RR schedulers with Multi-level queuing and Multi-processor scheduling

10-Aug-2016
CS303
Autumn 2016

Motivation

- Process types:
  - CPU-bound Vs I/O-bound
- There are several sched. Algos.: (we have seen...)
  - FCFS, SJF, RR, SJRT
  - SJF/SJRT provide better avg. wait time and Throughput

Can RR be made to exploit the nature of a process?

  - RR can be adapted to expedite IO-bound process than a CPU-bound process
  - Consequently RR also exhibits improved
    - avg. wait time and throughput etc.
- How?
  - With Multi-level queuing we can achieve this improvement

Multilevel queuing (1/2)

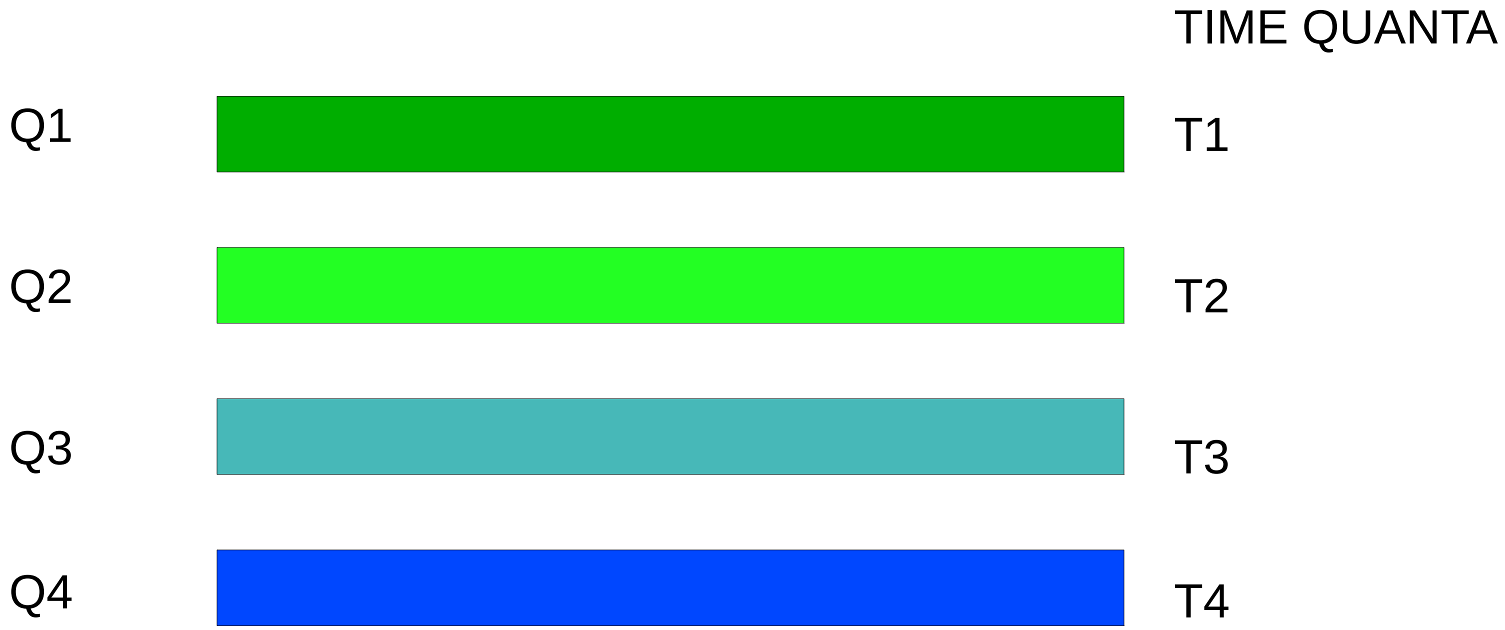
- Ready queue in RR can be conceptualised as a multi-level queue
  - consisting of multiple queues with different time-quanta
  - The queue with lower time-quanta is given more priority
  - Within the Queue Q_i, the processes go RR with timequanta T_i
    - Multi-level queue

	TIME QUANTA	CBT interval CI[i]
Q1	T1	(0,T1]
Q2	T2	(T1,T2]
Q3	T3	(T2,T3]
Q4	T4	(T3,T4]

Such that T1 < T2 < T3 < T4
And Priority: π(Q1) > π(Q2) > π(Q3) > π(Q4)

Multilevel queuing (2/2)

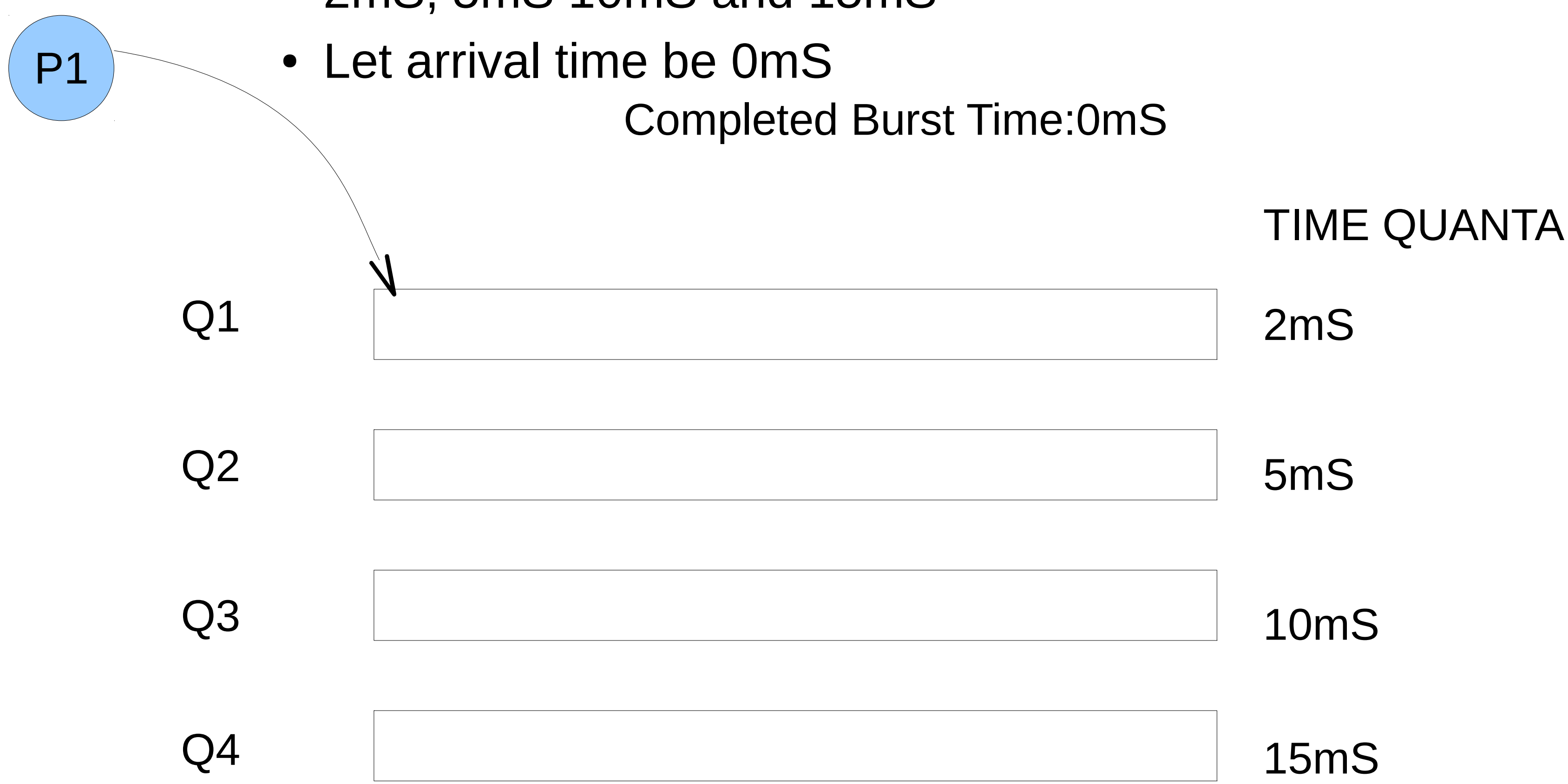
- Within every level/queue all the processes are scheduled in RR with corresponding time-quanta



Such that T1 < T2 < T3 < T4
And Priority is $\pi(Q1) > \pi(Q2) > \pi(Q3) > \pi(Q4)$

Multilevel queuing: Illustration (1/5)

