COT 4600 Operating Systems Fall 2009

Dan C. Marinescu

Office: HEC 439 B

Office hours: Tu-Th 3:00-4:00 PM

Lecture 25

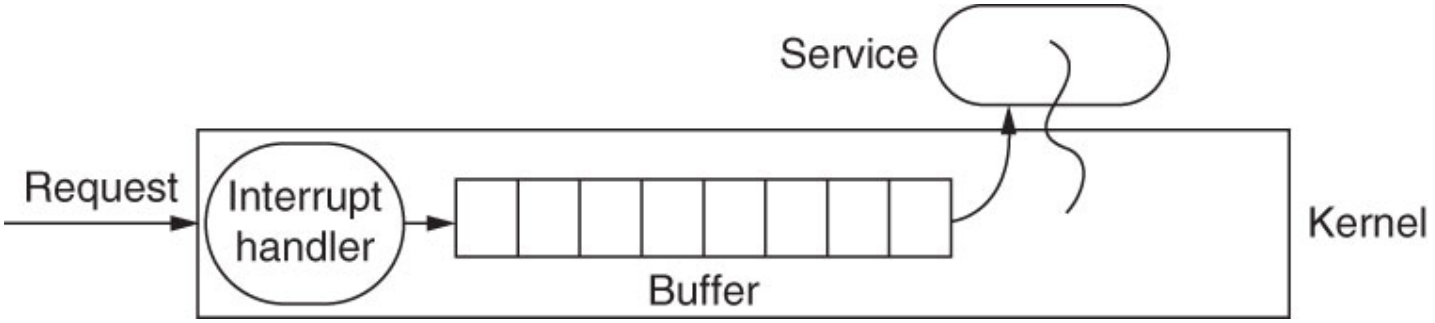
- Attention: project phase 4 and HW 6 – due Tuesday November 24
  - Final exam – Thursday December 10 4-6:50 PM
- Last time:
  - Multi-level memories
  - Memory characterization
  - Multilevel memories management using virtual memory
  - Adding multi-level memory management to virtual memory
- Today:
  - Scheduling
- Next Time:
  - Network properties (Chapter 7) - available online from the publisher of the textbook

Scheduling

- The process of allocating resource e.g., CPU cycles, to threads/processes.
- Distinguish
  - Policies
  - Mechanisms to implement policies
- Scheduling problems have evolved in time:
  - Early on: emphasis on CPU scheduling
  - Now: more interest in transaction processing and I/O optimization
- Scheduling decisions are made at different levels of abstraction and it is not always easy to mediate.

Example: an overloaded transaction processing system

- Incoming transaction are queued in a buffer which may fill up;
- The interrupt handler is constantly invoked as dropped requests are re-issued;
- The transaction processing thread has no chance to empty the buffer;
- Solution: when the buffer is full disable the interrupts caused by incoming transactions and allow the transaction processing thread to run.

