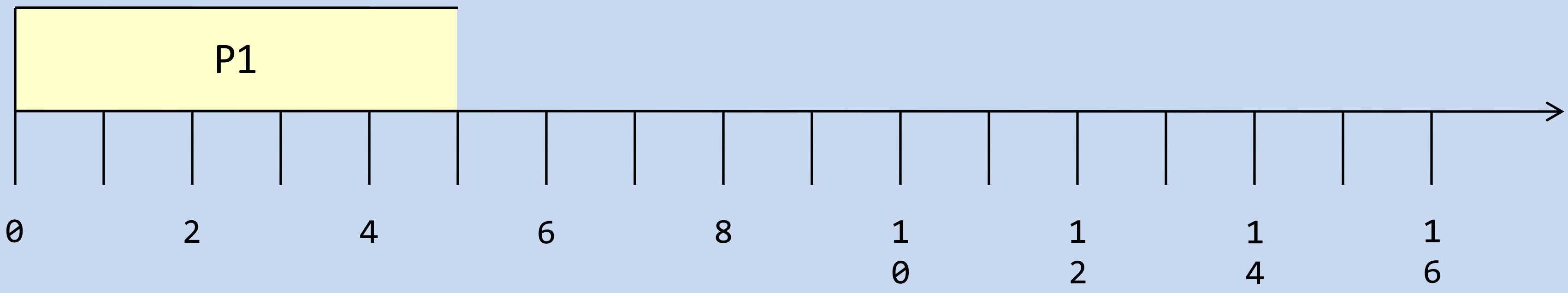
Number of <u>context switches</u>

Turnaround time	The time between the arrival of the task and the termination of the task.
Waiting time	The accumulated time that a task has waited for the CPU.

Different algorithms

Algorithms
Shortest-job-first (SJF)
Round-robin (RR)
Priority scheduling with multiple queues

Non-preemptive SJF (assume context switch is free)



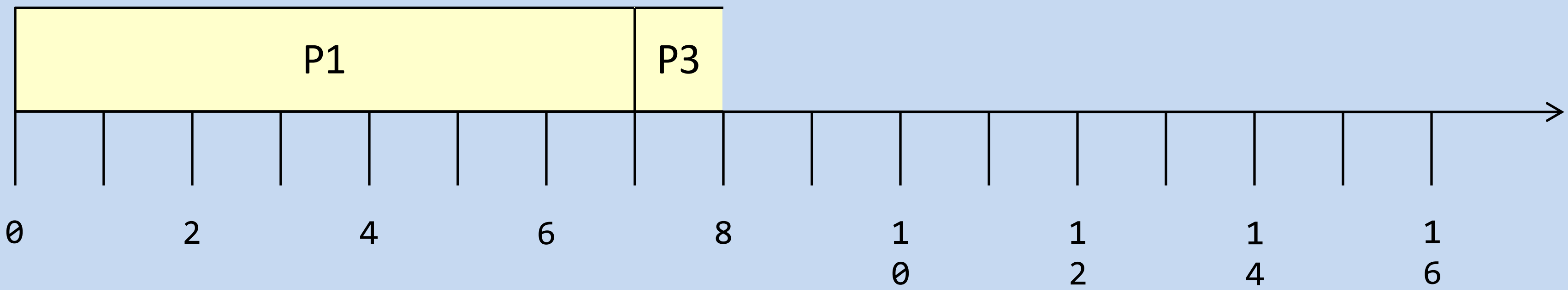
Time = 5

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



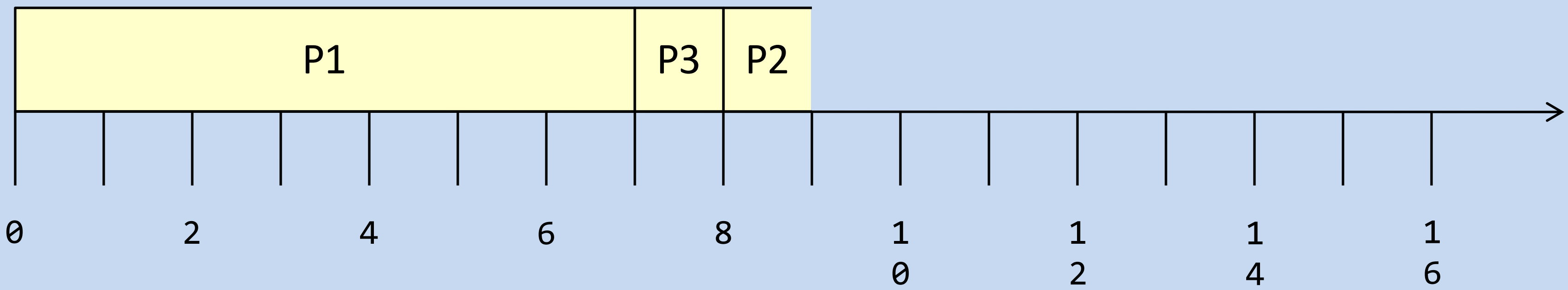
Time = 7

Set of processes

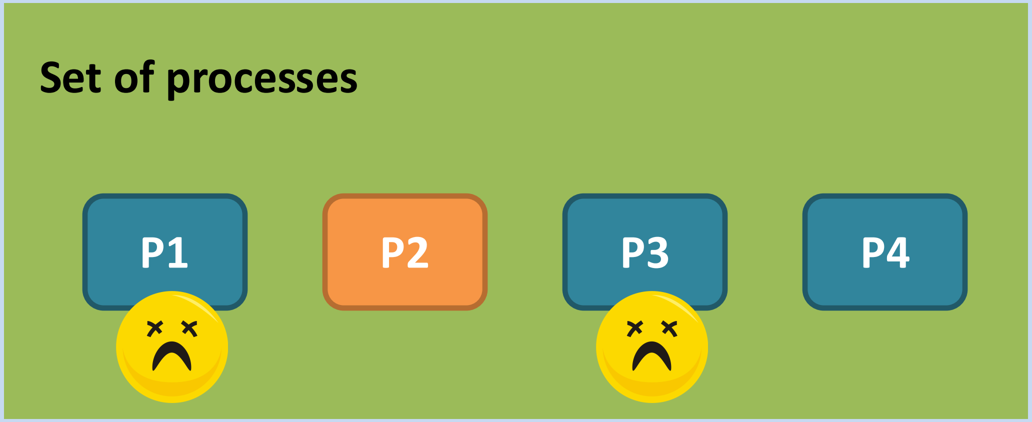
Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

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

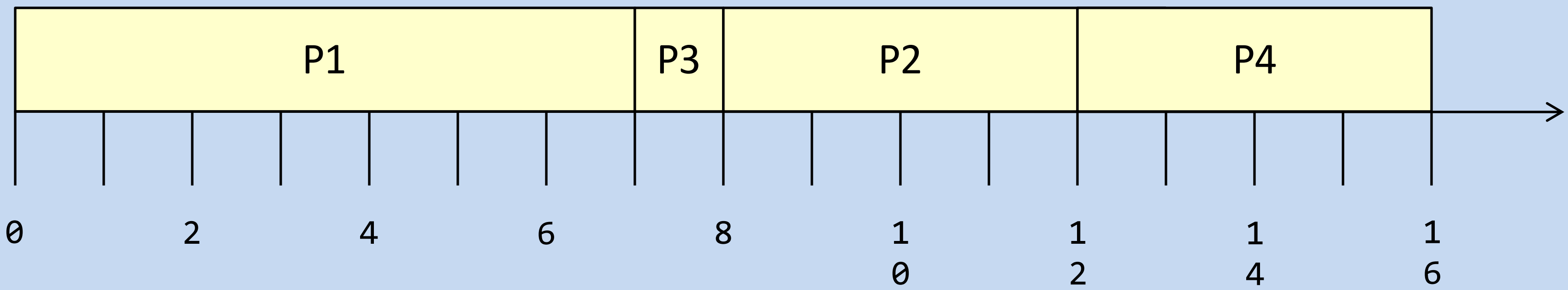


Time = 8

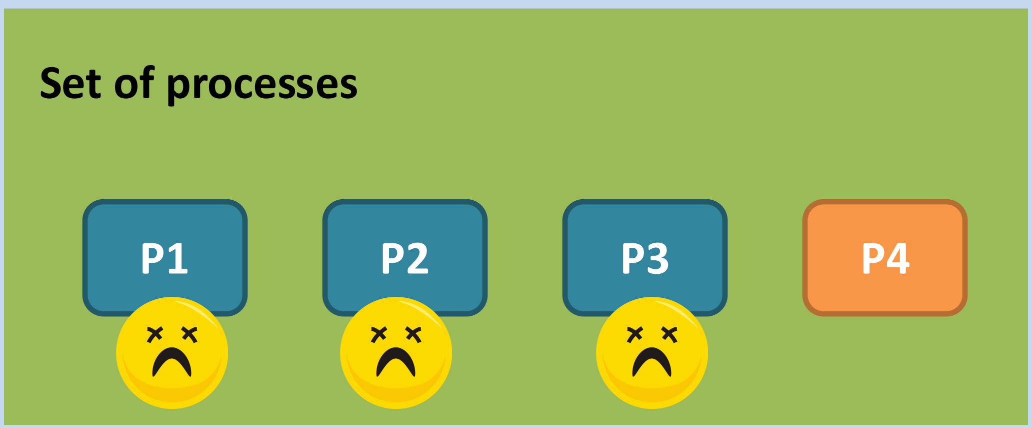


Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

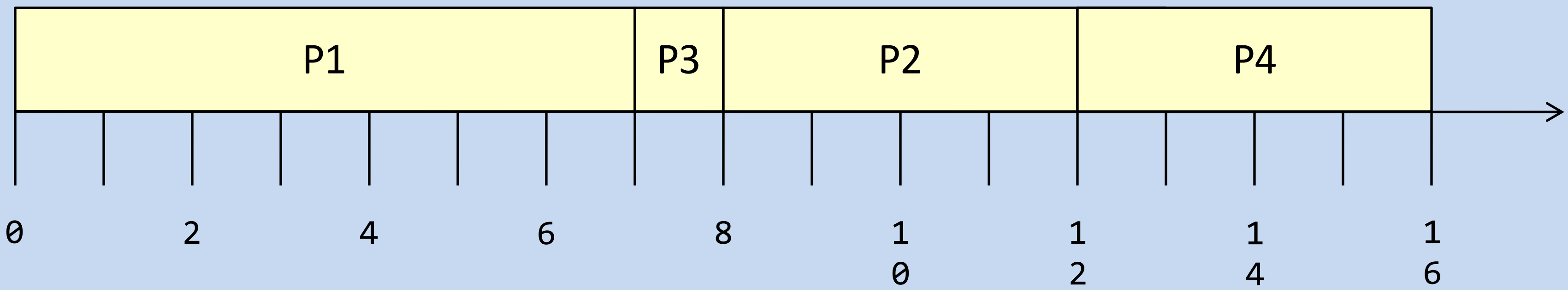


Time = 16



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;

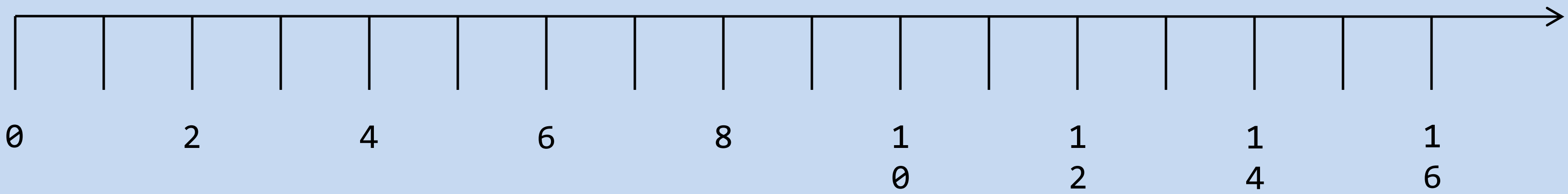
Average = (7 + 10 + 4 + 11) / 4 = 8.

