Lecture 9
CPU Scheduling

Prof. Yinqian Zhang

Spring 2022

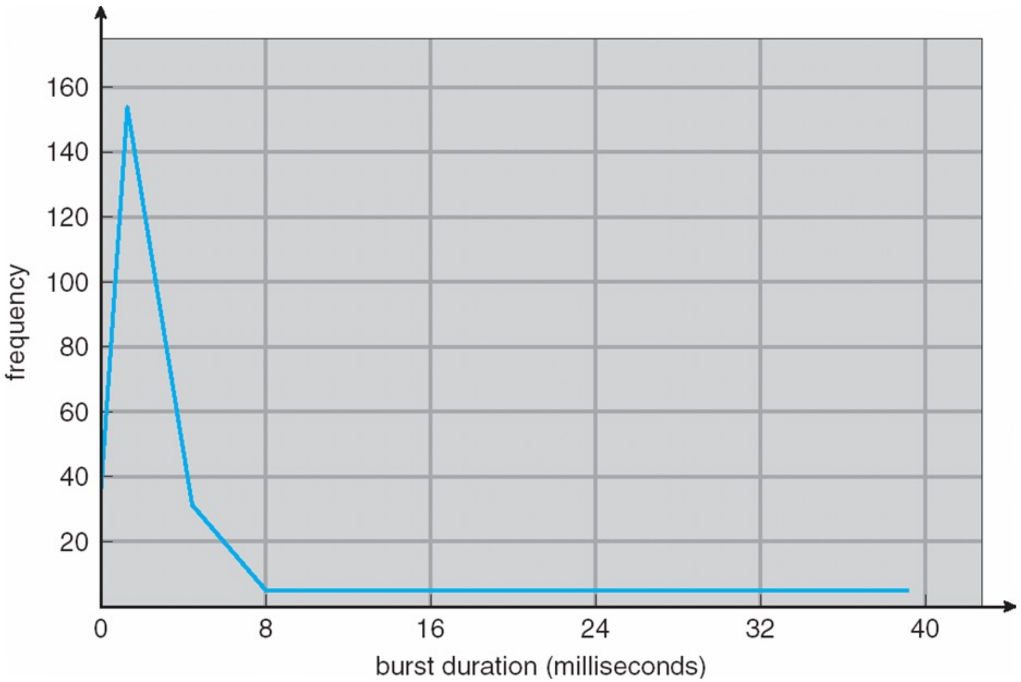
CPU Scheduling

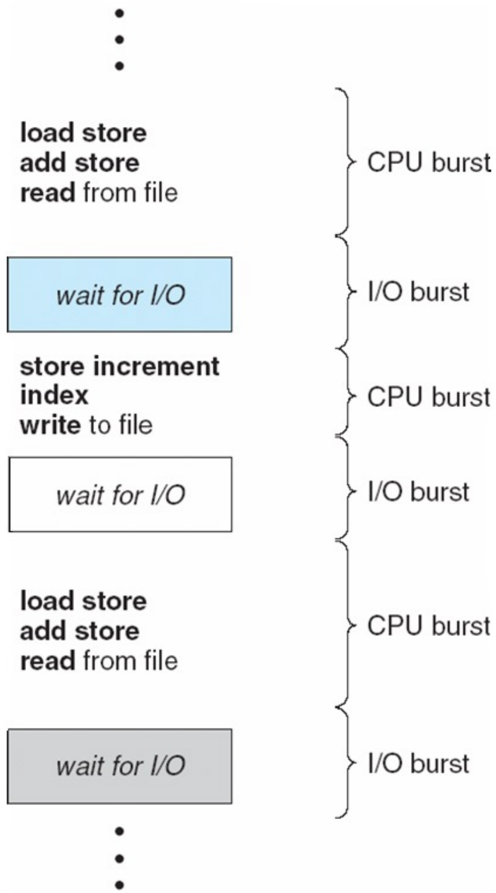
- Scheduling is important when multiple processes wish to run on a single CPU
  - CPU scheduler decides which process to run next
- Two types of processes
  - CPU bound and I/O bound

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**
<u>Examples</u> - AI course assignments.	**<u>Examples</u>** - **/bin/ls**, networking programs.

CPU Burst

- Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution





CPU Scheduler

- CPU scheduler selects one of the processes that are ready to execute and allocates the CPU to it
- CPU scheduling decisions may take place when a process:
  - 1. Switches from running to waiting state
  - 2. Switches from running to ready state
  - 3. Switches from waiting to ready
  - 4. Terminates
- A scheduling algorithm takes place **only** under circumstances 1 and 4 is **non-preemptive**
- All other scheduling algorithms are **preemptive**

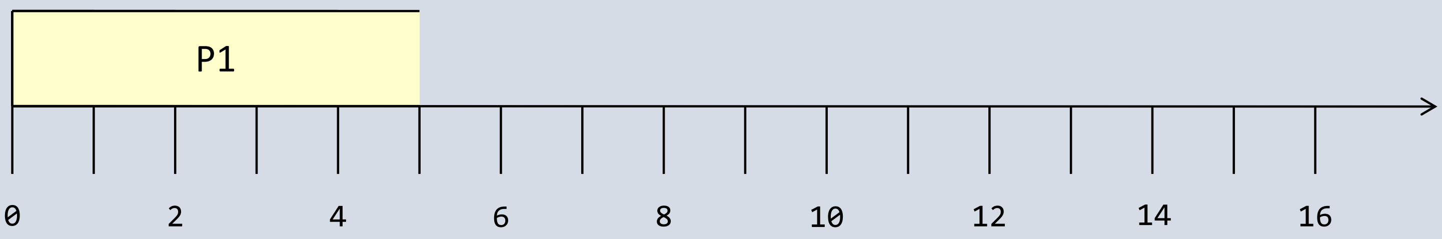
Scheduling Algorithm Optimization Criteria

- Given a set of processes, with
  - **Arrival time**: the time they arrive in the CPU ready queue (from waiting state or from new state)
  - **CPU requirement**: their expected CPU burst time
- Minimize average turnaround time
  - **Turnaround time**: The time between the arrival of the task and the time it is blocked or terminated.
- Minimize average waiting time
  - **Waiting time**: The accumulated time that a task has waited in the ready queue.
- Reduce the number of context switches

Different Algorithms

- Shortest-job-first (SJF)
- Round-robin (RR)
- Priority scheduling
- Multiple queue priority scheduler

