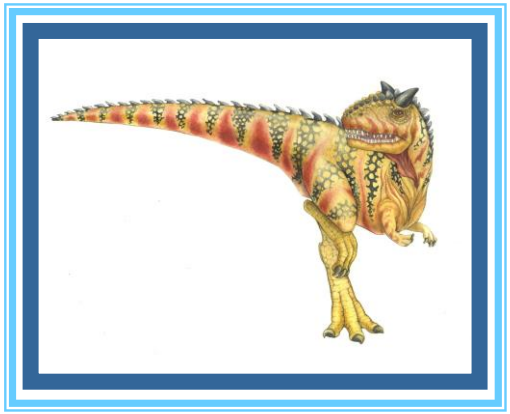
Chapter 5: CPU Scheduling



Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

CPU Scheduler

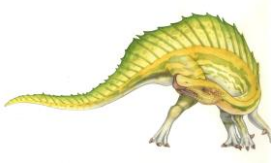
- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output



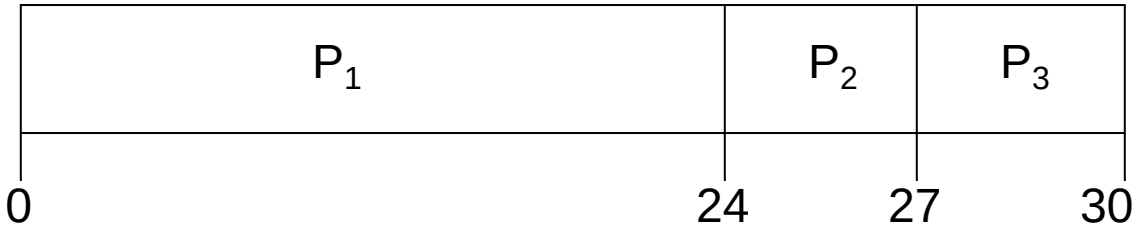
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



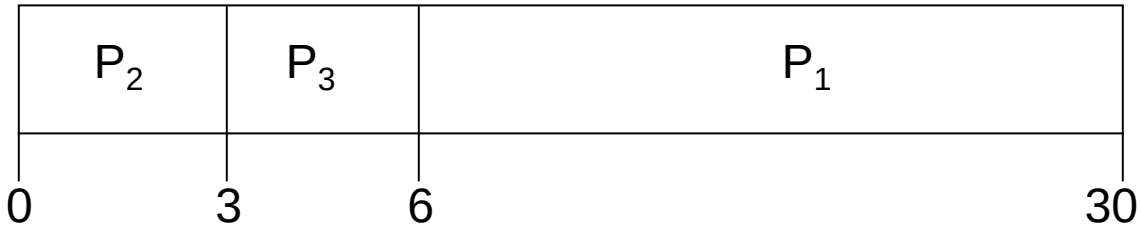
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

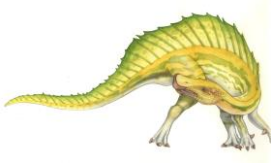


- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case

- convoy effect:
- I/0 processes end up waiting in the ready queue until the CPU-bound process is done; as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

Shortest-Job-First (SJF) Scheduling

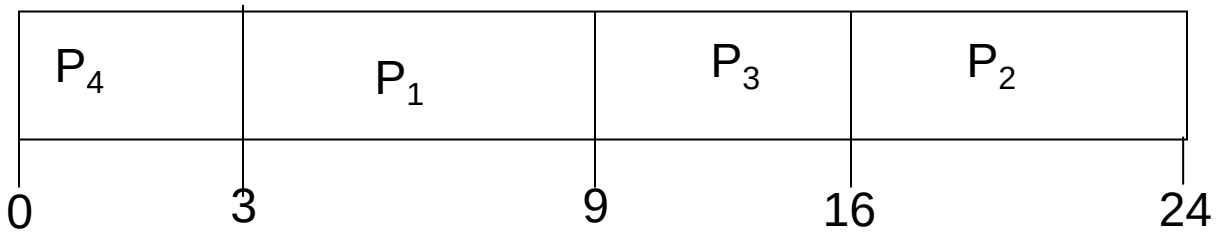
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- SJF-preemptive
- SJF-Non-preemptive
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request.



Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

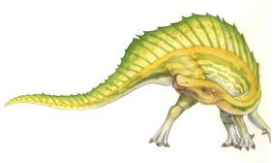


- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

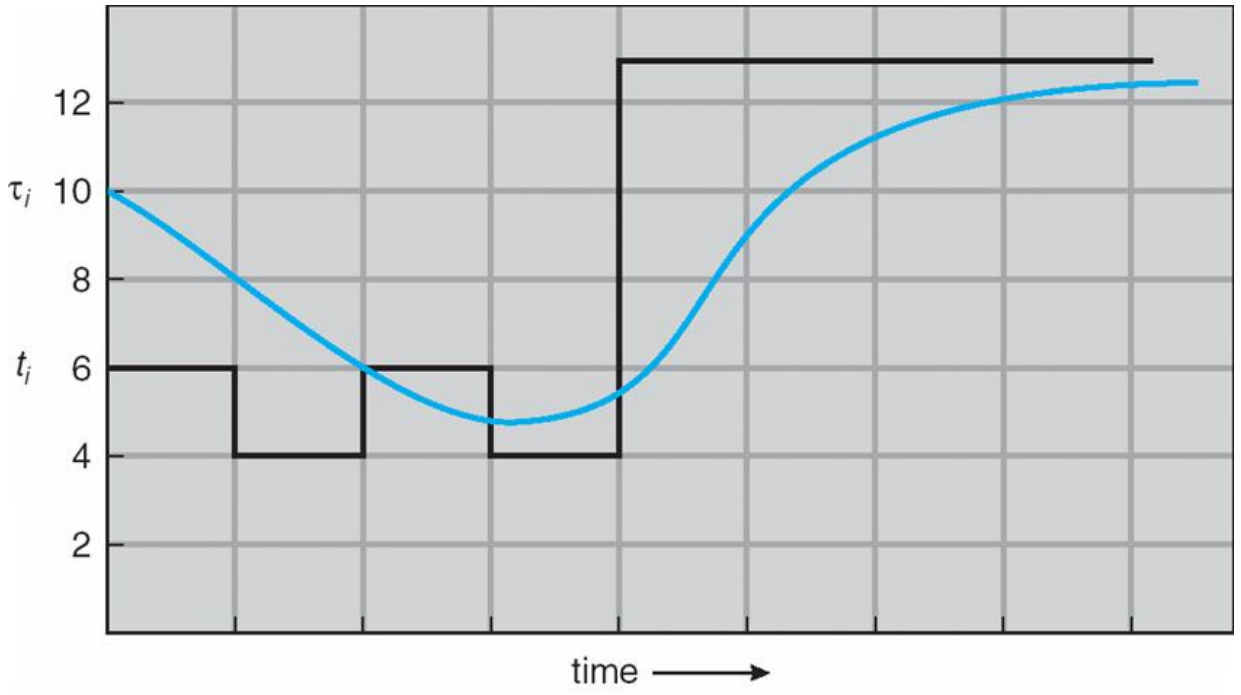
Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