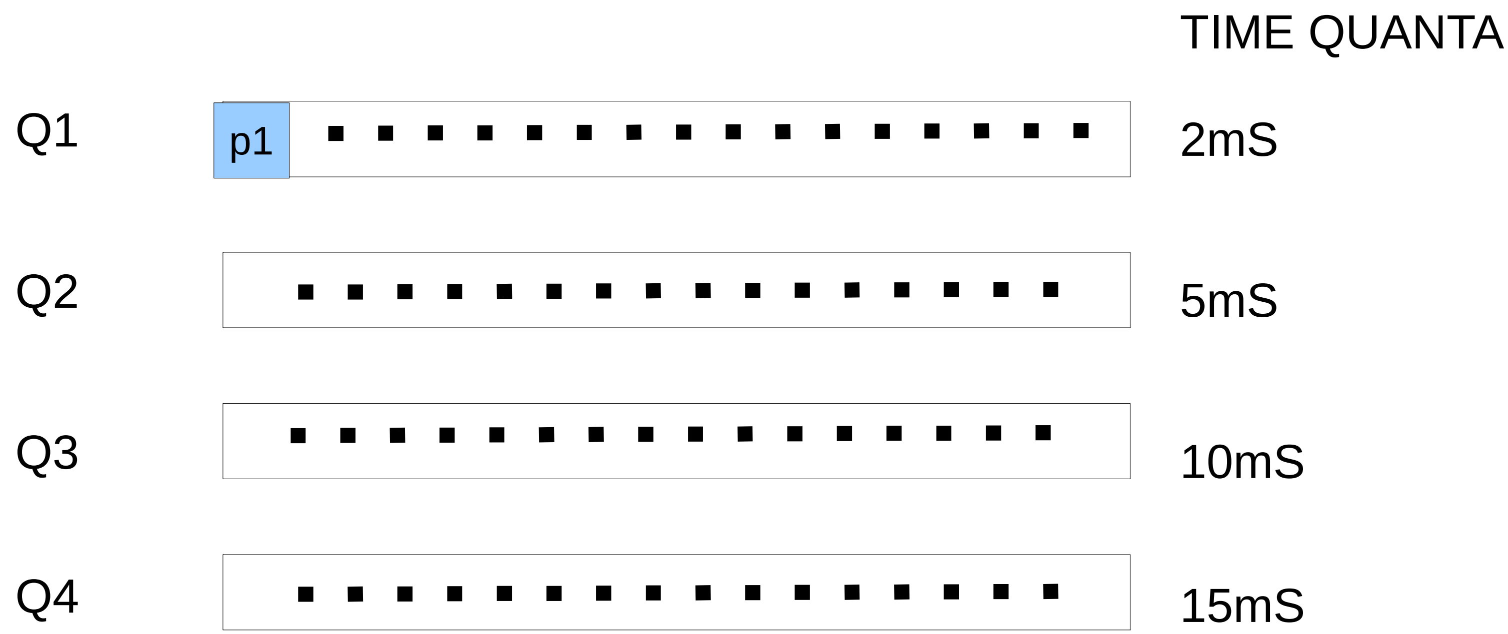
- Consider process P1 with.CBT 25 milli-Seconds(mSec)
  - There is a 4-level queue with timequantas:
    - 2mS, 5mS 10mS and 15mS
    - Let arrival time be 0mS



Multilevel queuing: Illustration (2/5)

- As per the algo., P1 is placed in Q1 with TQ of 2mS
- Waits for its turn on RR Scheduler of Q1

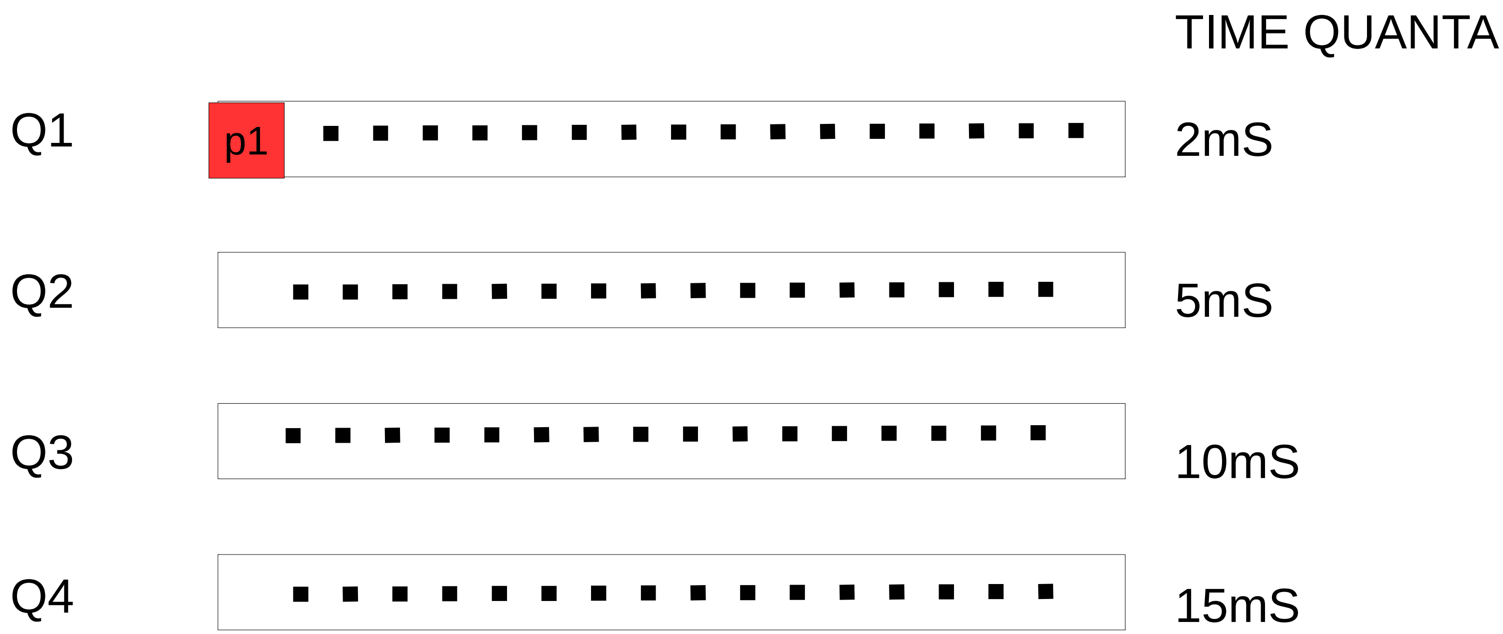
Completed Burst Time:0mS



Multilevel queuing: Illustration (2/5)

- As per the algo., P1 is placed in Q1 with TQ of 2mS
- Waits for its turn on RR Scheduler of Q1
- RR scheduler of Q1 dispatches P1, P1 starts running

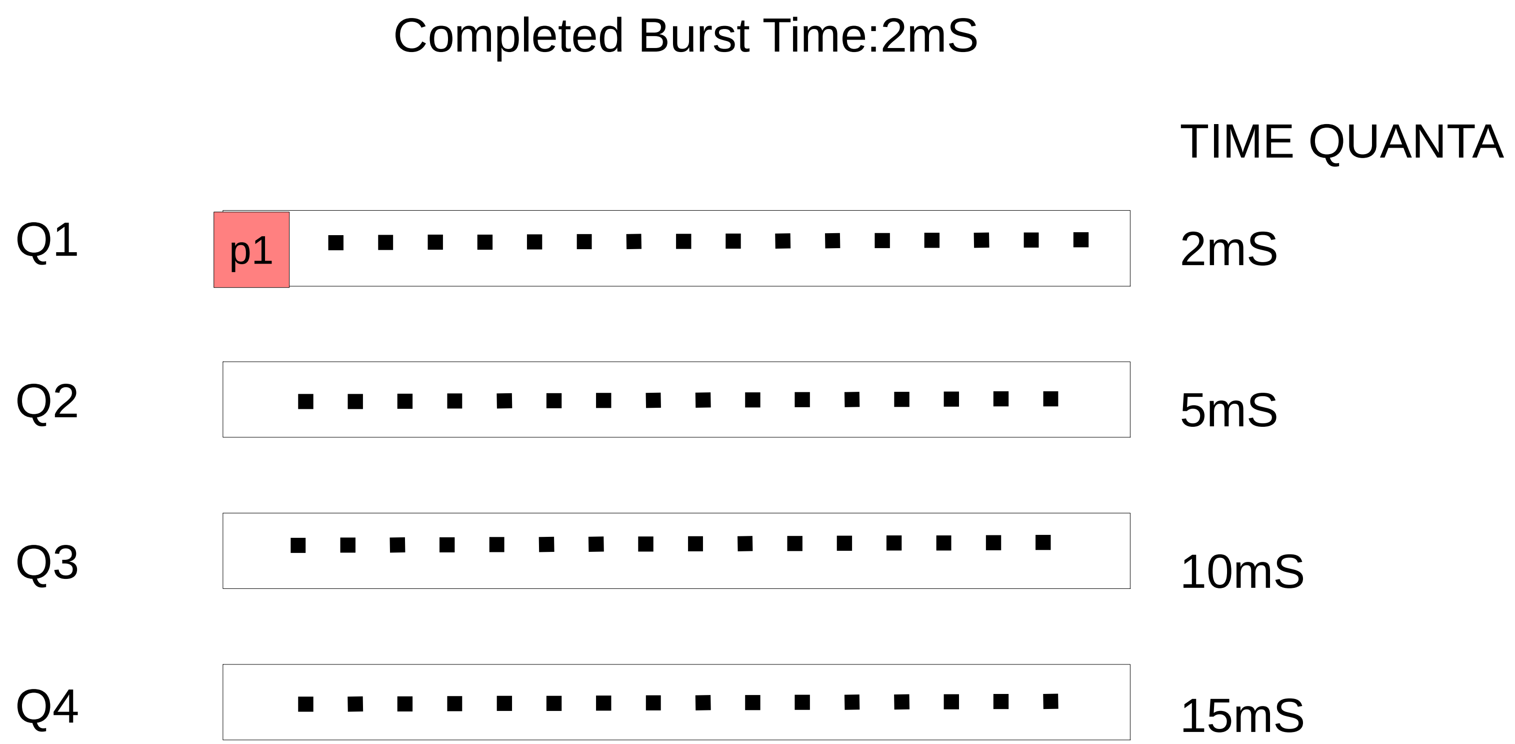
Completed Burst Time:0mS



Multilevel queuing: Illustration (3/5)

