Lecture 4
CPU Scheduling

Prof. Yinqian Zhang

Spring 2023

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

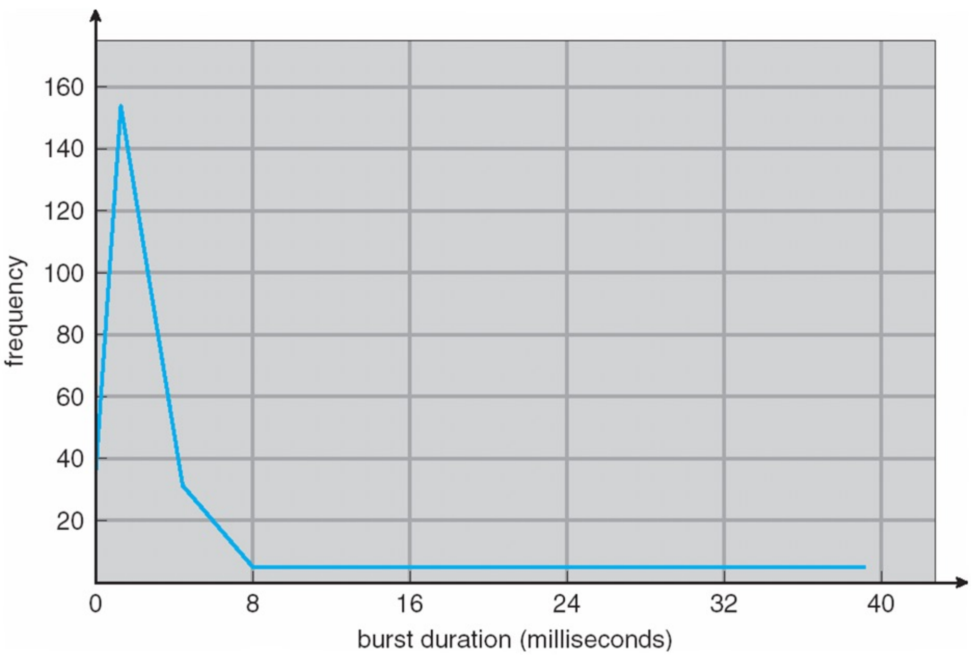
CPU-bound Process	I/O-bound process
Spends most of its running time on the CPU, i.e., **user-time > sys-time**	Spends most of its running time on I/O, i.e., **sys-time > user-time**
<u>Examples</u> - AI course assignments.	**<u>Examples</u>** - **/bin/ls**, networking programs.

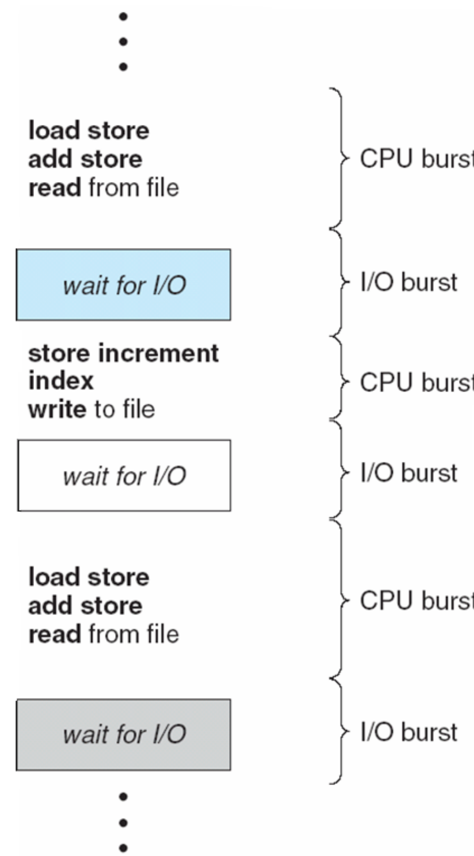
I/O burst is much more slower than CPU burst

majority of CPU Burst time is within 8 ms, so how to ues this property to design a better CPU Schedule?

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 e.x. I/O
  - 2. Switches from running to ready state e.x.1: interrupted e.x.2: all quota of process is occupied, has to be in ready state
  - 3. Switches from waiting to ready
  - 4. Terminates
- A scheduling algorithm takes place **only** under circumstances 1 and 4 is **non-preemptive** 指非抢占式的，即case1和case4是主动让出CPU的
- All other scheduling algorithms are **preemptive** 抢占式的，即被终止或其他情况致使CPU被抢用

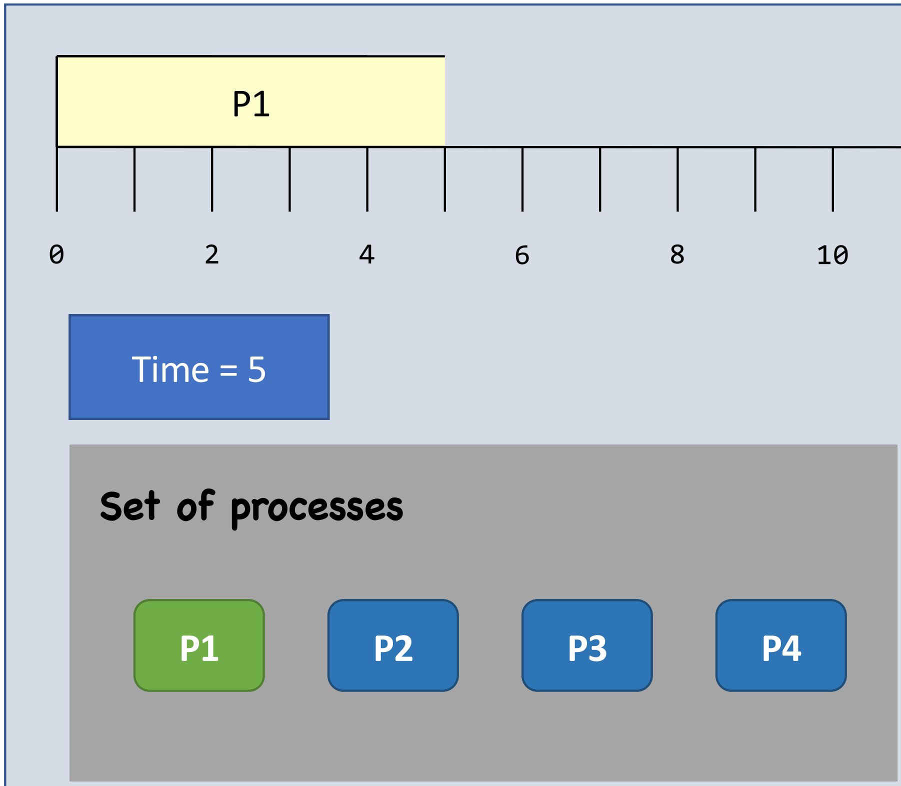
Scheduling Algorithm Optimization Criteria

- 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. time between getting in the ready queue and quitting the ready queue.
- Minimize average waiting time the existing time of a process in the ready queue
  - **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

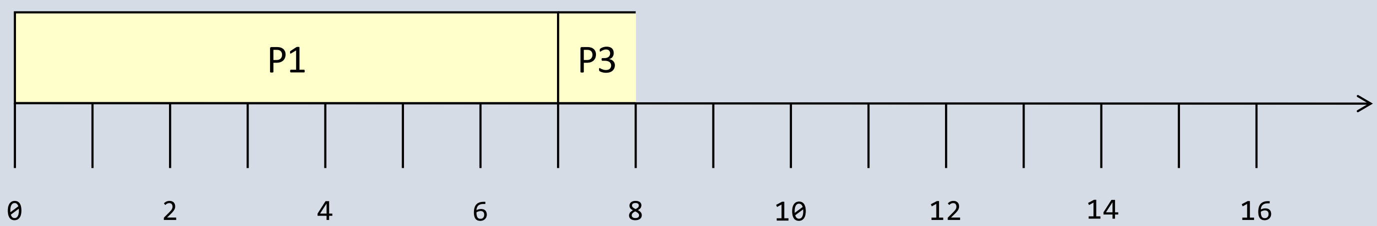
Non-preemptive SJF

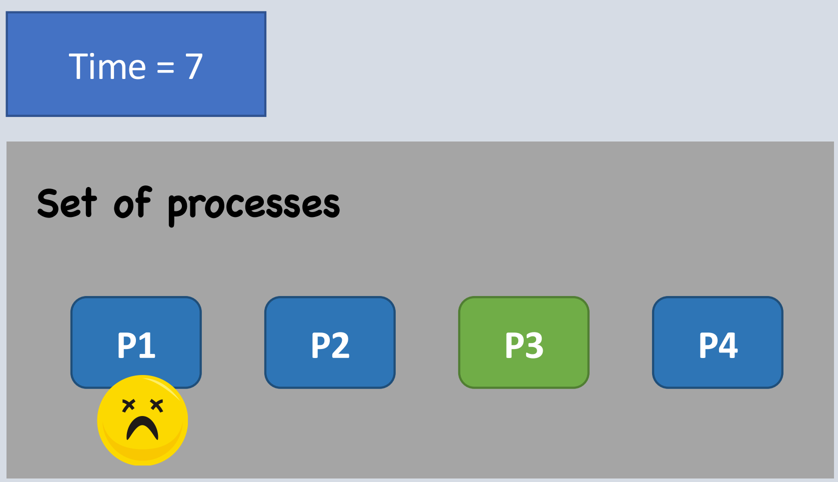


Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

shortest job first:
because of non-preemptive, so until P1 is terminated, then we find the process in ready queue with shortest CPU requirement, and then do it next.

