Operating System Concepts
Ch. 5: Scheduling

Silberschatz, Galvin & Gagne

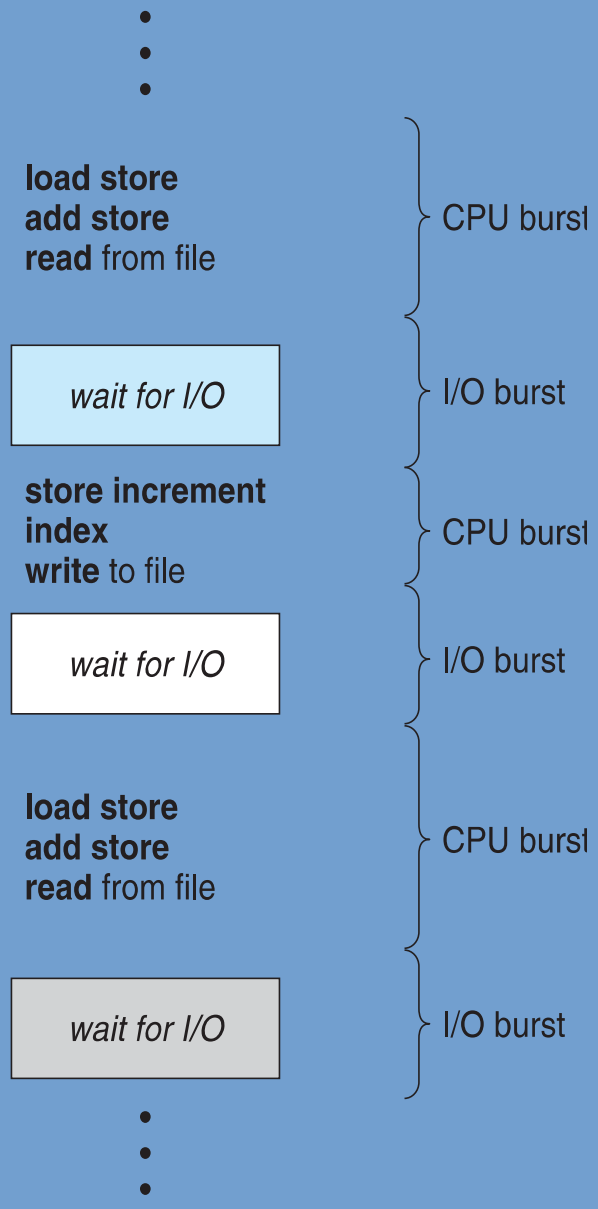
Universiteit Leiden
The Netherlands

Scheduling

- In a multi-programmed system, multiple processes may be loaded into memory at the same time.
- We need a procedure, or rather algorithm, to decide what process is assigned to the CPU.
  - This algorithm is the scheduling algorithm.
  - Various algorithms exist which tune for different criteria.
- Remember an OS serves as resource allocator; the CPU scheduler is the allocator for the CPU cycles resource.