– P1 executes for 2mS; execution does not finish

– Gets preempted



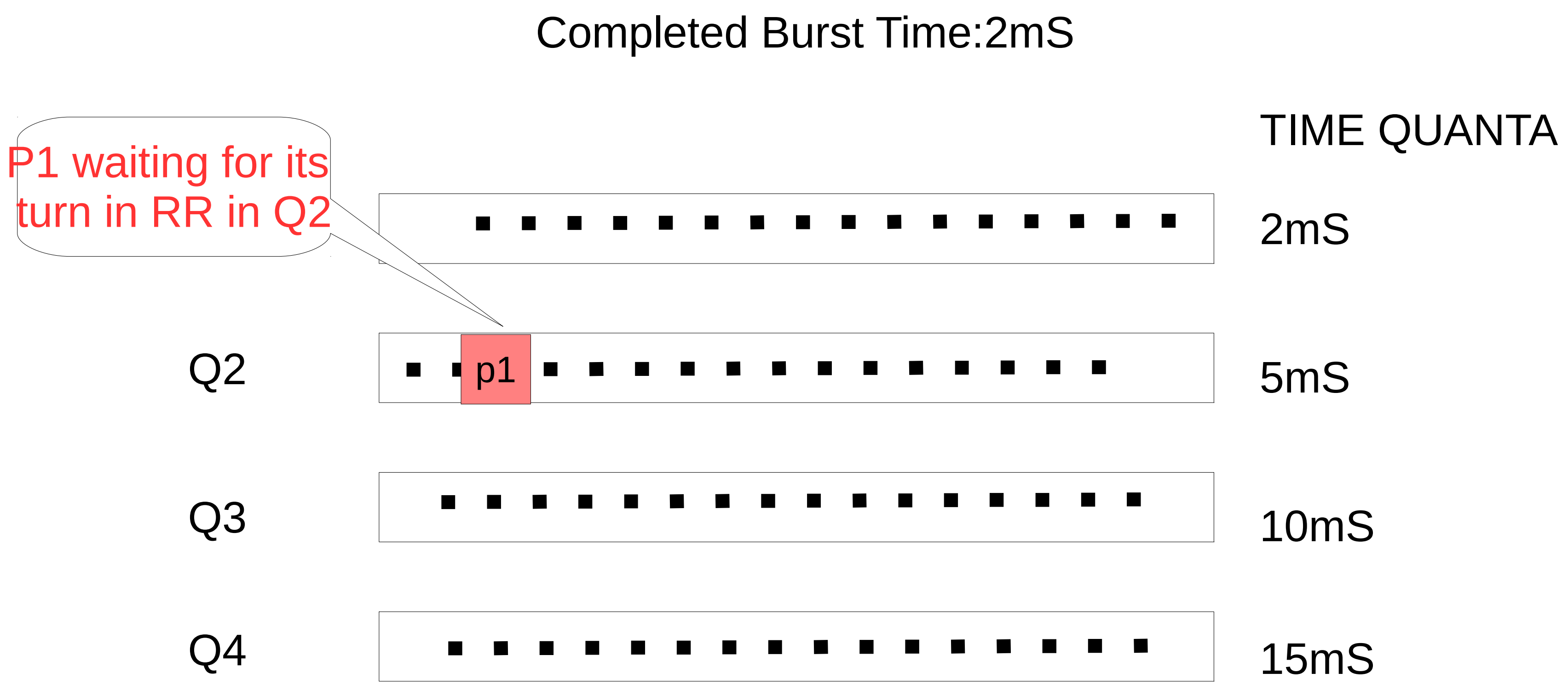
Multilevel queuing: Illustration (4/5)

- Since execution not finished
- It is moved down to level-2

Completed Burst Time:2mS

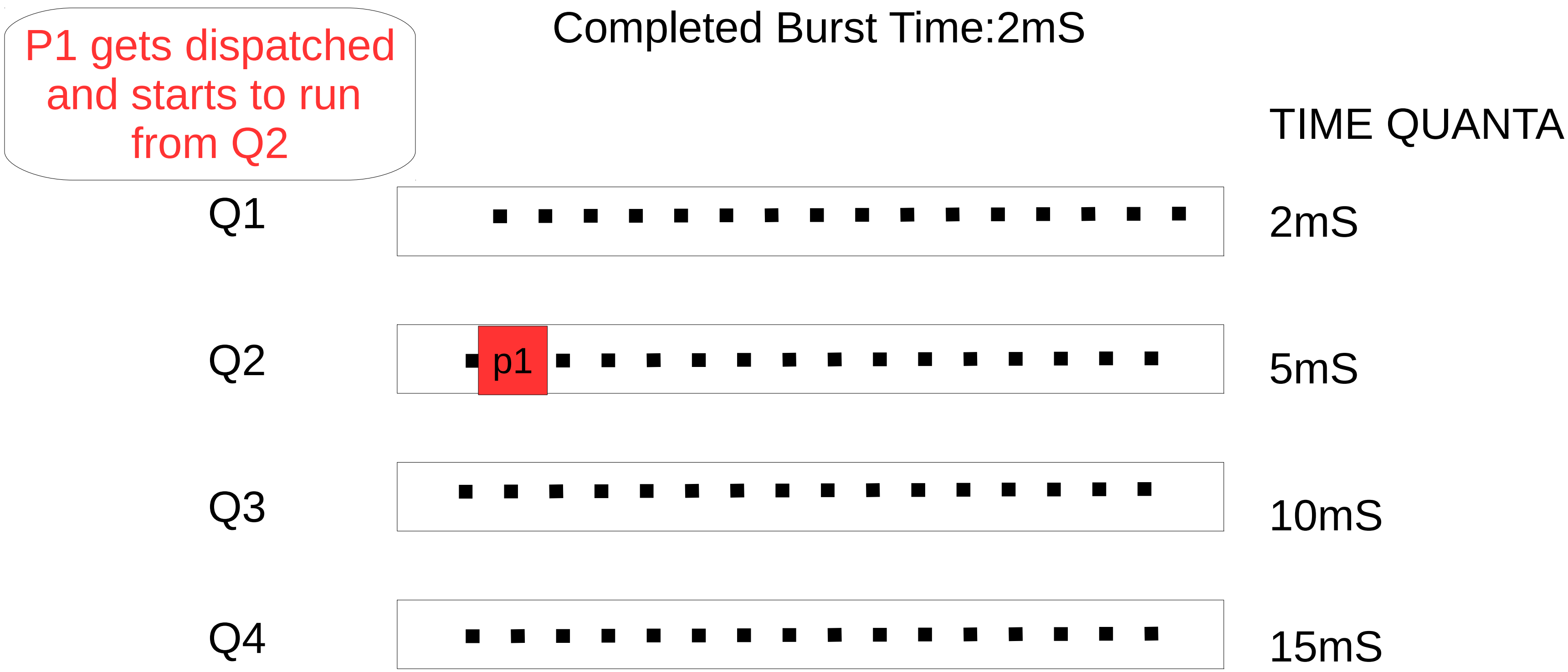
Multilevel queuing: Illustration (5)

- With 2mS the execution is not finished
- It is moved down to level-2



Multilevel queuing: Illustration (6/5)

- With 2mS the execution is not finished
- It is moved down to level-2
- **In Q2, on getting its turn in RR sched, starts execution**

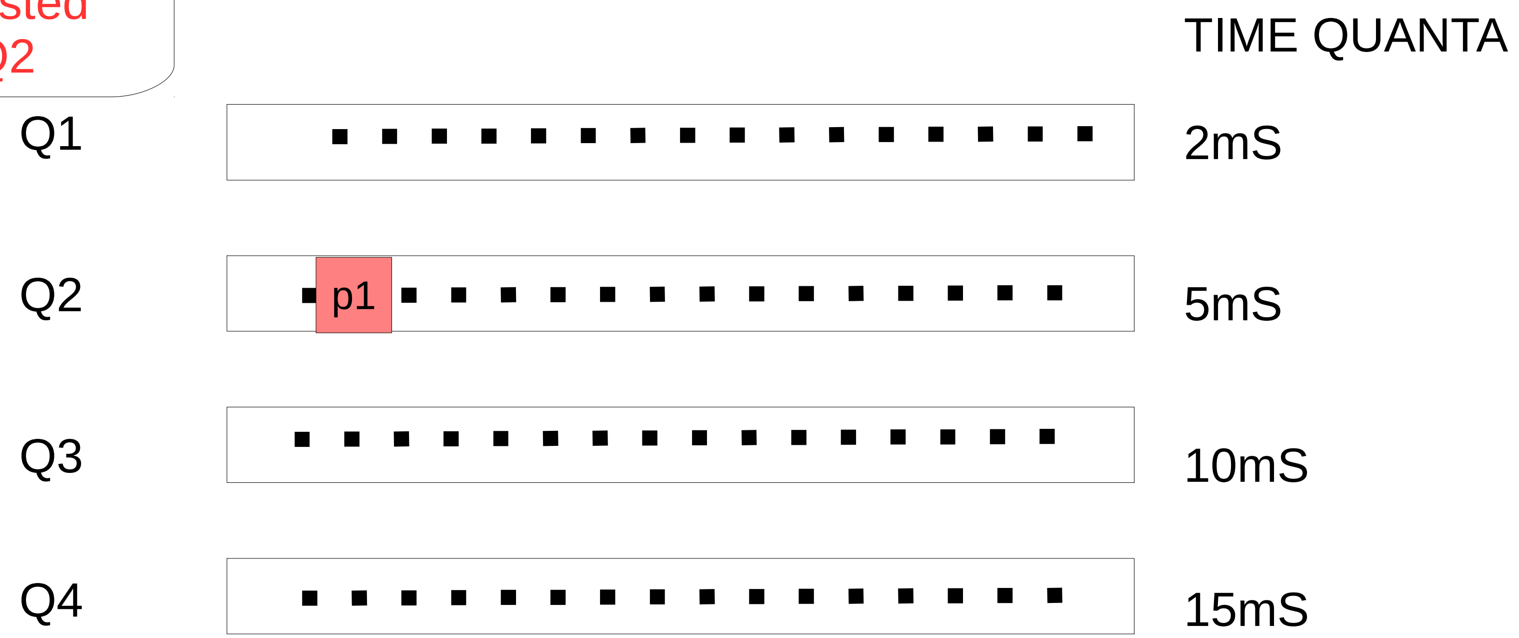


Multilevel queuing: Illustration (7/5)