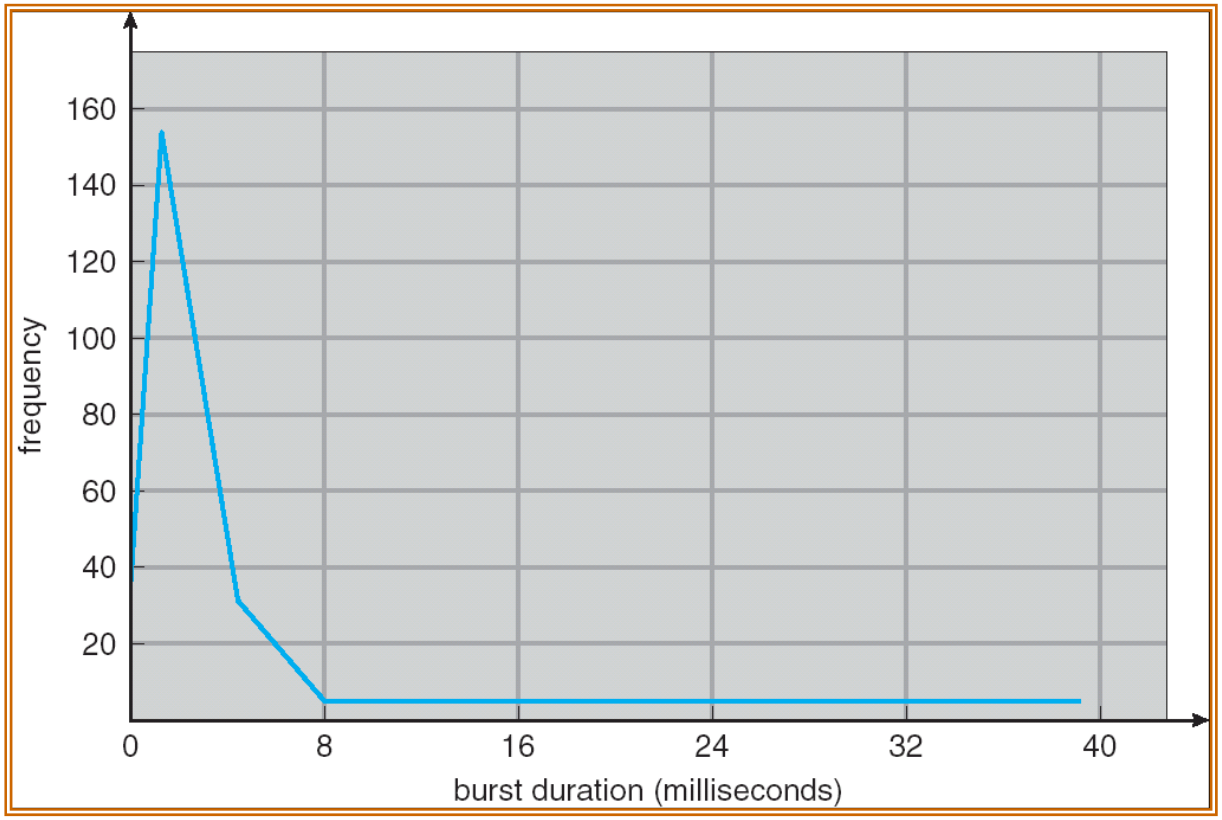


Scheduling objectives

- Performance metrics:
  - CPU Utilization ➔ Fraction of time CPU does useful work over total time
  - Throughput ➔ Number of jobs finished per unit of time
  - Turnaround time ➔ Time spent by a job in the system
  - Response time ➔ Time to get the results
  - Waiting time ➔ Time waiting to start processing
- All these are random variables ➔ we are interested in averages!!
- The objectives - system managers (M) and users (U):
  - Maximize CPU utilization ➔M
  - Maximize throughput ➔ M
  - Minimize turnaround time ➔ U
  - Minimize waiting time ➔ U
  - Minimize response time ➔ U

CPU burst

- CPU burst → the time required by the thread/process to execute



Scheduling policies

- First-Come First-Serve (FCFS)
- Shortest Job First (SJF)
- Round Robin (RR)
- Preemptive/non-preemptive scheduling

First-Come, First-Served (FCFS)

Thread	Burst Time
P_1	24
P_2	3
P_3	3

- Processes arrive in the order: P_1 ➔ P_2 ➔ P_3
  Gantt Chart for the schedule:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- *Convoy effect* ➔short process behind long process

FCFS Scheduling (Cont'd.)

- Now threads arrive in the order: P_2 ➔ P_3 ➔ P_1
- Gantt chart:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better!!

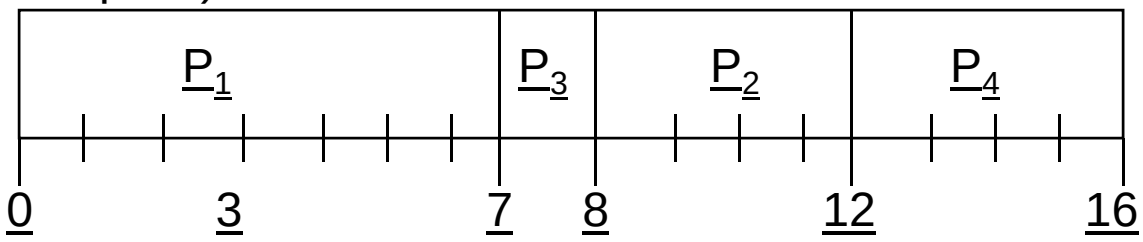
Shortest-Job-First (SJF)