Bursts

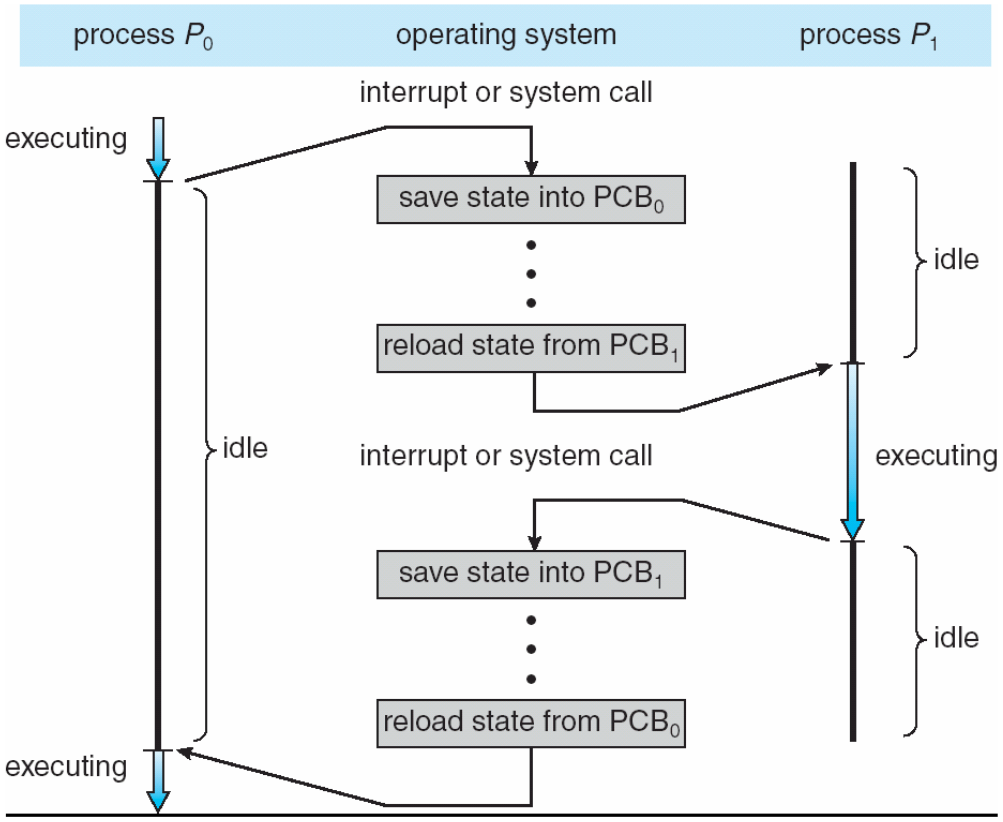
- The execution of a process can be seen as a sequence of alternating CPU and I/O *bursts*.
- In an I/O burst the process blocks waiting on I/O, it cannot make further progress until the I/O operation is completed.
- With multi-programming the CPU does not have to sit idle during I/O, we can assign another process to the CPU.
  - CPU utilization is maximized.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Process context switch

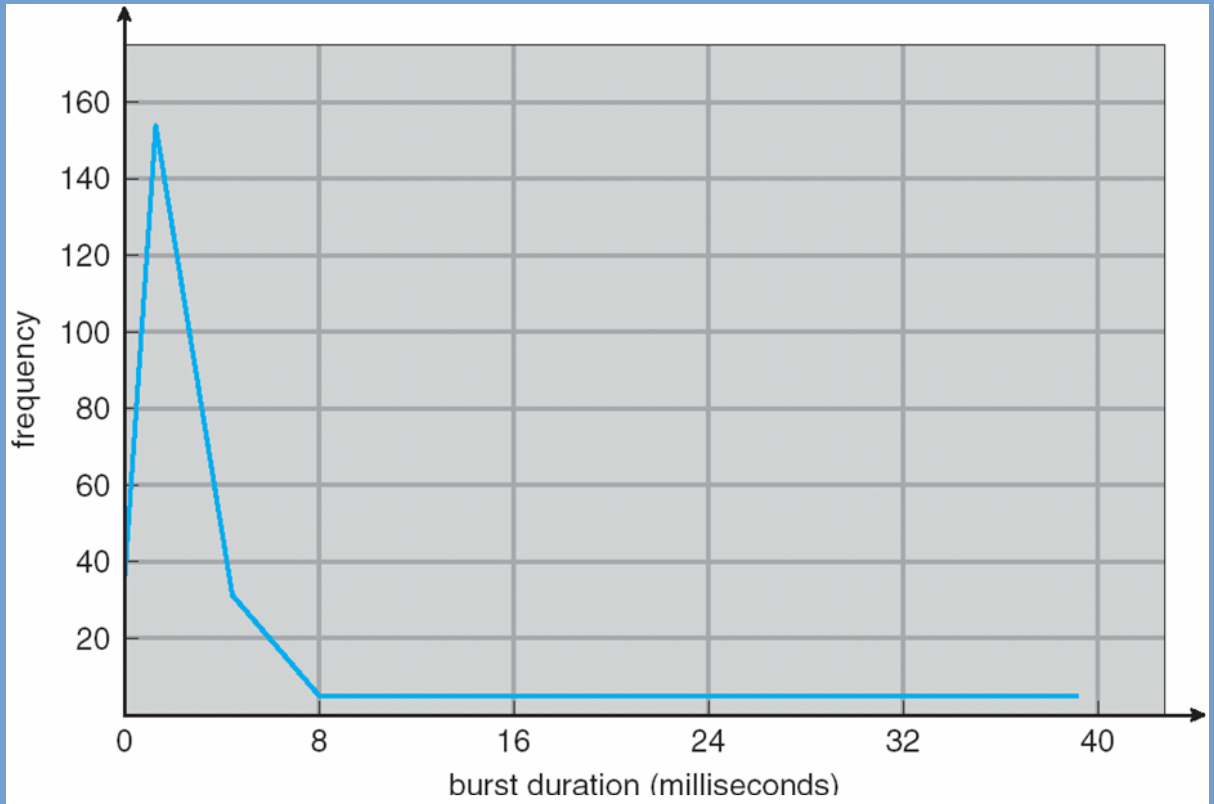
- Recall: the switch from one process to another is referred to as a context switch.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