Non-preemptive SJF



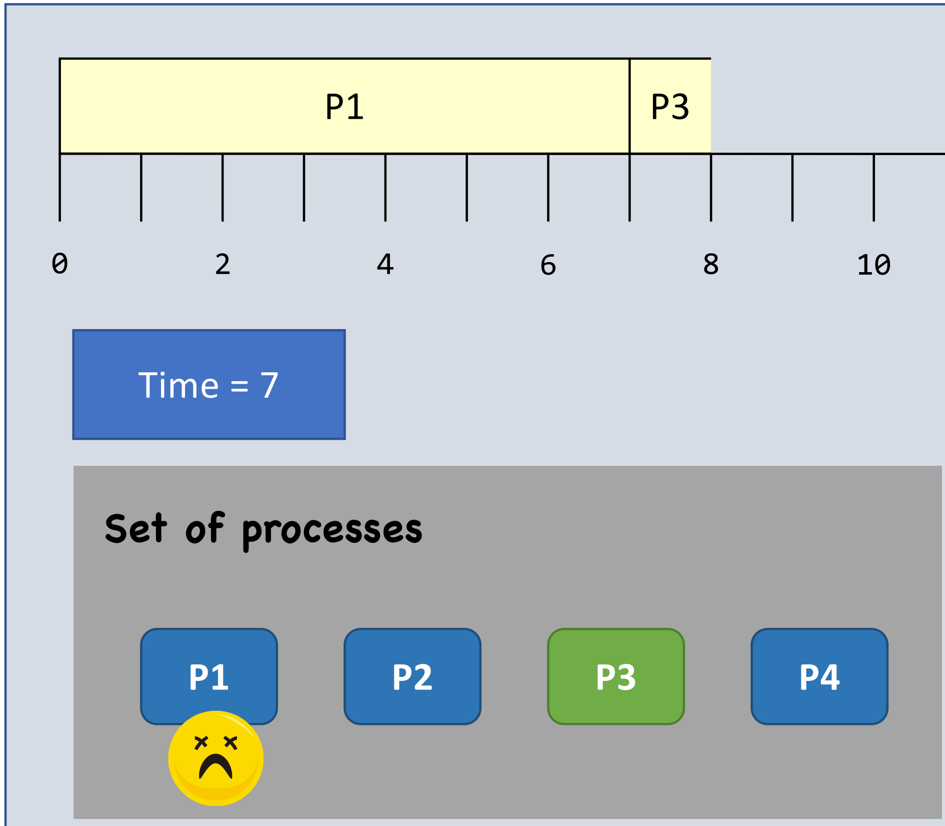
Time = 5

Set of processes

P1 P2 P3 P4

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

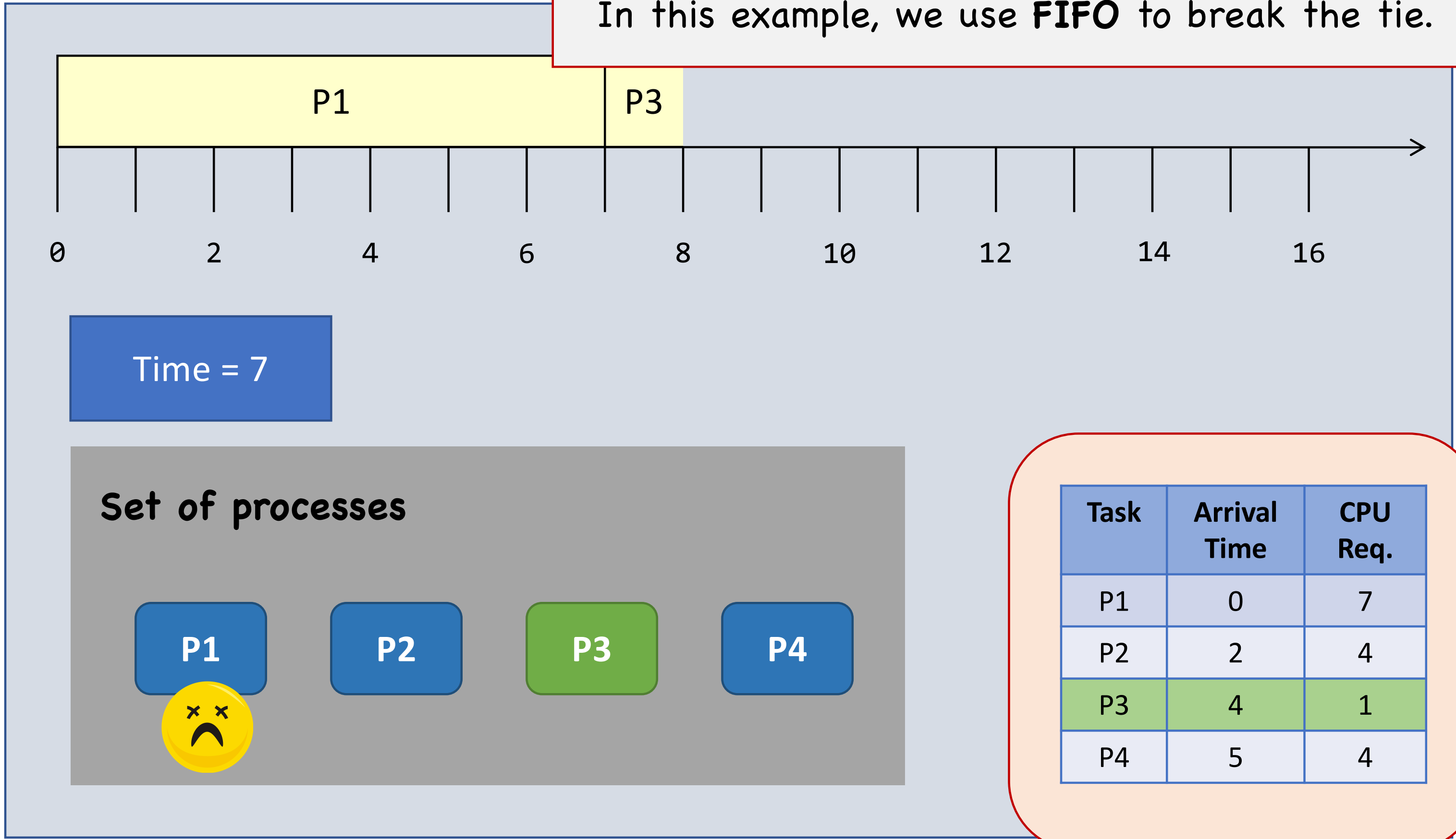
Non-preemptive SJF



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

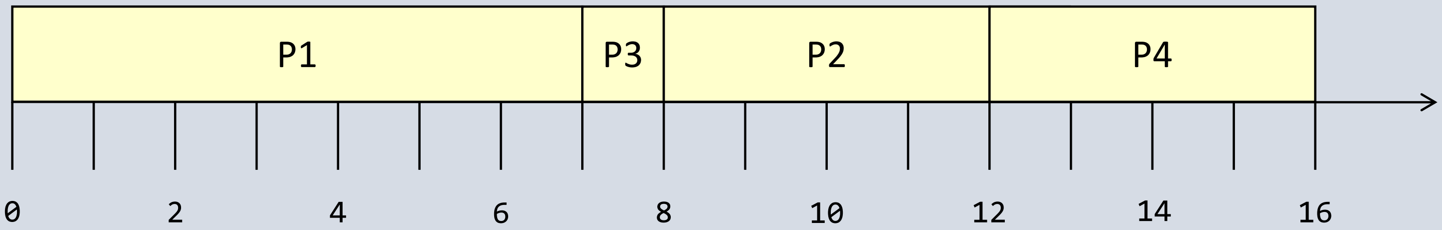
Non-preemptive SJF

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

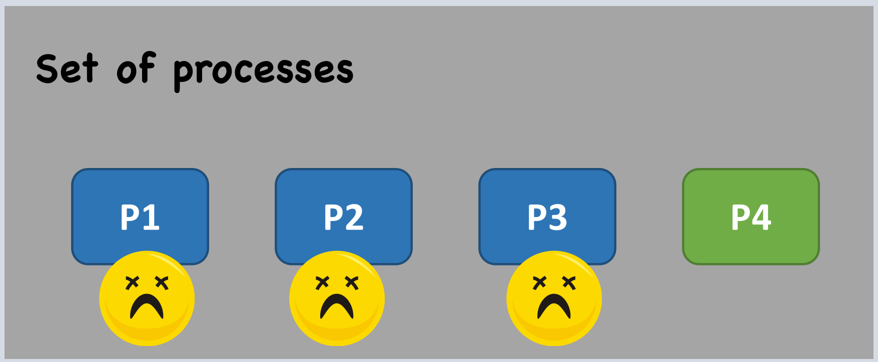


Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

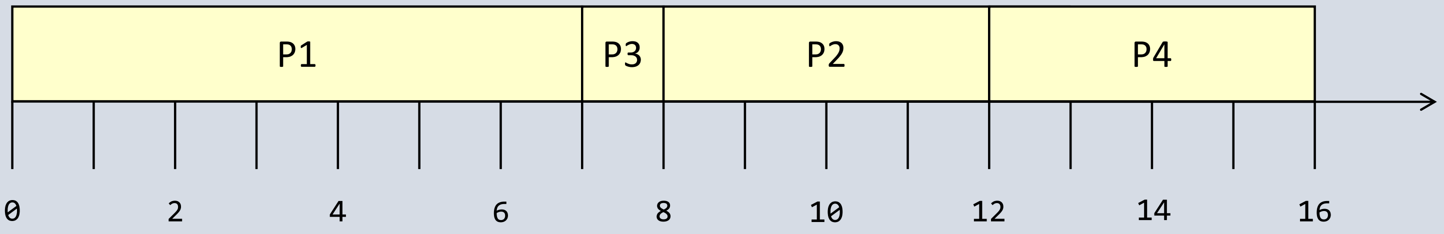


Time = 16



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.

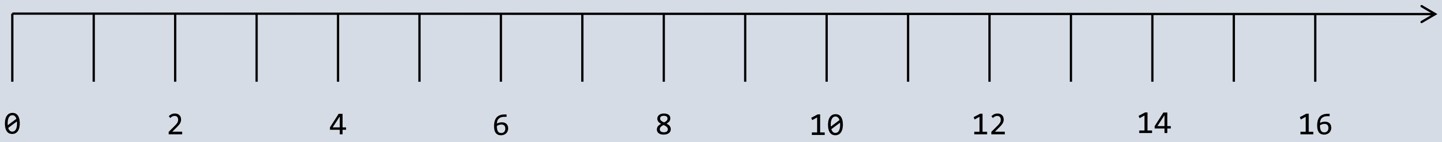
Turnaround time:

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

Preemptive SJF

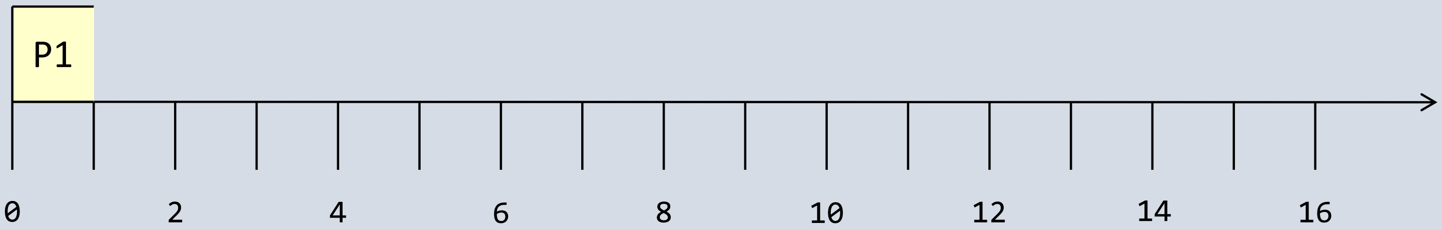


Whenever a new process arrives in the ready queue (either from waiting or from new state), the scheduler steps in and selects the next task based on **their remaining CPU requirements.**

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

Animation; don't print

Preemptive SJF



Time = 0

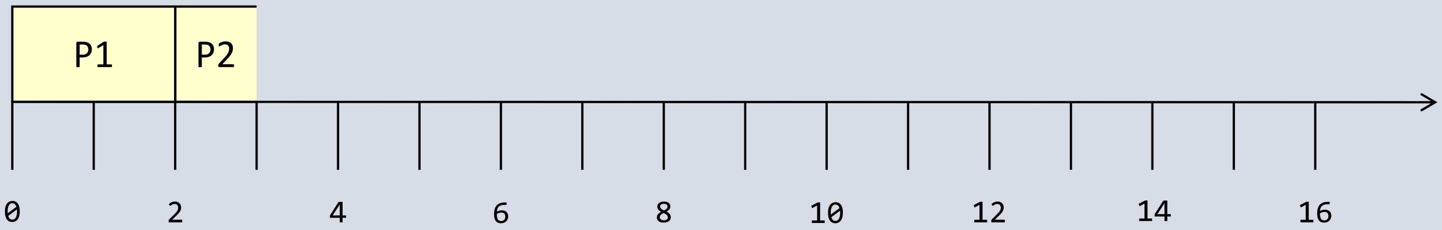
Set of processes

P1

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

Animation; don't print

Preemptive SJF



Time = 2

P2 is selected!