- With 2mS the execution is not finished
- It is moved down to level-2
- In Q2, on getting its turn in RR sched, starts execution
- **P1 exhausts its quanta of 5mS in Q2; gets preempted**

Completed Burst Time:7mS

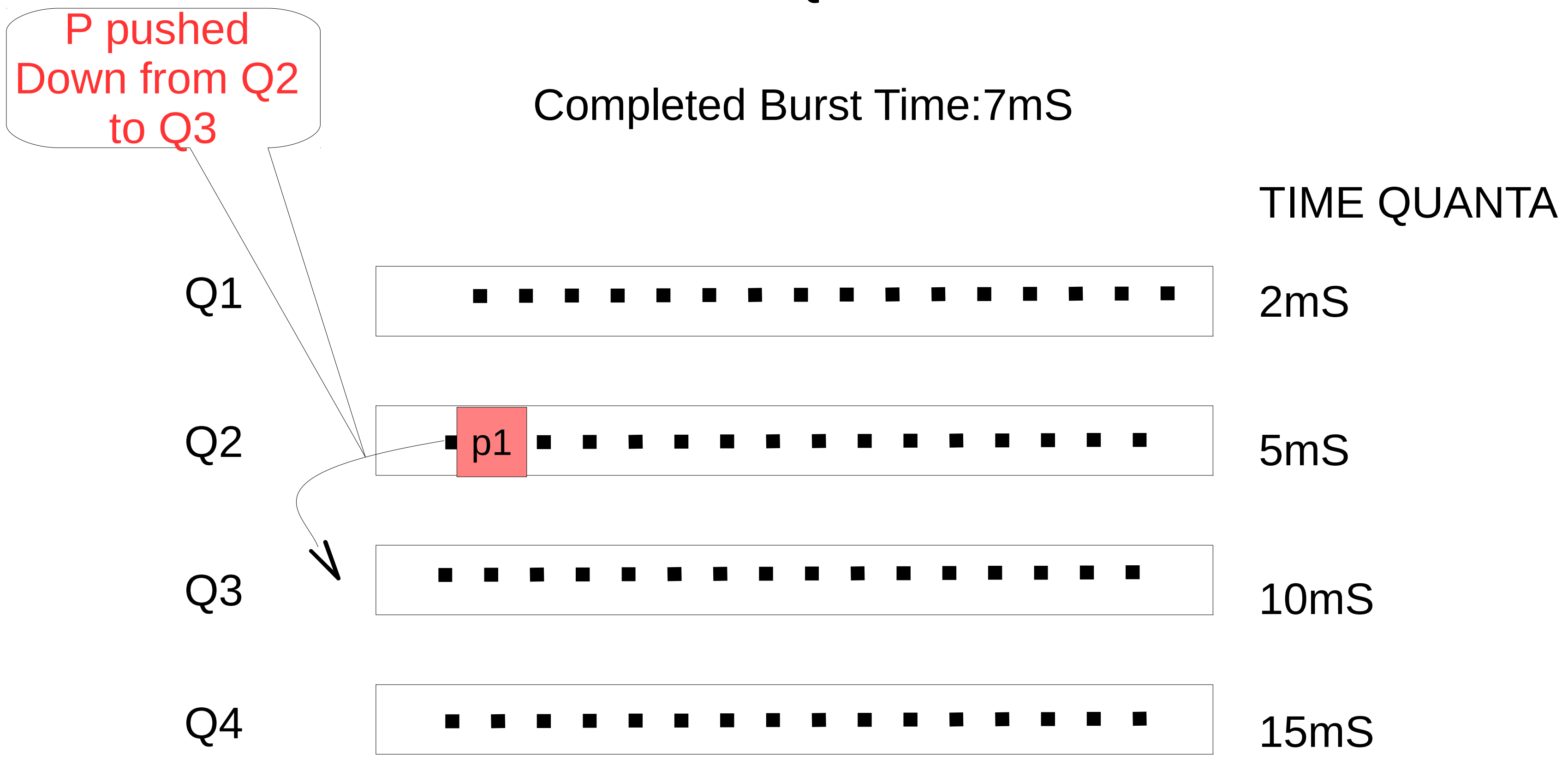
P1's Quanta exhausted in Q2



Multilevel queuing: Illustration (8/5)

In Q2, on getting its turn in RR sched, starts execution

- Process exhausts its quanta of 5mS in Q2
- P1 moves out to Q3

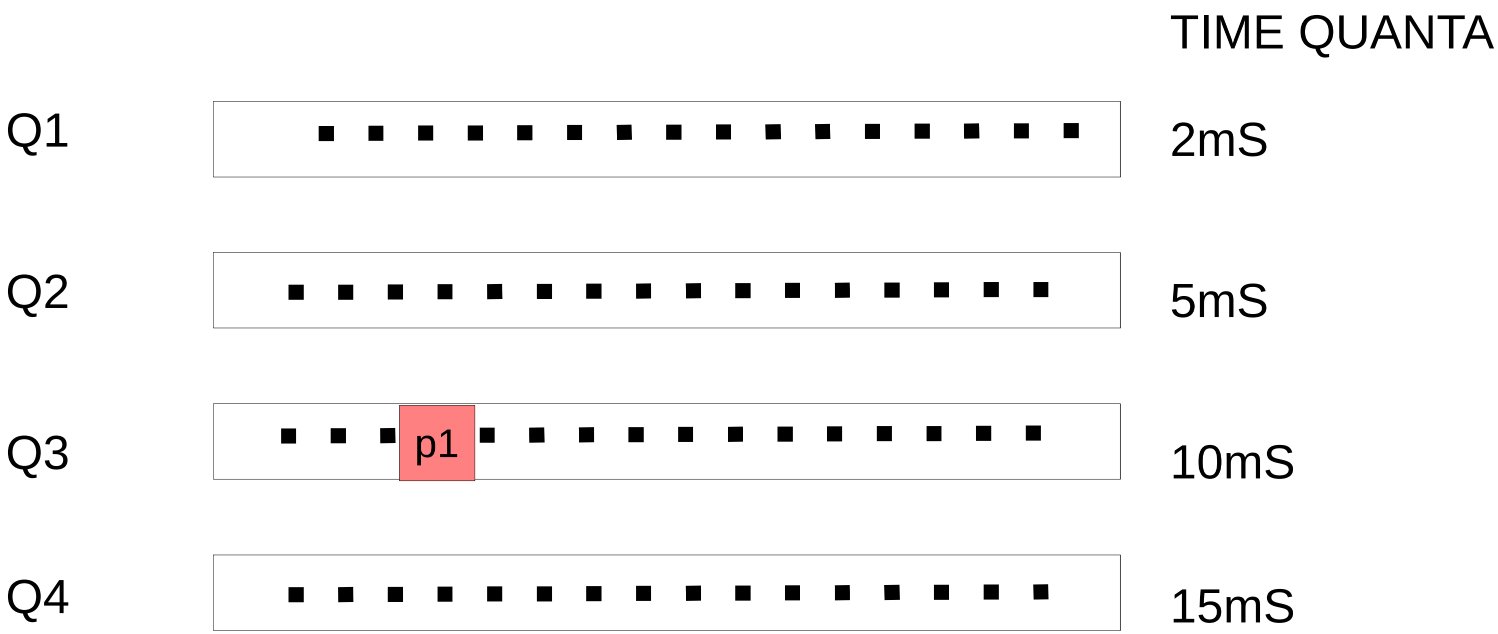


Multilevel queuing: Illustration (9/5)