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

SJF

- Problem:
  - What if tasks arrive after P2 all have CPU requirement < 3?
  - Problem persists even for its preemptive version

Preemptive SJF

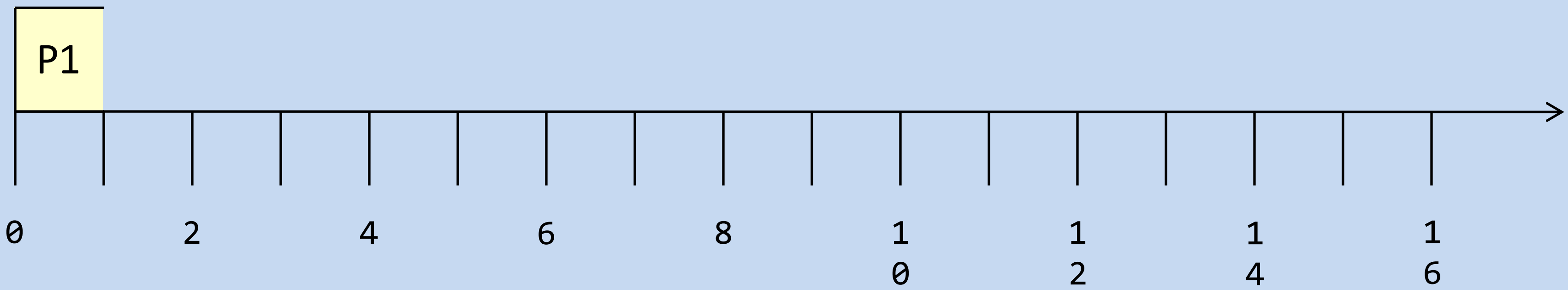


-Whenever a new process arrives at the system, the scheduler steps in and selects the next task based on **their remaining CPU requirements**.

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

Animation; don't print



Time = 0

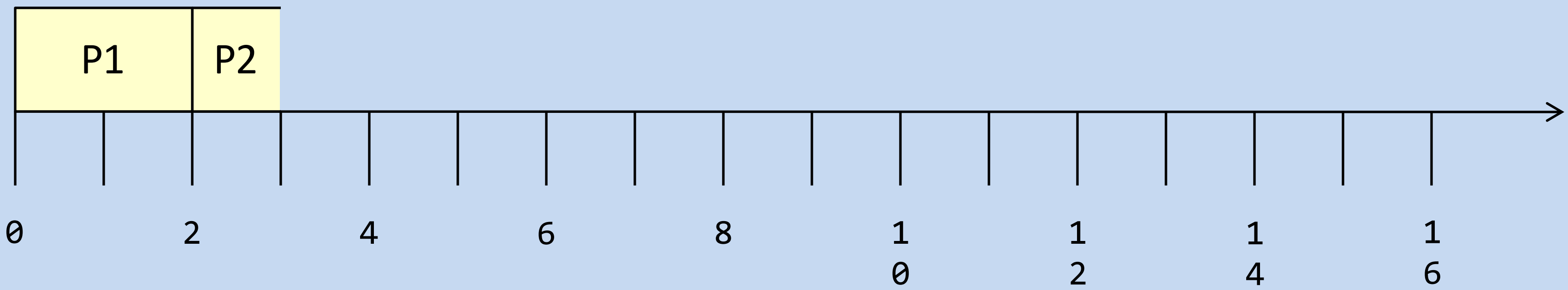
Set of processes

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

Animation; don't print



Time = 2

P2 is selected!

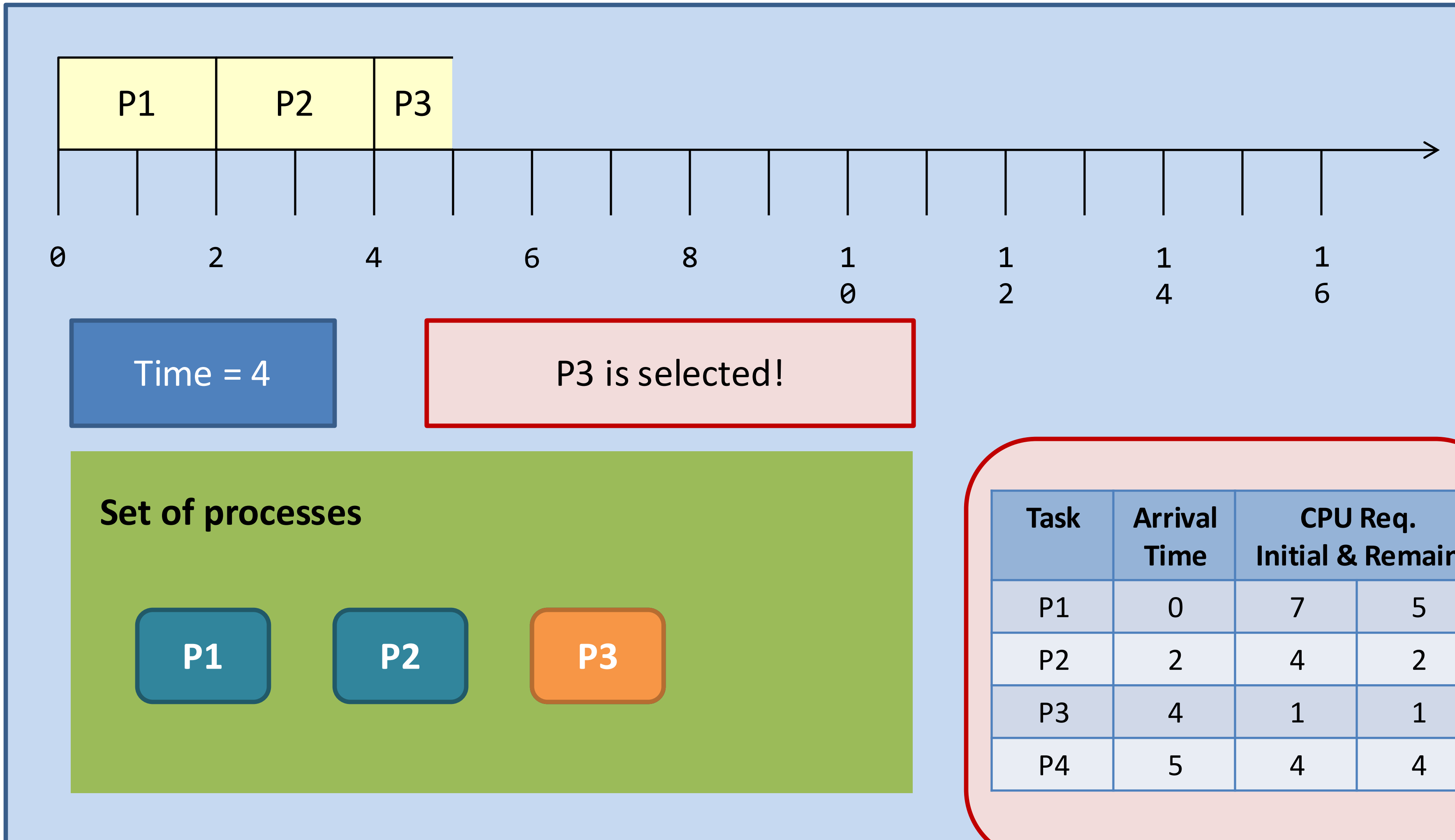
Set of processes

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

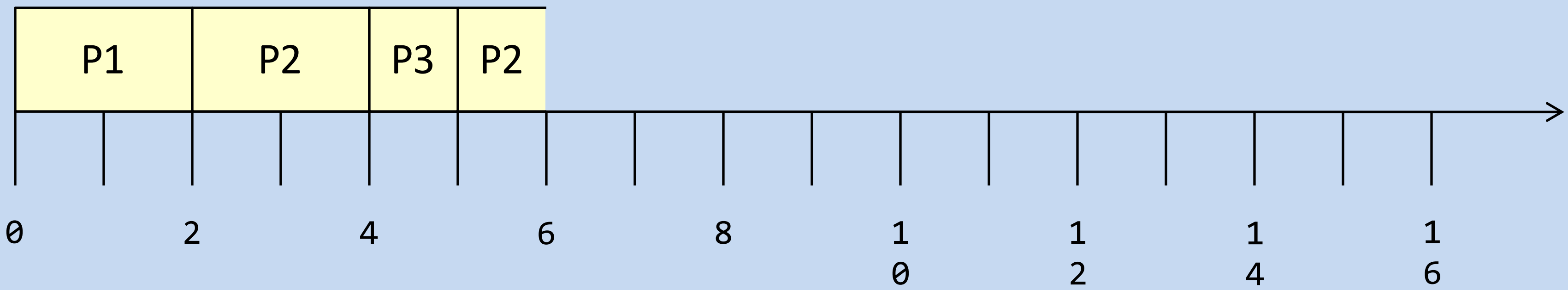
Animation; don't print



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

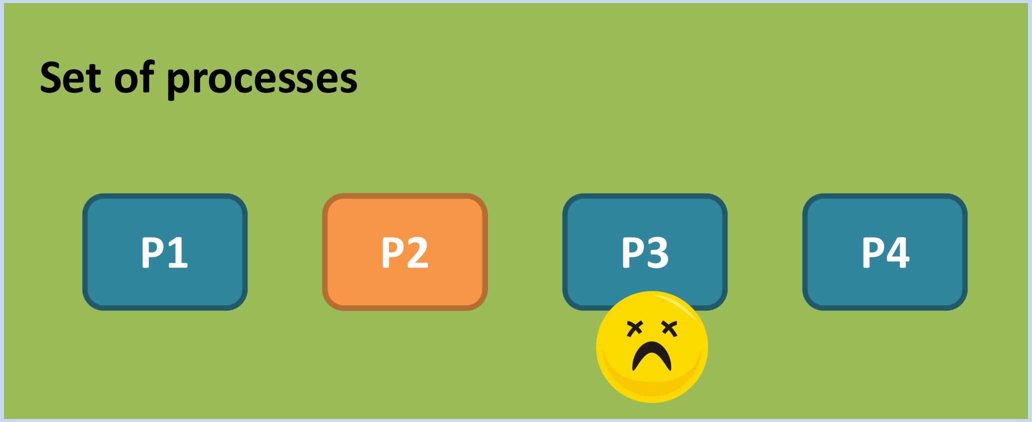
Preemptive SJF

Animation; don't print



Time = 5

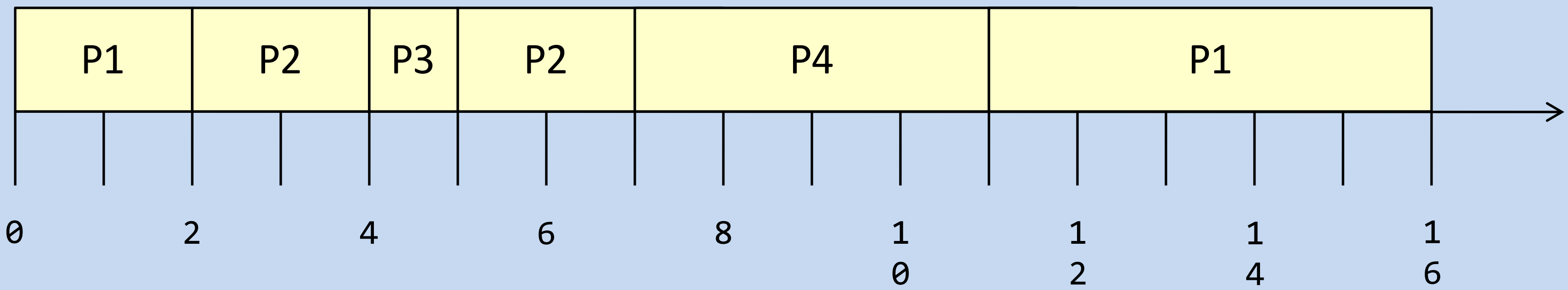
P2 is selected!