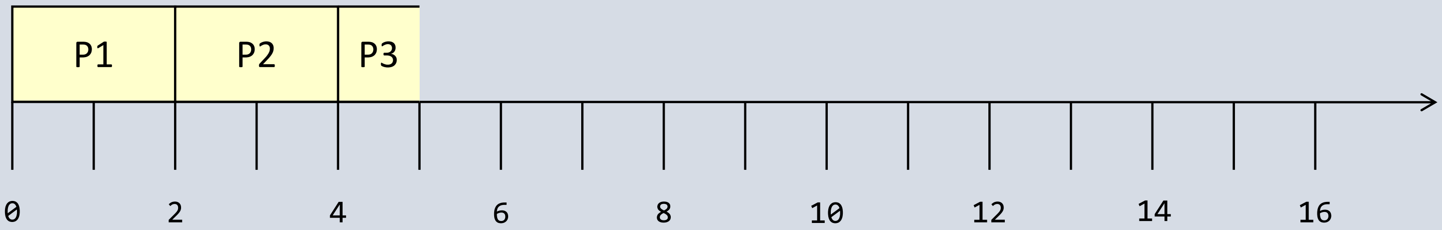
Set of processes

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

Animation; don't print

Preemptive SJF



Time = 4

P3 is selected!

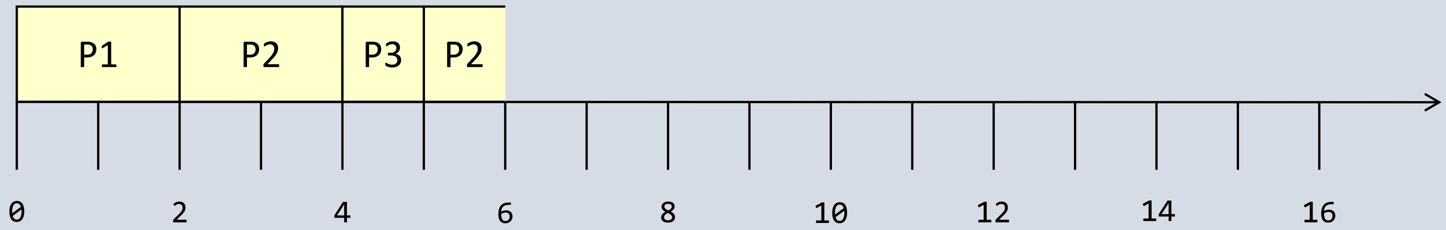
Set of processes

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

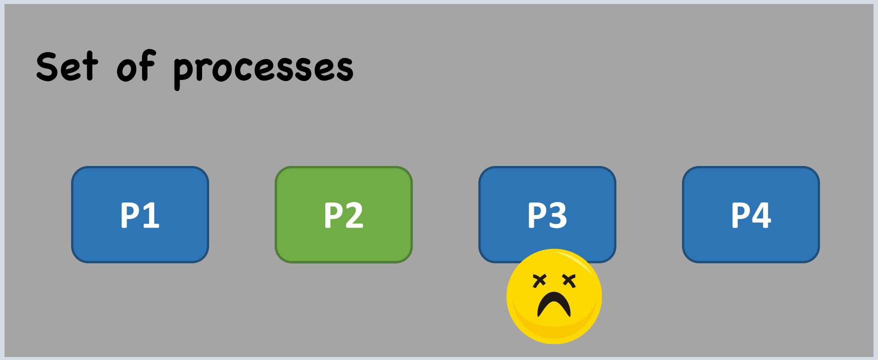
Animation; don't print

Preemptive SJF



Time = 5

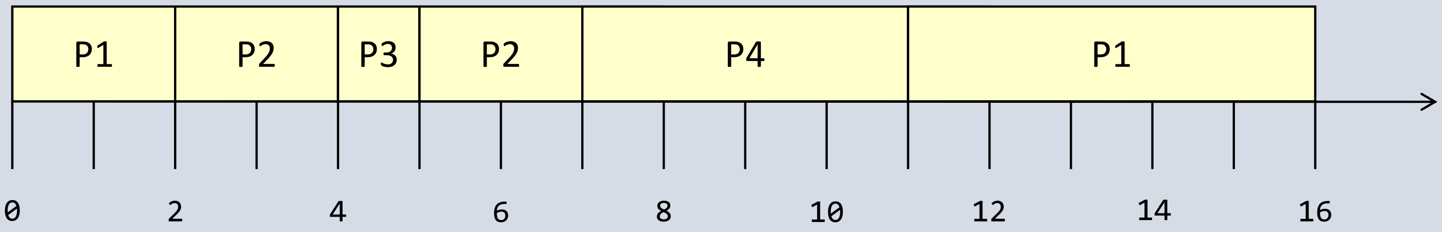
P2 is selected!



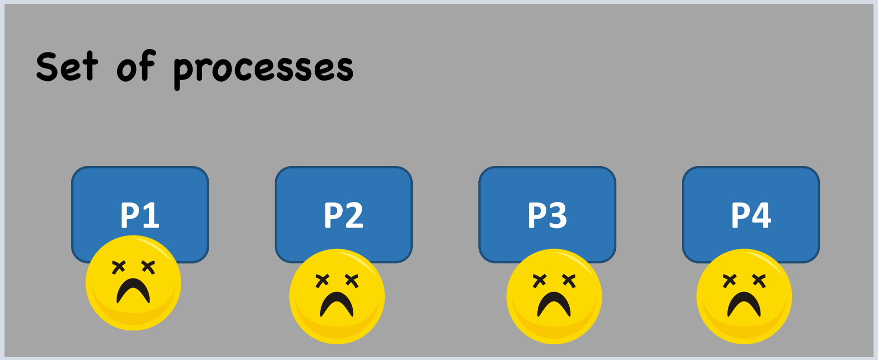
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	4

Animation; don’t print

Preemptive SJF

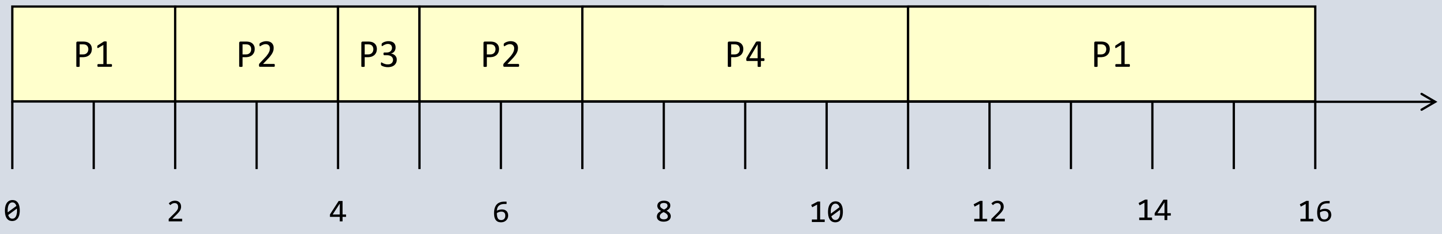


Time = 16



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**.

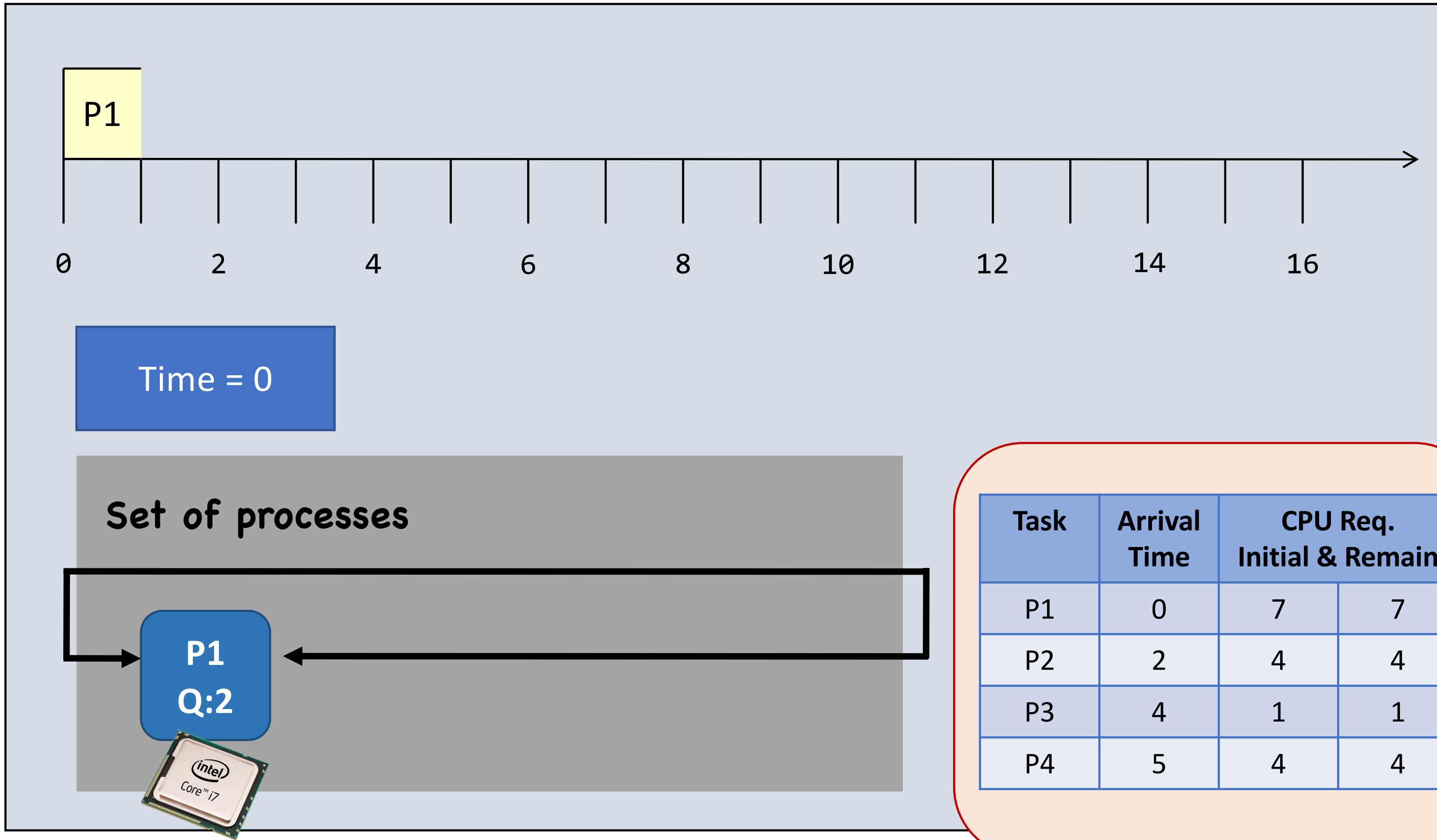
Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Round Robin (RR)