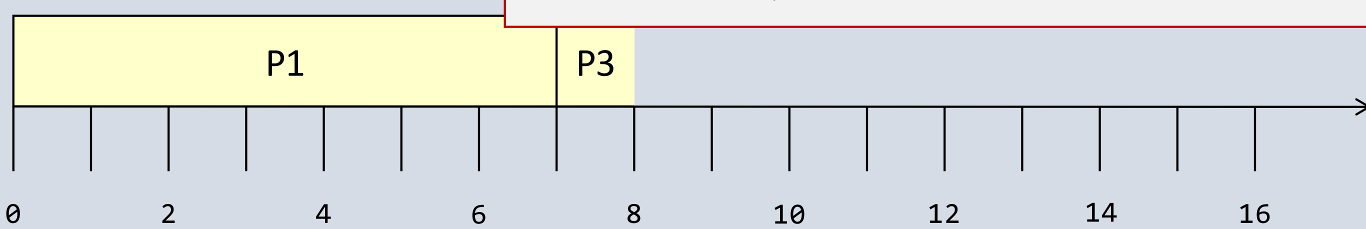




Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF

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

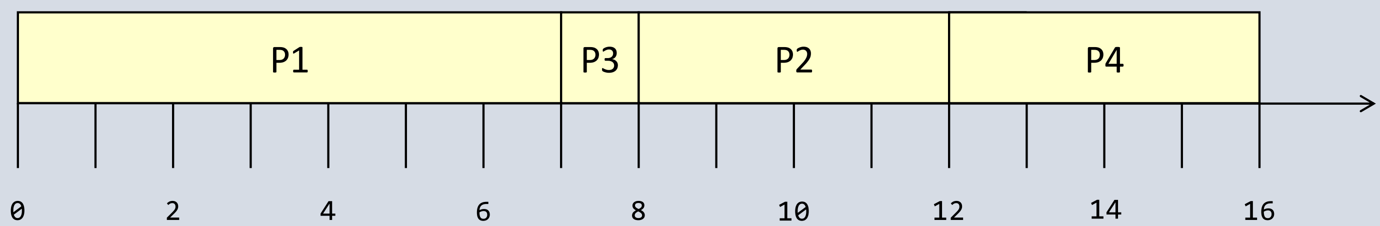


Time = 7

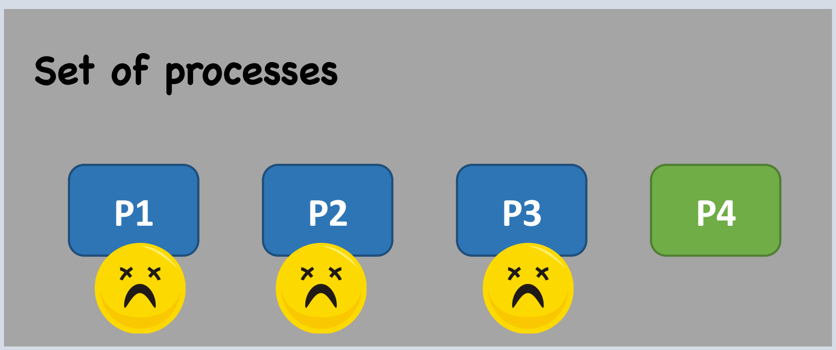
Set of processes

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

Non-preemptive SJF

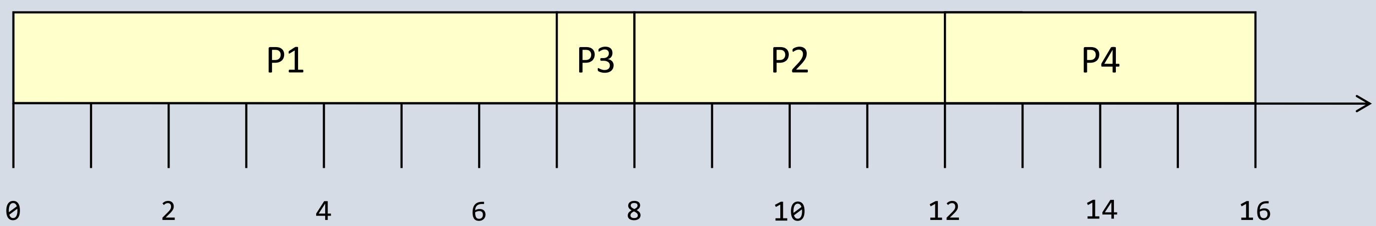


Time = 16



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:

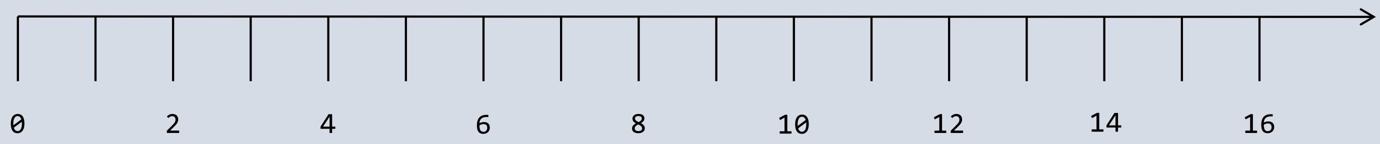
P1 = 7; P2 = 10; P3 = 4; P4 = 11;

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

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

Waiting time + CPU Burst time = Turnaround time

Preemptive SJF

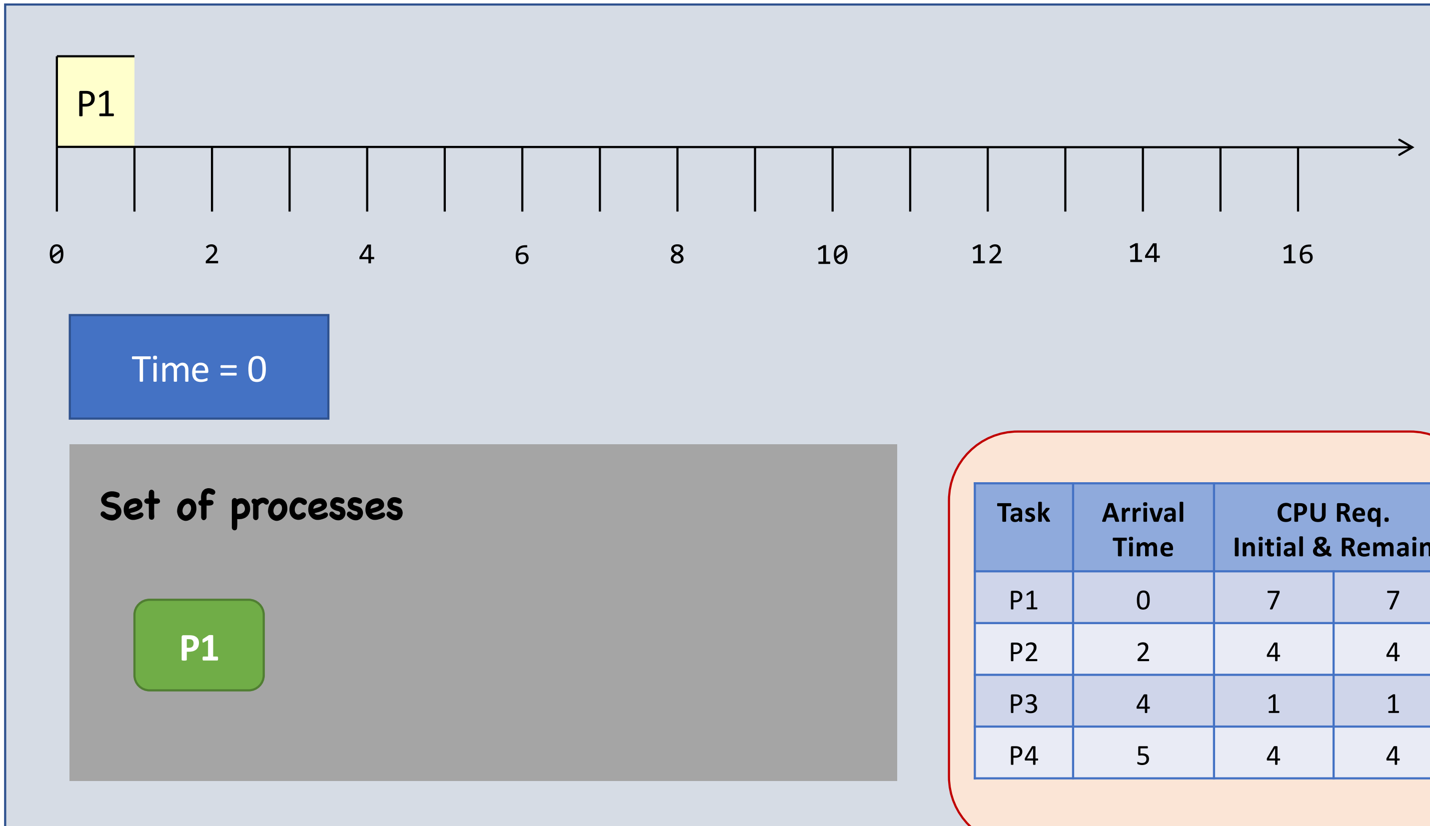


Whenever a new process arrives in the ready queue (either from waiting or from new state), the scheduler steps in and selects the next task based on **their remaining CPU requirements**.

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



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**

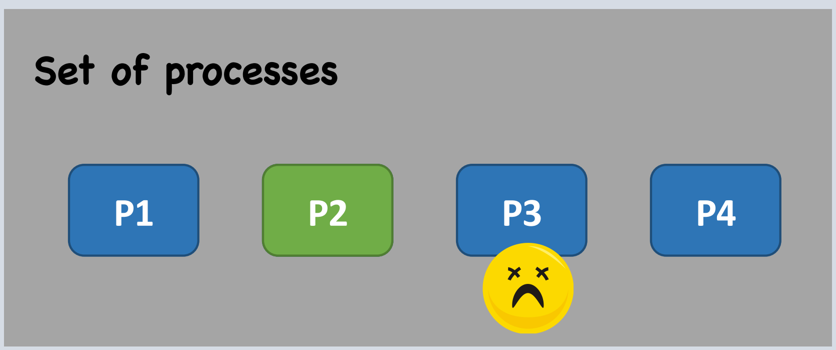
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	1
P4	5	4	4

Animation; don't print

Preemptive SJF

Time = 5

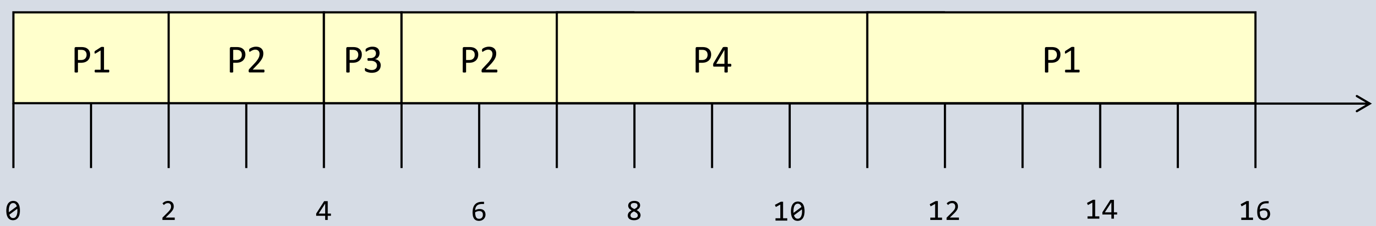
P2 is selected!



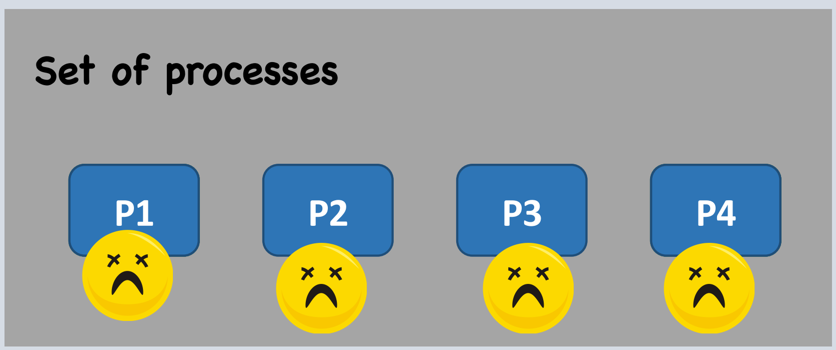
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	2
P3	4	1	0
P4	5	4	4