CPU burst time histogram

- When a CPU to I/O burst transition (system call involving blocking I/O) is encountered, we need to invoke the scheduler.
- The typical length of a CPU burst tells us how often scheduler invocations need to take place.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

The CPU scheduler (short-term scheduler)

- Short bursts (< 8 ms) most frequent; we need a scheduler that is invoked (very) often and makes quick decisions.
  - This is the task of the short-term scheduler.
  - Typically a ready queue is maintained of processes that are ready to run (and don't block on e.g. I/O).
  - Any process from this queue may be assigned to the CPU. Which is chosen exactly depends on the scheduling algorithm that is in use.

The CPU scheduler (2)

- In the simple case, a scheduling decision is only made when:
  - A process terminates itself (exit() system call).
  - A process calls a blocking system call (transition running -> waiting), relinquishing the CPU.
- If the scheduler is only invoked in these cases, the scheduler is called *non-preemptive*.
- Non-preemptive systems are also known as *cooperative systems*.
  - The context switches that are triggered are *voluntary*.
  - Note that a malicious process may choose to never terminate itself or call a system call (infinite loop).
  - Only works if everybody plays by the rules.
  - Used in past systems, e.g. classic Mac OS (before Mac OS X).

The CPU scheduler (3)

- Now, it is clear there also is a category of schedulers known as *preemptive schedulers*.
- These **additionally** invoke the scheduler in these cases:
  - A process' state changes from waiting to ready, so no longer blocks and is put back in the run queue.
    - Note that this may cause the currently running process to be interrupted and exchanged for another.
  - A process' state changes from running to ready, so it could have continued to run (no blockade) but its assignment of the CPU is revoked.
- Note that these additions concern *involuntary* context switches, or *process preemptions*.

The CPU scheduler (4)

- Our OS model of having a timer interrupt handler that periodically invokes the scheduler is dependent on preemptive scheduling.
  - Recall we could terminate “infinite loop processes” this way as a kernel always regain control of the system.
- Most modern OS kernels implement preemptive scheduling.
- This is not without problems however:
  - What if a process is involuntarily interrupted while manipulating shared data structures?
  - What if a process is involuntarily interrupted while manipulating (shared) kernel data structures from a system call?
  - This, and similar, problems need a solution when we want to enable preemptive scheduling.

The dispatcher

- The task of the CPU scheduler is clear: given a ready queue, give us the next process to run.
- Now another routine is necessary which replaces the currently running process with the next process to run.
  - This is the task of the dispatcher.
  - It handles, amongst others:
    - State saving (register state)
    - Switching context / address space
    - Resuming next process at the right location and switch back to user mode.
  - The time required to perform this switch is referred to as the dispatch latency (and is pure overhead).

Scheduling Algorithms

- We are now ready to see some scheduling algorithms.
- An important question first: what to optimize for?
- Common scheduling criteria:
  - **CPU utilization**: maximize use of the CPU.
  - **Throughput**: maximize tasks that complete per time unit (e.g. requests or transactions per second).
  - **Turnaround time**: minimize time it takes to complete a process from start to finish.
  - **Wait time**: minimize time that processes spend waiting in queues.
  - **Response time**: minimize time between request submission (or device input) and first response of the system.