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

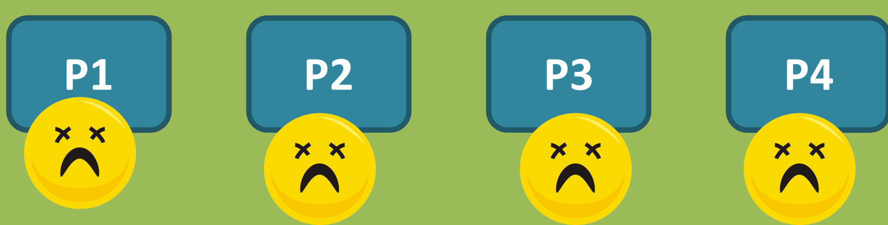
Preemptive SJF

Animation; don't print



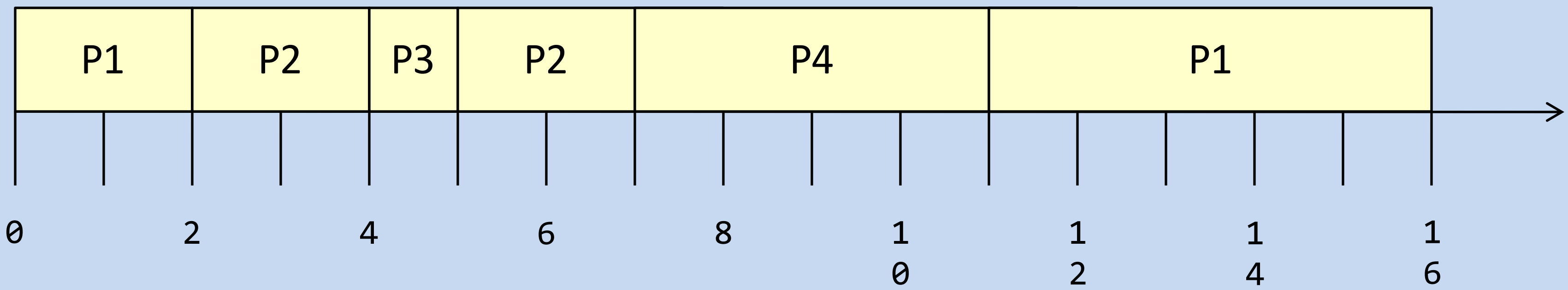
Time = 16

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: Preemptive or not?

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

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

Context switch is expensive. (That's why we shall minimize the # of sys calls as well; on a syscall, the program switch from user-process to kernel-"process".)

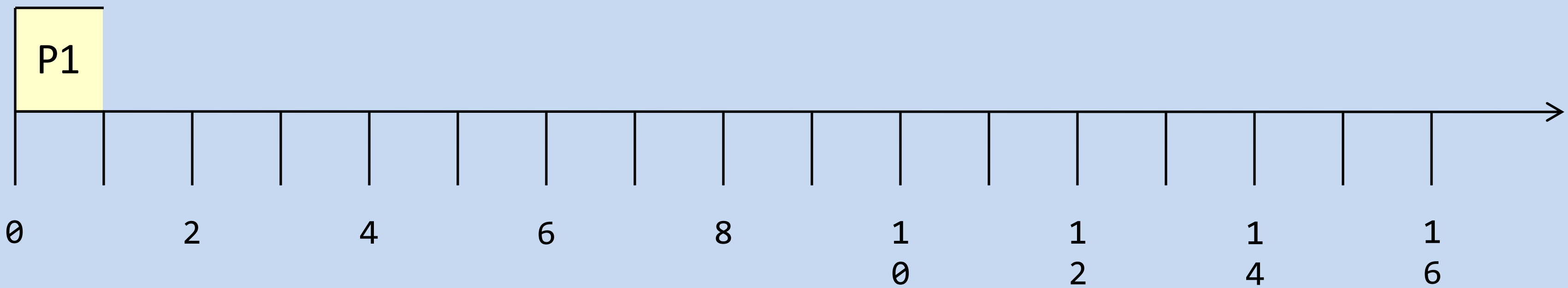
Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Round-robin

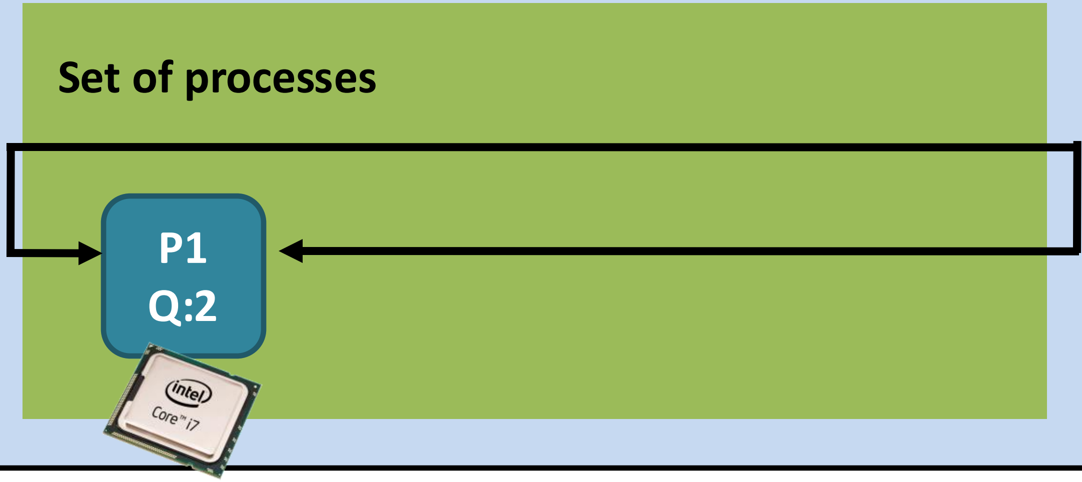
- Round-Robin (RR) scheduling is preemptive.
  - Every process is given a **quantum**, or the amount of time allowed to execute.
  - Whenever the quantum of a process is **used up** (i.e., 0), the process releases the CPU and **this is the preemption**.
  - Then, the scheduler steps in and it chooses **the next process which has a non-zero quantum** to run.
  - If all processes in the system have used up the quantum, they will be re-charged to their initial values.
  - Processes are therefore running one-by-one as a **circular queue**, for the basic version (i.e., no priority)
    - New processes are added to the tail of the ready queue
    - New process arrives won't trigger a new selection decision

Round-robin (quantum =2)

Animation; don't print



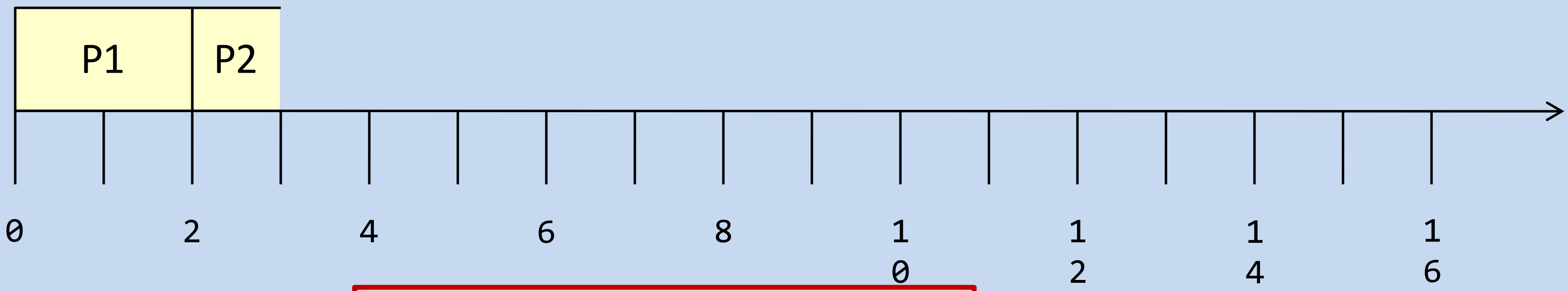
Time = 0



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

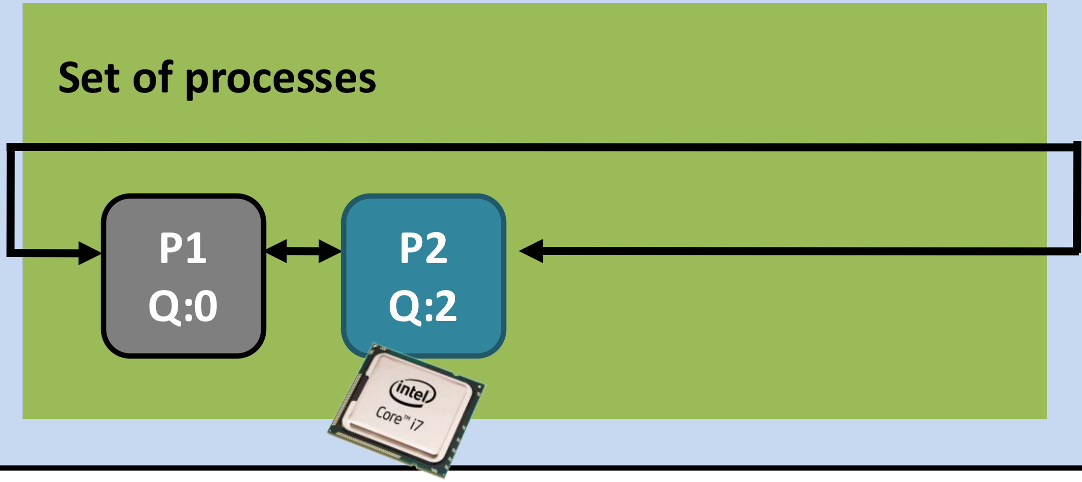
Round-robin

Animation; don't print



Time = 2

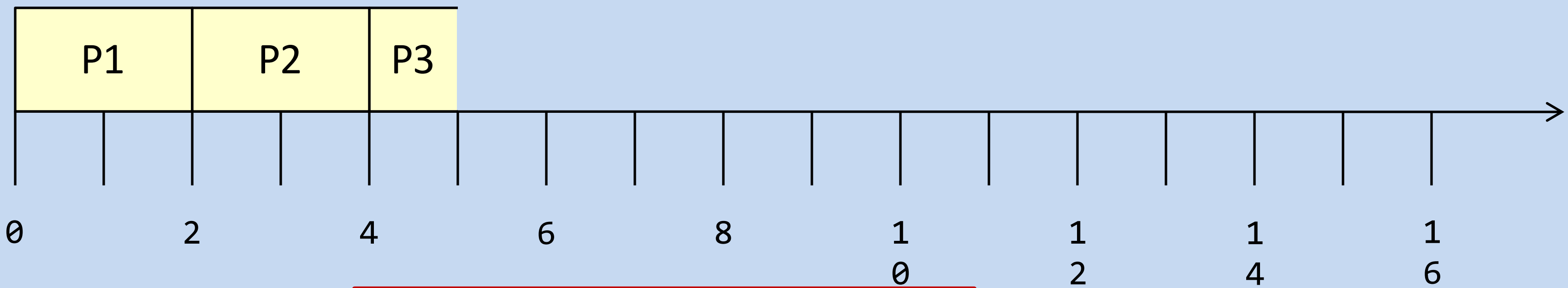
P1's quantum is 0;
P2 is selected!