Animation; don't print

Preemptive SJF



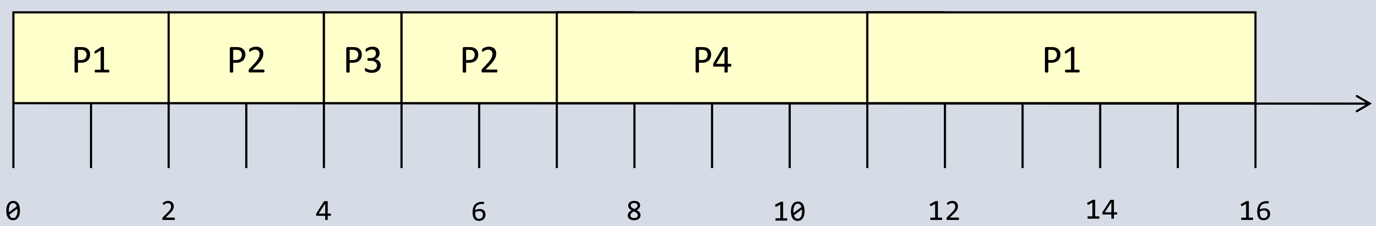
Time = 16



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

抢占式的，原理为当每次有新进程进入ready queue后，比较queue内各进程的remaining CPU requirement，执行剩余最少的进程。

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

理论模型中，不确定确切的CPU Requirement

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

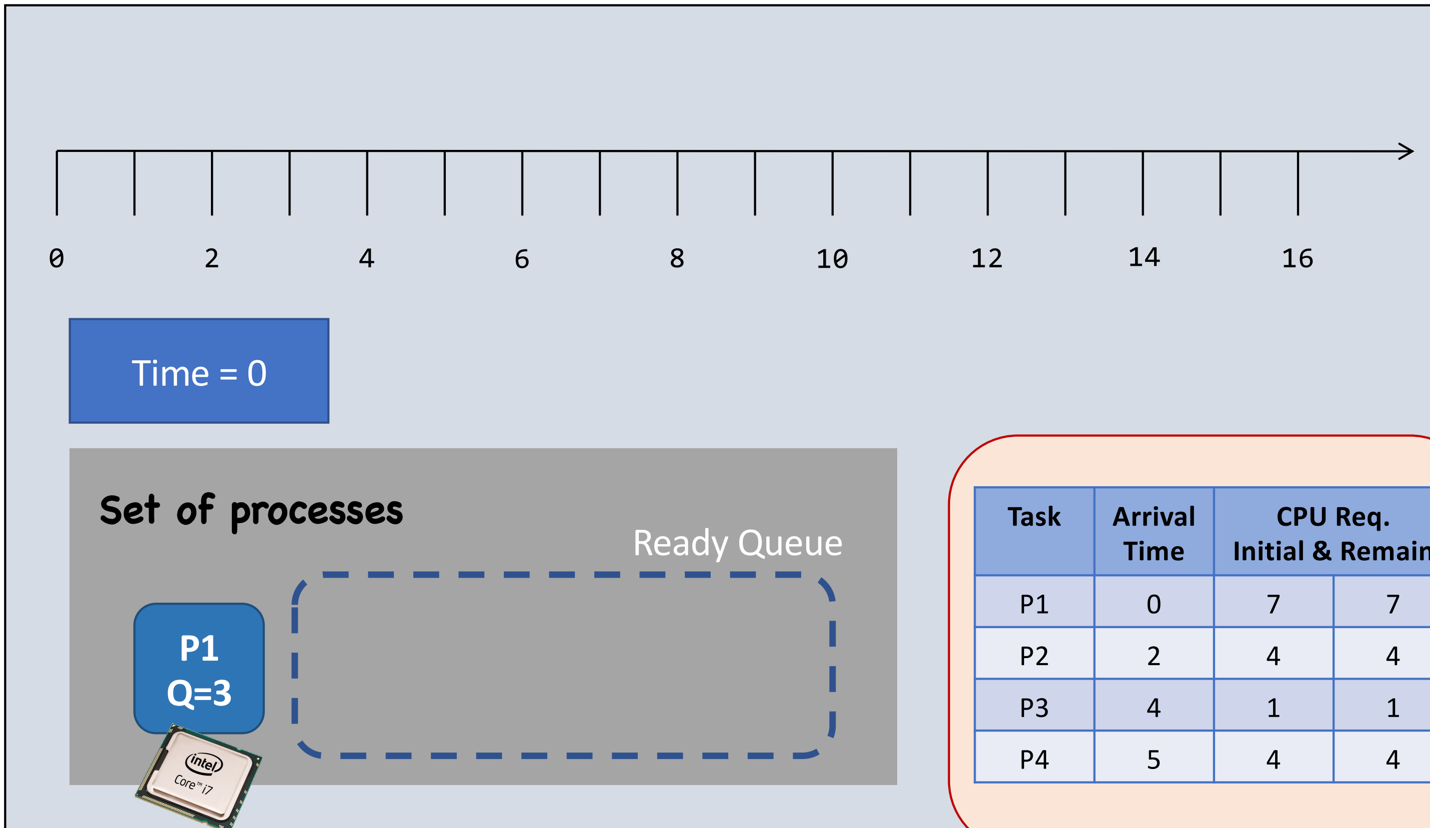
Round Robin (RR)

每个进程分配一个份额（quantum），当用完后被抢占并将该进程重新添加进queue，并recharge quantum。新来的进程则直接添加至queue结尾，CPU并不会在新进程进来时触发selection decision即新进入queue的进程不会影响正在进行的进程。

- 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, placed at the end of the queue, with its quantum re-charged
  - Then, the scheduler steps in and it chooses the next process which has a non-zero quantum to run.
  - 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

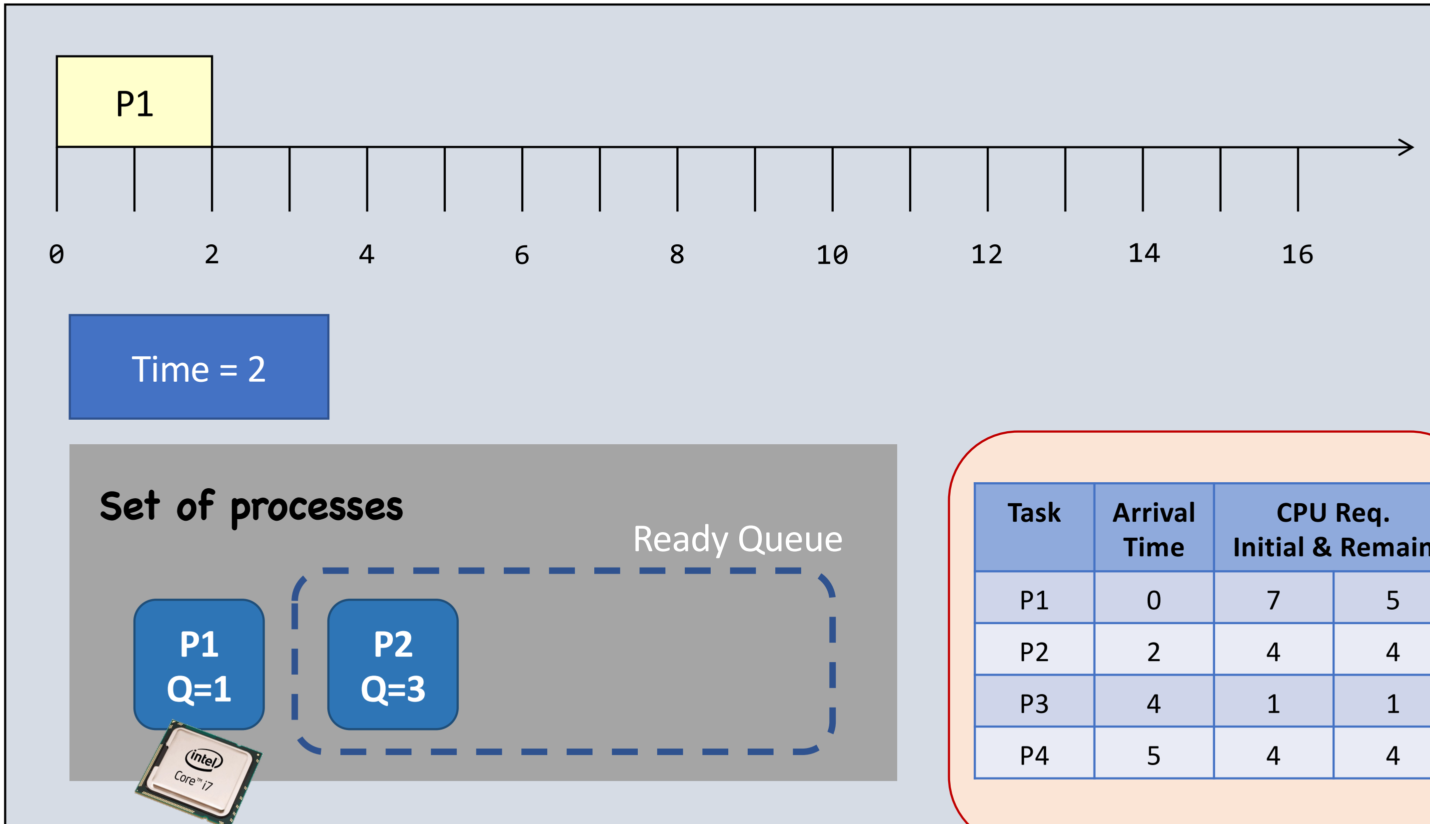
Round Robin (Quantum = 3)



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

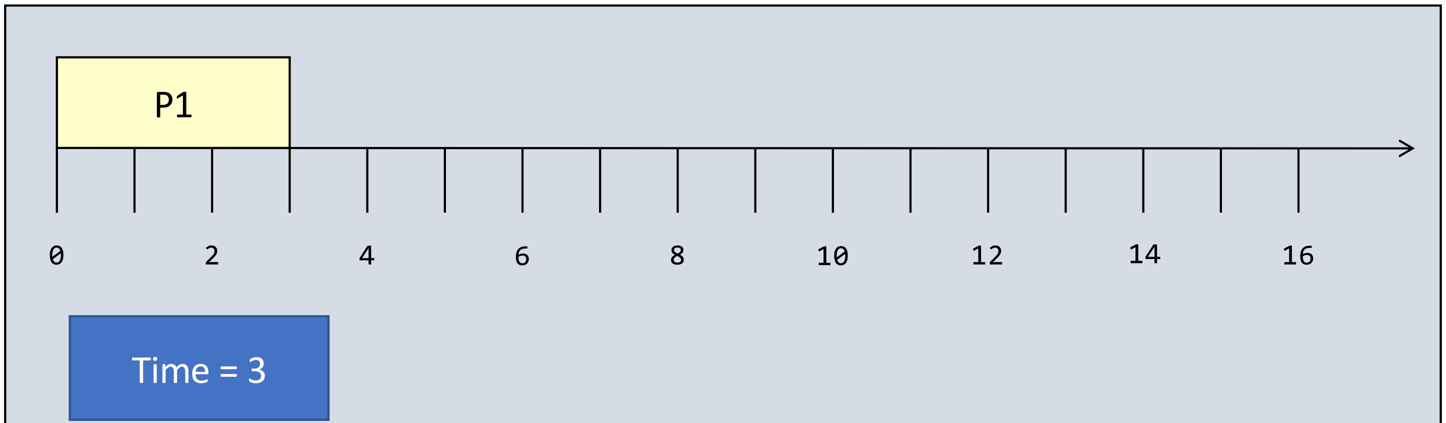
Round Robin (Quantum = 3)

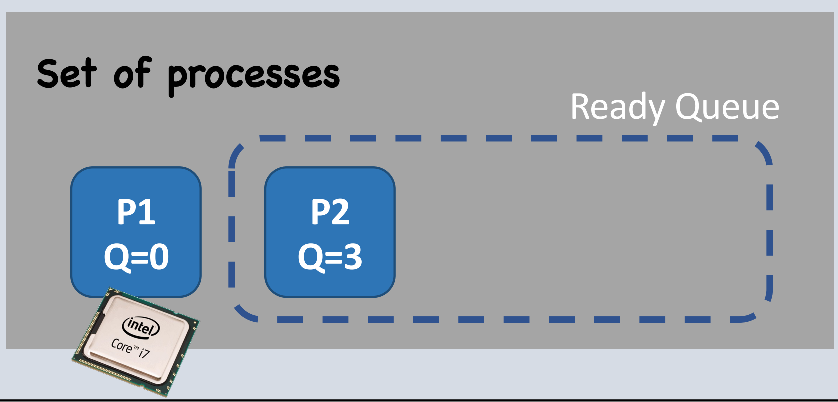


Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	5
P2	2	4	4
P3	4	1	1
P4	5	4	4