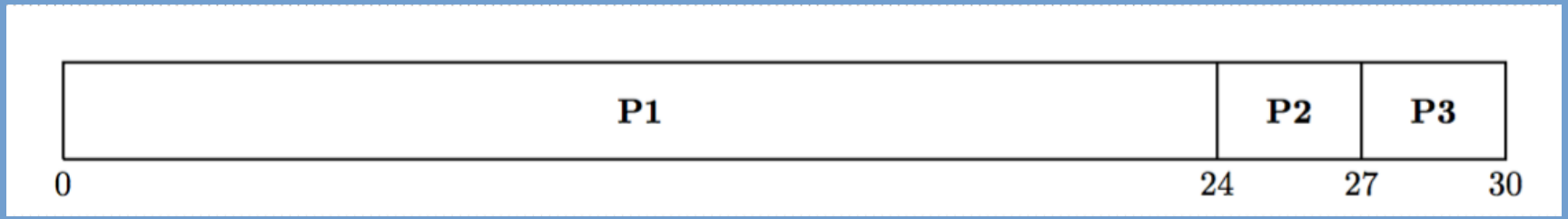
Scheduling Algorithms (2)

- We will now discuss a number of (simple) scheduling algorithms.
  - To keep things simple and manageable a single CPU burst (in milliseconds) per process is considered.
  - We optimize for average waiting time.
  - We chart the schedules using a Gantt chart.
  - The algorithms are non-preemptive unless otherwise noted.

FCFS: First-come First-Serve

Process	Burst time
P1	24
P2	3
P3	3

- Given is that all processes are present in the ready queue at time 0 in the order: P1, P2, P3.
- This results in the following Gantt chart:

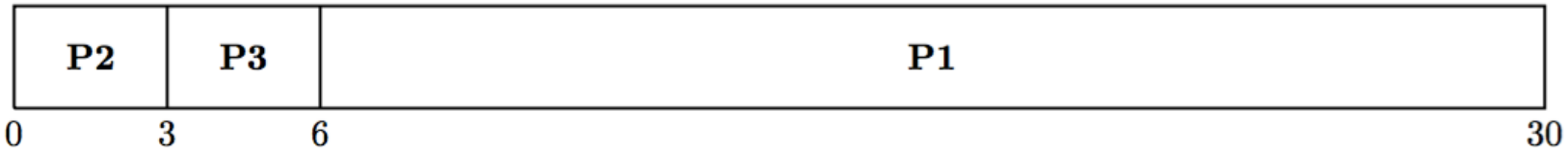


- Compute waiting times for each process: P1 = 0; P2 = 24; P3 = 27.
- This gives as average waiting time (0+24+27)/3 = 17

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

FCFS: First-come First-Serve (2)

- Now, assume the processes are present in the ready queue at time 0 in a different order: P2, P3, P1.
- The schedule changes and we obtain a different Gantt chart:



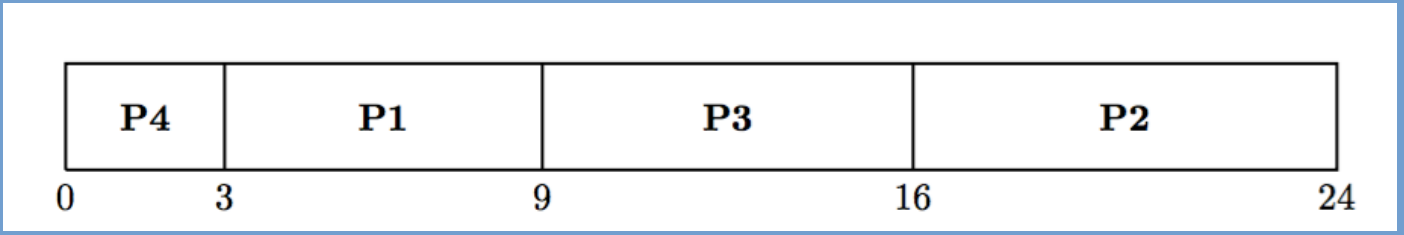
- New average waiting time: (0+3+6)/3=3
  - Much better result.
- We see here that the achieved performance of the algorithm depends on the order of the processes in the run queue.
  - *Convoy effect*: short processes waiting behind a long process.
  - For example consider a (long) CPU bound processes and many (interactive) I/O bound processes.

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

SJF: Shortest Job First

Process	Burst time
P1	6
P2	8
P3	7
P4	3

- Idea: select the process with the shortest CPU burst.
- All processes present in the ready queue at time 0.



- Average waiting time: (3+16+9+0)/4 = 7

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

SJF: Shortest Job First (2)

- The good news: SJF is proven optimal! It results in the minimum average waiting time for any set of processes.

- Now the bad news: We usually don't know the length of the CPU burst beforehand.
  - Except in batch scheduling systems where you could ask the user.
  - Or if you design a system such that a prediction for the length of the CPU burst must be given when entering the ready queue: difficult!