- Round-Robin (RR) scheduling is preemptive.
  - Every process is given a **quantum (**the amount of time allowed to execute).
  - Whenever the quantum of a process is used up (i.e., 0), the process is preempted
  - 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
- New processes are added to the tail of the ready queue
  - New process's arrival won't trigger a new selection decision

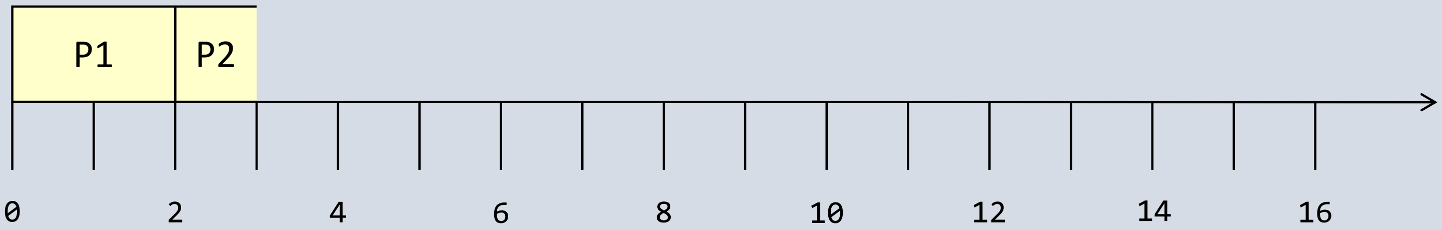
Animation; don't print

Round Robin (Quantum = 2)



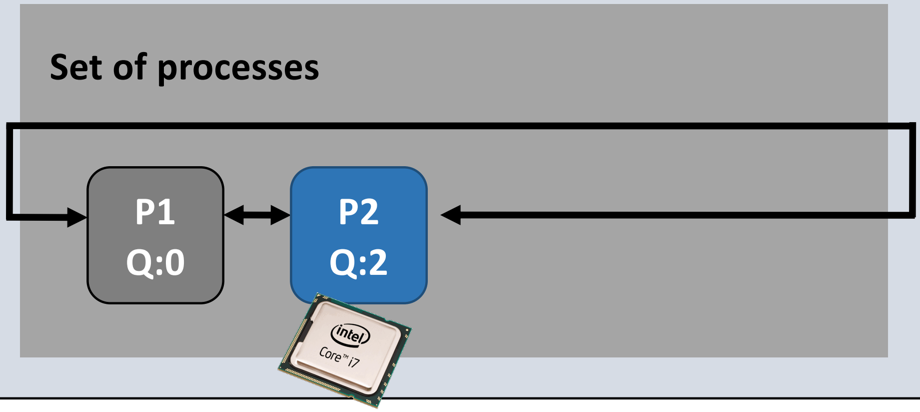
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round Robin (Quantum = 2)



Time = 2

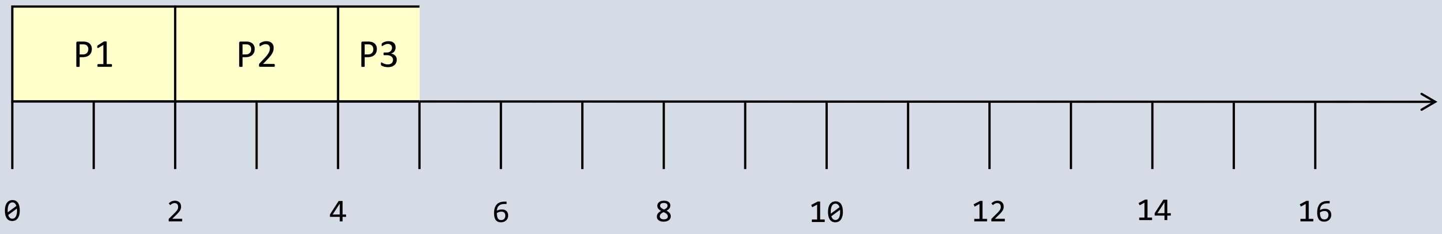
P1's quantum is 0;
P2 is selected!



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

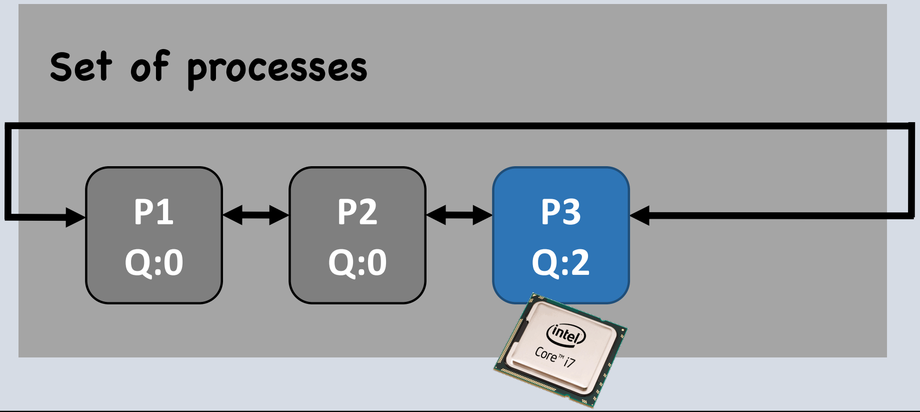
Animation; don't print

Round Robin (Quantum = 2)



Time = 4

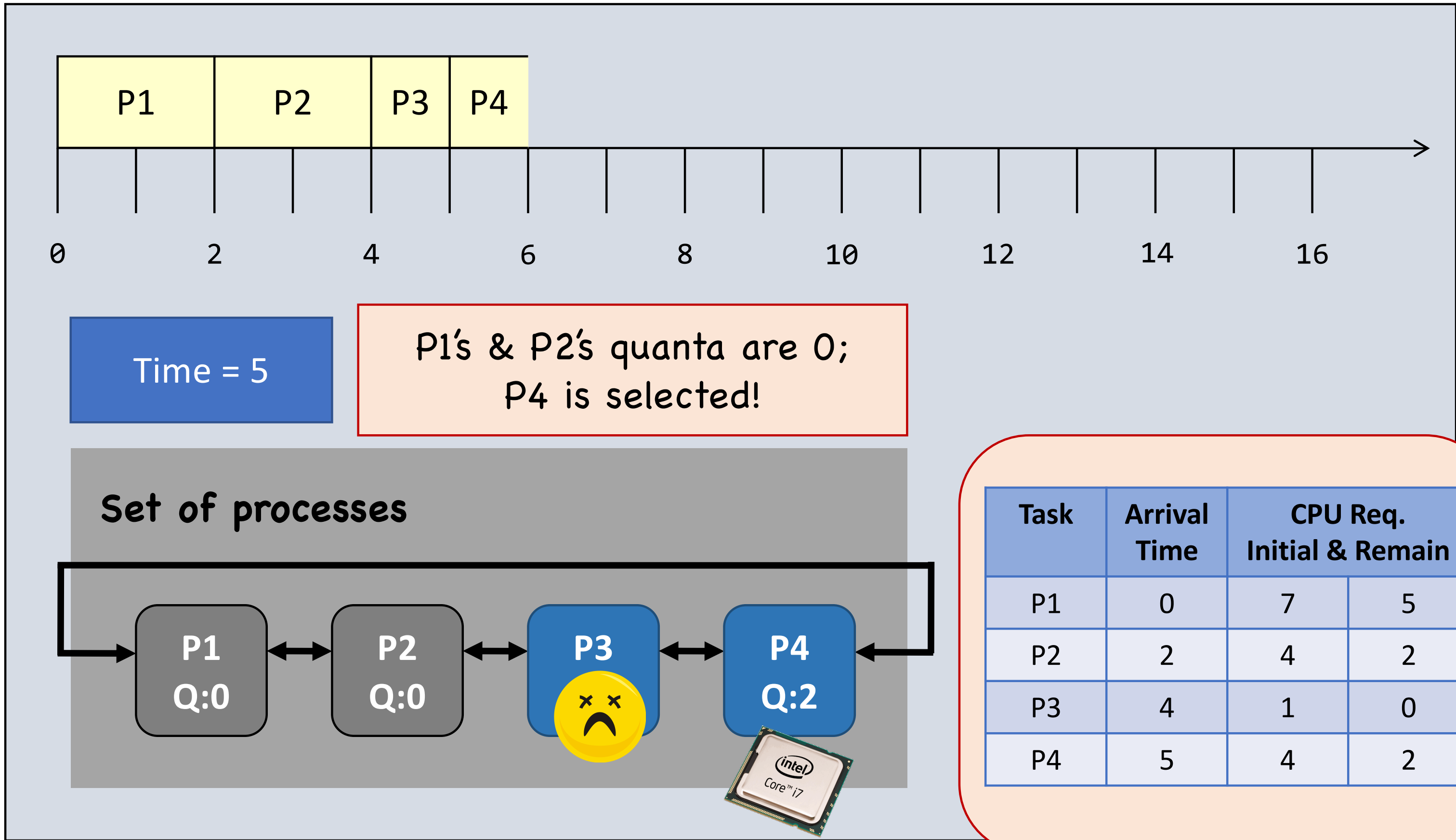
P1's & P2's quanta are 0;
P3 is selected!