Animation; don't print

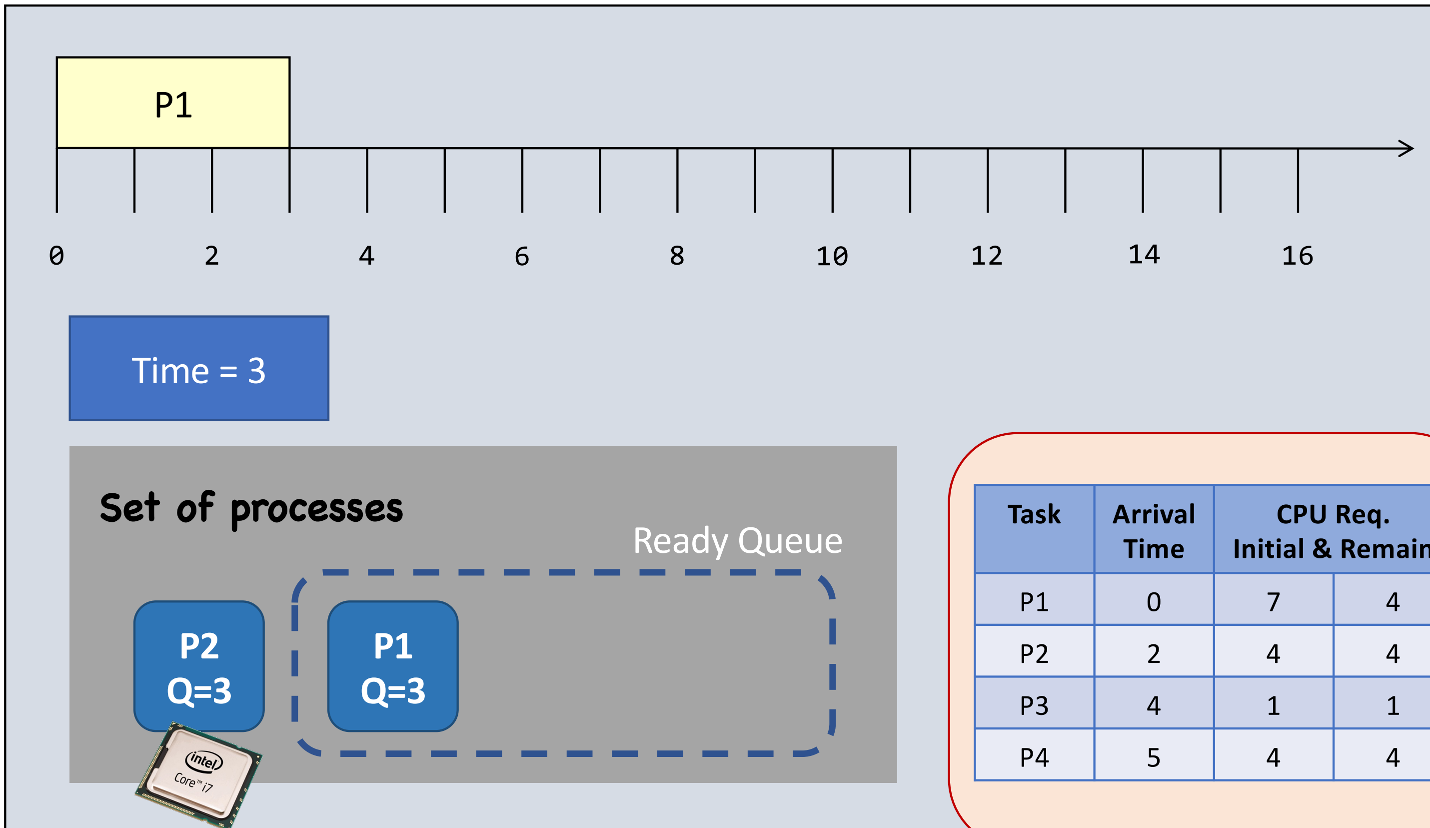
Round Robin (Quantum = 3)





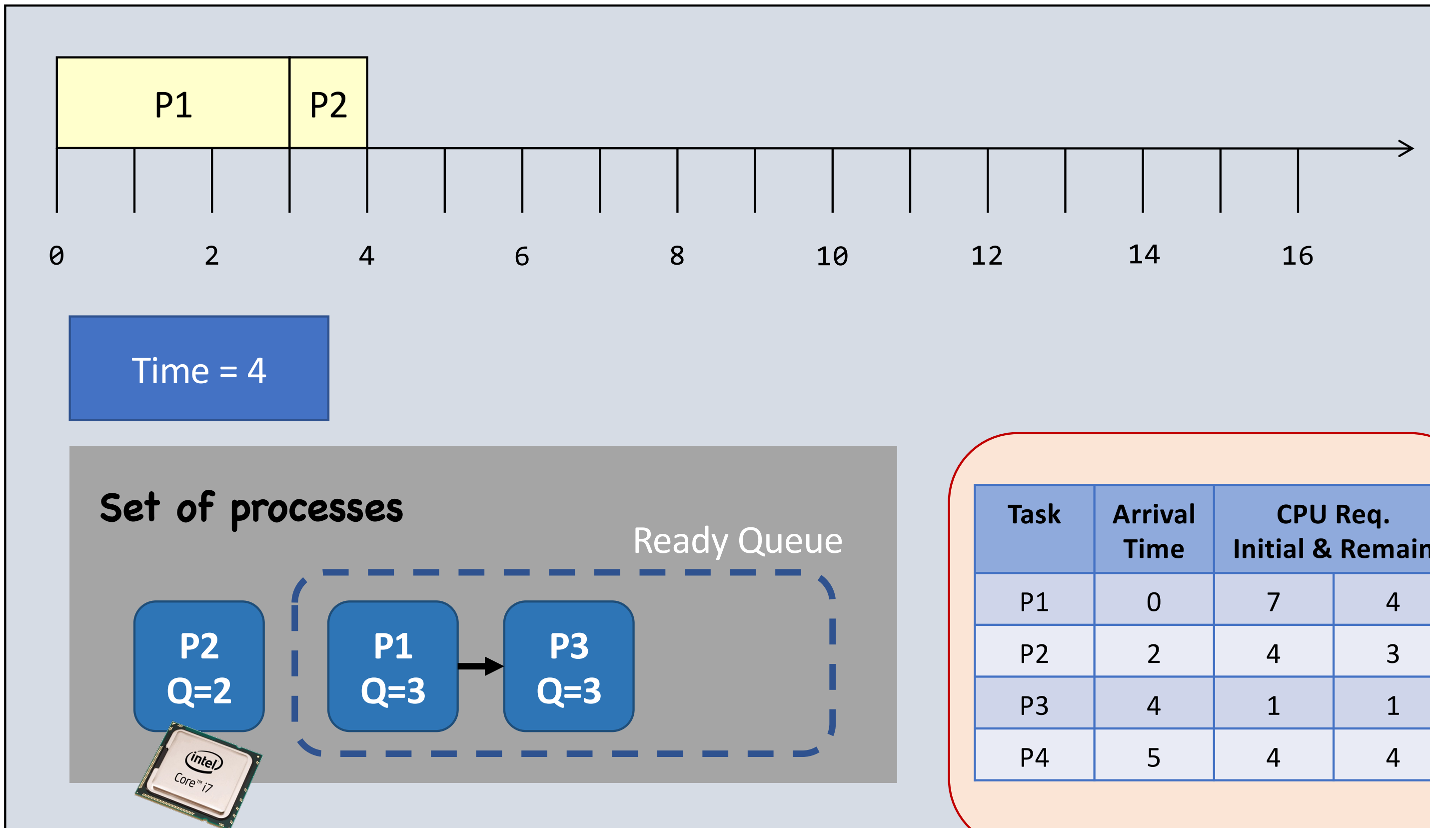
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	4
P2	2	4	4
P3	4	1	1
P4	5	4	4

Round Robin (Quantum = 3)



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

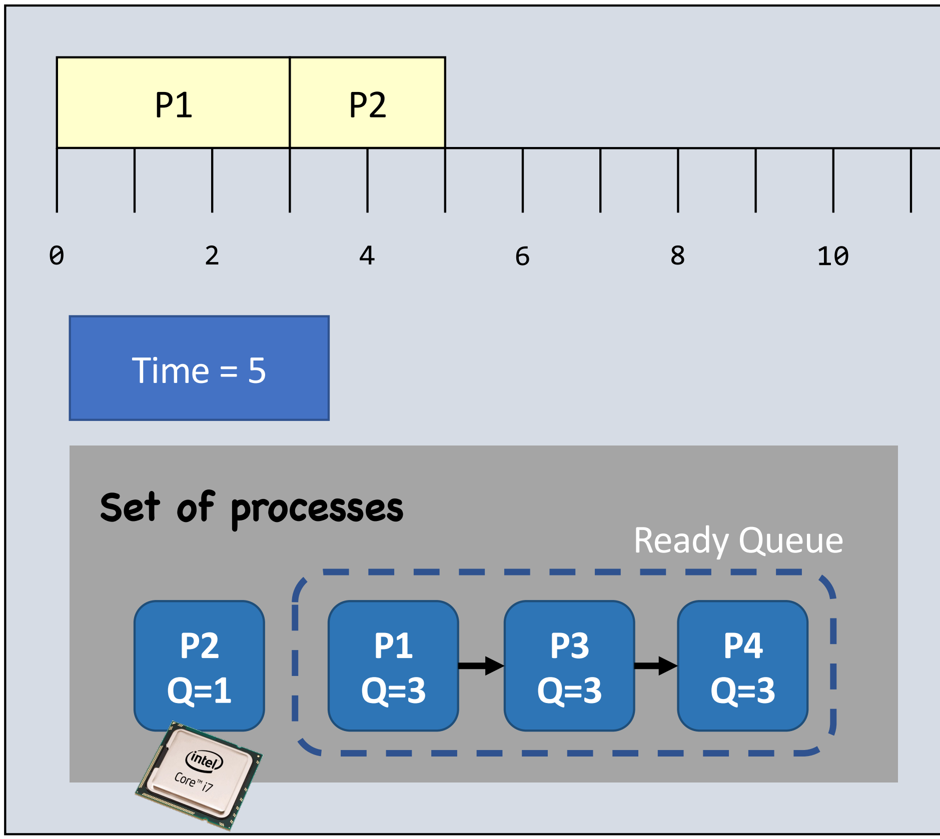
Round Robin (Quantum = 3)



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

Animation; don't print

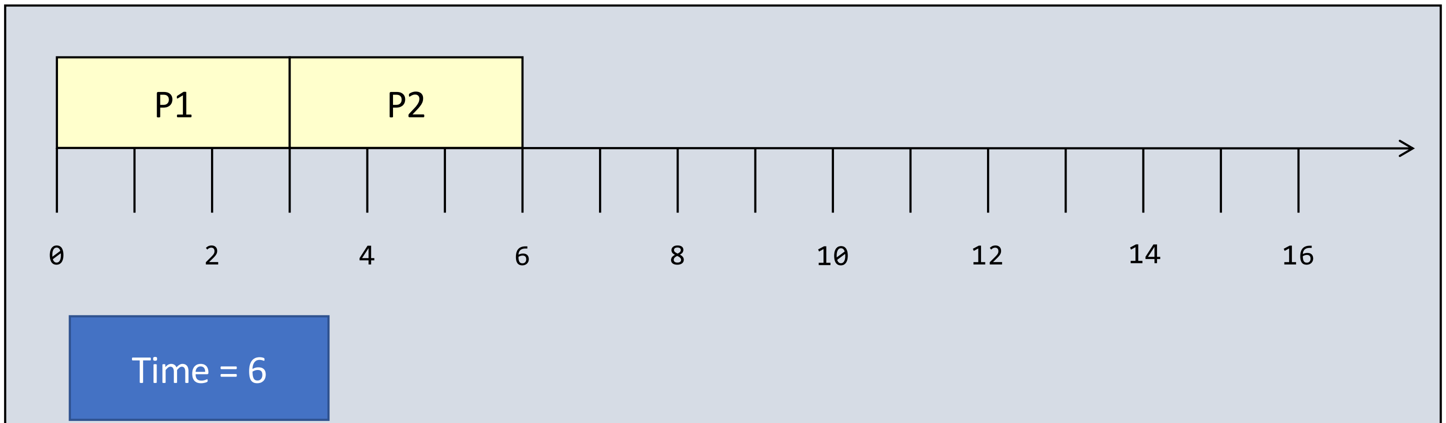
Round Robin (Quantum = 3)