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



Prediction of the Length of the Next CPU Burst



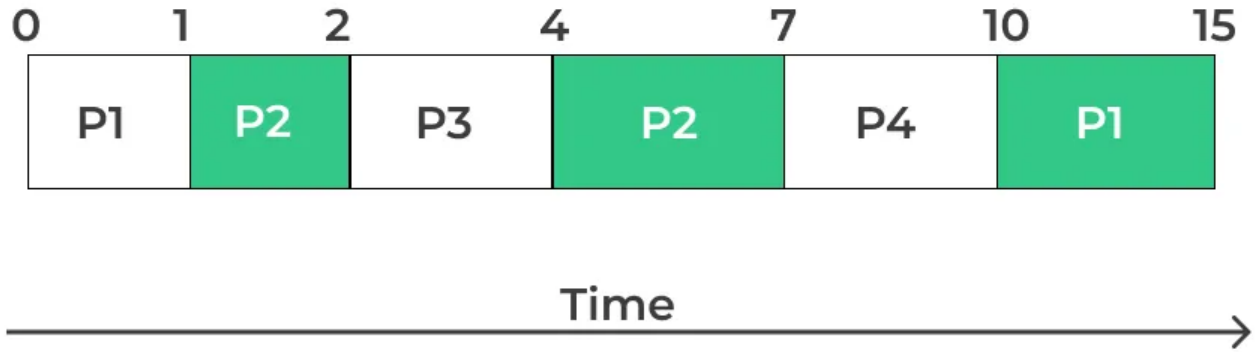
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

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

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

Shortest Job First (SJF) Preemptive Algorithm



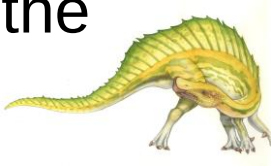
Process	Arrival Time	CPU Burst Time
P1	0	6
P2	1	4
P3	2	2
P4	3	3

Shortest Remaining Time First (Preemptive SJF) Scheduling Algorithm

Process	Burst Time	Arrival Time
P1	6 ms	2 ms
P2	2 ms	5 ms
P3	8 ms	1 ms
P4	3 ms	0 ms

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)
  - Preemptive
  - Non-preemptive
- Note that SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process



Now we calculate **Turn Around Time (TAT)** and **Waiting Time (WT)** using the following formula:

TAT = CT - AT, WT = TAT – BT

And

Response Time (RT) = FR (First Response) - AR (Arrival Time)

Note: Here waiting time and response time same because it is non-preemptive.

For every non-preemptive scheduling waiting time = response time.

Disadvantages:

A **major problem** with priority scheduling algorithms is indefinite blocking, or starvation. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Aging:

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time.

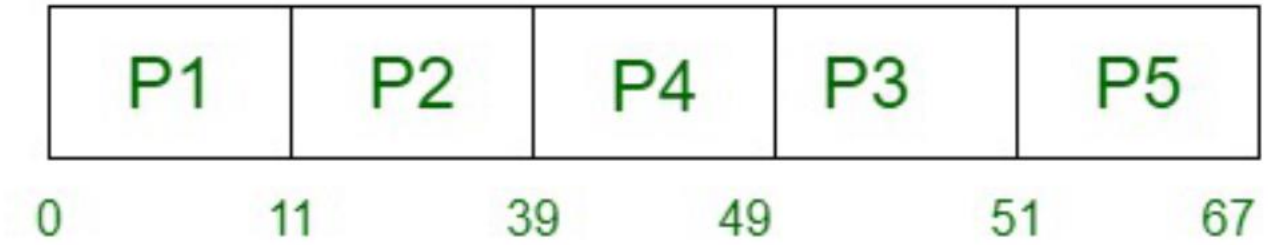
For example

If priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority -127 process to age to a priority -0 process.

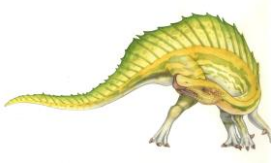
Example Problem – Non preemptive

Process	Arrival Time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

Example Problem – Non preemptive



#	AT	BT	CT	TAT	WT	RT
1	0	11	11	11	0	0
2	5	28	39	34	6	6
3	12	2	51	39	37	37
4	2	10	49	47	37	37
5	9	16	67	58	42	42
			AVG	**37.8**	**24.4**	