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

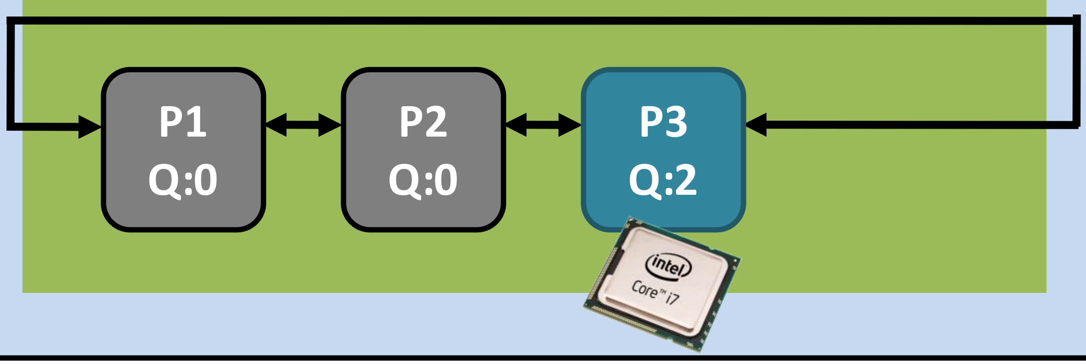
Animation; don't print



Time = 4

P1's & P2's quanta are 0;
P3 is selected!

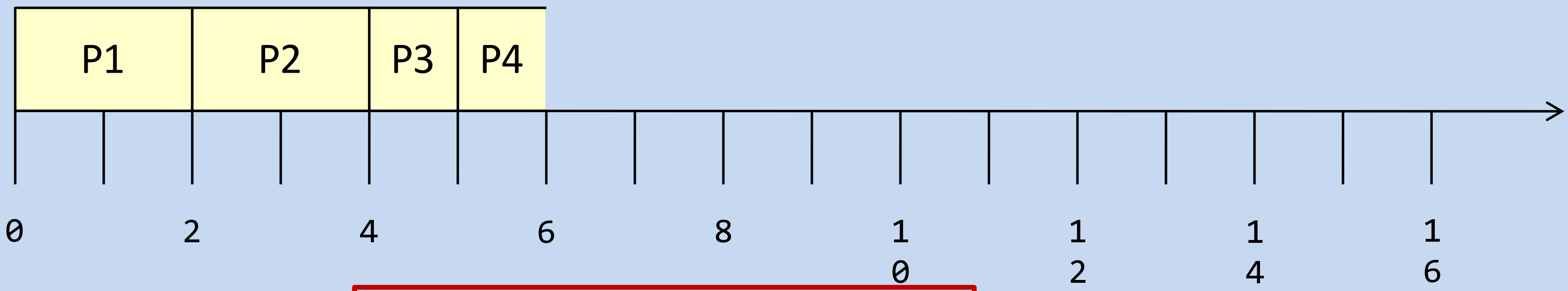
Set of processes



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

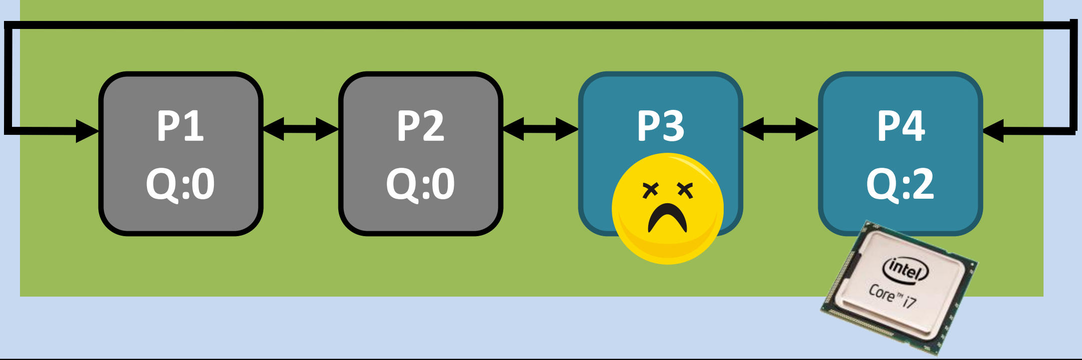
Animation; don't print



Time = 5

P1's & P2's quanta are 0;
P4 is selected!

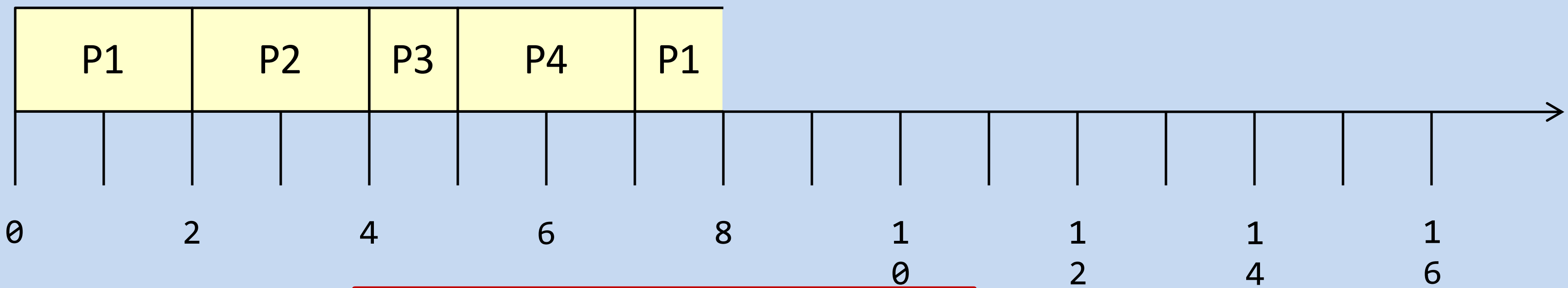
Set of processes



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

Round-robin

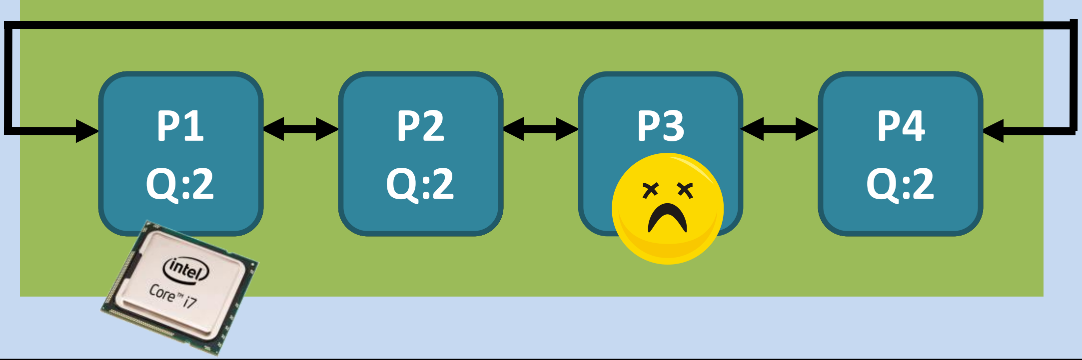
Animation; don't print



Time = 7

Now, recharge is needed.
P1 is selected.

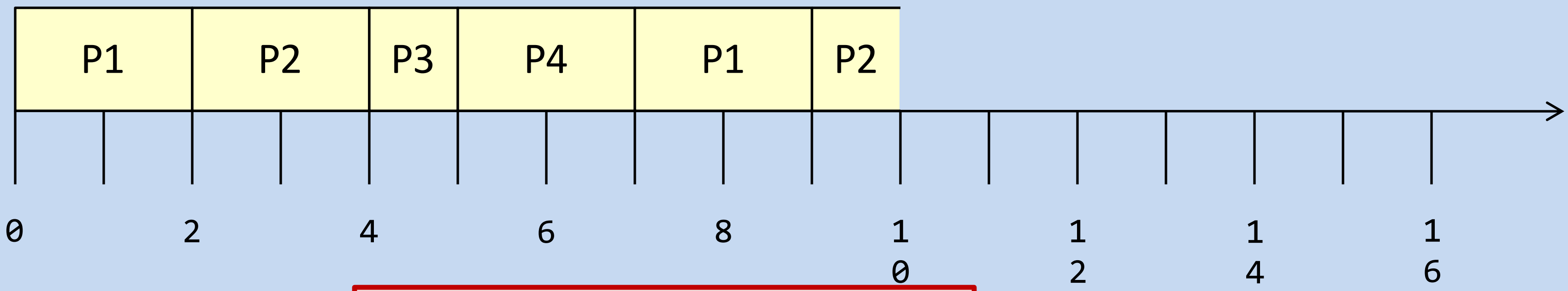
Set of processes



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

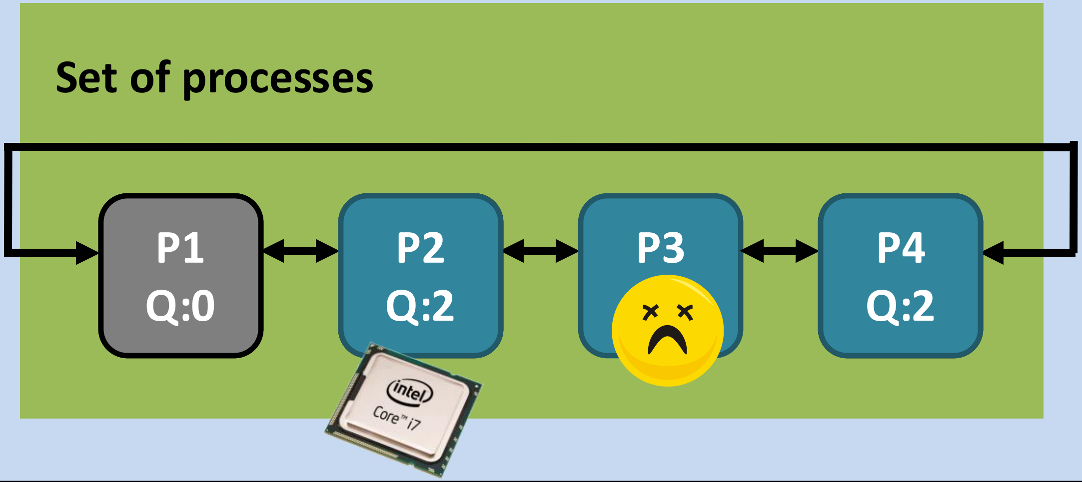
Round-robin

Animation; don't print



Time = 9

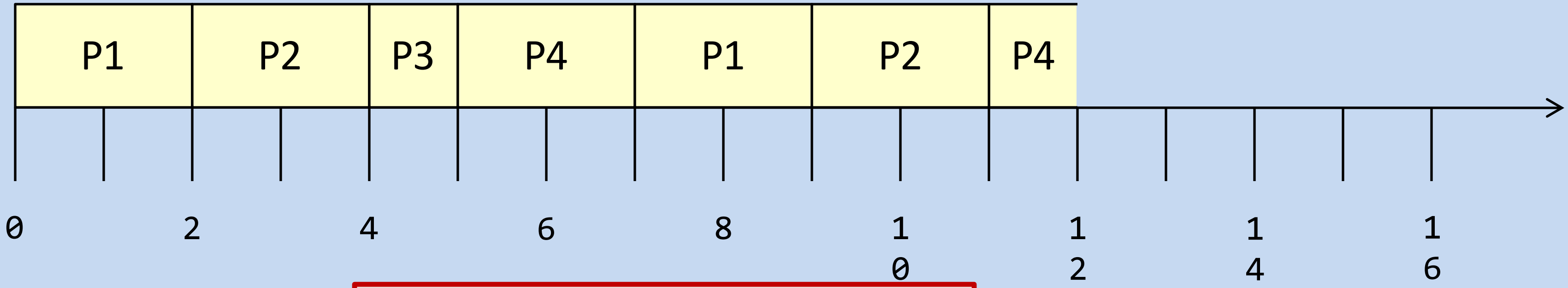
P1's quantum is 0;
P2 is selected!