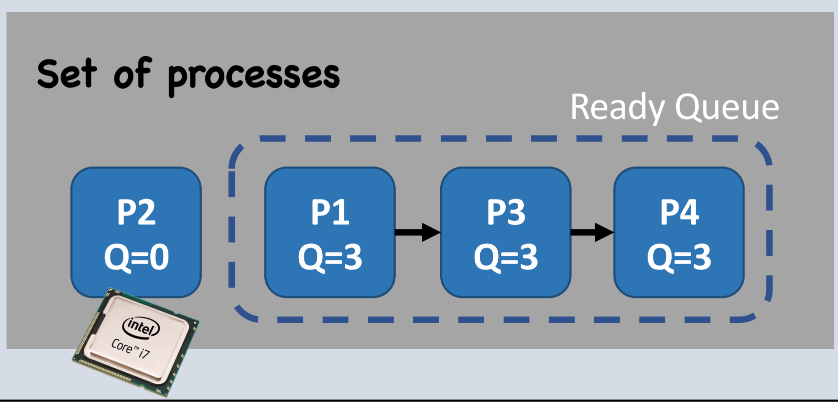


Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	4
P2	2	4	2
P3	4	1	1
P4	5	4	4

Animation; don't print

Round Robin (Quantum = 3)

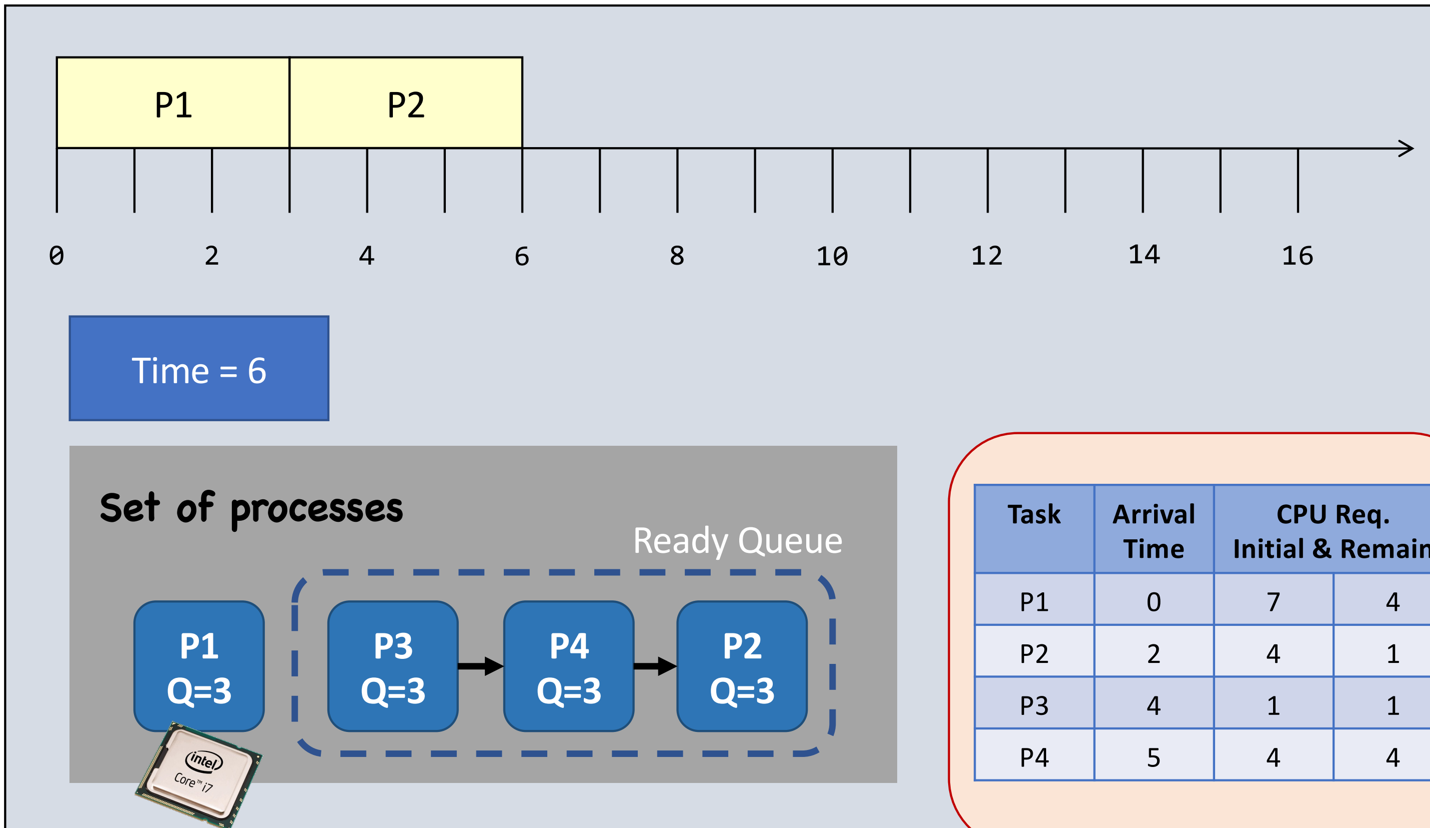




Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	4
P2	2	4	1
P3	4	1	1
P4	5	4	4

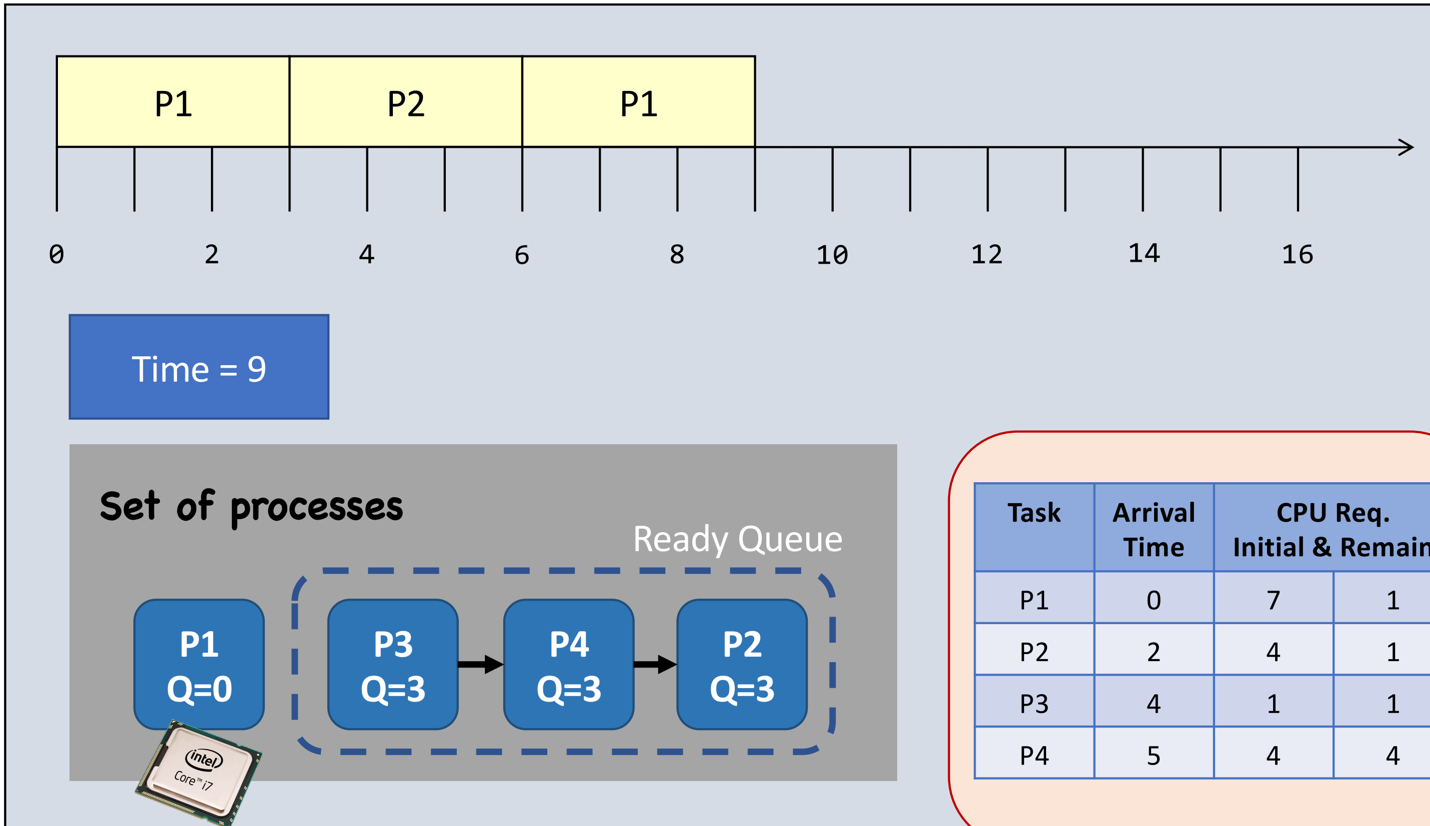
Animation; don't print

Round Robin (Quantum = 3)



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

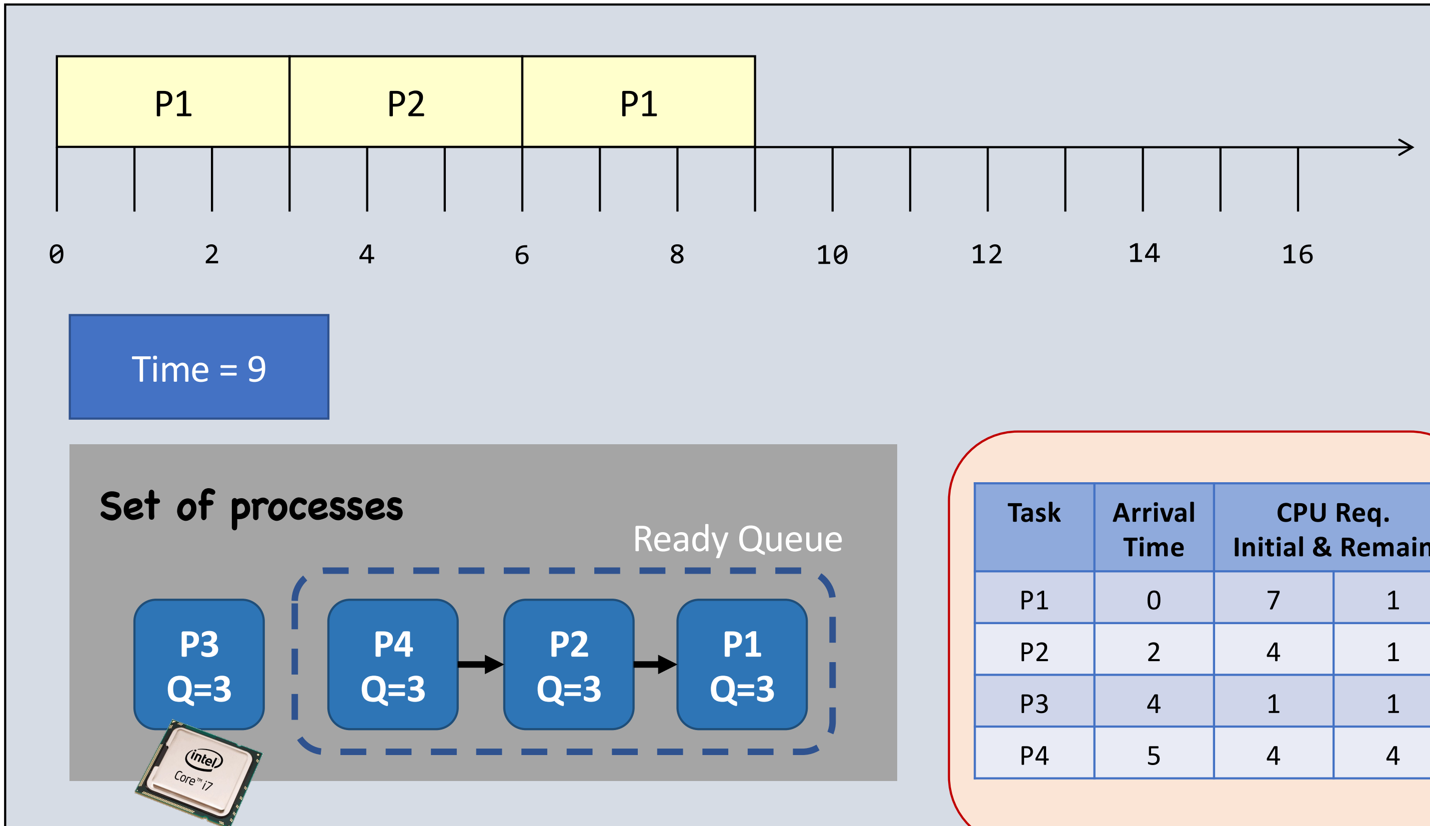
Round Robin (Quantum = 3)