Process exhausts its quanta of 5mS in Q2

- P1 moves out to Q3
- P1 in Q3
- P1 waits for its turn on RR sched of Q3

Completed Burst Time:7mS

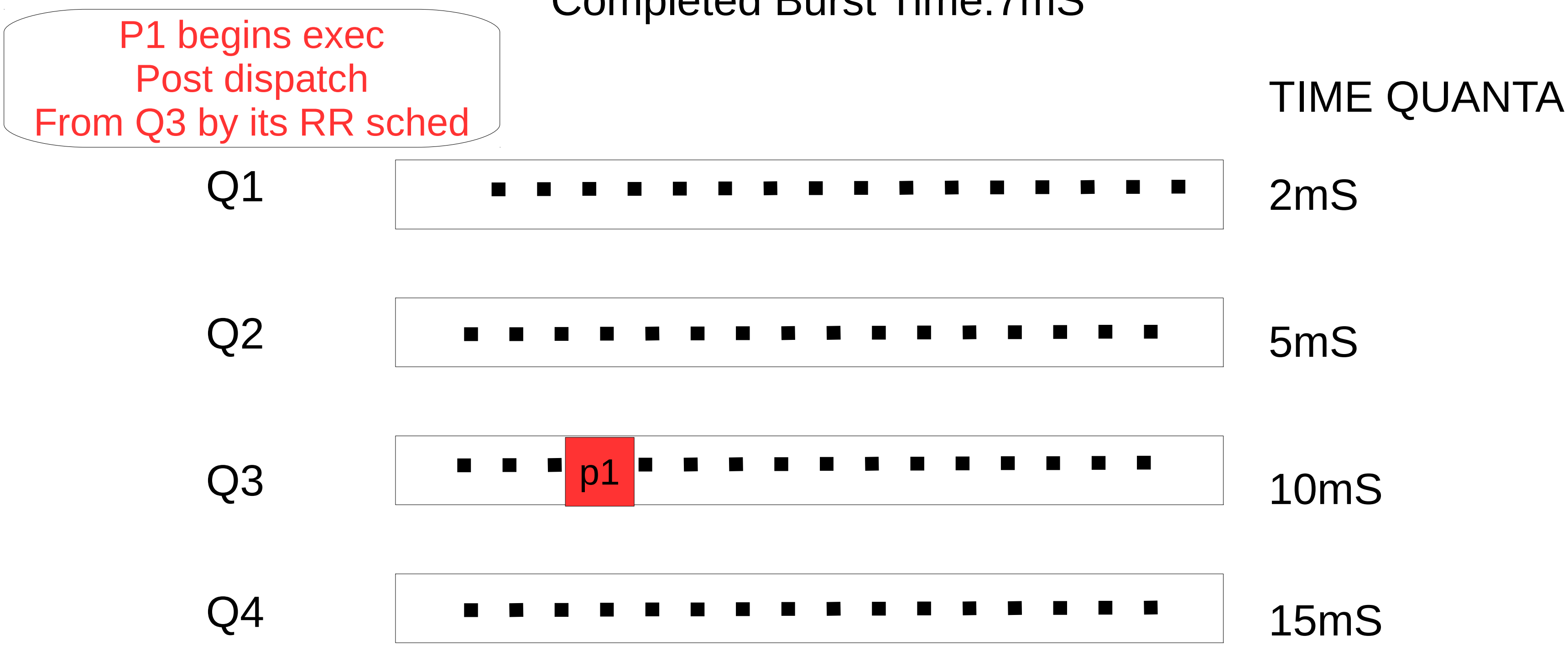


Multilevel queuing: Illustration (10/5)

P1 moves out to Q3

- P1 in Q3
- P1 waits for its turn on RR sched of Q3
- P1 is dispatched and start execution

Completed Burst Time:7mS



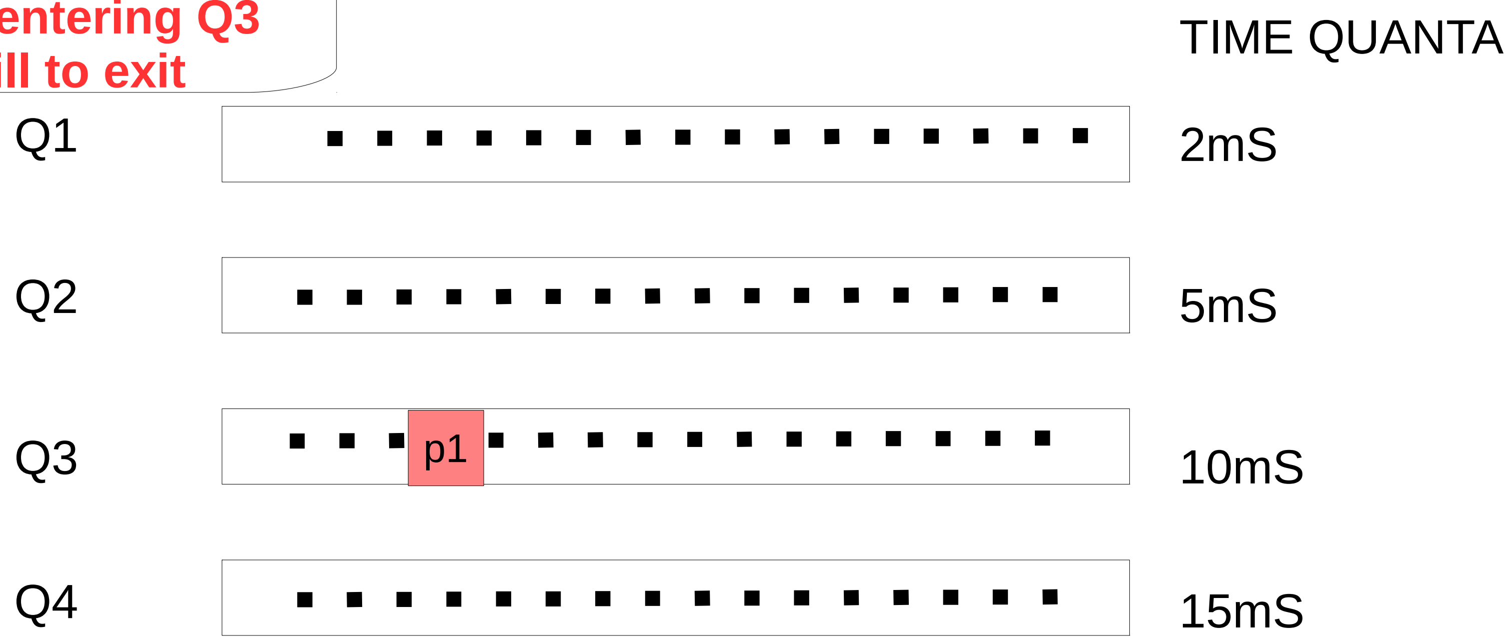
Multilevel queuing: Illustration (11)

P1 in Q3

- P1 waits for its turn on RR sched of Q3
- P1 is dispatched and start execution
- P1 exhausts its quanta of 10 mS in Q3, gets prempted

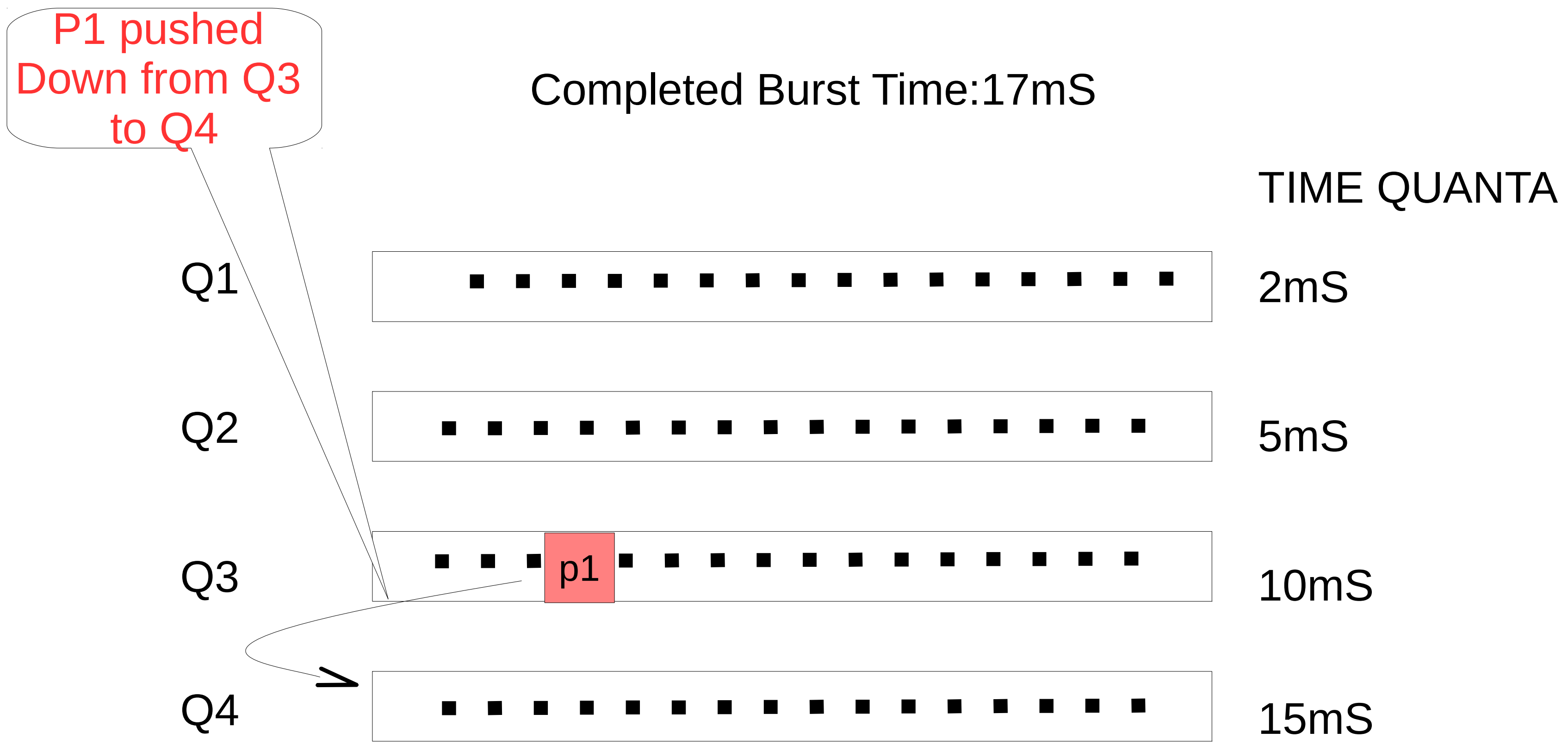
Completed Burst Time:17mS

P1 executes for 10mS
After entering Q3
Still to exit



Multilevel queuing: Illustration (11)