- Use the length of the next CPU burst to schedule the thread/process with the shortest time.
- SJF is optimal ➔minimum average waiting time for a given set of threads/processes
- Two schemes:
  - Non-preemptive ➔ the thread/process cannot be preempted until completes its CPU burst
  - Preemptive ➔ if a new thread/process arrives with CPU burst length less than remaining time of current executing process, preempt. known as Shortest-Remaining-Time-First (SRTF)

Example of non-preemptive SJF

Thread	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

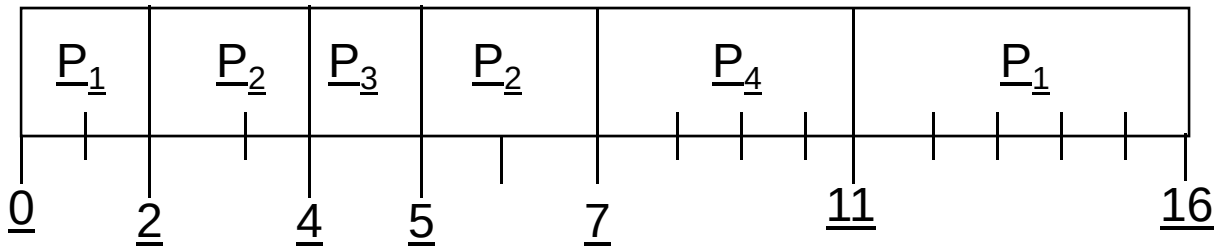


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Shortest-Remaining-Time-First (SRTF) (Preemptive SJF)

Thread	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- Shortest-Remaining-Time-First



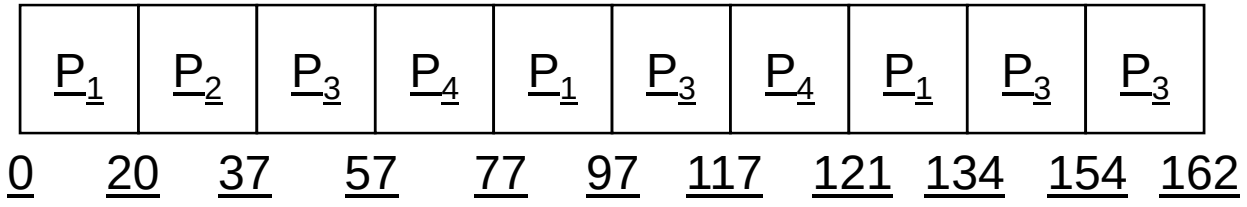
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the thread/process is preempted and added to the end of the ready queue.
- If there are *n* threads/processes in the ready queue and the time quantum is *q*, then each thread/process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No thread/process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