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

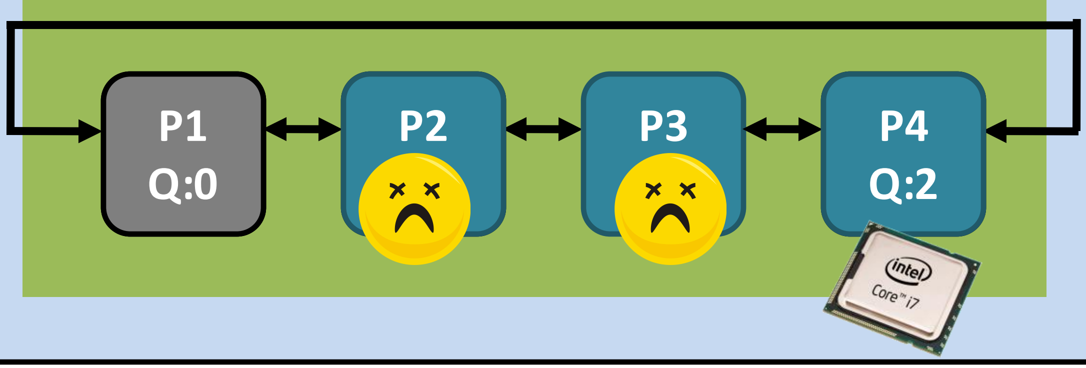
Animation; don't print



Time = 11

P1's quantum is 0;
P4 is selected!

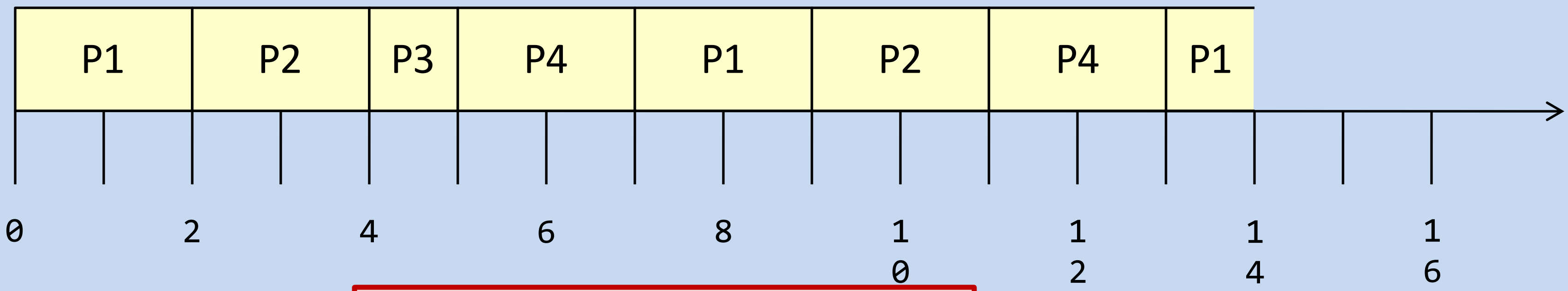
Set of processes



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

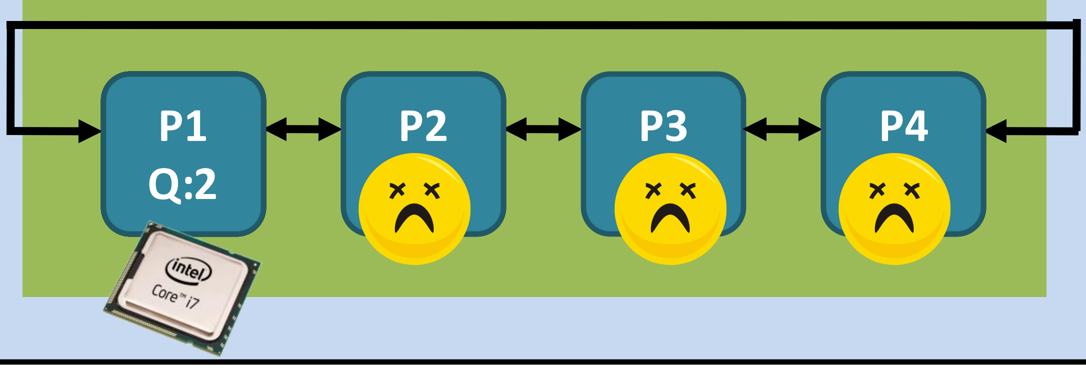
Animation; don't print



Time = 13

Now, recharge is needed.
P1 is selected.

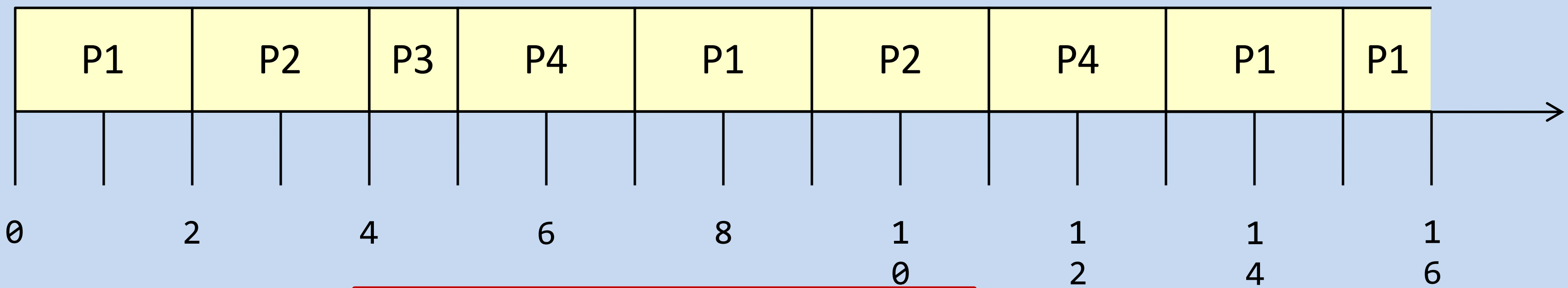
Set of processes



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

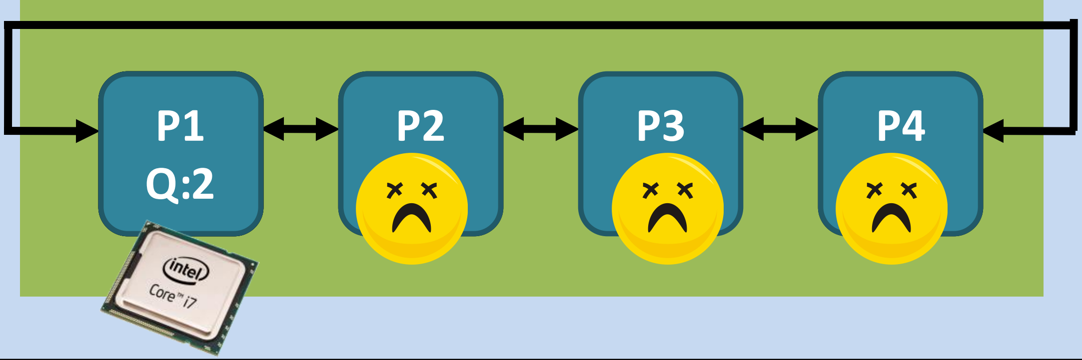
Animation; don't print



Time = 15

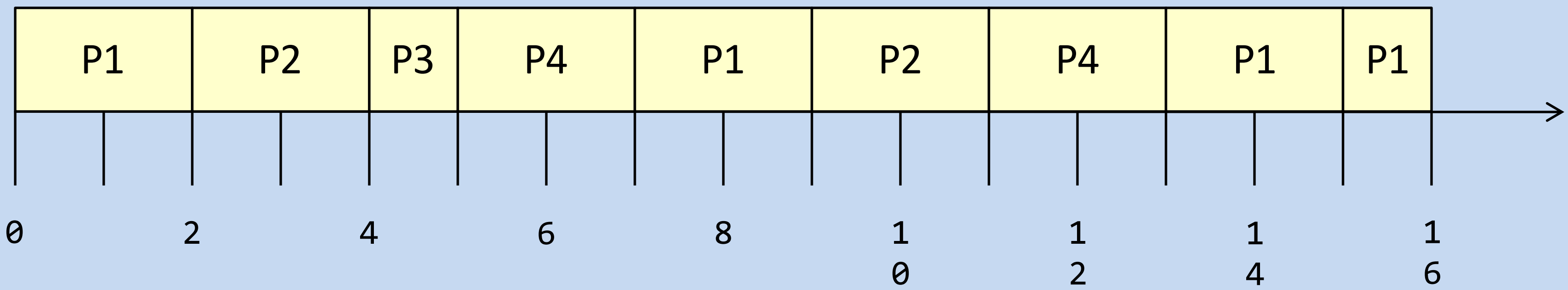
Now, recharge is needed.
P1 is selected.

Set of processes



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 gets the CPU from time to time!

Priority Scheduling

- A task is given a priority (and is usually an integer).
- A scheduler selects the next process based on the priority
- New process arrival triggers a new selection
  - Same priority? RR, SJF, etc.