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

Animation; don't print

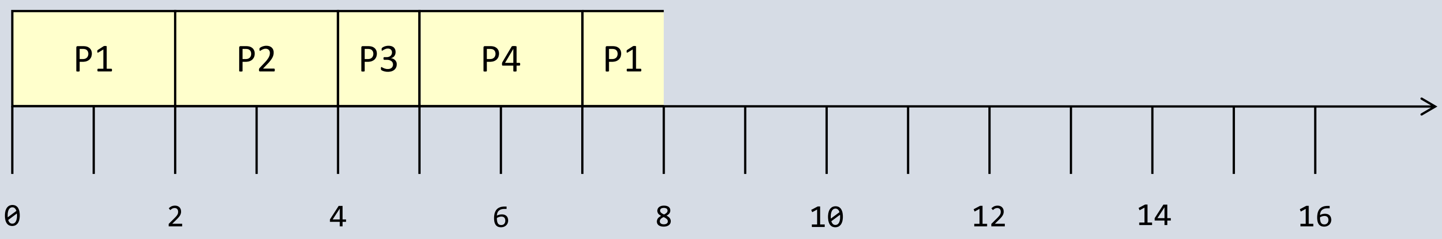
Round Robin (Quantum = 2)



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

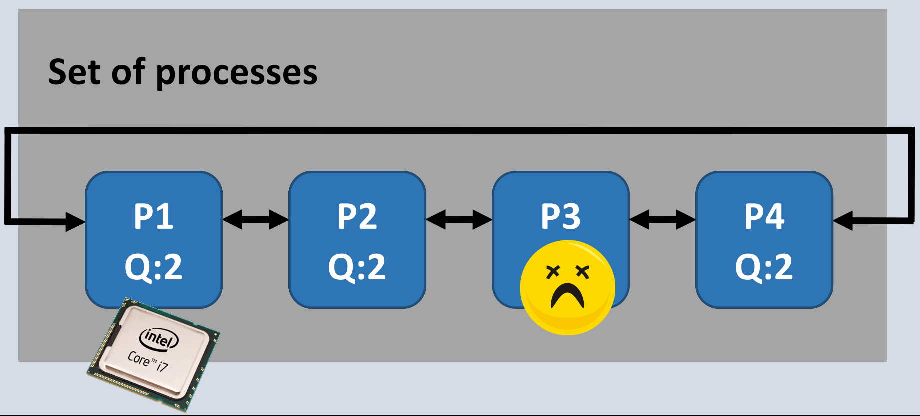
Animation; don't print

Round Robin (Quantum = 2)



Time = 7

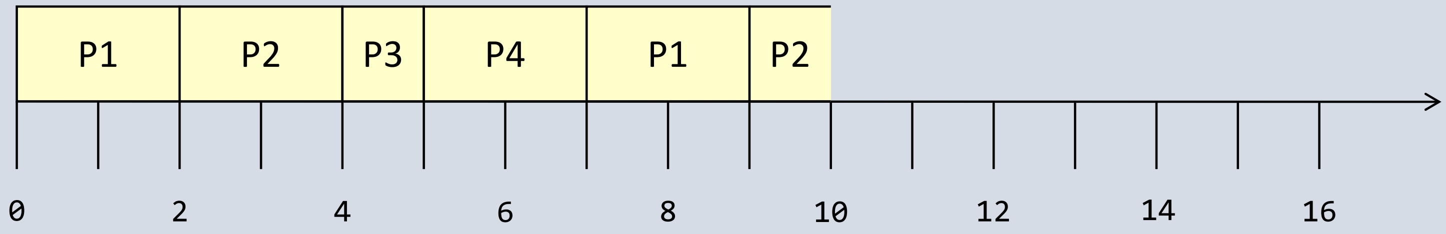
Now, recharge is needed.
P1 is selected.



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

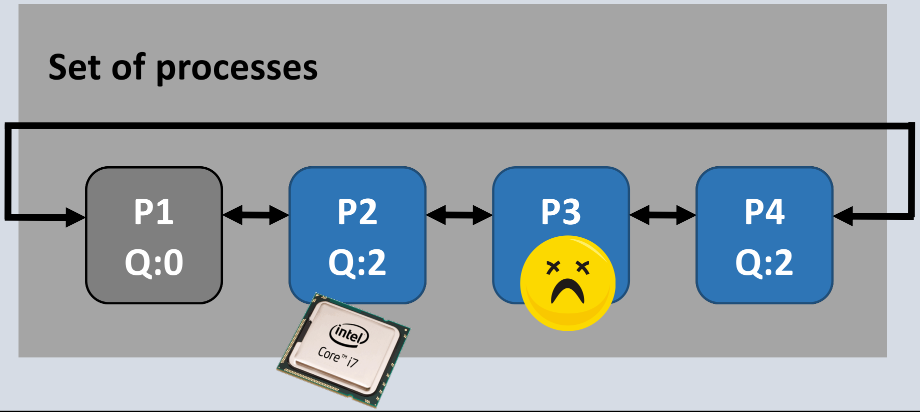
Animation; don't print

Round Robin (Quantum = 2)



Time = 9

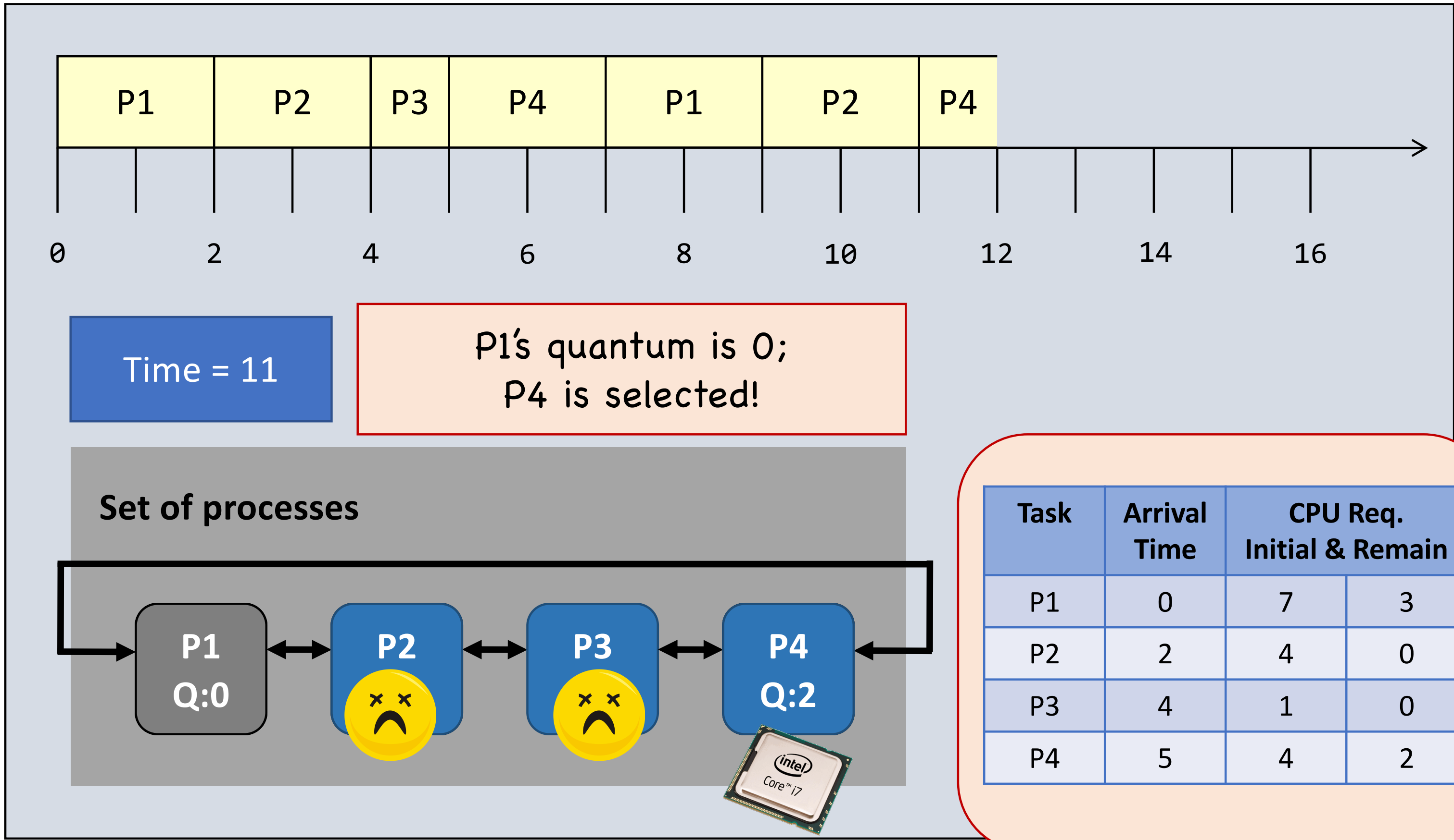
P1's quantum is 0;
P2 is selected!