Example Problem – preemptive

PROCESS	ARRIVAL TIME	BURST TIME		PRIORITY
		TOTAL	REMAINING	
P1	0	4	4	4
P2	1	3	3	3
P3	3	4	4	1
P4	6	2	2	5
P5	8	4	4	2

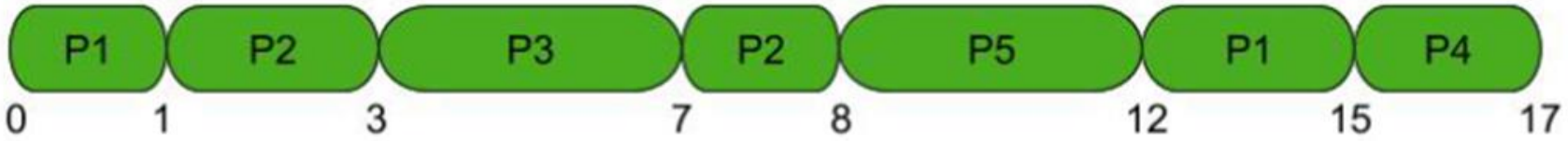
GANTT CHART:

Preemptive Priority Scheduling Algorithm at time = 0

As higher priority processes keep on adding, prior processes get preempted and get the CPU later on.

Example Problem – preemptive

GANTT CHART:



# (PRIORITY)	AT	BT	CT	TAT	WT	RT
1 (4)	0	4	15	15	11	0
2 (3)	1	3	8	7	4	0
3 (1)	3	4	7	4	0	0
4 (5)	6	2	17	11	9	9
5 (2)	8	4	12	4	0	0
			AVG	**8.2**	**4.8**	

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Performance
  - *TS* large $\Rightarrow$ FIFO
  - *TS* small $\Rightarrow$ may hit the context switch wall, overhead is too high.

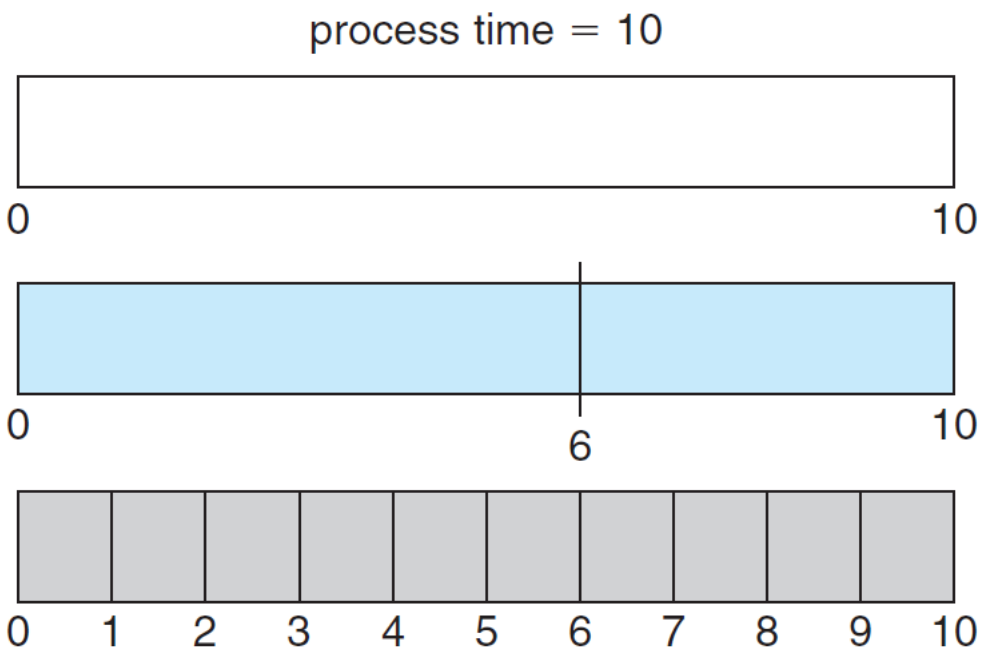


Figure 6.4 How a smaller time quantum increases context switches.