- P1 exhausts its quanta of 10 mS in Q3
- P1 is pushed down to Q4



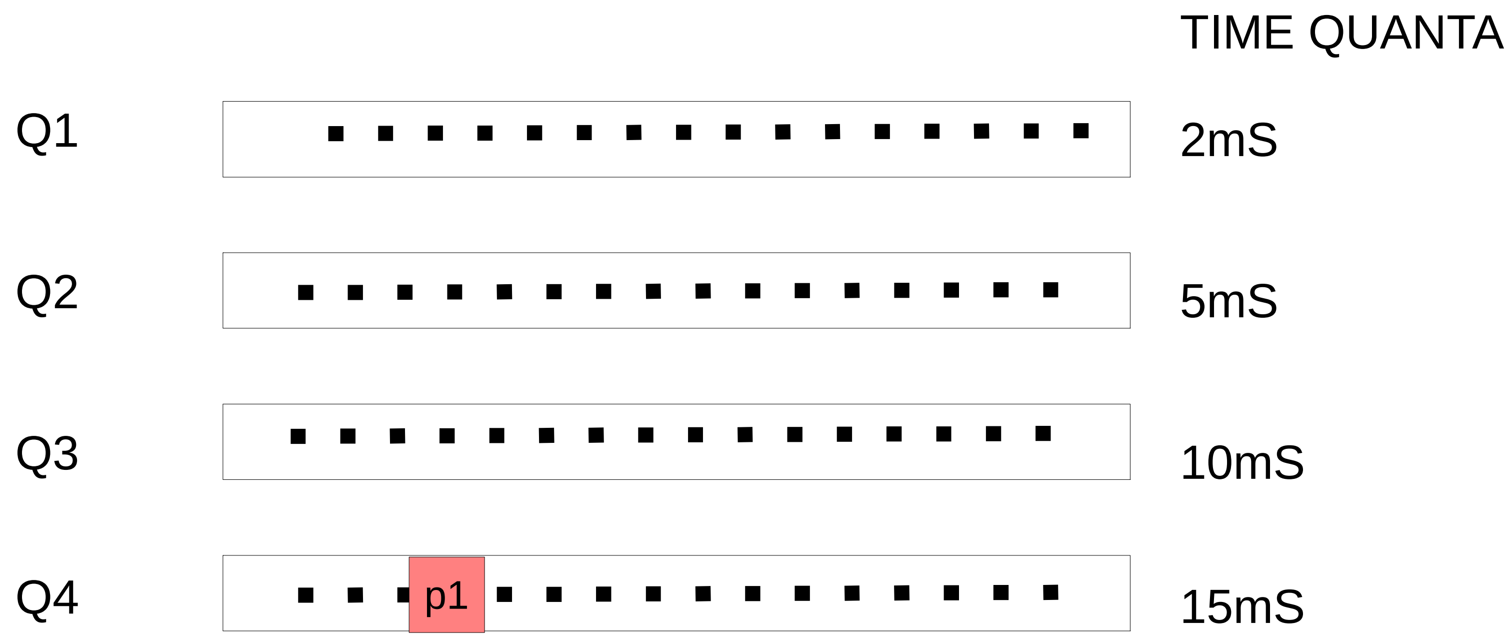
Multilevel queuing: Illustration (11)

P1 exhausts its quanta of 10 mS in Q3

- P1 is pushed down to Q4
- P1 waits from Q4 for its turn in the RR sched. Of Q4

P1 waits in Q4
For its turn in RR
Of Q4

Completed Burst Time:17mS



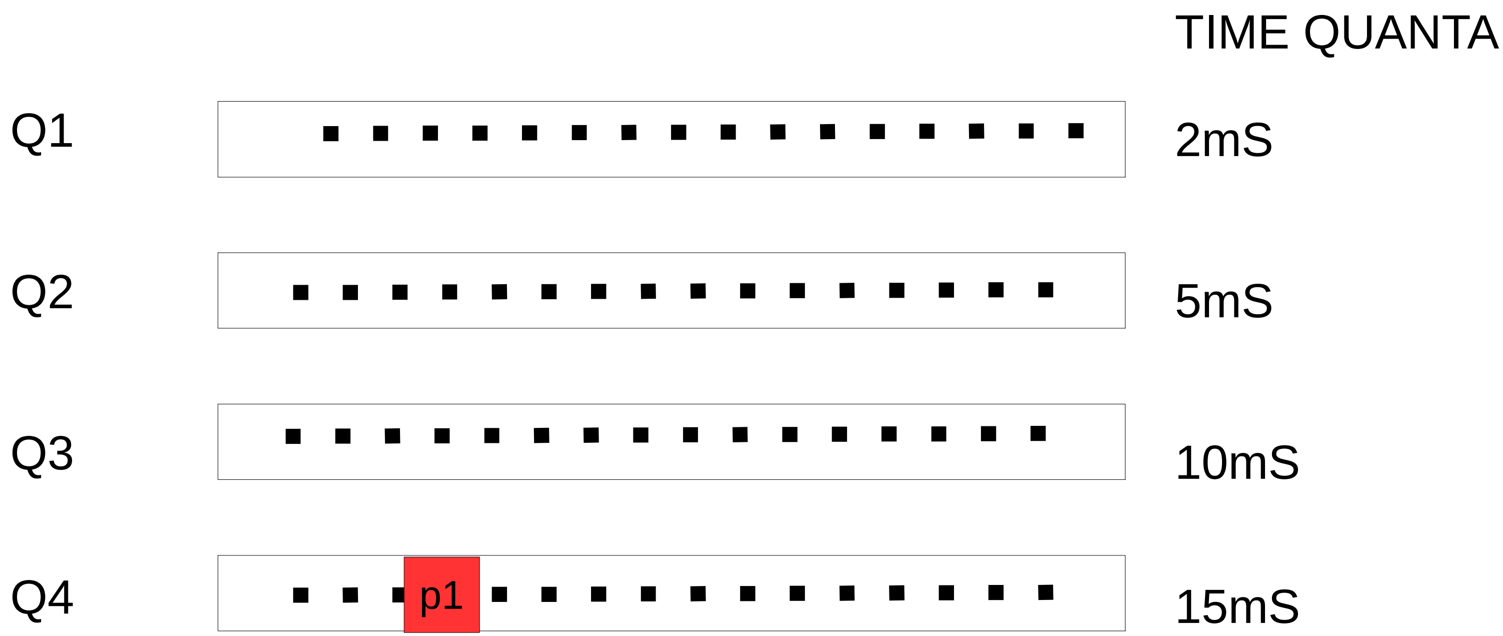
Multilevel queuing: Illustration (11)

P1 is pushed down to Q4

- P1 waits from Q4 for its turn in the RR sched. Of Q4
- RR Scheduler of Q4 when turn for P1 dispatches it, P1 execution begins

P1 gets dispatched by RR scheduler of Of Q4

Completed Burst Time:17mS



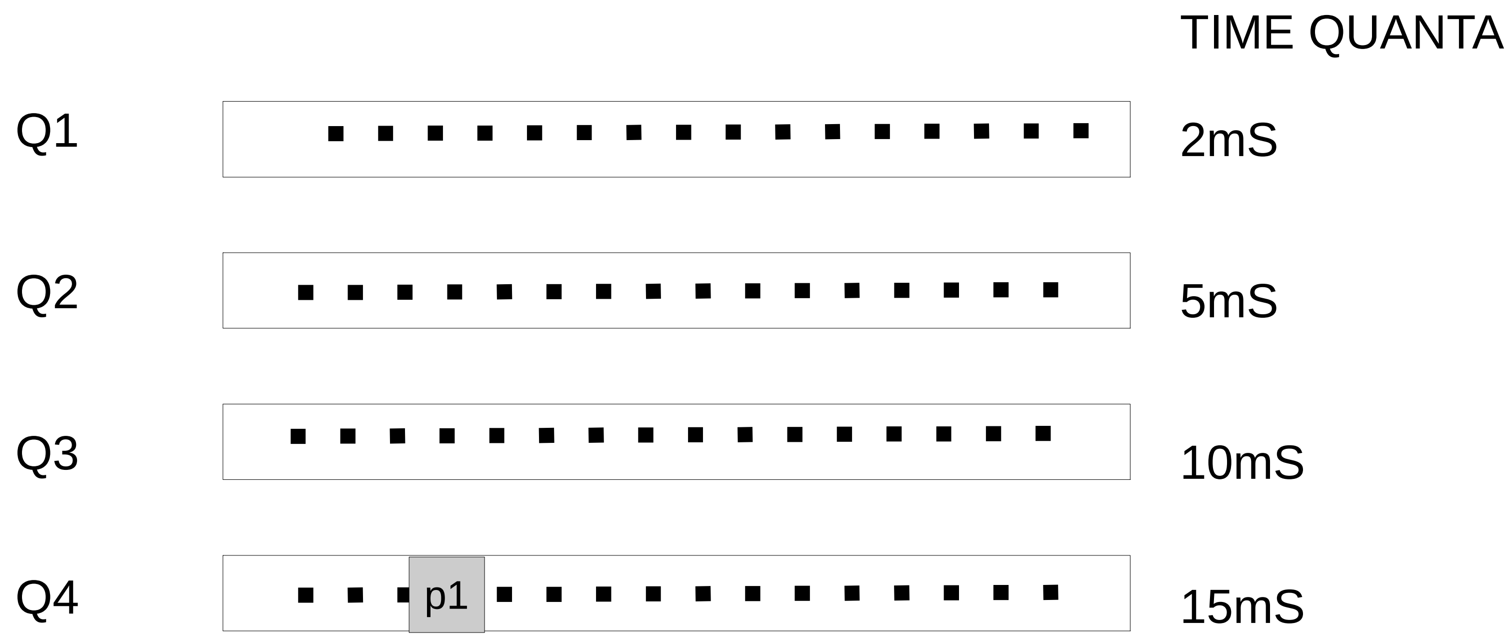
Multilevel queuing: Illustration (11)