SJF: Shortest Job First (3)

- A way out is to estimate the length and use these predictions to select the (predicted) shortest job.
- For instance, this can be achieved by recording the length of previous CPU bursts and using exponential averaging.

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$.

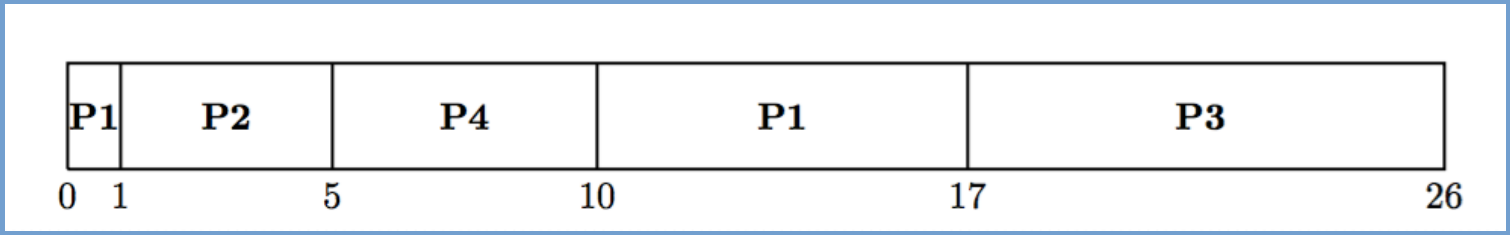
- Alpha is commonly set to 0.5.

SRTF: Shortest Remaining Time First

- SRTF is the preemptive version of SJF.
- We now also take the arrival times of processes into the ready queue into account. When a process arrives in the ready queue the scheduler is invoked.

Process	Arrival Time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- The following Gantt chart is obtained:



- Note the process preemption at time 1.
- Average waiting time: ((10-1)+(1-1)+(17-2)+5-3))/4 = 26/4 = 6.5

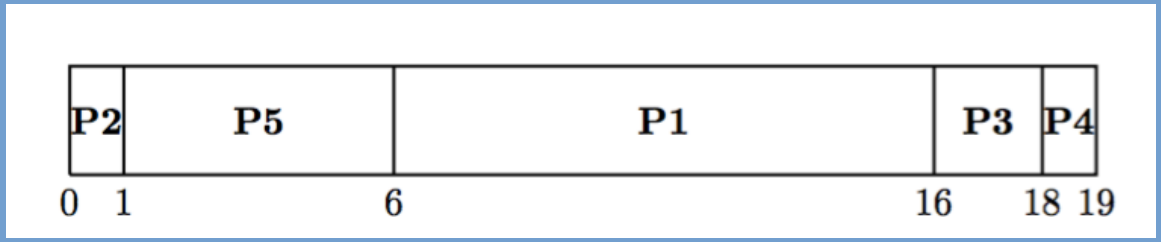
Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Priority Scheduling

- In fact, SJF and STRF are instances of a priority scheduling algorithm.
- In a priority scheduling algorithm, a priority must be computed for each process. The process with the highest priority is scheduled first.
  - The priority is often an integer number.
  - For SJF the priority is the inverse of the predicted duration of the next CPU burst.
- Priority scheduling algorithms suffer from starvation. Low priority processes may never execute.
  - A solution is to implement aging, in which the priority of a process is gradually increased as it ages (spends more time waiting).

Priority Scheduling (2)

Process	Priority	Burst time
P1	3	10
P2	1	1
P3	4	2
P4	5	1
P5	2	5



- Average waiting time: (0+1+6+16+18)/5 = 8.2

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

RR: Round Robin

- A preemptive scheduling algorithm devised for interactive systems is Round Robin scheduling.
- Idea: give each process a small time slice (or time quantum). If the process still runs when the time quantum expires, it is preempted and put at the end of the ready queue.
  - (Otherwise the process performed a blocking system call within its time quantum causing it to be replaced).
- Time quantum length typically 10 – 100 milliseconds.
  - 10 ms: compare with histogram of CPU bursts.