Four jobs to be executed on a single processor system arrive at time 0 in the order A, B, C, D. Their burst CPU time requirements are 4, 1, 8, 1 time units respectively. The completion time of A under round robin scheduling with time slice of one time unit is-

1. 10
2. 4
3. 8
4. 9

Process Id	Arrival time	Burst time
A	0	4
B	0	1
C	0	8
D	0	1

Gantt chart-

Ready Queue-

C, A, C, A, C, A, D, C, B, A

Gantt Chart

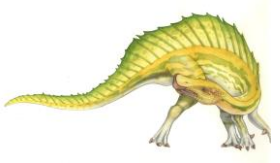
Process	BT	CT	TAT=CT-AT	WT=TAT-BT
A	4	9	9	5
B	1	2	2	1
C	8	14	14	6
D	1	4	4	3
		AVG	**7.25**	**3.75**

RR with different AT

Consider the set of 5 processes whose arrival time and burst time are given below-

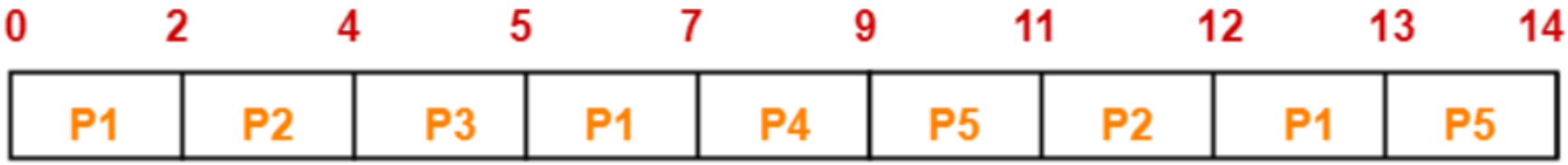
Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turn around time.



Ready Queue-

P5, P1, P2, P5, P4, P1, P3, P2, P1



Gantt Chart

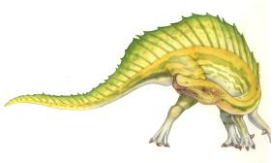
Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

Now,

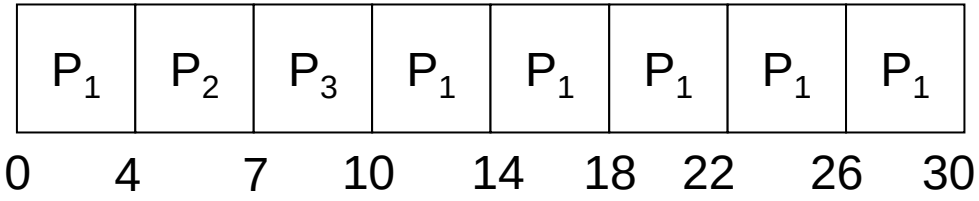
- Average Turn Around time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit



Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

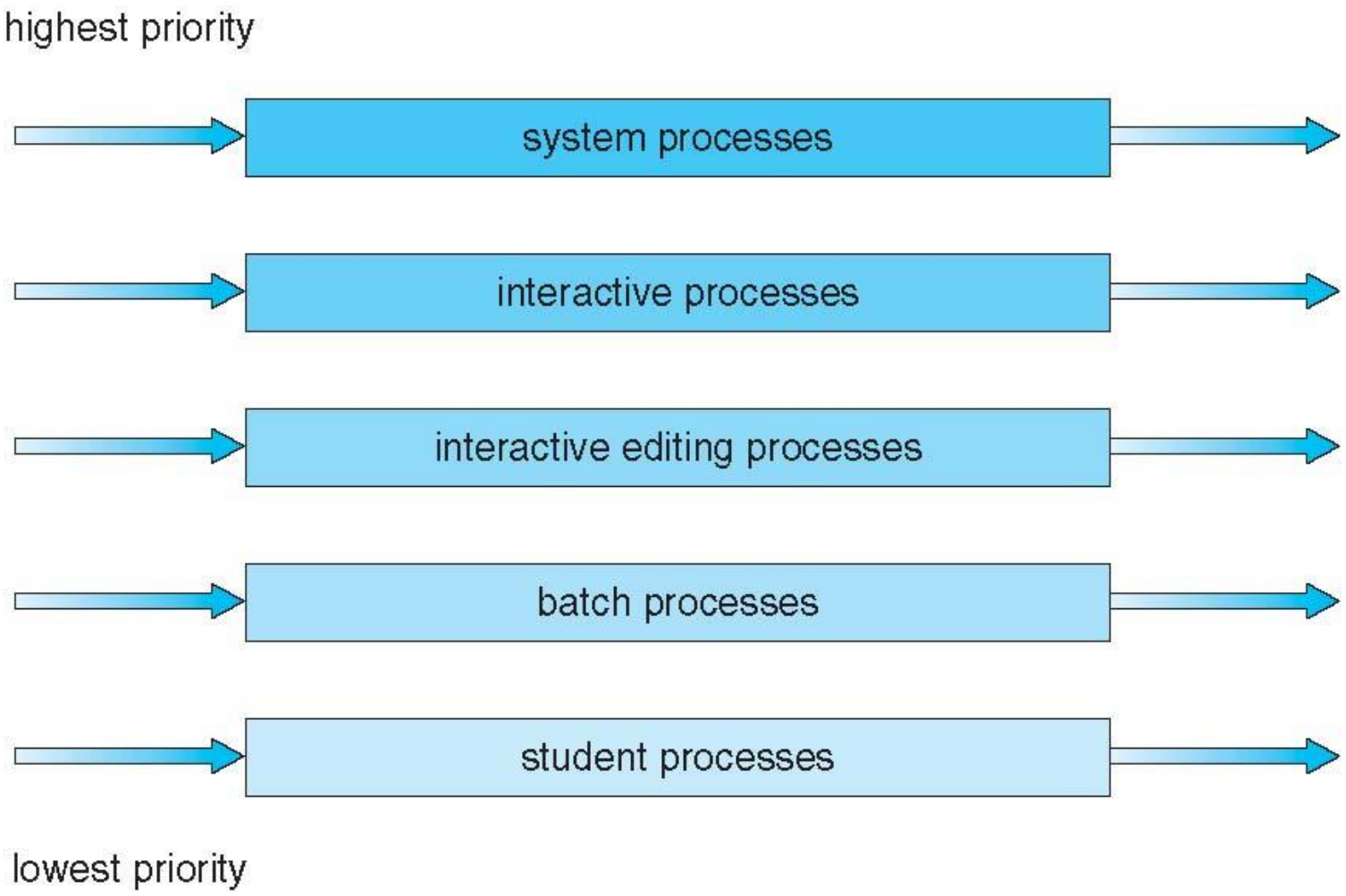
- The Gantt chart is:



Multilevel Queue

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done 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



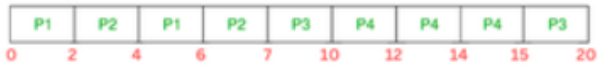
Example Problem:

Consider the below table of four processes under Multilevel queue scheduling. Queue number denotes the queue of the process.

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

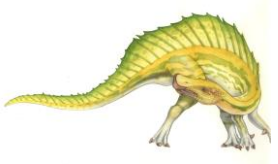
Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.

Below is the **Gantt chart** of the problem:



Working:

- At starting, both queues have process so process in queue 1 (P1, P2) runs first (because of higher priority) in the round-robin fashion and completes after 7 units
- Then process in queue 2 (P3) starts running (as there is no process in queue 1) but while it is running P4 comes in queue 1 and interrupts P3 and start running for 5 seconds and
- After its completion P3 takes the CPU and completes its execution.