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

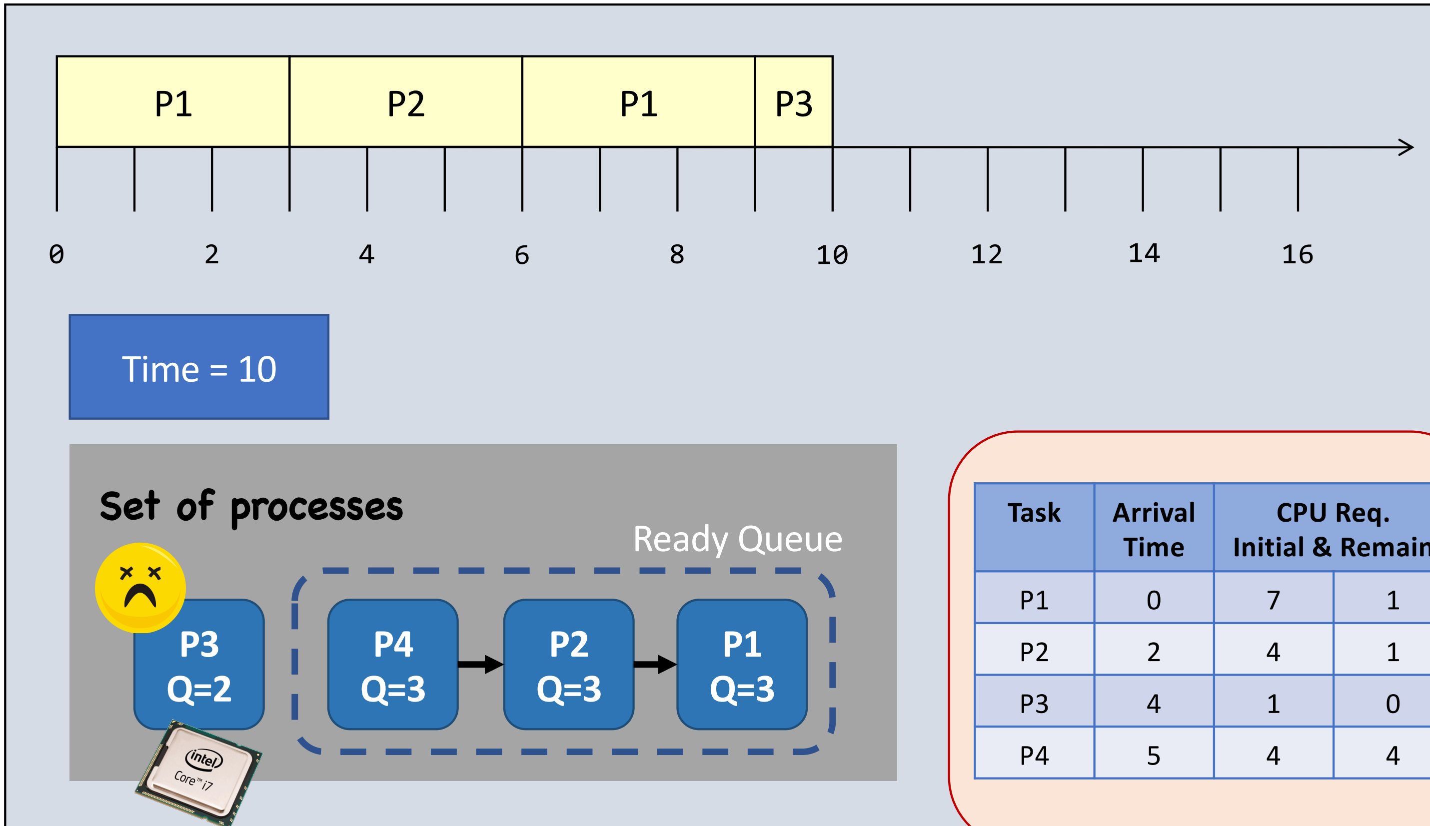
Animation; don't print

Round Robin (Quantum = 3)



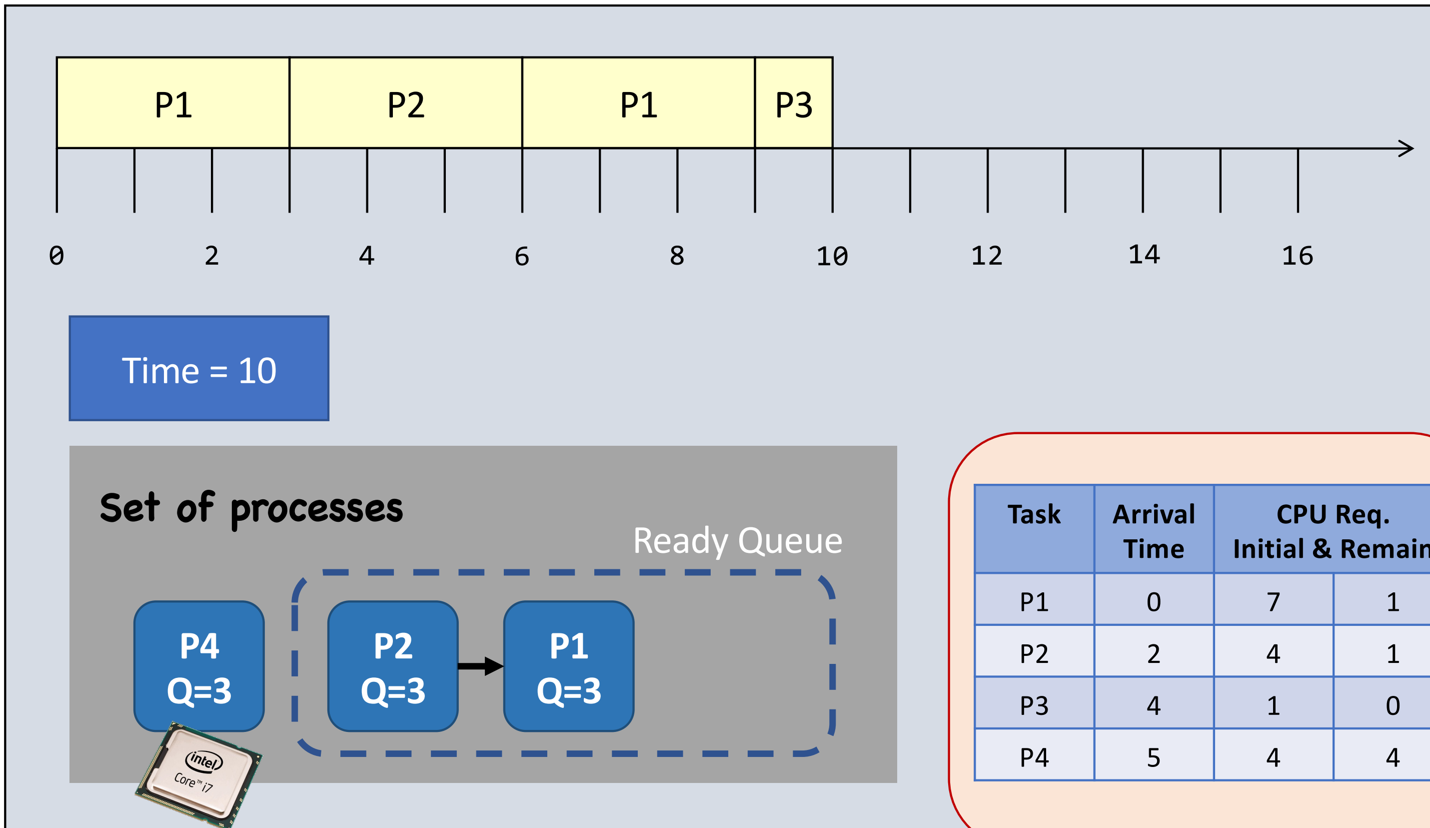
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	1
P2	2	4	1
P3	4	1	1
P4	5	4	4

Round Robin (Quantum = 3)



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

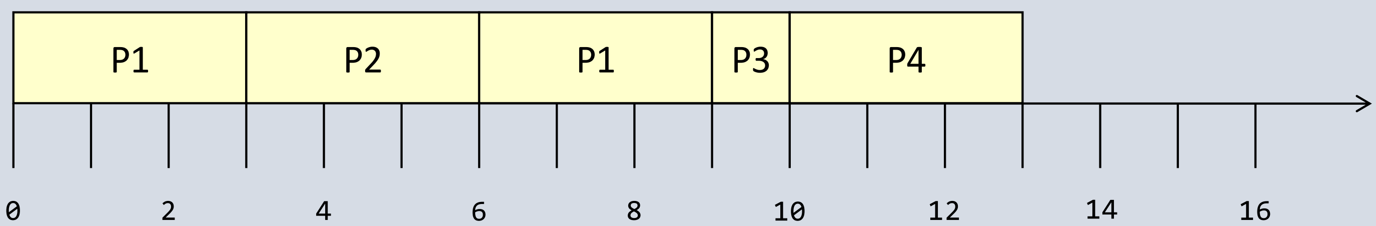
Round Robin (Quantum = 3)



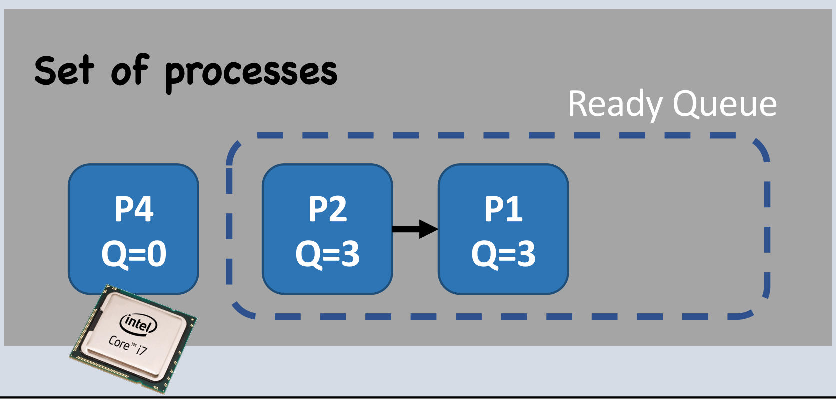
Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	1
P2	2	4	1
P3	4	1	0
P4	5	4	4

Animation; don't print

Round Robin (Quantum = 3)