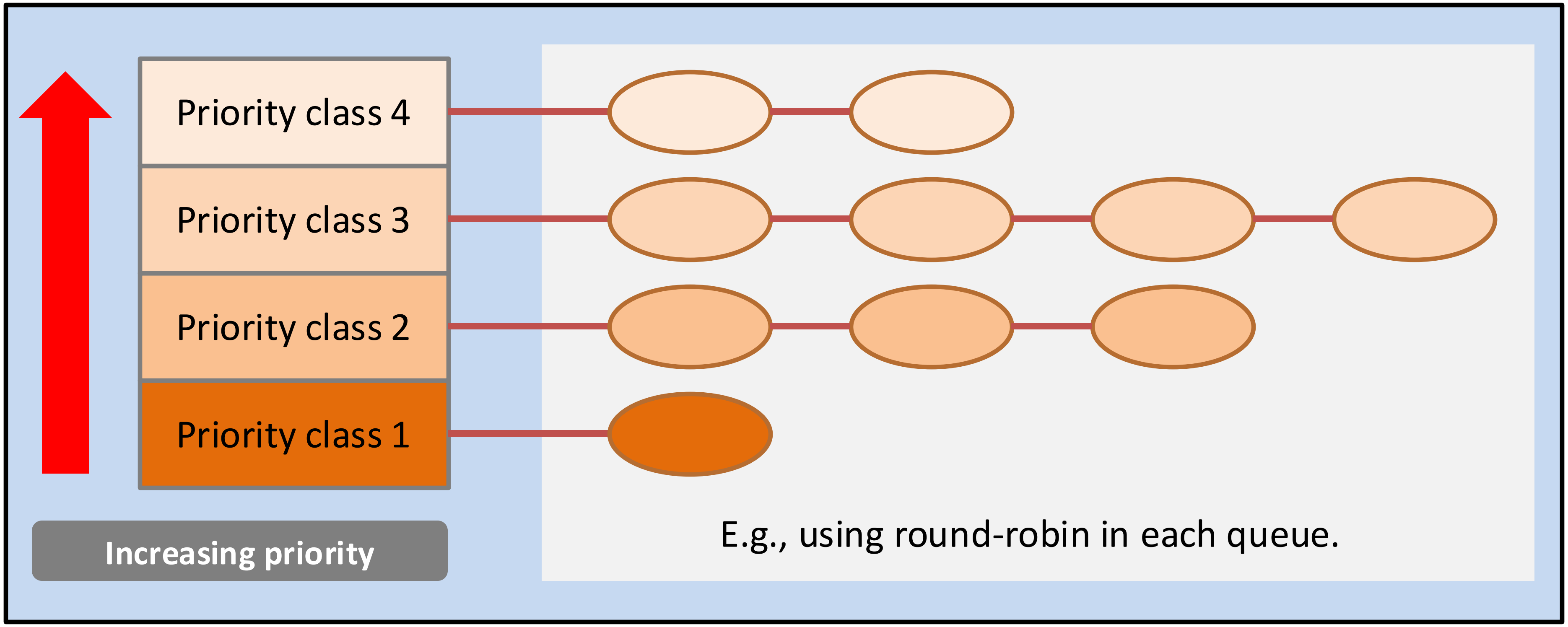
2 Classes	
Static priority	**Dynamic priority**
Every task is given a fixed priority.	Every task is given an initial priority.
The priority is **fixed** throughout the life of the task.	The priority is **changing** throughout the life of the task.

If a task is preempted in the middle

- Note:
  - it has already been dequeued
  - Re-enqueue back to the queue
  - Quantum preserved / recharge?
    - Depends
      - Preserved: need more book keeping
      - Recharge: easy (assumed in this course)

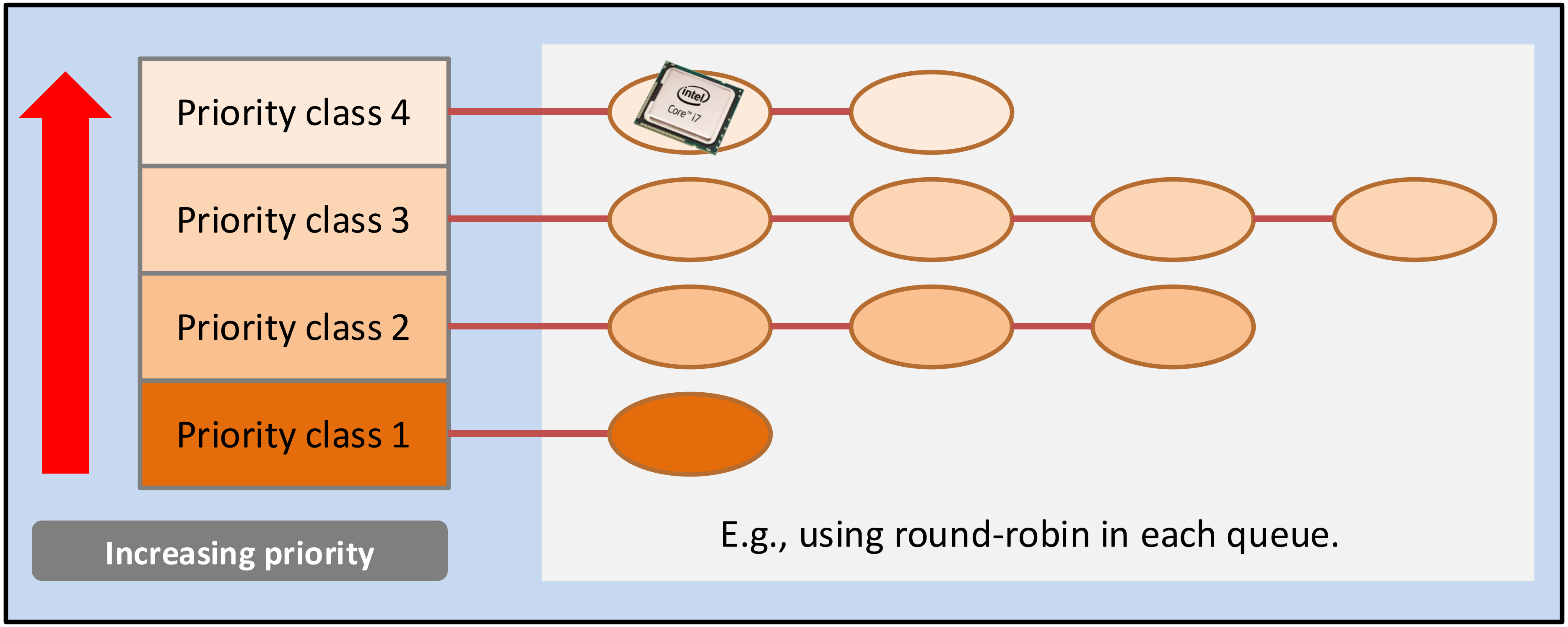
Static priority scheduling – an example

- **Properties**: process is assigned a fix priority when they are submitted to the system.
  - E.g., Linux kernel 2.6 has 100 priority classes, [0-99].



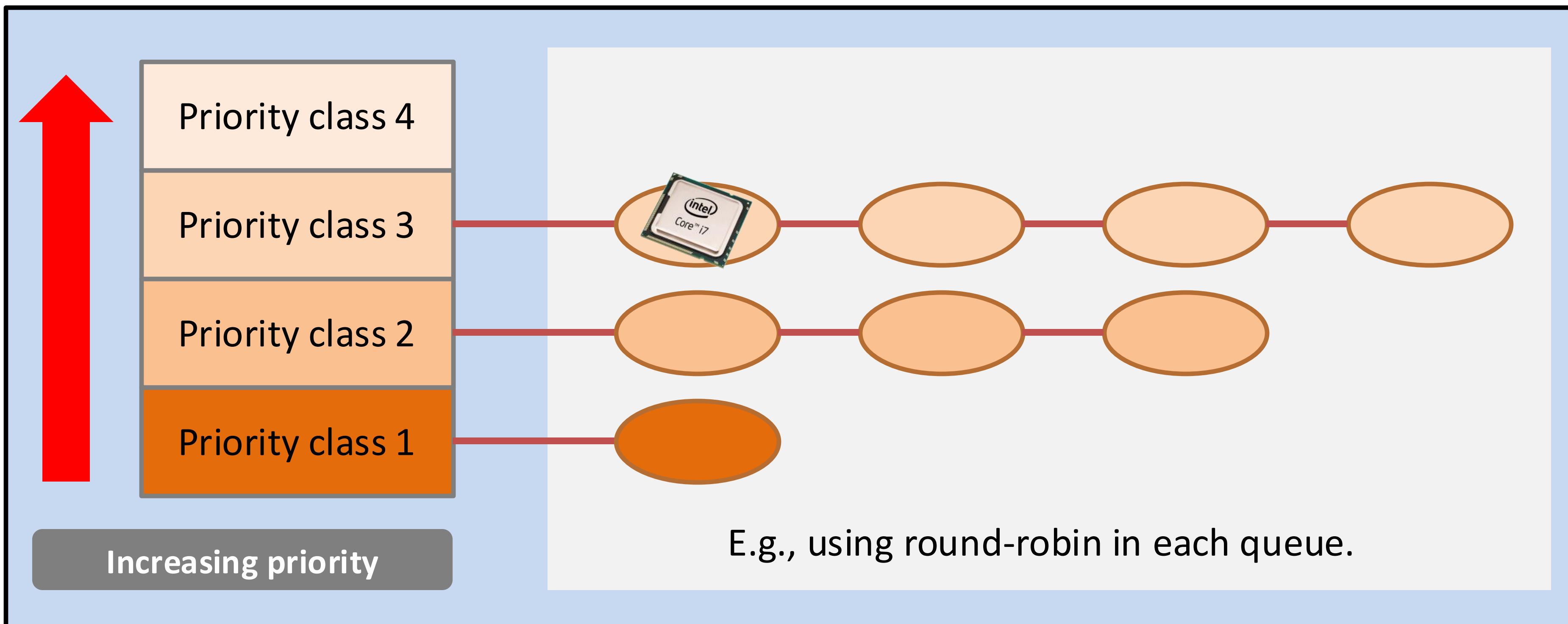
Static priority scheduling – an example

- The highest priority class will be selected.
  - The tasks are usually short-lived, but important;
    - To prevent high-priority tasks from running indefinitely.



Static priority scheduling – an example

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



Multiple queue priority scheduling

- **Definitions.**
  - It is still a priority scheduler.
  - But, at each priority class, **different schedulers** may be deployed.
  - The priority can be a mix of static and dynamic.