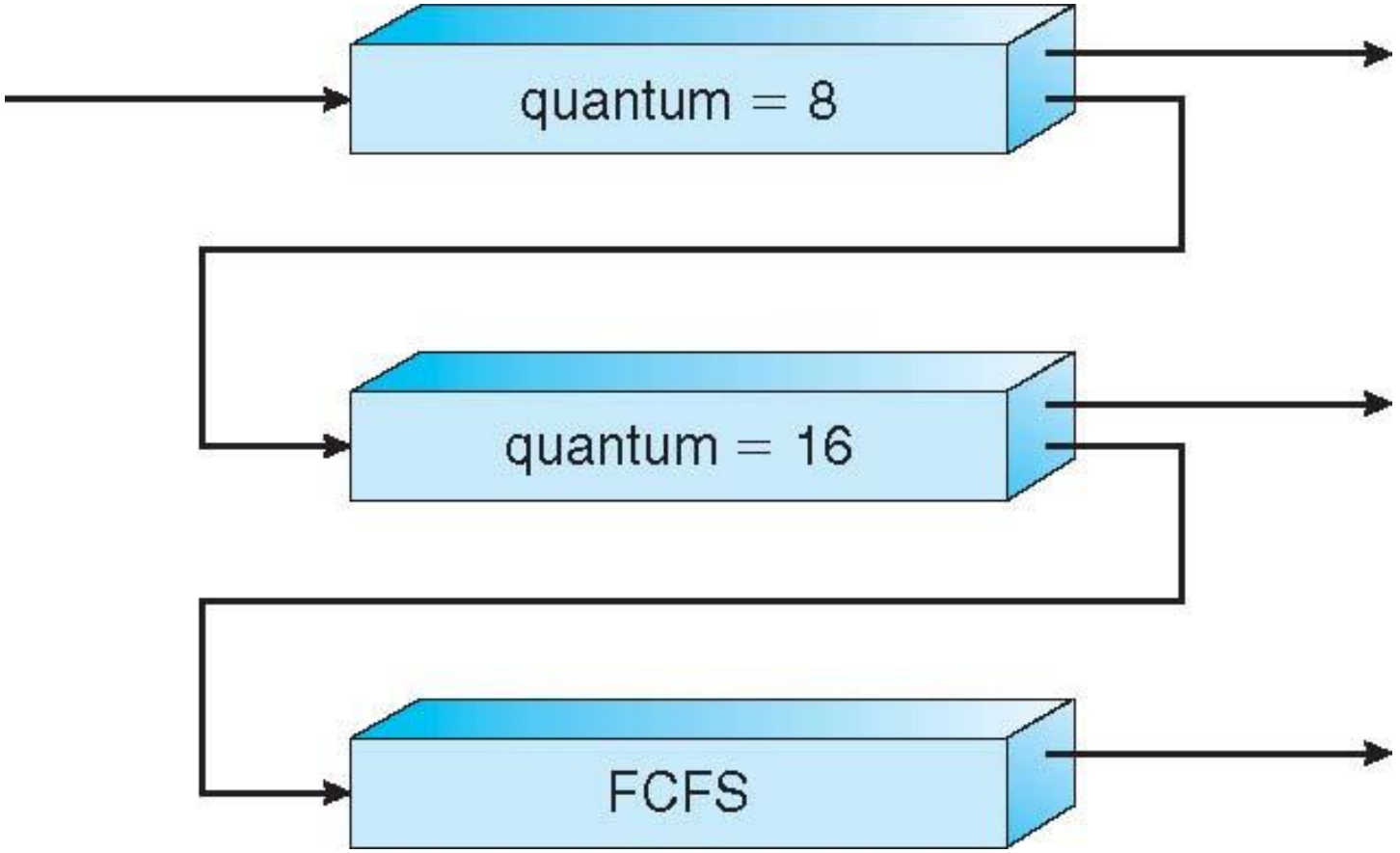
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- 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 RR. 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 RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Example: Consider a system that has a CPU-bound process, which requires a burst time of 40 seconds. The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum ‘2’ seconds and in each level it is incremented by ‘5’ seconds. Then how many times the process will be interrupted and in which queue the process will terminate the execution?

Solution:

Process P needs 40 Seconds for total execution.
At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.
At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.
At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.
At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.
At Queue 5 it executes for 2 seconds and then it completes.
Hence the process is interrupted 4 times and completed on queue 5.

End of Chapter 5