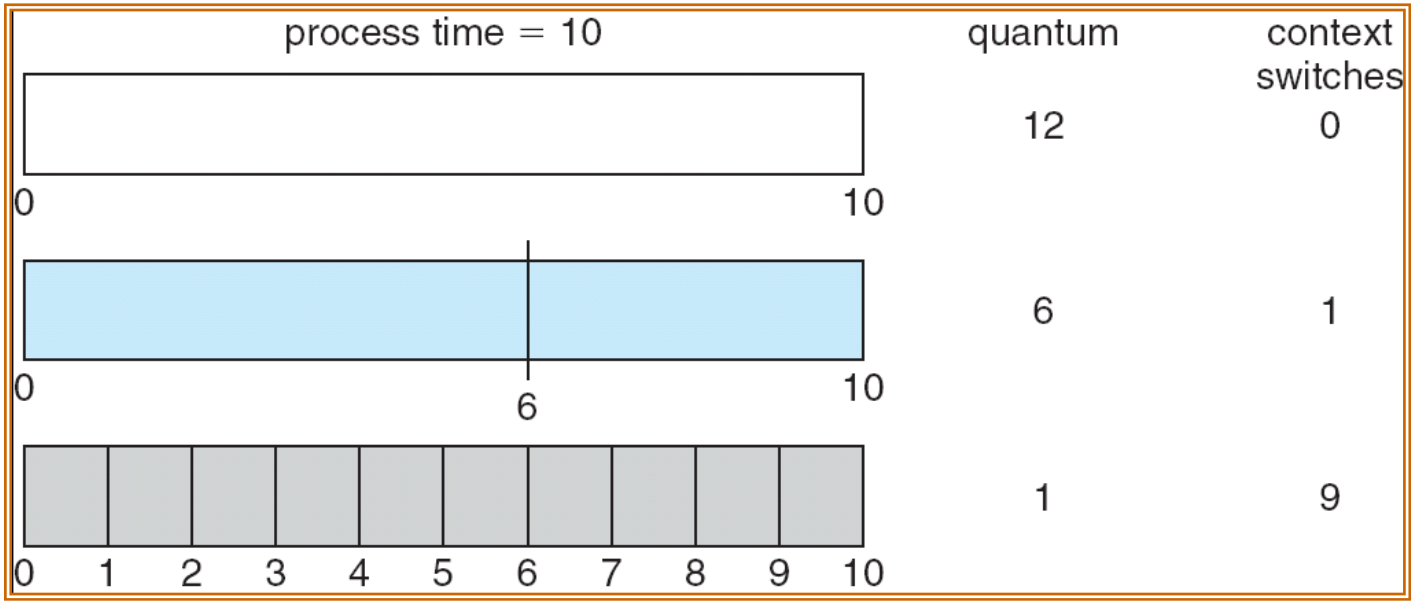
RR with time slice q = 20

Thread	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

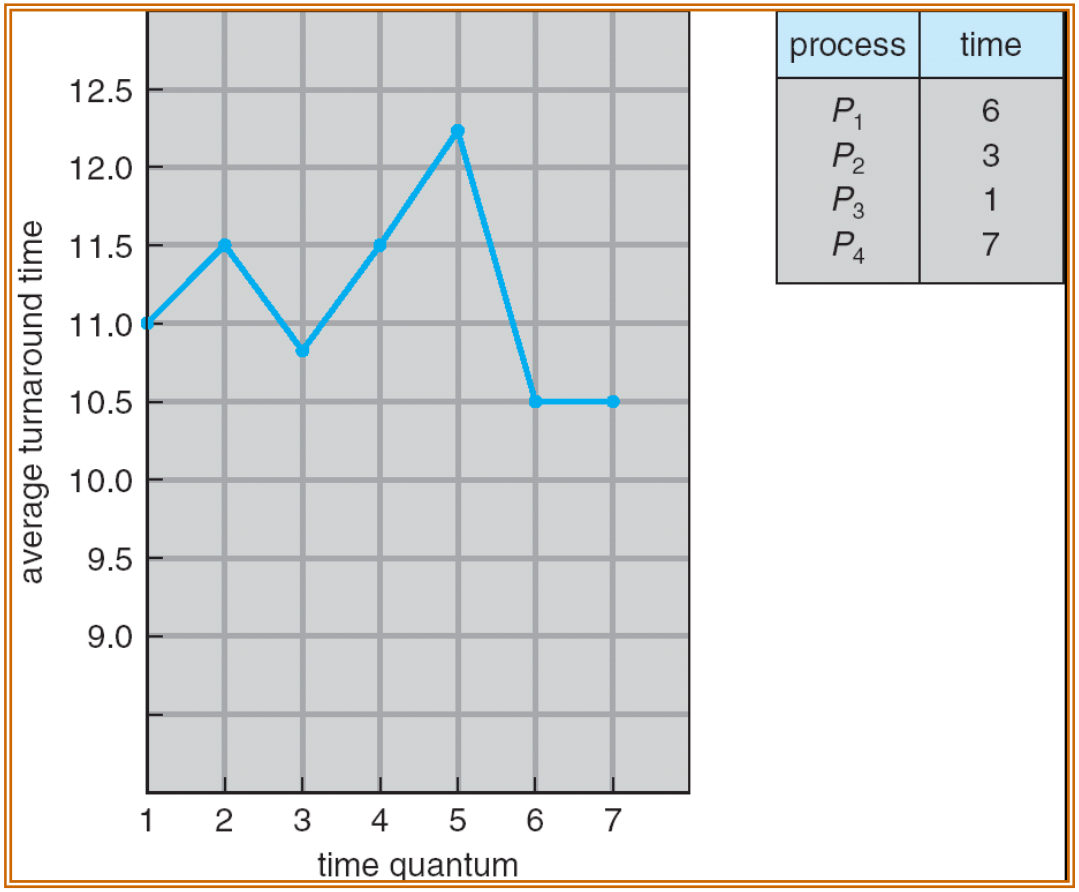


Typically, higher average turnaround than SJF, but better *response*

Time slice (quantum) and context switch time



Turnaround time function of time quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Job	Arrival time	Work	Start time	Finish time	Wait time till start	Time in system
A	0	3	0	3	0	3
B	1	5	3	3 + 5 = 8	3 – 1 = 2	8 – 1 = 7
C	3	2	8	8 + 2 = 10	8 – 3 = 5	10 – 3 = 7

A	0	3	0	3	0	3
B	1	5	5	5 + 5 = 10	4	10 – 1 = 9
C	3	2	3	3 + 2 = 5	0	5 – 3 = 2

A	0	3	0	6	0	6 – 0 = 6