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

Animation; don't print

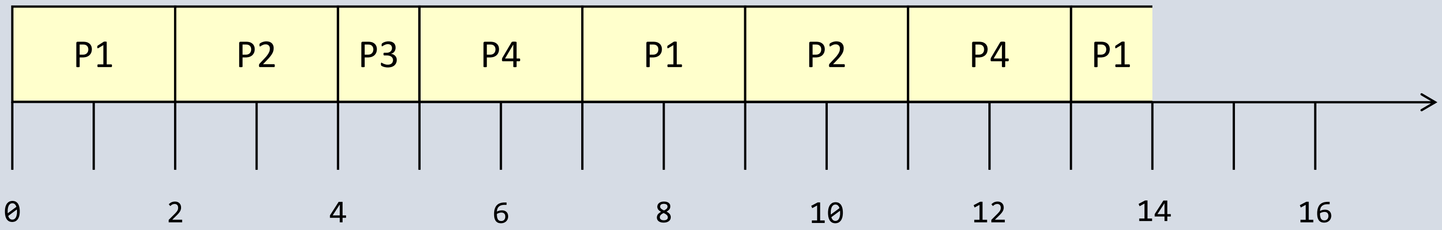
Round Robin (Quantum = 2)



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

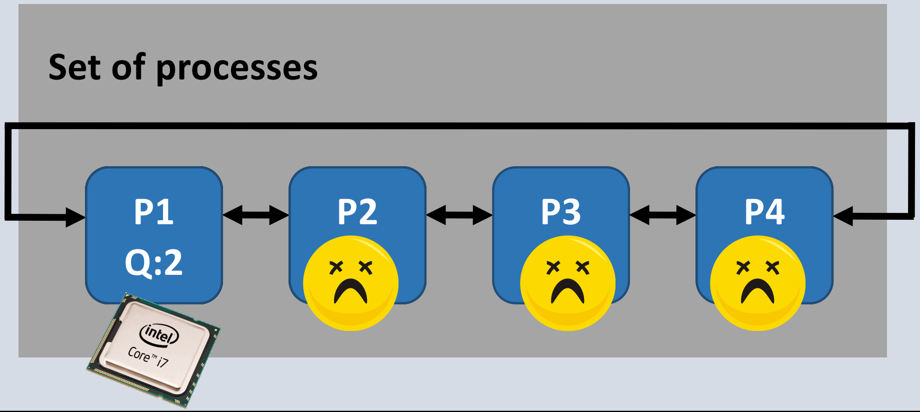
Animation; don't print

Round Robin (Quantum = 2)



Time = 13

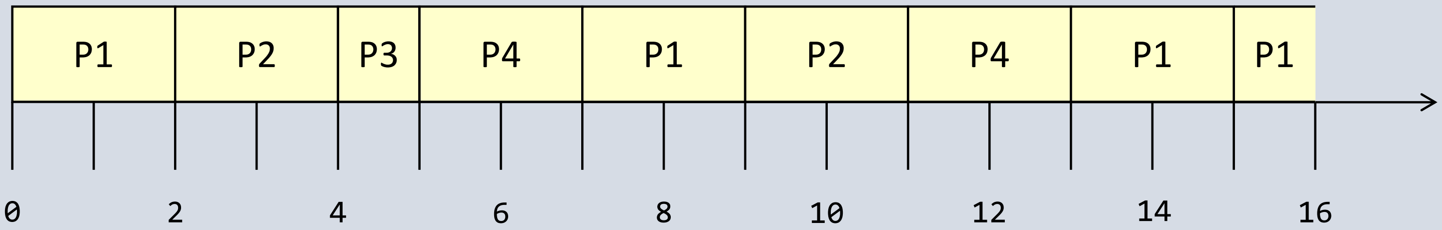
Now, recharge is needed.
P1 is selected.



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

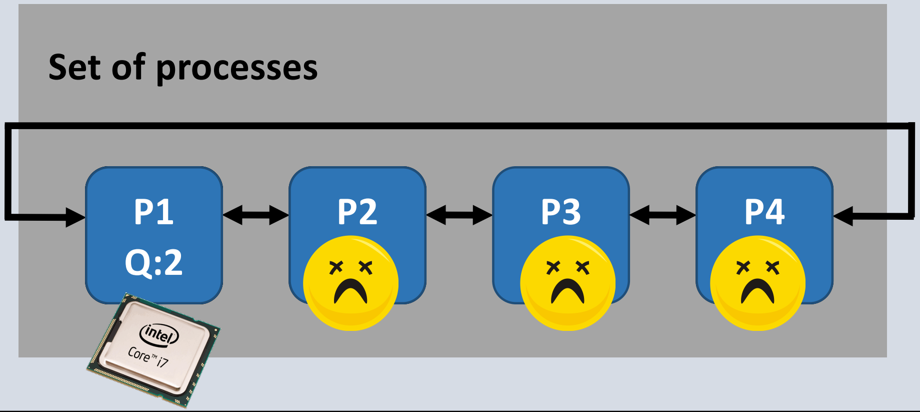
Animation; don't print

Round Robin (Quantum = 2)



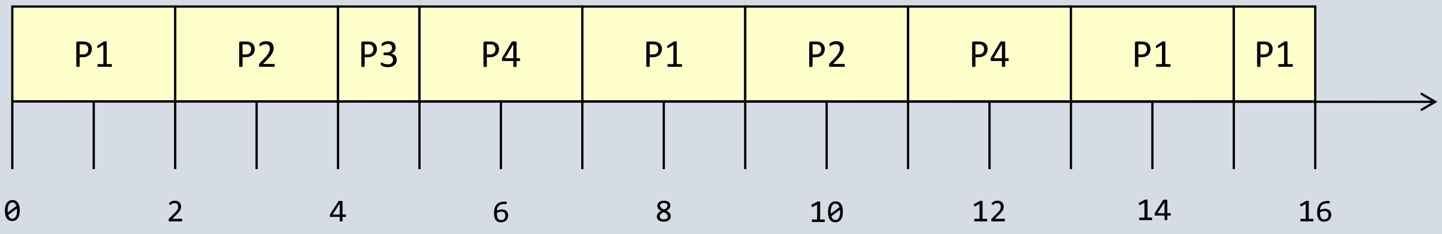
Time = 15

Now, recharge is needed.
P1 is selected.



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 (Quantum = 2)



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 v.s. 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

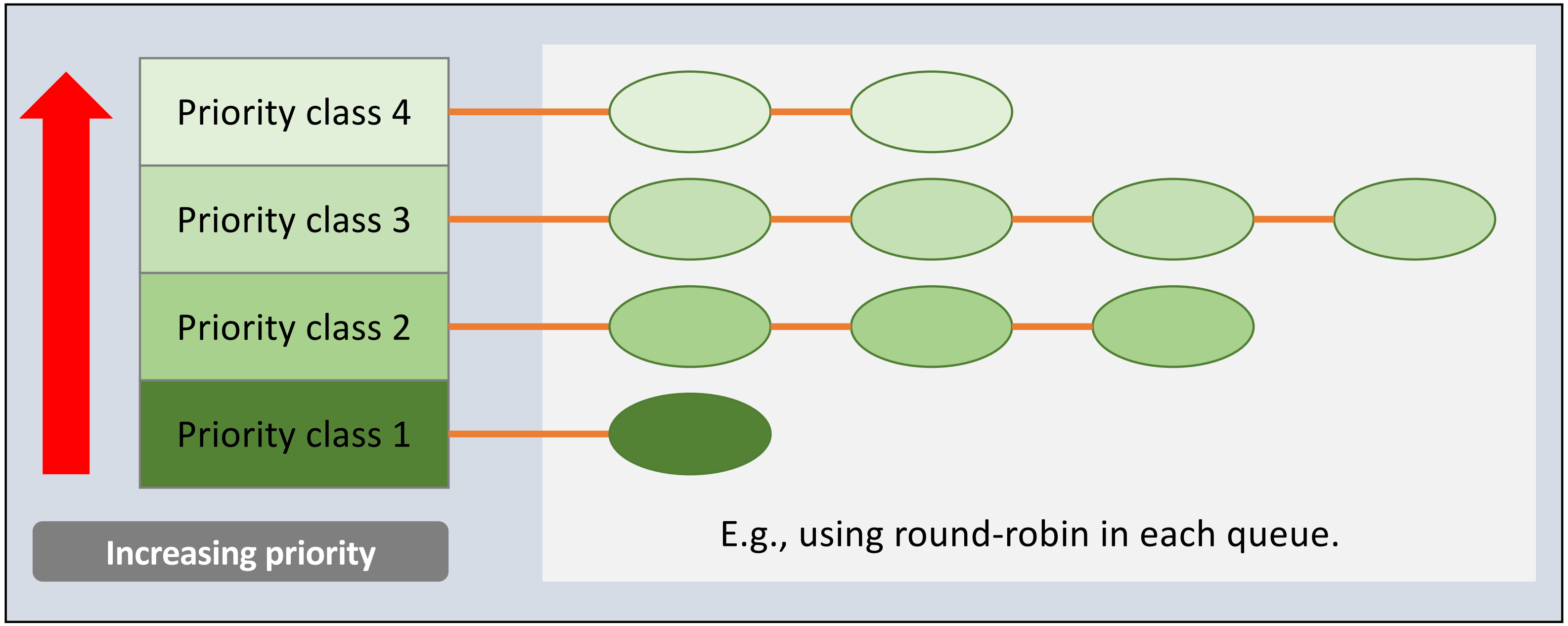
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)
  - Nonpreemptive: newly arrived process simply put into the queue
  - Preemptive: if the priority of the newly arrived process is higher than priority of the currently running process---preempt the CPU
- Static priority and dynamic priority
  - static priority: fixed priority throughout its lifetime
  - dynamic priority: priority changes over time
- SJF is a priority scheduling where priority is the next CPU burst time