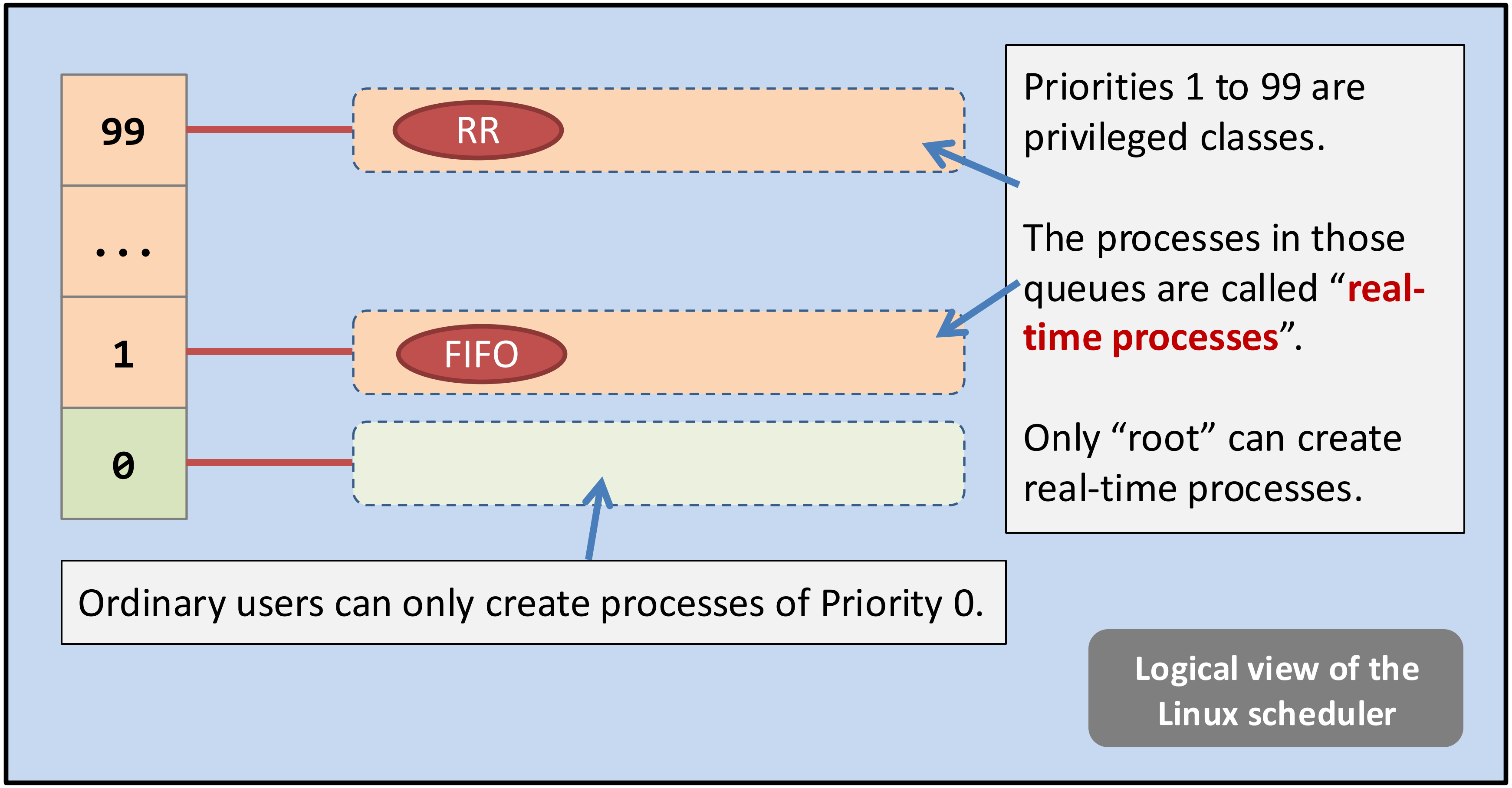
Priority class 5	Non-preemptive, FIFO
Priority class 4	Non-preemptive, SJF
Priority class 3	RR with quantum = 10 units.
Priority class 2	RR with quantum = 20 units.
Priority class 1	RR with quantum = 40 units.

Just an example.

A process drops to a lower priority class after it has used up its quantum and has the quantum recharged.

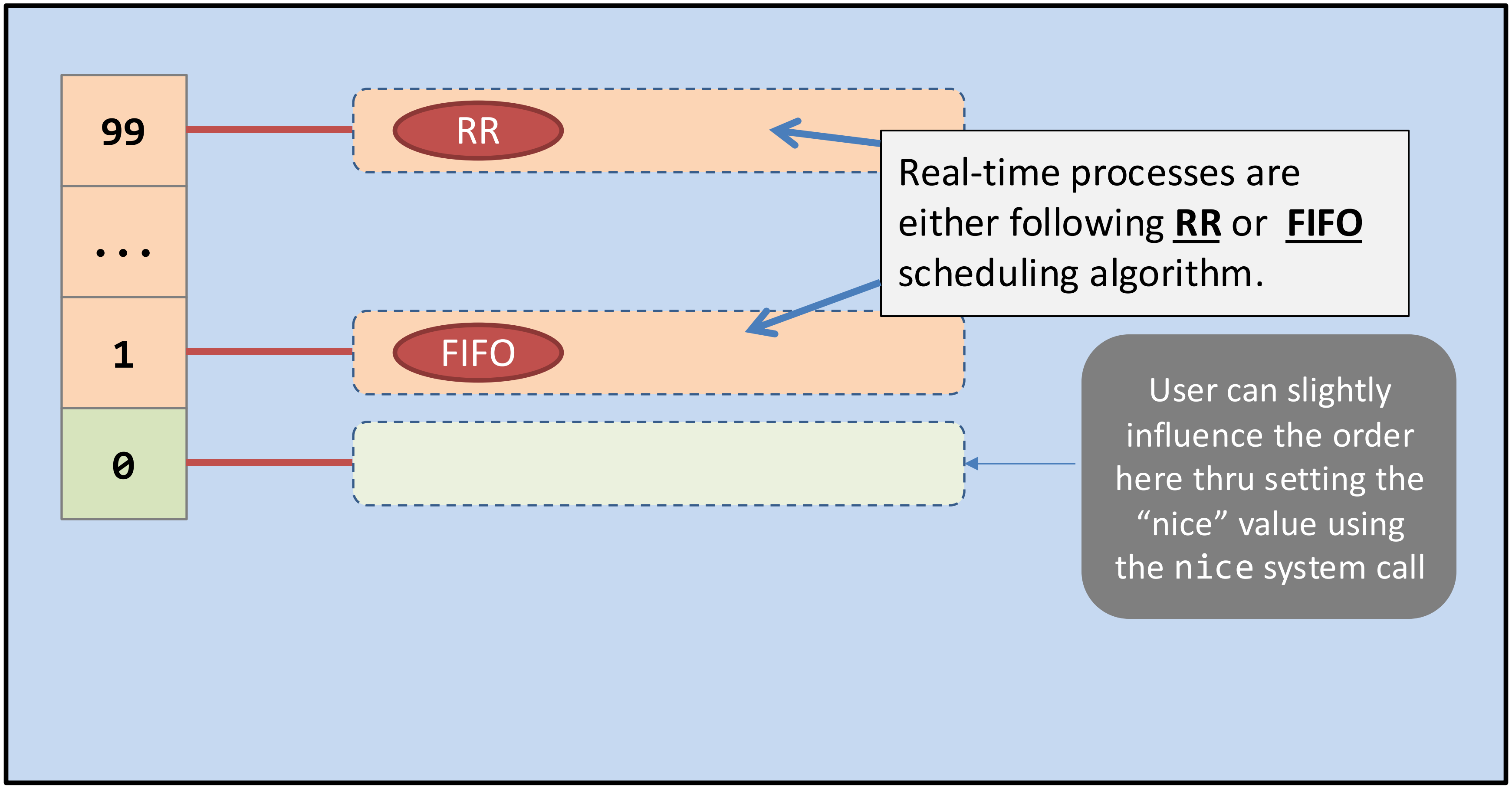
Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



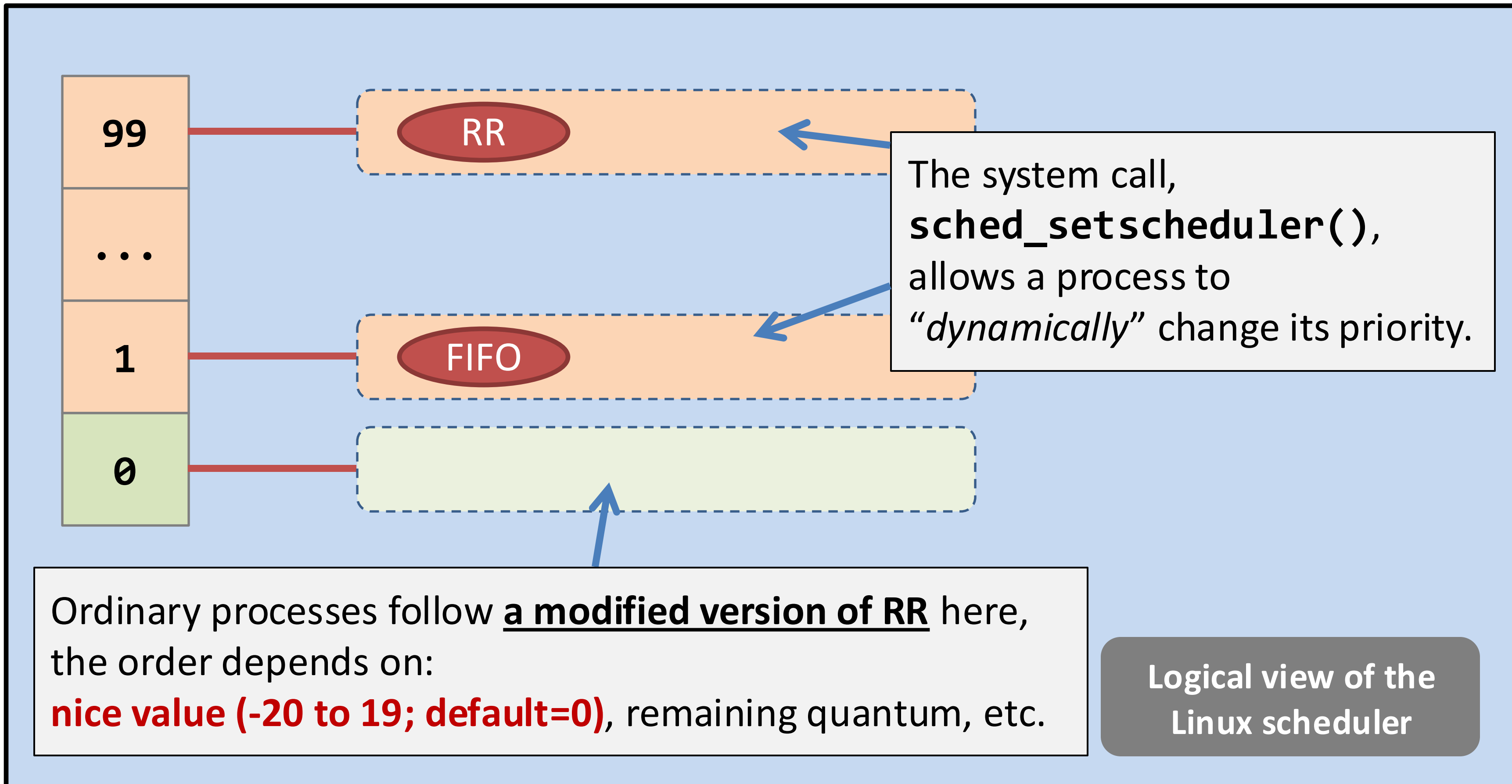
Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