B	1	5	1	10	1 – 1 = 0	10 – 1 = 9
C	3	2	5	8	5 – 3 = 2	8 – 3 = 5

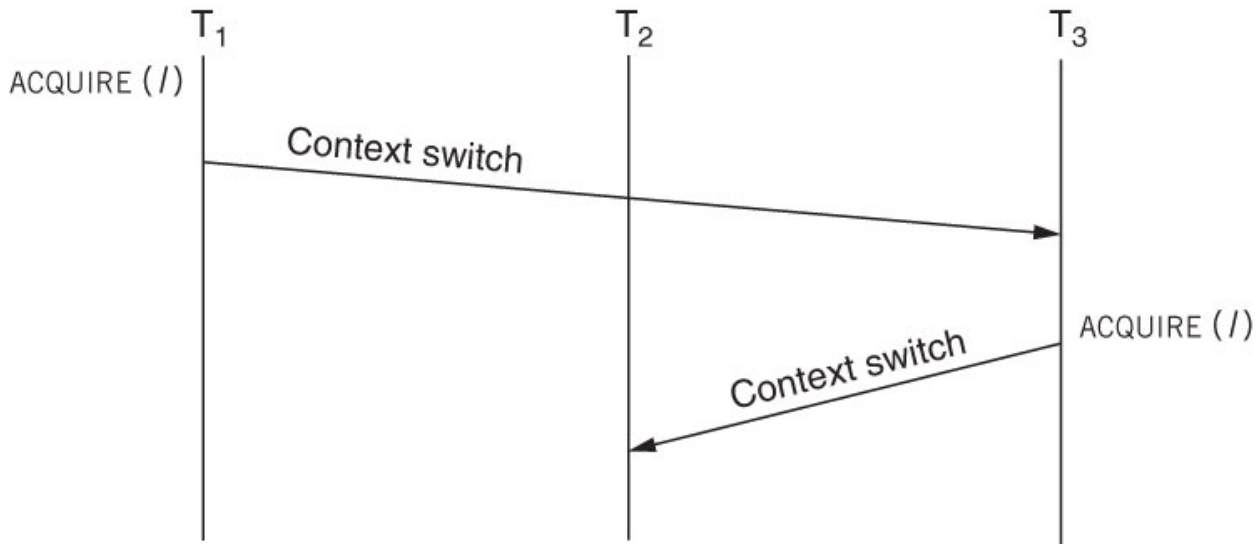
Scheduling policy	Average waiting time till the job started	Average time in system
FCFS	7/3	17/3
SJF	4/3	14/3
RR	3/3	20/3

Priority scheduling

- Each thread/process has a priority and the one with the highest priority (smallest integer ≡ highest priority) is scheduled next.
  - Preemptive
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ➔ Starvation – low priority threads/processes may never execute
- Solution to starvation ➔ Aging – as time progresses increase the priority of the thread/process
- Priority my be computed dynamically