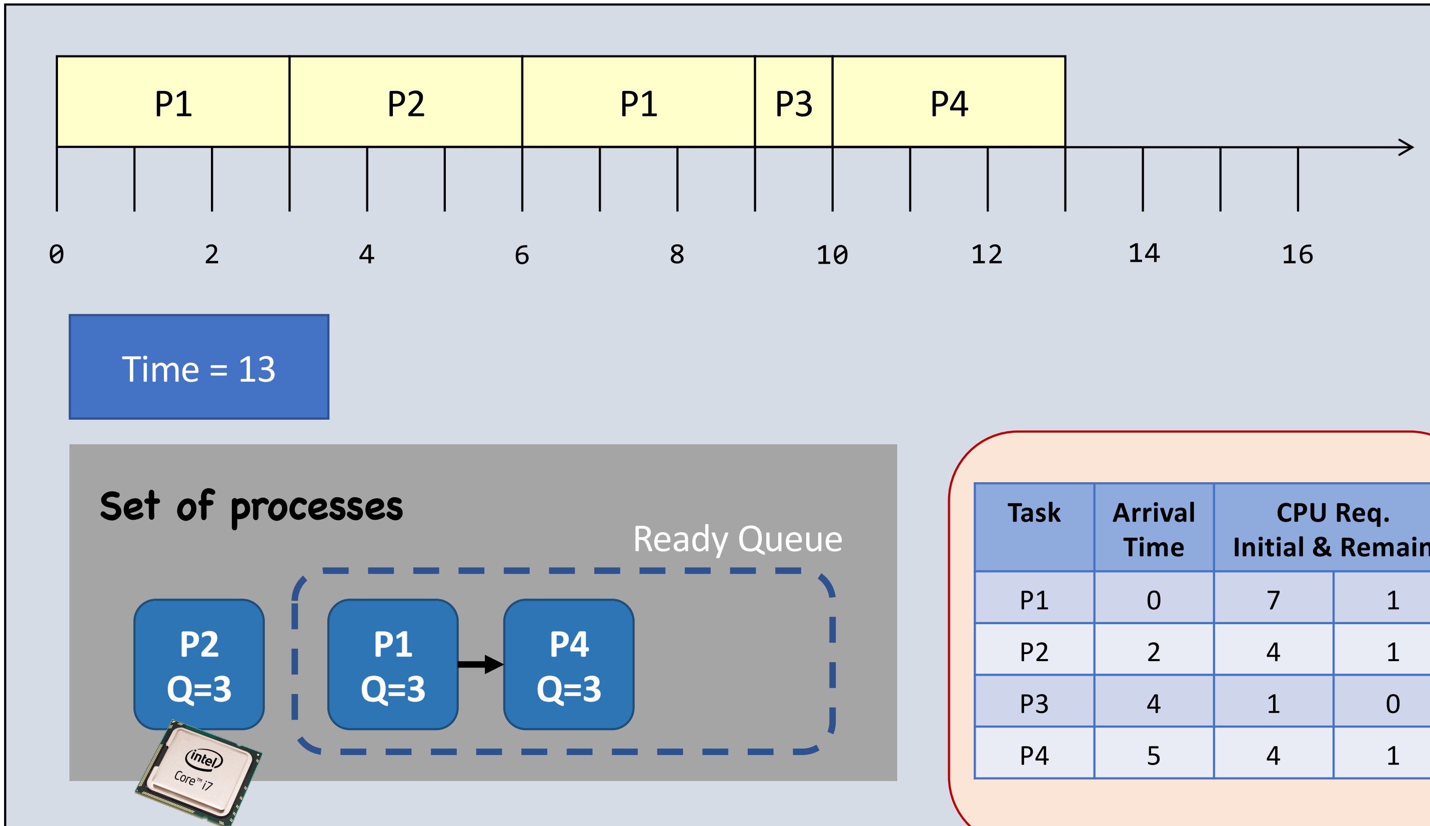
Time = 13



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

Animation; don't print

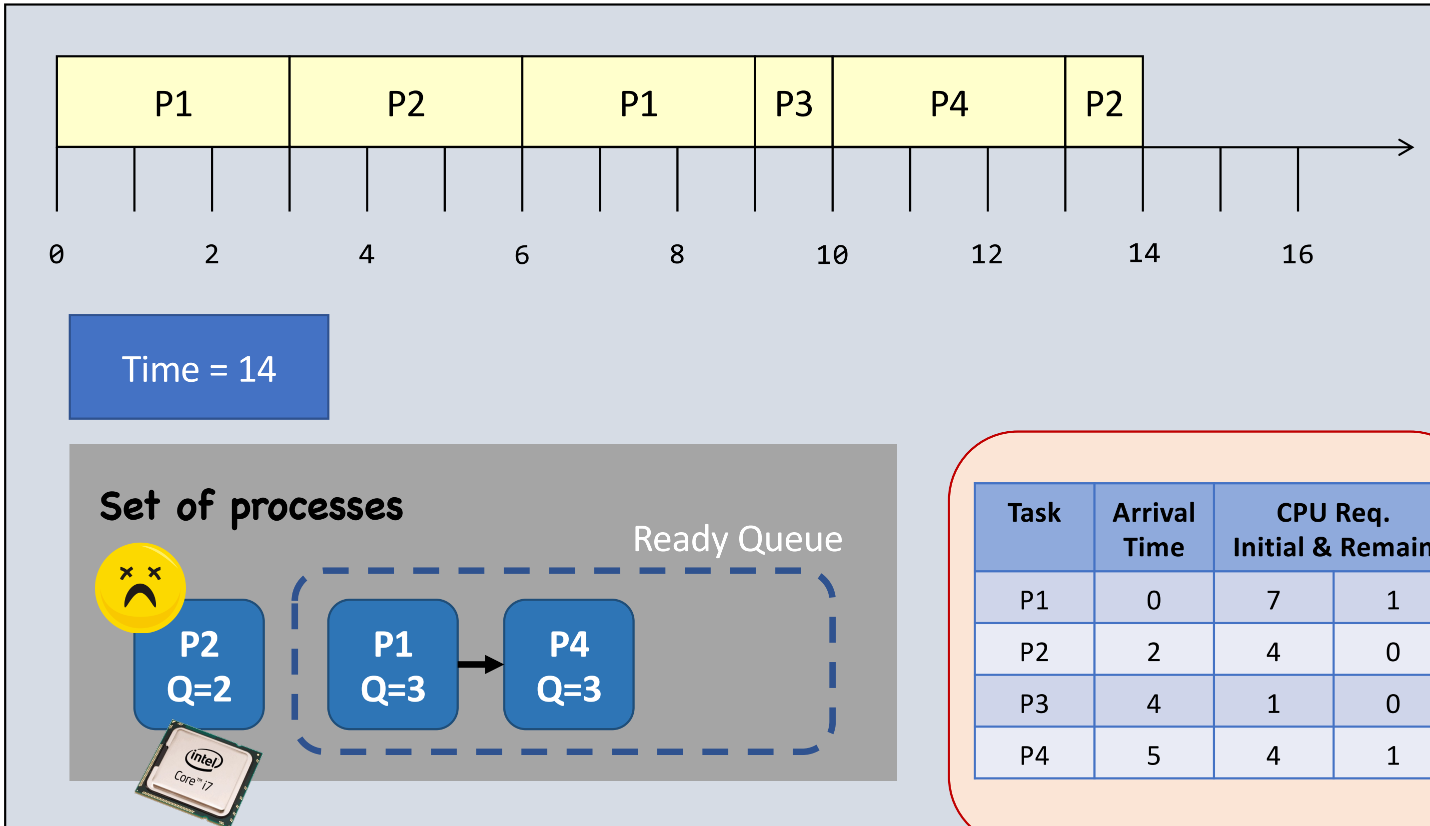
Round Robin (Quantum = 3)



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

Animation; don't print

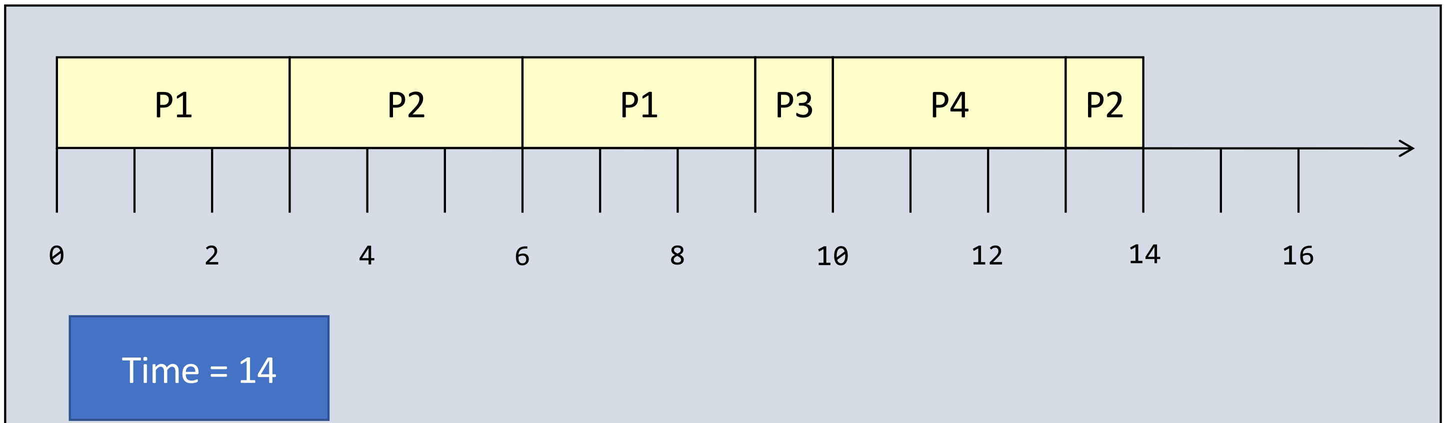
Round Robin (Quantum = 3)

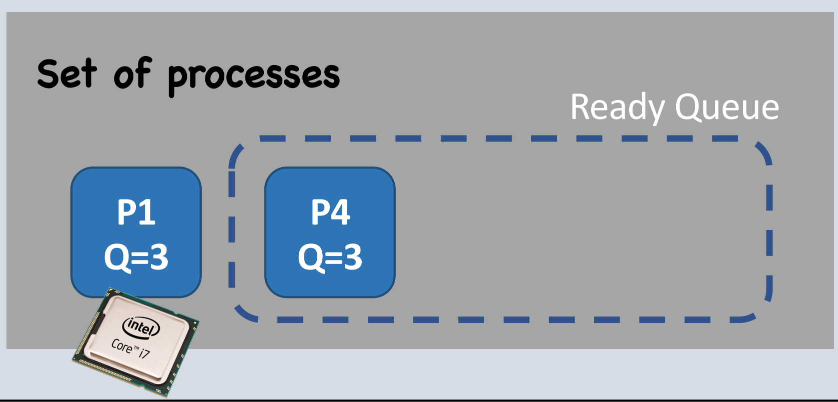


Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	1
P2	2	4	0
P3	4	1	0
P4	5	4	1

Animation; don't print

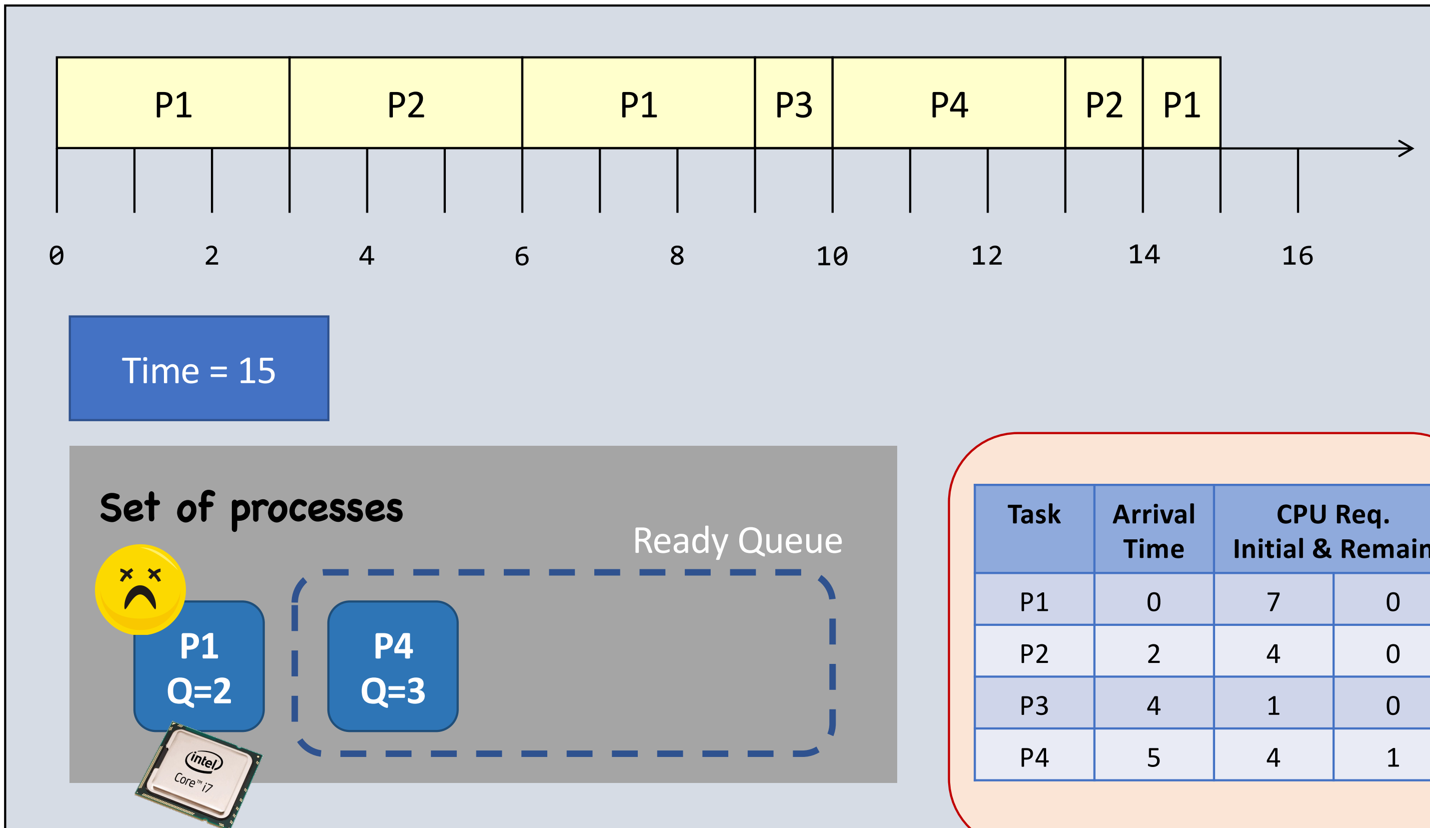
Round Robin (Quantum = 3)