P1 runs for another 8 mS and exits

RR Scheduler of Q4 when turn for P1 dispatches it, P1 execution begins

- Runs for another 8mS and exits voluntaritly
- The queue-level-association is done ONLY after EXIT

Completed Burst Time:17mS



Proc-Queue Association

process	Queue

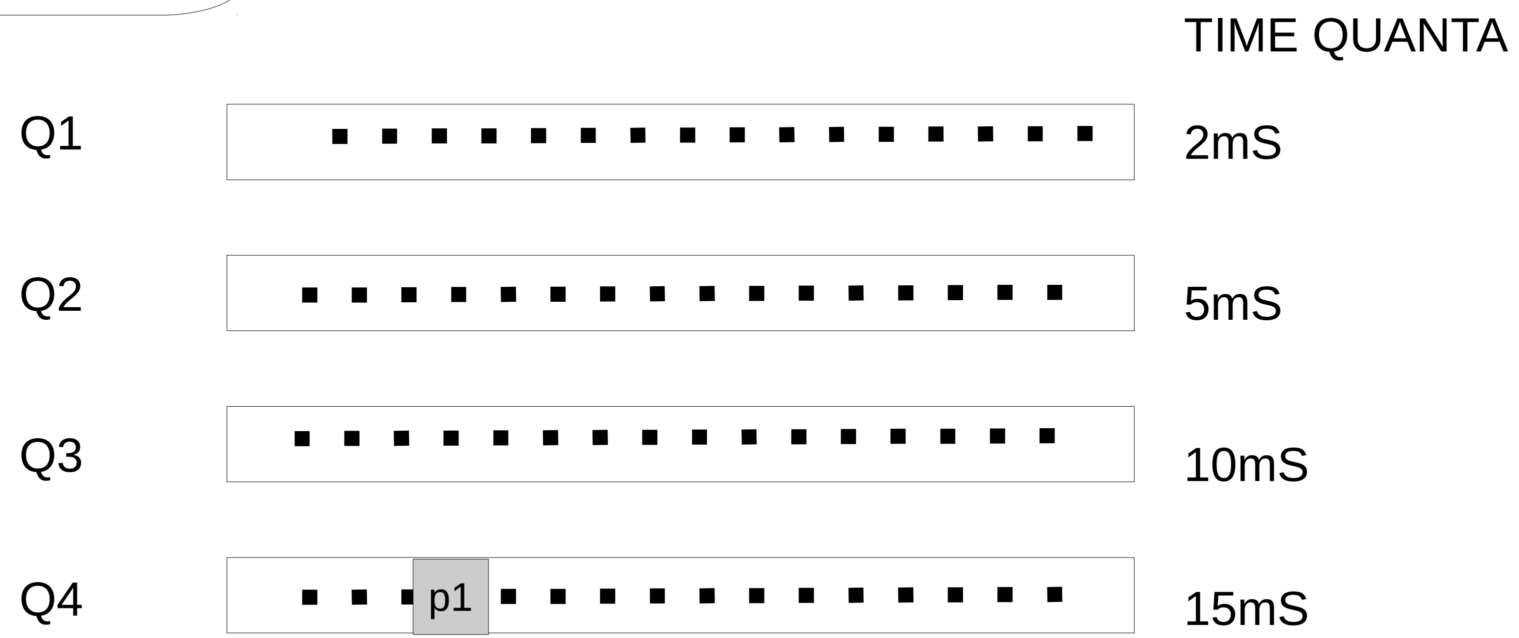
Multilevel queuing: Illustration (11)

P1 waits from Q4 for its turn in the RR sched. Of Q4

- RR Scheduler of Q4 when turn for P1 dispatches it, P1 execution begins
- Runs for another 8mS and exits voluntaritly

Completed Burst Time:17mS

Process P1 – Q4 Association is done



On exit, only Queue-mapping gets fixed

Mapping Table

process	Queue
P1	Q4

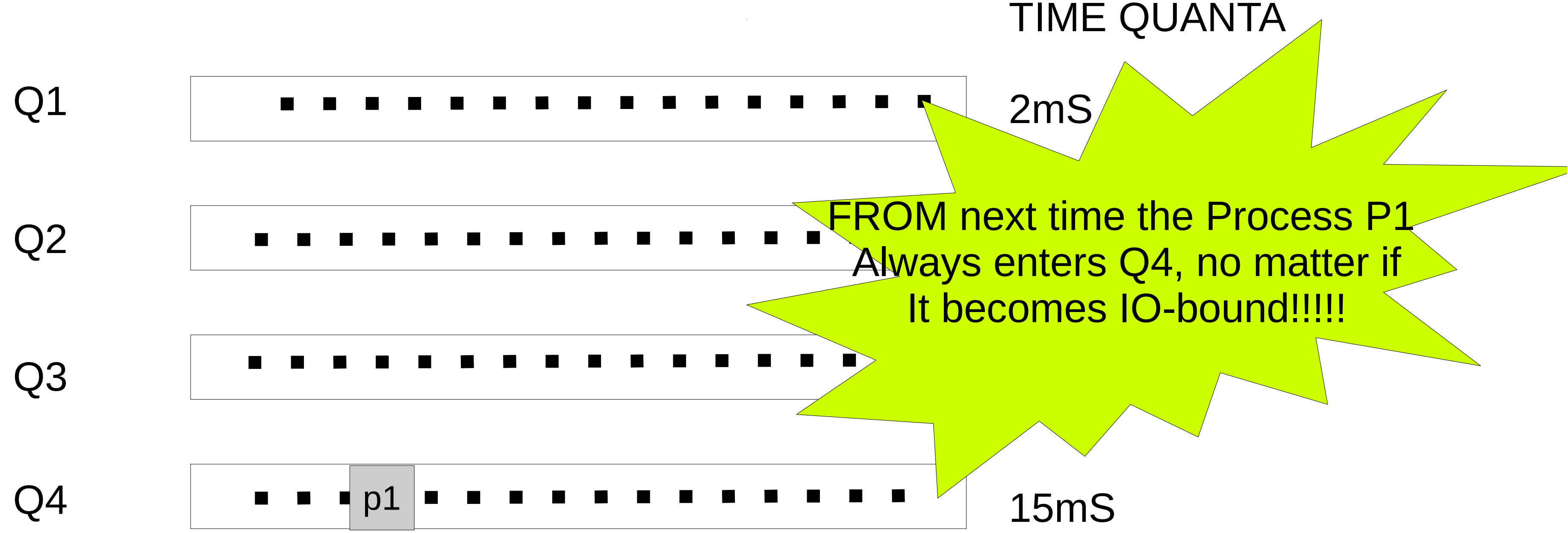
Multilevel queuing: Illustration (11)

Runs for another 8mS and exits voluntaritly

- NEXT time onwards the P1 **always** enters Q4

P1 runs for another 8 mS and exits

Completed Burst Time:17mS



Mapping Table

process	Queue
P1	Q4

Process allocation to each queue-level

- **Whenever a process becomes ready the process is allocated to a queue as following:**

1. place it in the top-level queue Q_i (initialise i=1)
2. **IF** it's (remaining) CPU-Burst within T_i

THEN It is associated to this queue (permanently)

// all processes within a queue-level get RR-scheduled

//in ALL subsequent rounds it enters Q1 always

ELSE **IF** (i<n) {

i++;//move it to next-level

move process to Q_(i) ;

GOTO Step2

}

ELSE associate it to Q_n;

3. From next time a process becoming RDY, enters its associated Q_i
4. If queues above Q_i are empty, processes in Q_i are scheduled in RR with quanta T_i

This association is permanent: a process always enters that Q

Process allocation to each queue-level

- Whenever a process becomes ready the process is allocated to a queue as following:

1. **place it in the top-level queue Q_i (initialise i=1)**

2. **IF** it's (remaining) CPU-Burst within T_i

THEN It is associated to this queue (permanently)

// all processes within a queue-level get RR-scheduled

//in ALL subsequent rounds it enters Q1 always

ELSE **IF** (i<n) {

i++;//move it to next-level

move process to Q_(i) ;

GOTO Step2

}

ELSE associate it to Q_n;

3. From next time a process becoming RDY, enters its associated Q_i
4. If queues above Q_i are empty, processes in Q_i are scheduled in RR with quanta T_i

This association is permanent: a process always enters that Q

Process allocation to each queue-level

- Whenever a process becomes ready the process is allocated to a queue as following:

    1. place it in the top-level queue Q_i (initialise i=1)
    2. **IF** it's (remaining) CPU-Burst within T_i