Priority Scheduling (Cont'd)

- Problem ≡ **Starvation** – low priority processes may never execute
  - Rumors has it that when they shut down the IBM 7094 at MIT in 1973, they found a low priority process that had been submitted in 1967 and had not yet been run.
- Solution ≡ **Aging** – as time progresses increase the priority of the process
  - Example: priority range from 127 (low) to 0 (high)
  - Increase priority of a waiting process by 1 every 15 minutes
  - 32 hours to reach priority 0 from 127

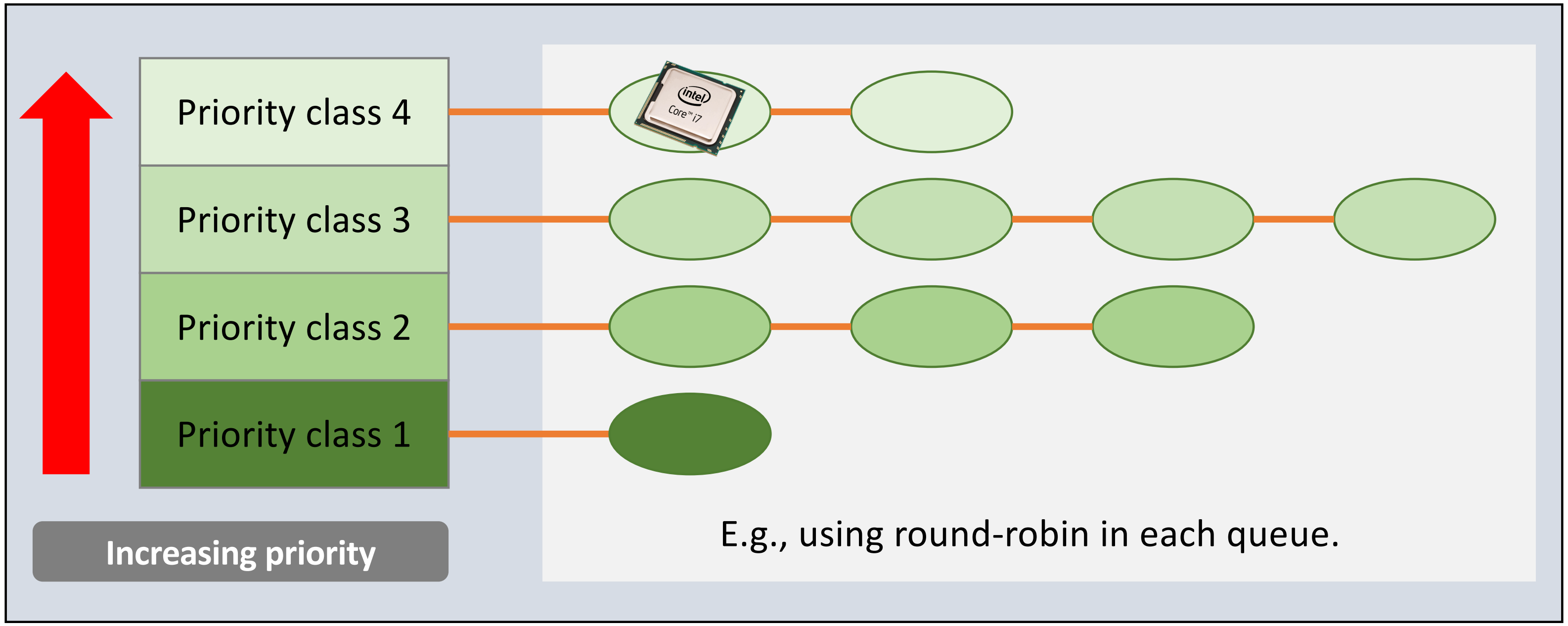
Static Priority Scheduling

- Each process is assigned a fix priority when they are submitted



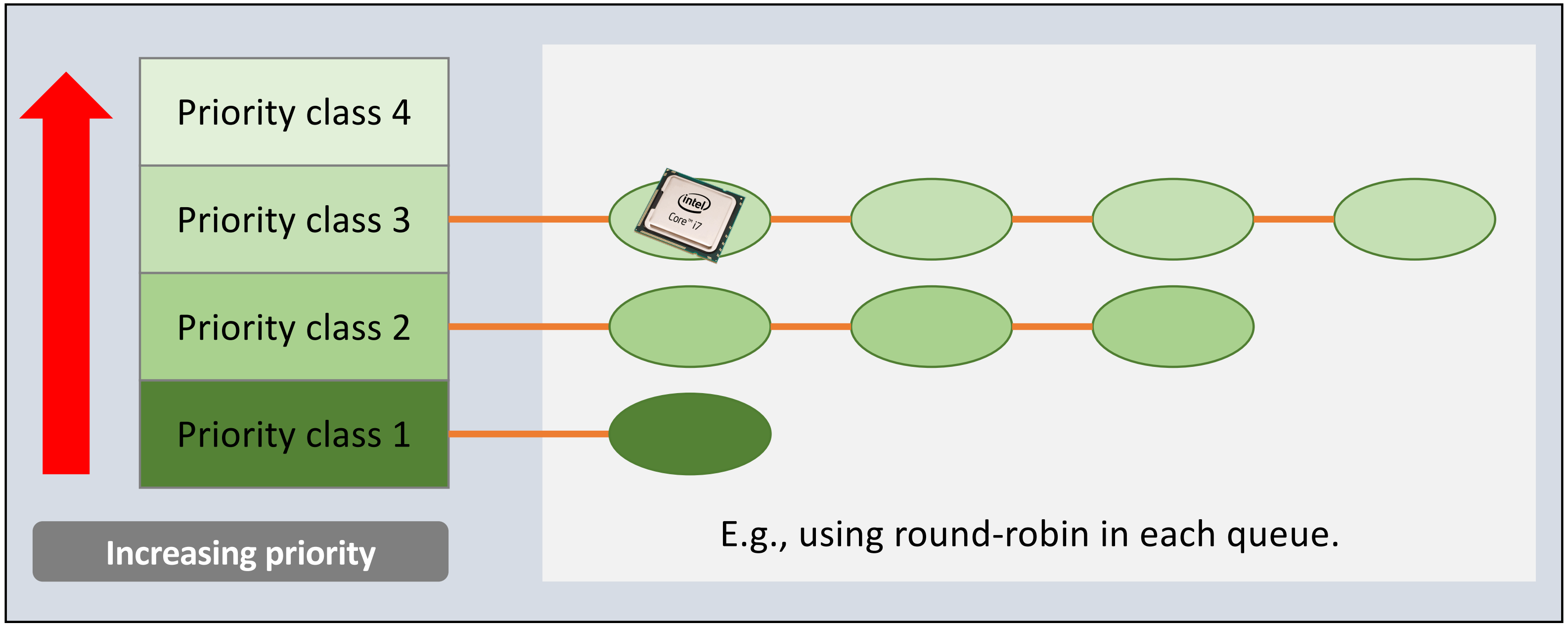
Static Priority Scheduling

- The highest priority class will be selected.
  - The tasks are usually be short-lived, but important



Static Priority Scheduling

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



Multiple Queue Priority Scheduling

- Multiple priority classes
- In each priority class, **different schedulers** may be deployed.
  - Can be a mix of static priority and dynamic priority.