RR: Round Robin (2)

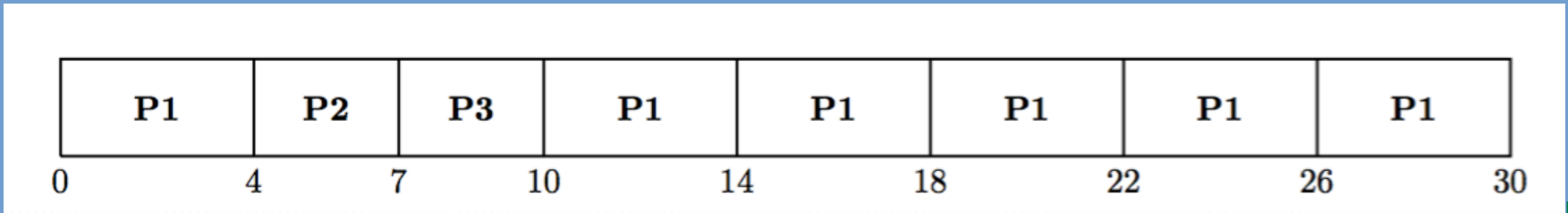
- Given n processes in the ready queue with time quantum is q, then each process gets $1/n$ of the CPU time in slices of (at most) q time units. Because processes rotate, no process waits longer than $(n\text{-}1)q$ time units for its turn.

- How are the processes interrupted? We need to program the timer interrupt for this. The timer interrupt handler must call a function which checks whether the time quantum of the currently running process has expired.

RR: Round Robin (3)

Process	Burst time
P1	24
P2	3
P3	3

- We apply Round Robin scheduling with a time quantum of 4. All processes are in the ready queue at time 0.

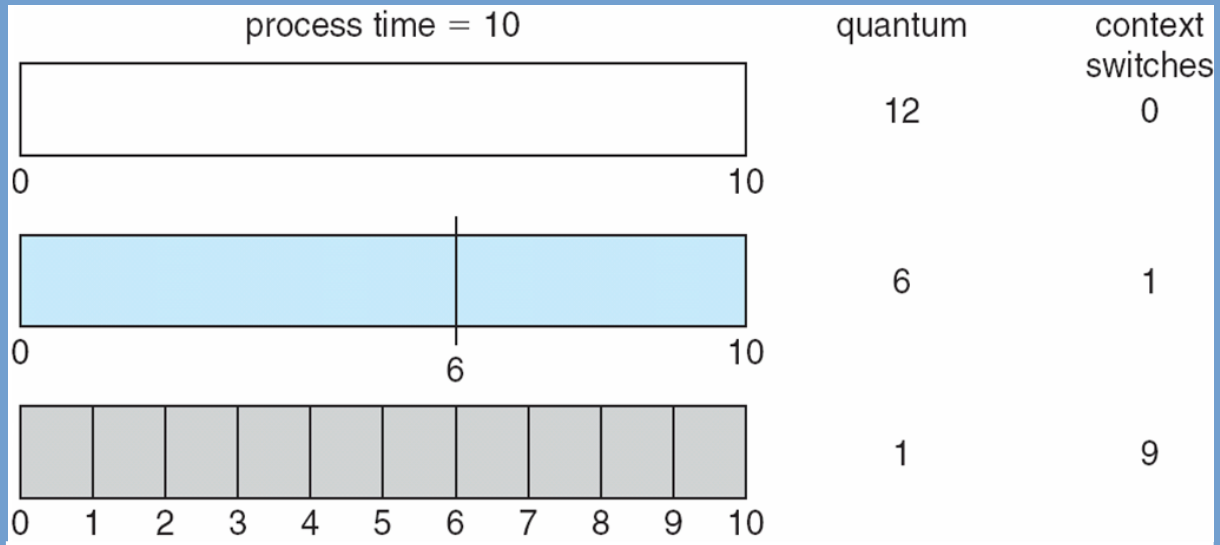


- Average waiting time: ((0+(10-4))+4+7)/3 = 5 2/3
- Higher average waiting time than SJF. This is typical and to be expected. However, for RR response time is the optimization objective.

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

RR: Round Robin (4)

- Tuning the time quantum is important.
  - Large quantum: many processes may run until their next voluntary context switch, so the scheduler behaves like a FIFO.
  - Small quantum: processes are always preempted and many more context switches are required to get the work done. Recall that context switches are pure overhead.
    - Naturally, time quantum should be larger than the context switch time.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

- Continue with the slides provided by the textbook, Chapter 6 (CPU Scheduling), starting at topic “Multilevel Queue”:

  http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html

End of Chapter 5.