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

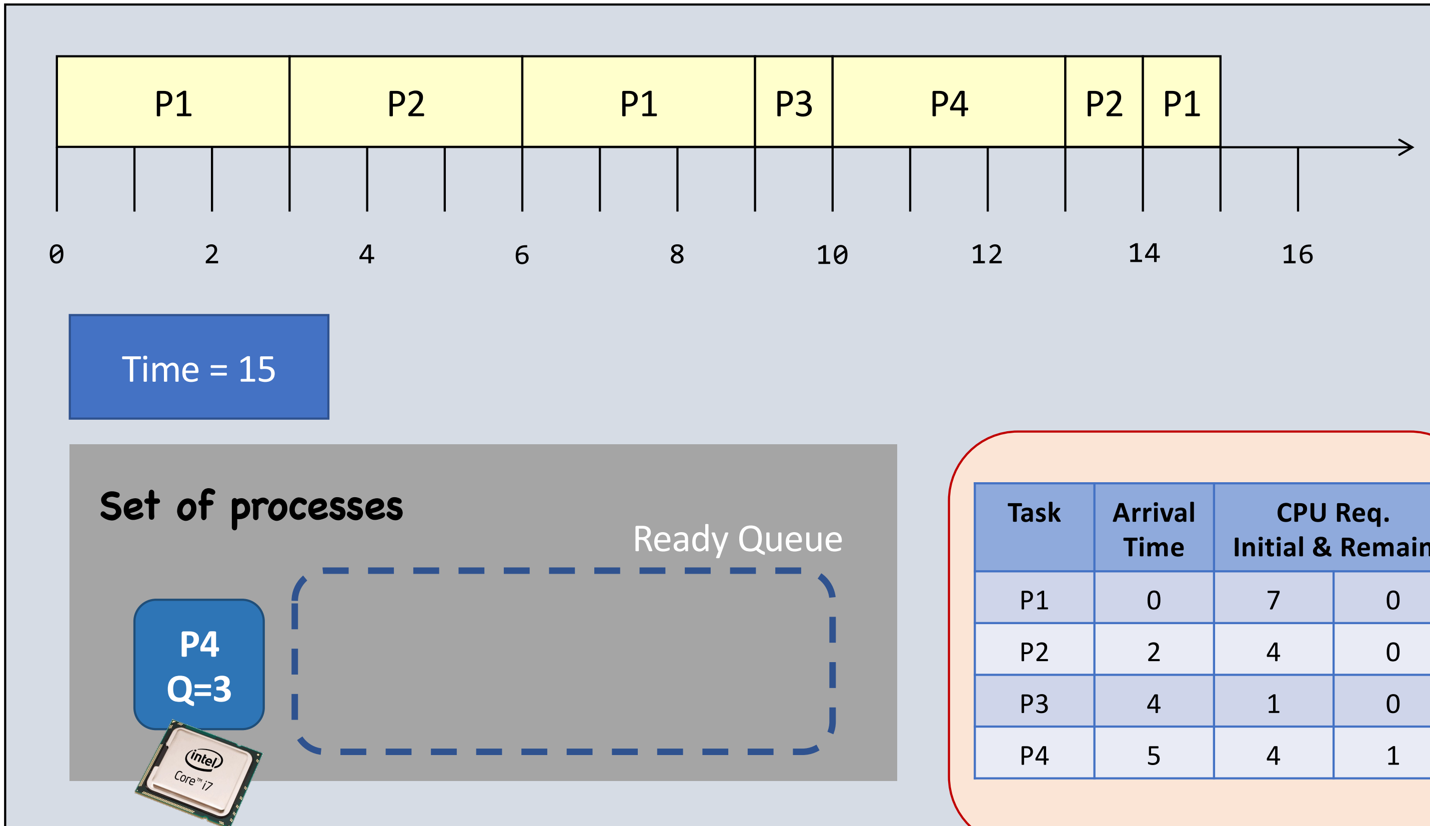
Round Robin (Quantum = 3)



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

Animation; don't print

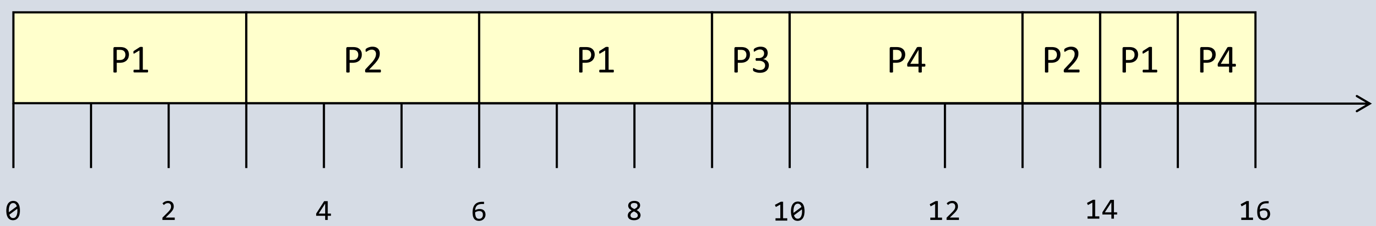
Round Robin (Quantum = 3)

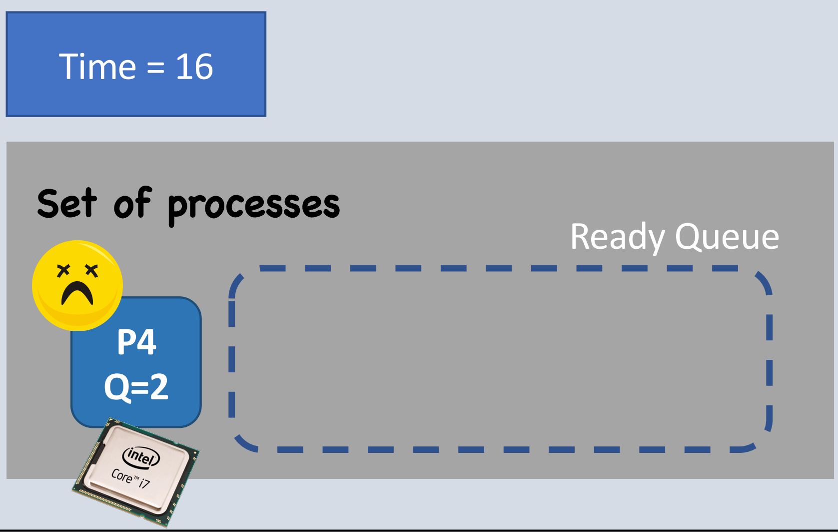


Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	1

Animation; don't print

Round Robin (Quantum = 3)

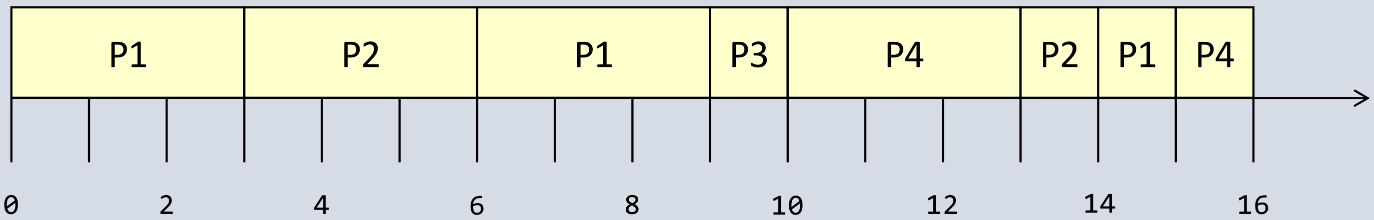




Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

Animation; don't print

Round Robin (Quantum = 3)



Waiting time:

P1 = 8; P2 = 8; P3 = 5; P4 = 7;

Average = (8 + 8 + 5 + 7) / 4 = 7

Turnaround time:

P1 = 15; P2 = 12; P3 = 6; P4 = 11;

Average = (15 + 12 + 6 + 11) / 4 = 11

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

事实上在SJF中，CPU并不清楚准确的每个进程的CPU burst time，即理论建立于理想情况的分析下

RR的应用一般也能起到很好的资源调度效果

RR v.s. SJF

	Non-preemptive SJF	Preemptive SJF	RR
Average waiting time	4	3	**7 (largest)**
Average turnaround time	8	7	**11 (largest)**
# of context switching	3	5	**7 (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 ≡ 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

如 preemptive SJF 中的 remaining 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 dynamic priority

Linux Scheduling

- Before Linux kernel version 2.5, traditional UNIX scheduling, not adequately support SMP
- Linux kernel version 2.5, O(1) scheduler
  - Constant scheduling time regardless number of tasks
  - Better support for SMP
  - Poor response time for interactive processes
- After Linux kernel version 2.6.23, CFS-completely fair scheduler
  - Default scheduler now

Completely Fair Scheduler

- Scheduling class
  - Standard Linux kernel implements two scheduling classes
  - (1) Default scheduling class: CFS
  - (2) Real-time scheduling class
- Varying length scheduling quantum
  - Traditional UNIX scheduling uses 90ms fixed scheduling quantum
  - CFS assigns a proposition of CPU processing time to each task
- Nice value
  - -20 to +19, default nice is 0
  - Lower nice value indicates a higher relative priority
  - Higher value is "being nice"
  - Task with lower nice value receives higher proportion of CPU time

即nice意味着它容忍没他nice的进程抢占它

Completely Fair Scheduler (Cont'd)

- Virtual run time
  - Each task has a per-task variable **vruntime**
  - Decay factor
    - Lower priority has higher rate of decay
    - nice = 0 virtual run time is identical to actual physical run time
    - A task with nice > 0 runs for 200 milliseconds, its **vruntime** will be higher than 200 milliseconds
    - A task with nice < 0 runs for 200 milliseconds, its **vruntime** will be lower than 200 milliseconds
- Lower virtual run time, higher priority
  - To decide which task to run next, scheduler chooses the task that has the smallest **vruntime** value
  - Higher priority can preempt lower priority

Completely Fair Scheduler (Cont'd)

- Example: Two tasks have the same nice value
- One task is I/O bound and the other is CPU bound
- **vruntime** of I/O bound will be shorter than **vruntime** of CPU bound
- I/O bound task will eventually have higher priority and preempt CPU-bound tasks whenever it is ready to run

Thank you!