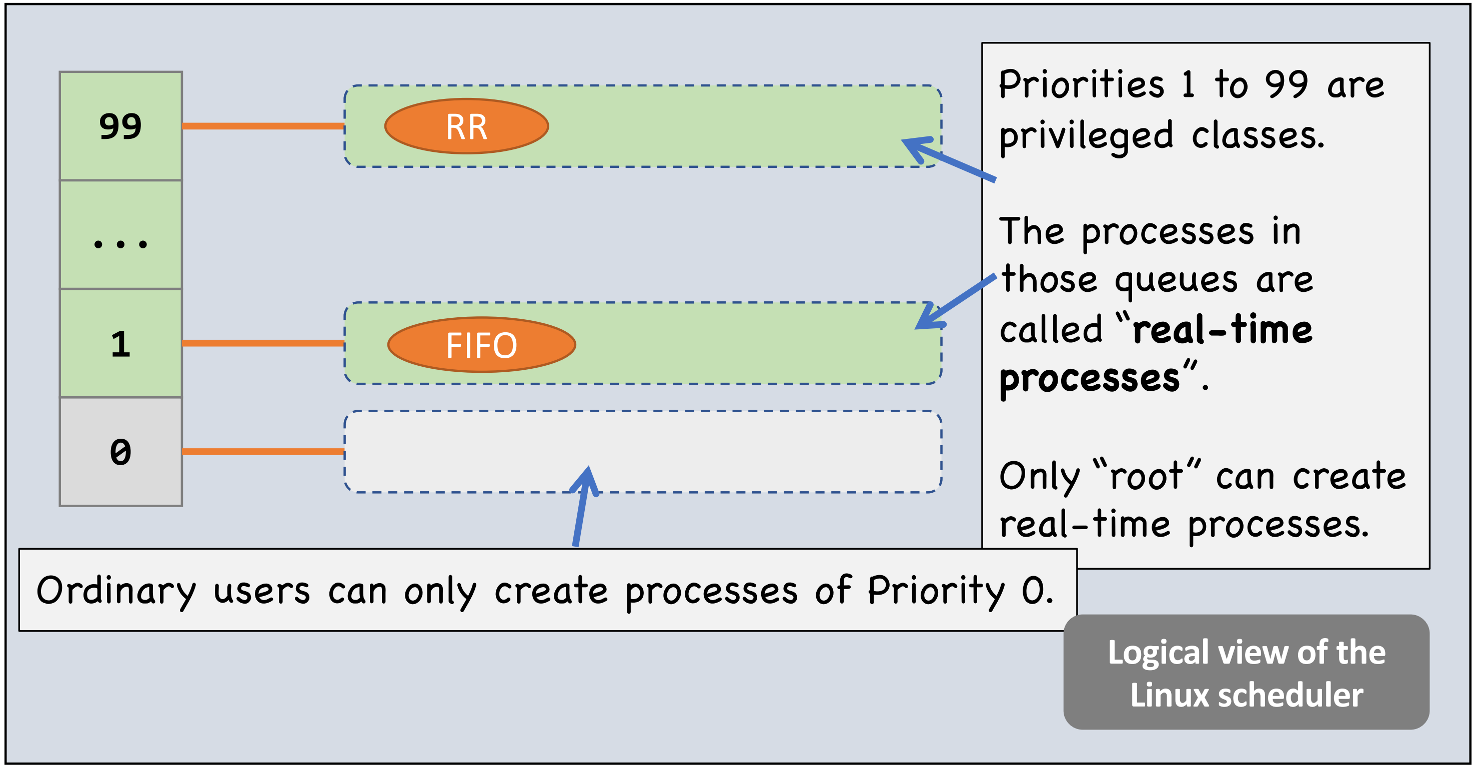
Priority class 5	Non-preemptive, FIFO
Priority class 4	Non-preemptive, SJF
Priority class 3	RR with quantum = 10 units.
Priority class 2	RR with quantum = 20 units.
Priority class 1	RR with quantum = 40 units.

Just an example.

Multi-level feedback queue scheduling: A process drops to a lower priority class after it has used up its quantum and has the quantum recharged.

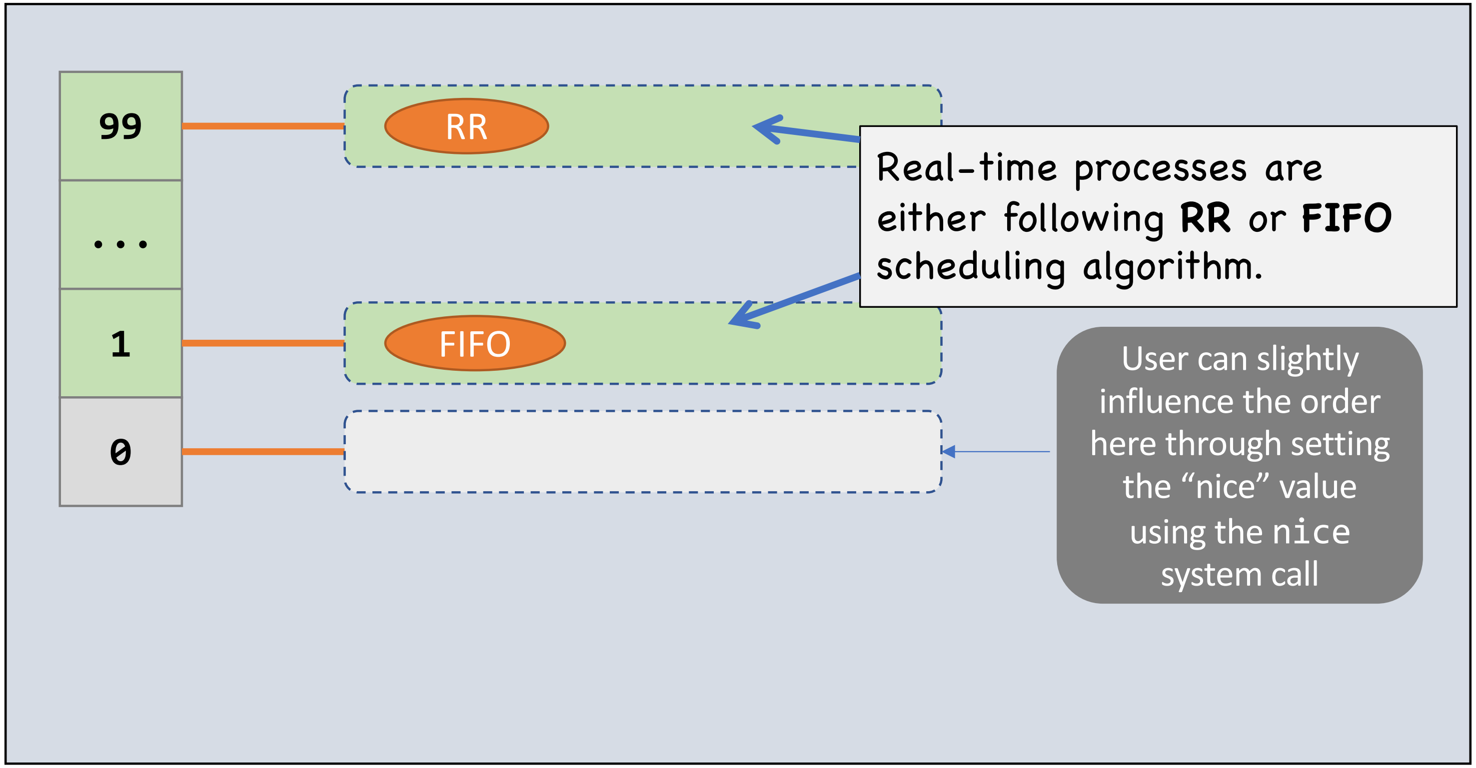
Multiple Queue Priority Scheduling

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



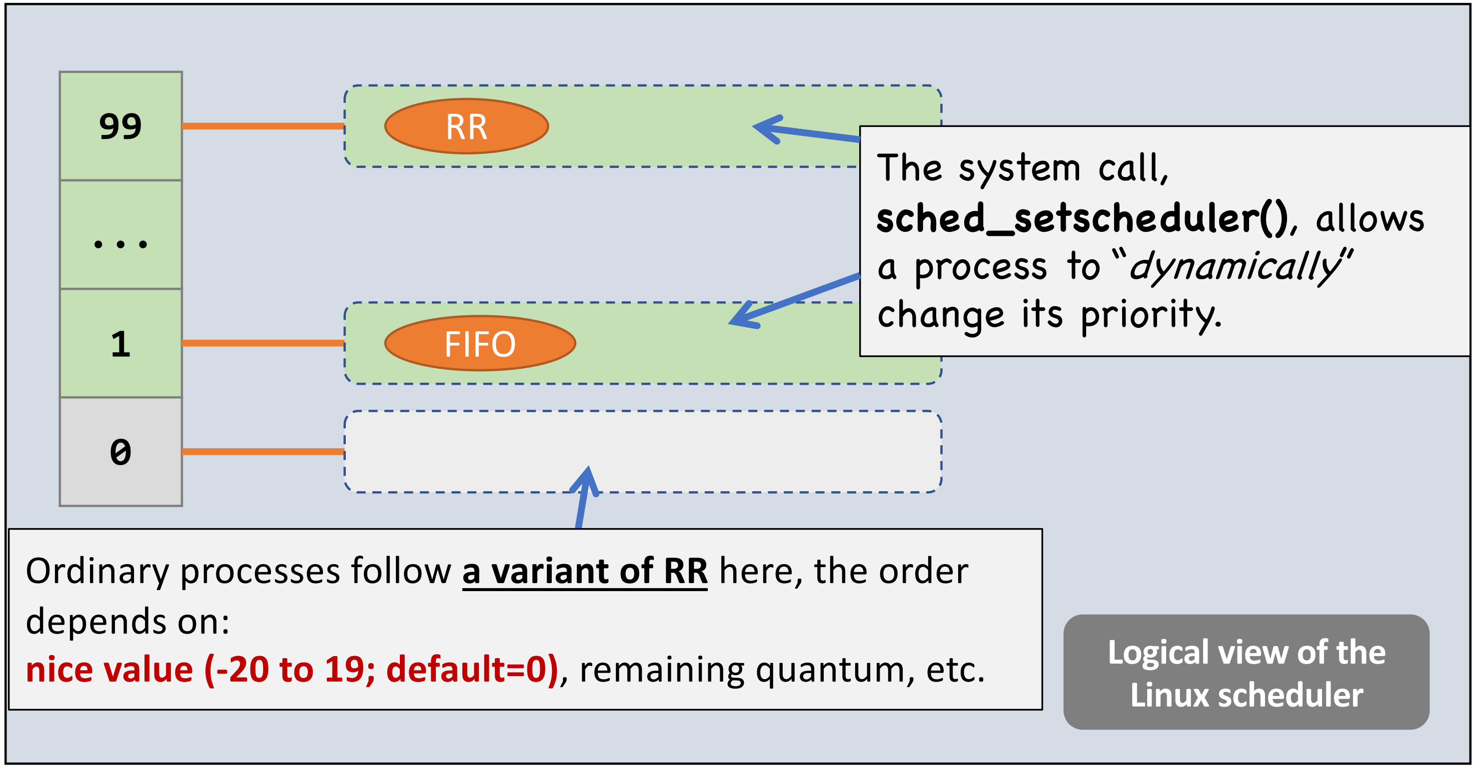
Multiple Queue Priority Scheduling

- **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.



Thank you!