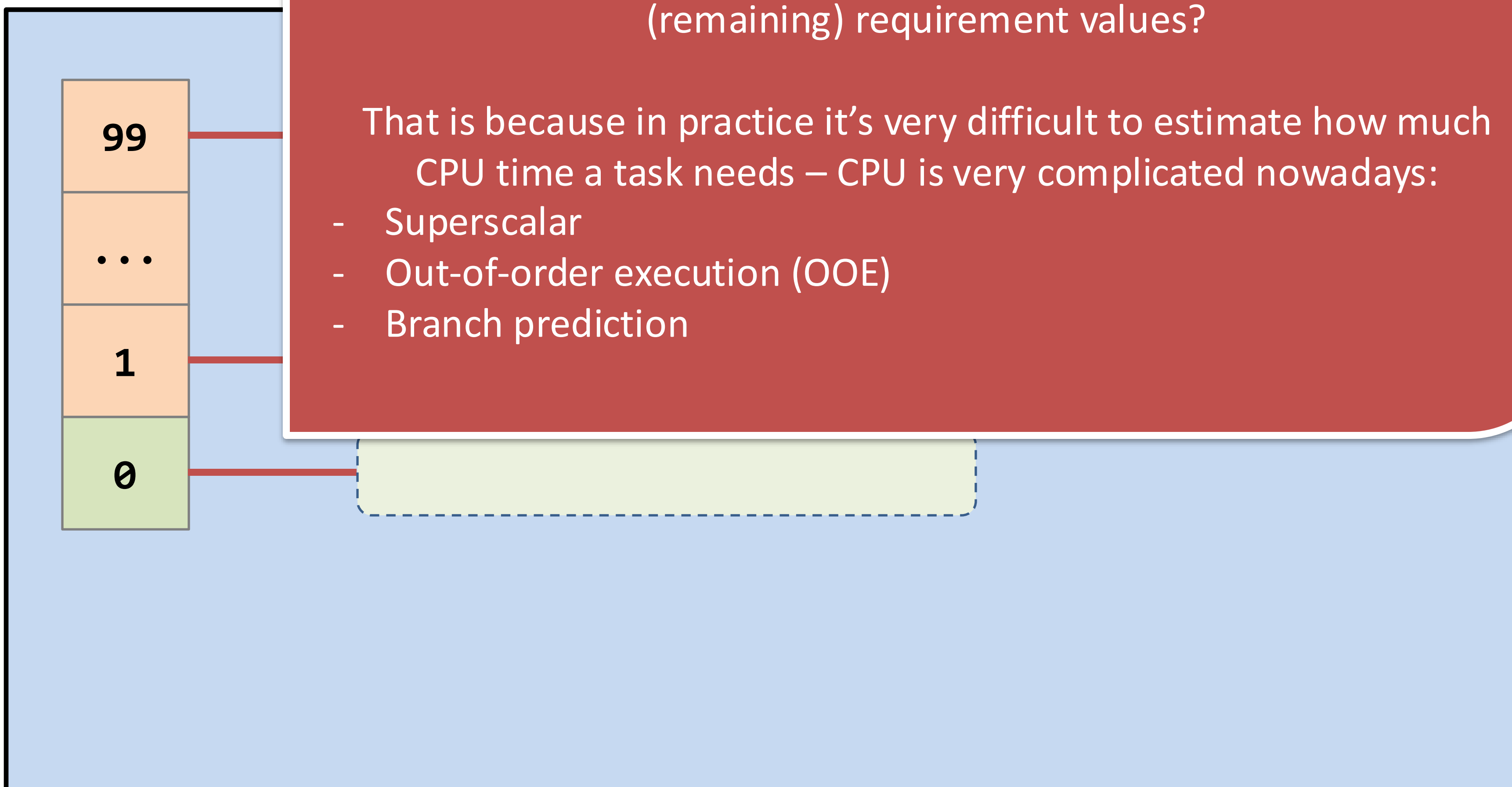
Multiple queue priority scheduling

- **Real example, the Linux Scheduler.**
  - A multiple queue, (kind of) static priority scheduler.



Multiple queue priority scheduling

- **Real example, the Linux Scheduler**
  - A multip



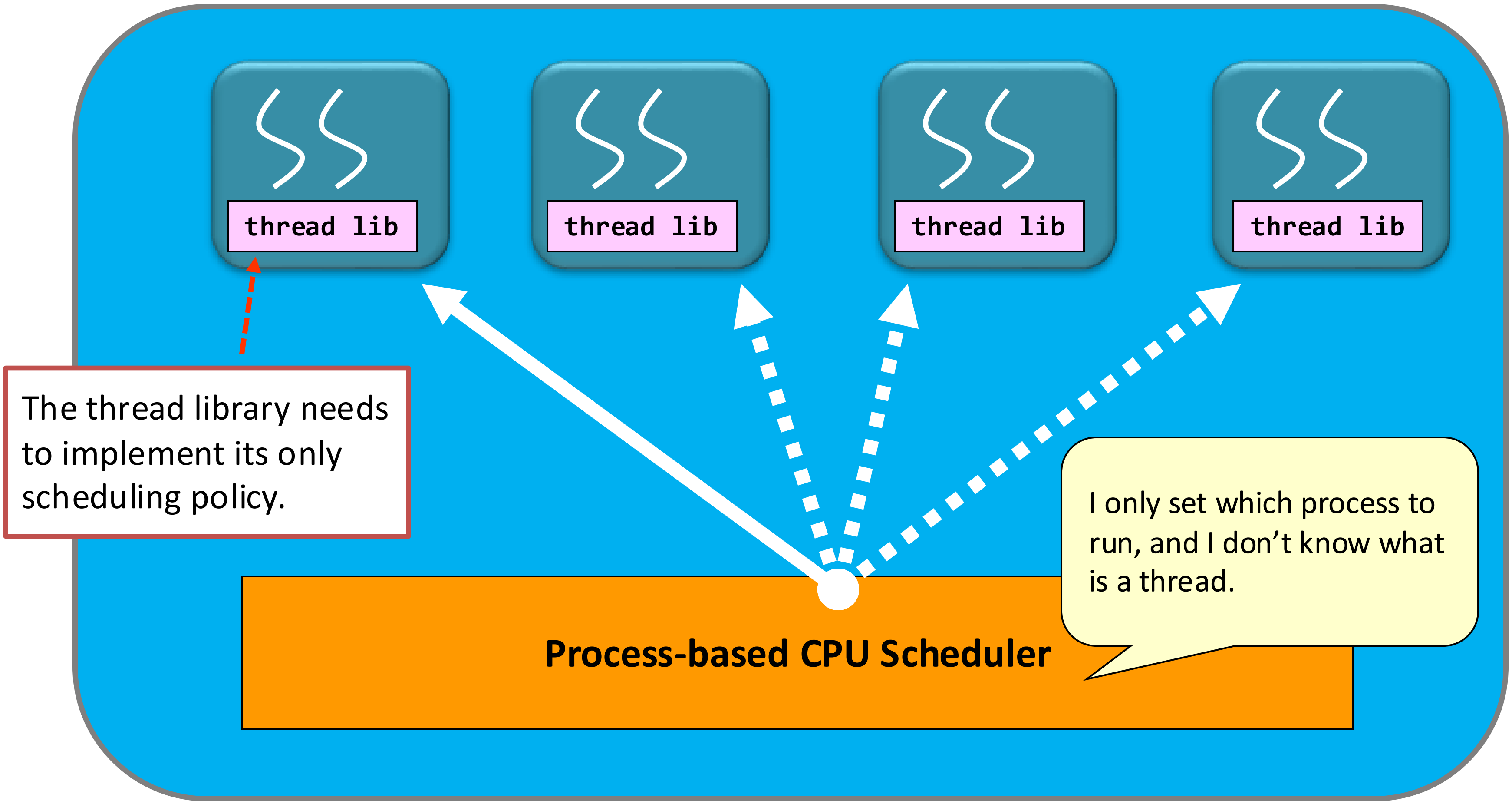
Do you notice that real (Linux) scheduler does not rely on task's CPU (remaining) requirement values?

That is because in practice it's very difficult to estimate how much CPU time a task needs – CPU is very complicated nowadays:

- Superscalar
- Out-of-order execution (OOE)
- Branch prediction

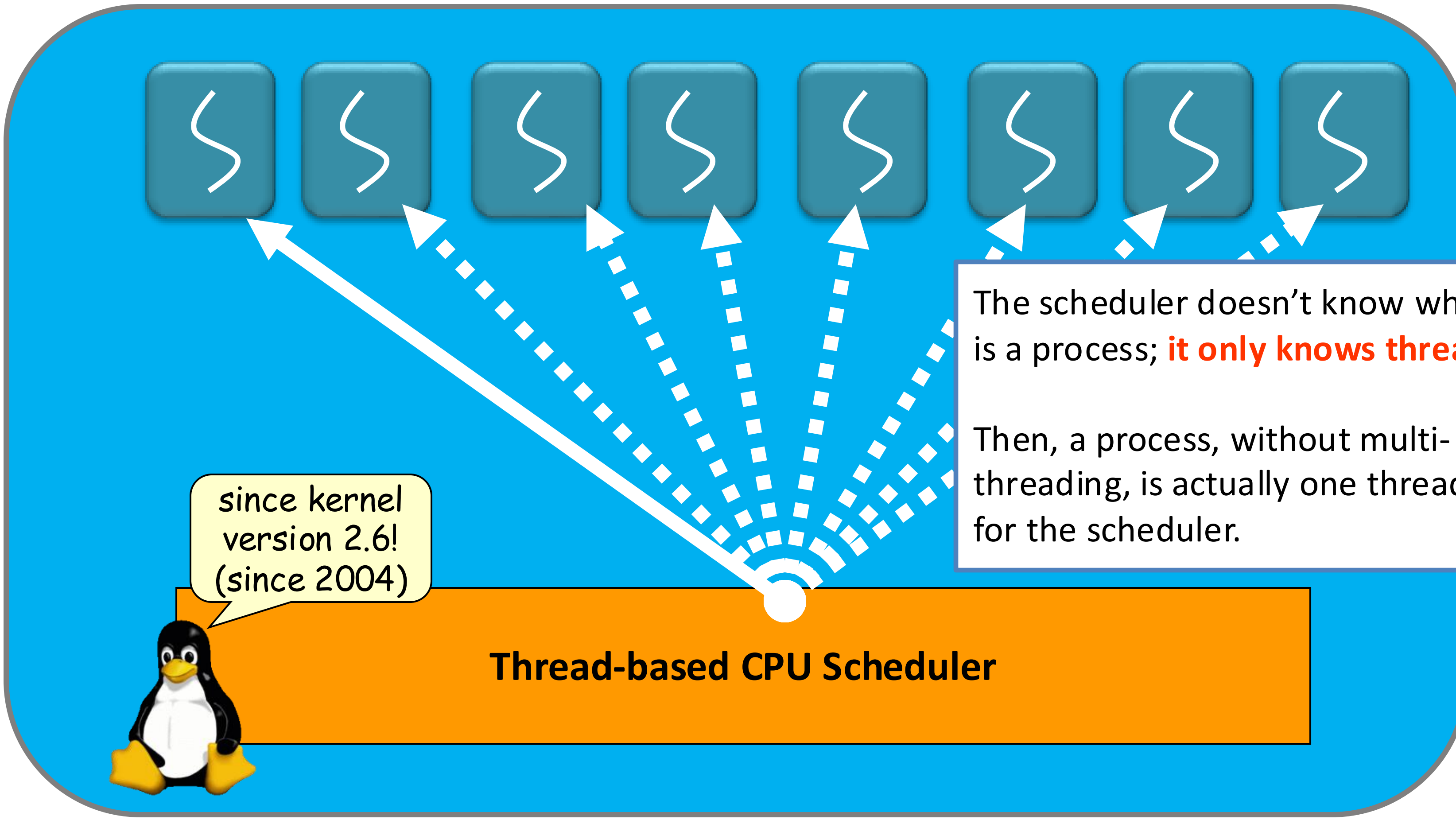
Thread Scheduling

- If a scheduler only interests in **processes**...



Thread Scheduling

- If a scheduler only interests in **threads**...



The scheduler doesn't know what is a process; **it only knows threads**.

Then, a process, without multi-threading, is actually one thread for the scheduler.

Scheduling for modern servers

- Symmetric multiprocessing (SMP)
  - All processes/threads **share a common ready queue**
    - ➔ **Race Condition**
  - Scheduling:
    - Each processor/core scheduler examines the common ready queue and selects a process to execute
  - Affinity Scheduling:
    - Soft affinity: **attempt** to keep the same process/thread on the same core
    - vs. Hard affinity: use `pthread_attr_setaffinity_np()` to manually bind a thread to a specific core in your pThread program
    - Especially important when we are reaching NUMA world