Priority inversion

- A lower priority thread/process prevents a higher priority one from running.
- T_3 has the highest priority, T_1 has the lowest priority; T_1 and T_3 share a lock.
- T_1 acquires the lock, then it is suspended when T_3 starts.
- Eventually T_3 requests the lock and it is suspended waiting for T_1 to release the lock.
- T_2 has higher priority than T_1 and runs; neither T_3 nor T_1 can run; T_1 due to its low priority, T_3 because it needs the lock help by T_1.
- Allow a low priority thread holding a lock to run with the higher priority of the thread which requests the lock



Estimating the length of next CPU burst

- Done using the length of previous CPU bursts, using exponential averaging

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

1. $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
3. $\alpha, 0 \le \alpha \le 1$

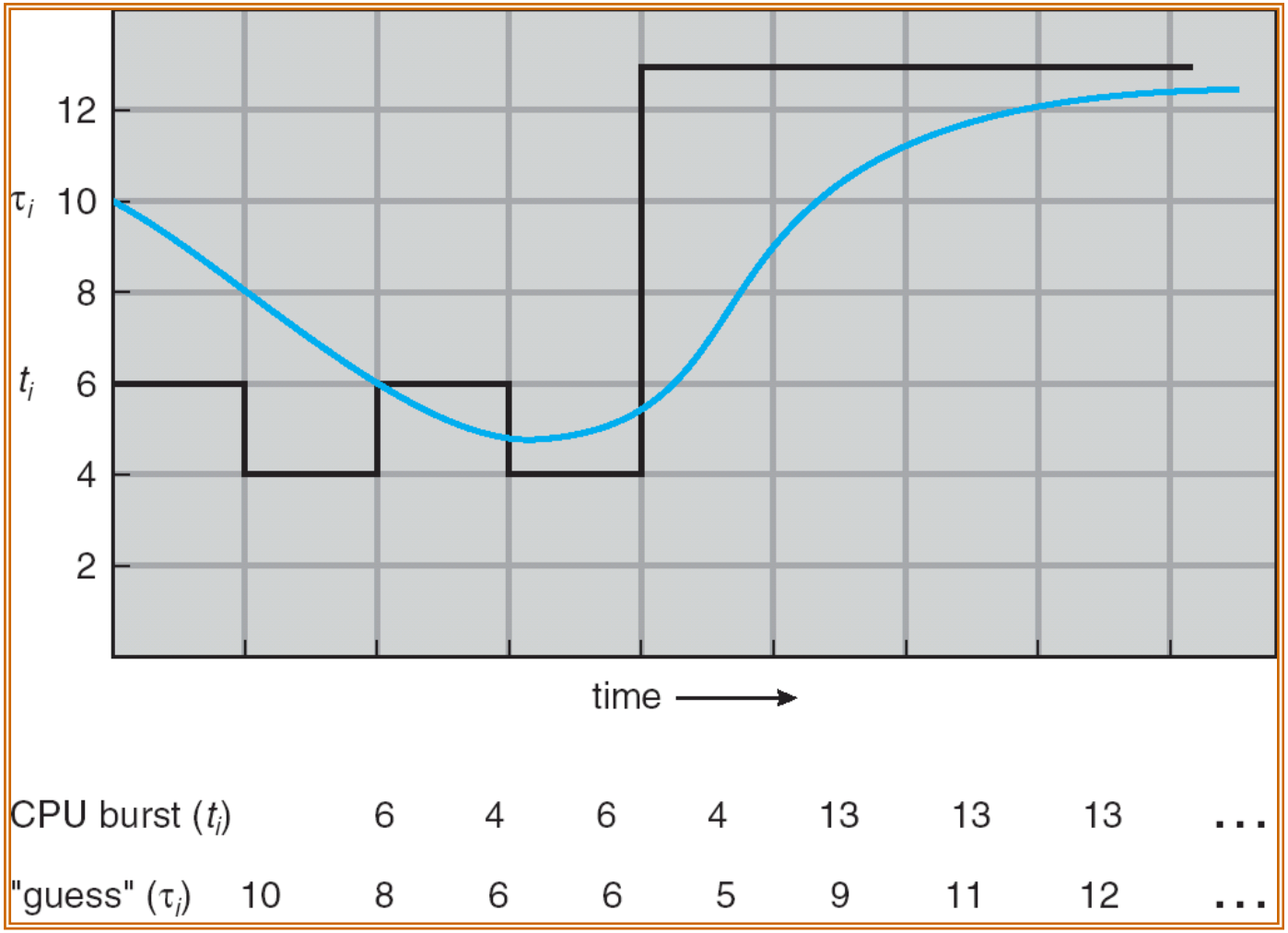
Exponential averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Predicting the length of the next CPU burst