        THEN It is associated to this queue (permanently)

        // all processes within a queue-level get RR-scheduled

        //in ALL subsequent rounds it enters Q1 always

        ELSE **IF** (i<n) {

        i++;//move it to next-level

        move process to Q_(i) ;

        GOTO Step2

        }

        ELSE associate it to Q_n;

    3. From next time a process becoming RDY, enters its associated Q_i
    4. If queues above Q_i are empty, processes in Q_i are scheduled in RR with quanta T_i

This association is permanent: a process always enters that Q

Process allocation to each queue-level

- Whenever a process becomes ready the process is allocated to a queue as following:

1. place it in the top-level queue Q_i (initialise i=1)
2. **IF** it's (remaining) CPU-Burst within T_i

THEN It is associated to this queue (permanently)

// all processes within a queue-level get RR-scheduled

//in ALL subsequent rounds it enters Q1 always

ELSE **IF** (i<n) {

i++;//move it to next-level

move process to Q_(i) ;

GOTO Step2

}

ELSE associate it to Q_n;

3. From next time a process becoming RDY, enters its associated Q_i
4. If queues above Q_i are empty, processes in Q_i are scheduled in RR with quanta T_i

This association is permanent: a process always enters that Q

Process allocation to each queue-level

- Whenever a process becomes ready the process is allocated to a queue as following:

1. place it in the top-level queue Q_i (initialise i=1)
2. **IF** it's (remaining) CPU-Burst within T_i

THEN It is associated to this queue (permanently)

// all processes within a queue-level get RR-scheduled

//in ALL subsequent rounds it enters Q1 always

ELSE **IF** (i<n) {

i++;//move it to next-level

move process to Q_(i) ;

GOTO Step2

}

ELSE associate it to Q_n;

3. From next time a process becoming RDY, enters its associated Q_i
4. If queues above Q_i are empty, processes in Q_i are scheduled in RR with quanta T_i

This association is permanent: a process always enters that Q

Process allocation to each queue-level

- Whenever a process becomes ready the process is allocated to a queue as following:

1. place it in the top-level queue Q_i (initialise i=1)
2. **IF** it's (remaining) CPU-Burst within T_i

THEN It is associated to this queue (permanently)

// all processes within a queue-level get RR-scheduled

//in ALL subsequent rounds it enters Q1 always

ELSE **IF** (i<n) {

i++;//move it to next-level

move process to Q_(i) ;

GOTO Step2

}

ELSE associate it to Q_n;

3. From next time a process becoming RDY, enters its associated Q_i
4. If queues above Q_i are empty, processes in Q_i are scheduled in RR with quanta T_i

This association is permanent: a process always enters that Q

Draw backs of static multi-level-Q RR sched.

- ML-RR scheduler is
  - insensitive to processes that change their nature from round to round
    - i.e. IO-bound becoming CPU-bound or vice versa
    - Prediction cannot be so quick
- Solution: Make association NOT fixed for more than one round
  - i.e. dynamic mapping
  - Consider the association only for next round, then evaluate again and decide the future association
  - Consequently
    - IF a process shifts from IO-bound to CPU-bound we migrate it downwards
    - ELSE IF a process shifts from CPU-bound to IO -bound we migrate it upwards sadf
- In dynamic version, we can have:
  - Feed-back multi-level queue RR scheduling

The whole motive is to improve efficiency from wait-time, TAT, CPU-utilisation, etc.

Feed-back Multi-level RR Scheduling

- Post first-time association:

    1. CPU-burst (CBT_p) of each process P_p finishing in Q_i is recorded // I is the queue-level

    2. IF CBT_p in CI[j], for some level 'j' above level i:

        THEN push the process to that level 'j' upwards //FEED-BACK

        ELSE IF ExecutionTime of p 'ET_p' exceeds T_i

            THEN move p to queue one-level downwards //FEED-FORWARD /

            // ELSE otherwise the process sticks to the same queue

THEREFORE, any process on changing its nature is sensed by this RR ML-scheduling THUS, exploits this change by juggling the processes up or down!

Accordingly we have connections between the heads and tails of various levels.

But, a caution: more context switches means reduction in efficiency, (RECALL the tutorial problem on GANTT CHARTS), more context switches means reduced CPU-utilisation, and increased wait time and TAT !!!!!!

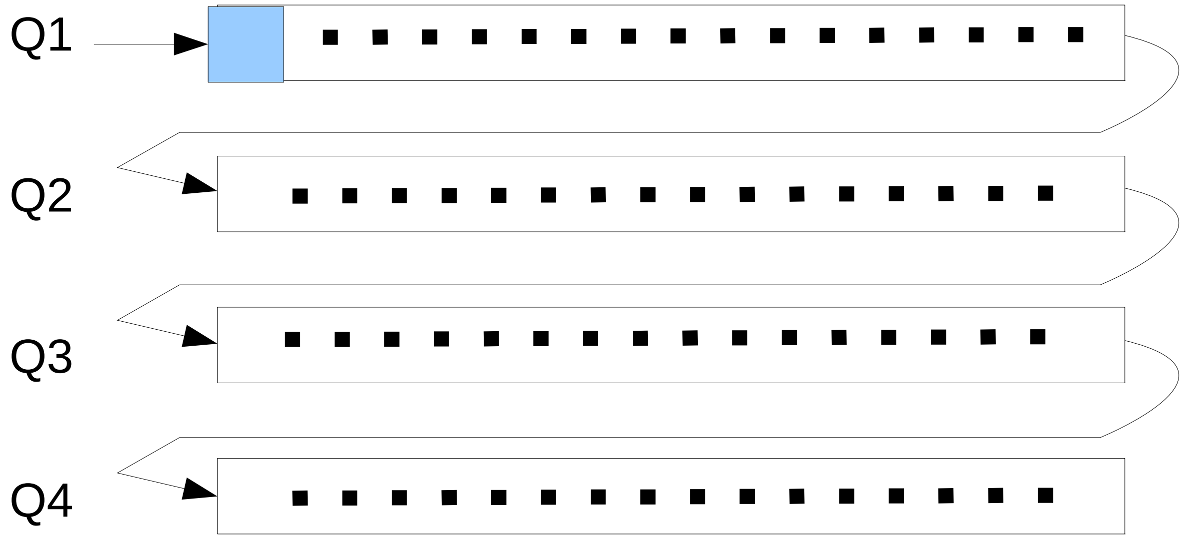
Such checking of times and comparison is costly, we can't afford to loose the efficiency by investing more processor time on scheduling!!!!!

Feed-back Multi-level RR Scheduling

> any process on changing its nature is sensed by this RR ML-scheduling

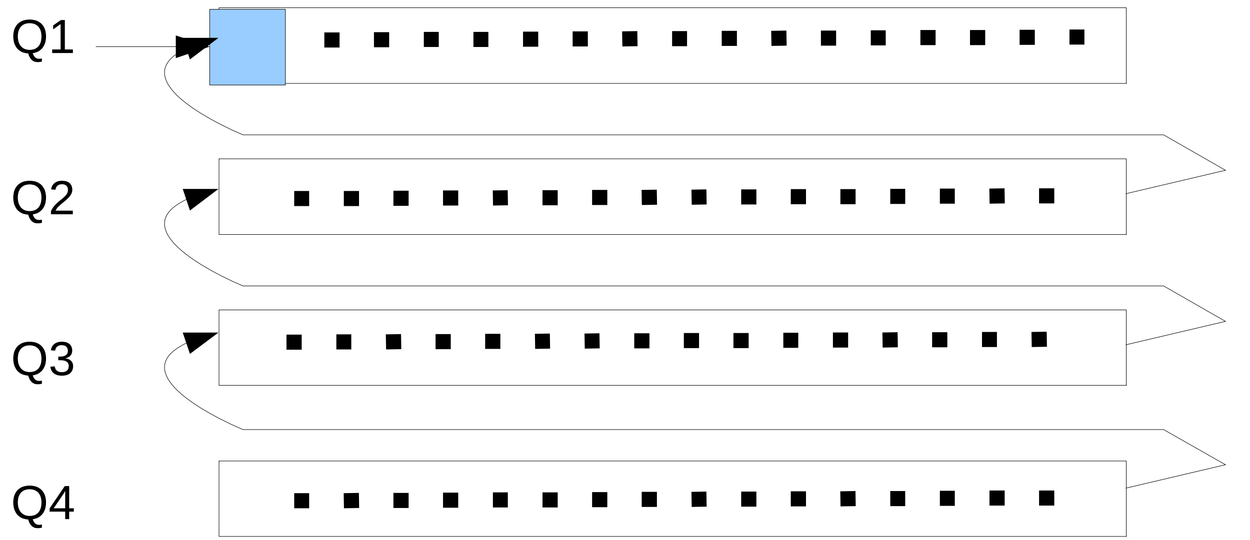
> THUS juggling the processes up or down accordingly improves efficiency

> Accordingly we have connections between the heads and tails of various levels.



Feed-back Multi-level RR Scheduling

> Feed-forward: IO-bound becoming CPU-bound
> Several configs are possible connecting tails of lower levels to heads of multiple levels up



Feed-back Multi-level RR Scheduling CAUTION

- But, a caution:
  - more context switches means reduction in efficiency, (RECALL the tutorial problem on GANTT CHARTS), more context switches means reduced CPU-utilisation, and increased wait time and TAT !!!!!!

- Such checking of times and comparison is costly, we can't afford to loose the efficiency by investing more processor time on scheduling!!!!!