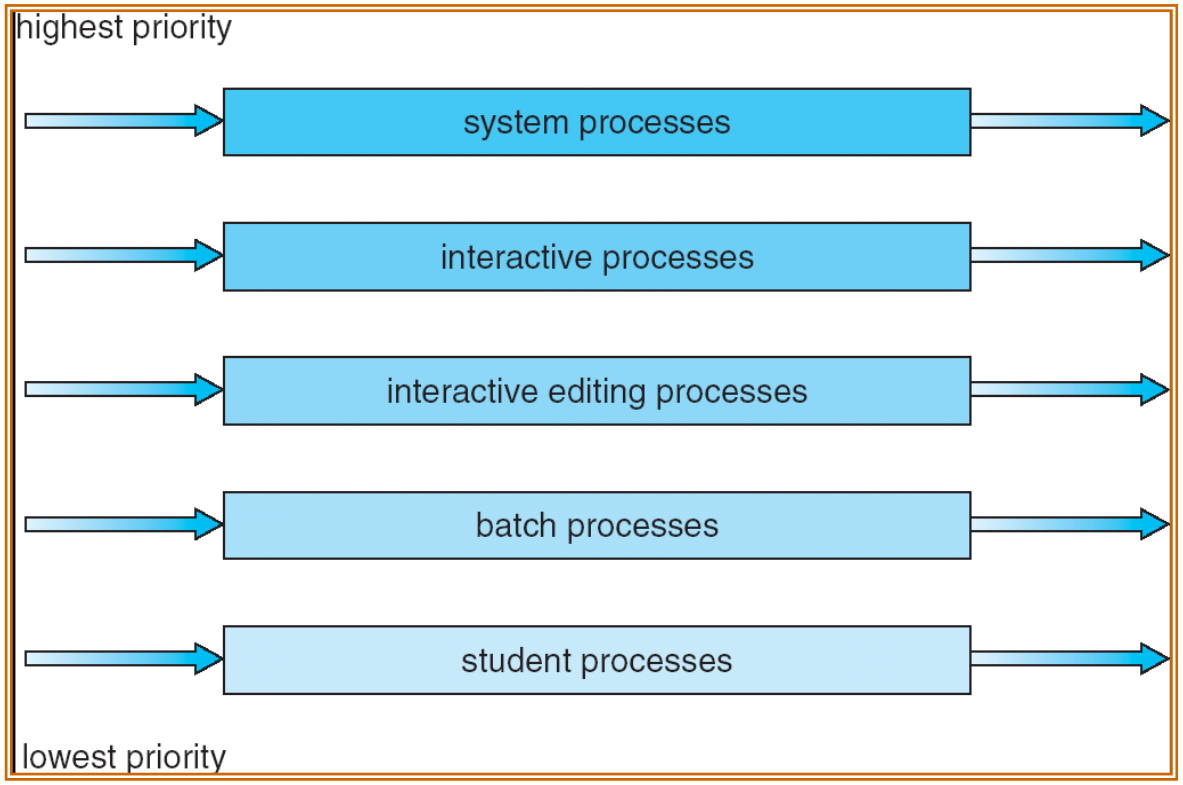


CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Multilevel queue

- Ready queue is partitioned into separate queues each with its own scheduling algorithm :
  - foreground (interactive) ➔ RR
  - background (batch) ➔ FCFS
- Scheduling between the queues
  - Fixed priority scheduling - (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e.,
    - 80% to foreground in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling



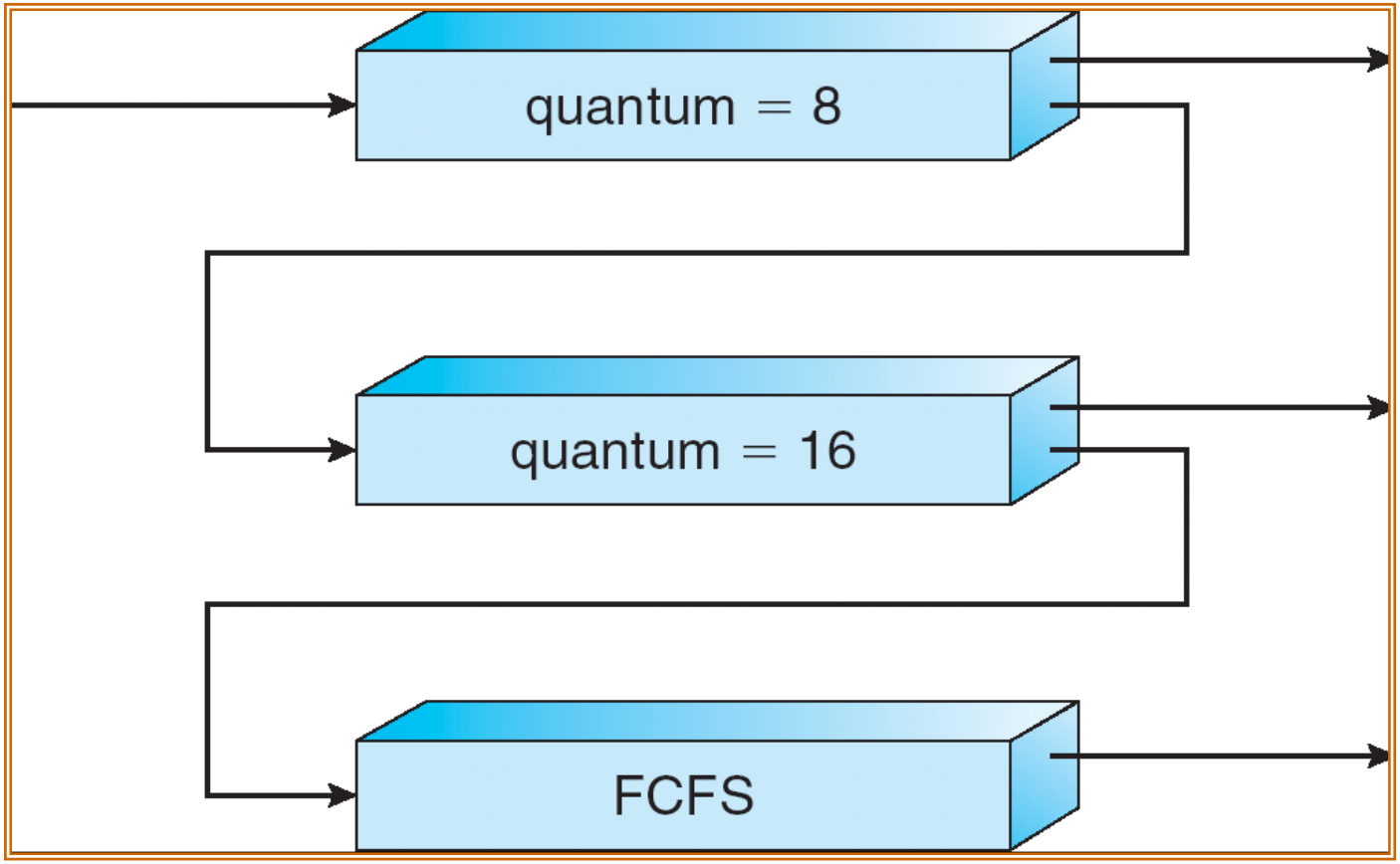
Multilevel feedback queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler characterized by:
  - number of queues
  - scheduling algorithms for each queue
  - strategy when to upgrade/demote a process
  - strategy to decide the queue a process will enter when it needs service

Example of a multilevel feedback queue exam

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Unix scheduler

- The higher the number quantifying the priority the lower the actual process priority.
- Priority = (recent CPU usage)/2 + base
- Recent CPU usage ➔ how often the process has used the CPU since the last time priorities were calculated.
- Does this strategy raises or lowers the priority of a CPU-bound processes?
- Example:
  - base = 60
  - Recent CPU usage: P1 =40, P2 =18, P3 = 10