CPU SCHEDULING

Operating Systems

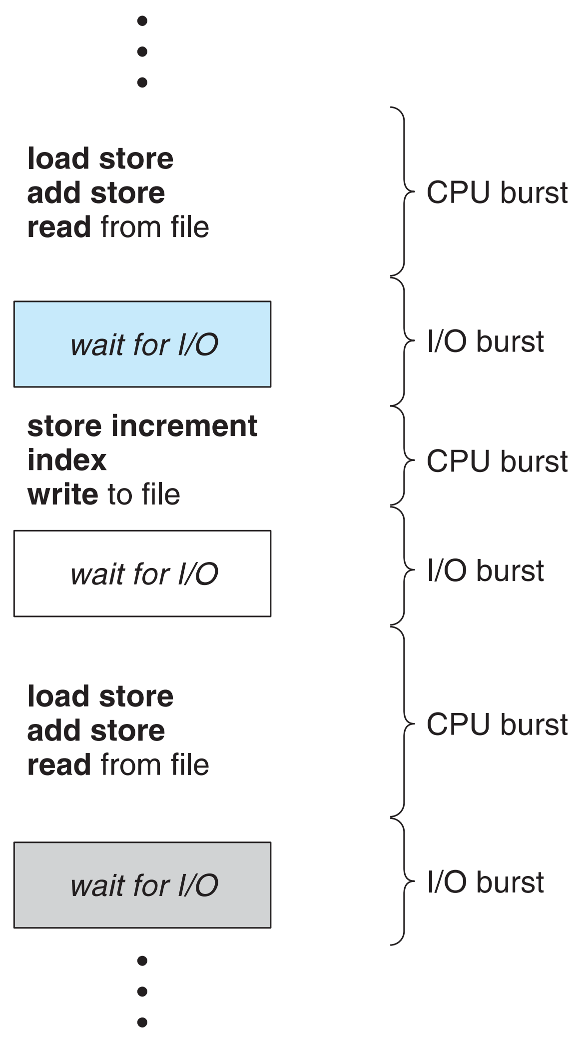
(CS-220)
SPRING 2020
FAST NUCES

anaum.hamid@nu.edu.pk

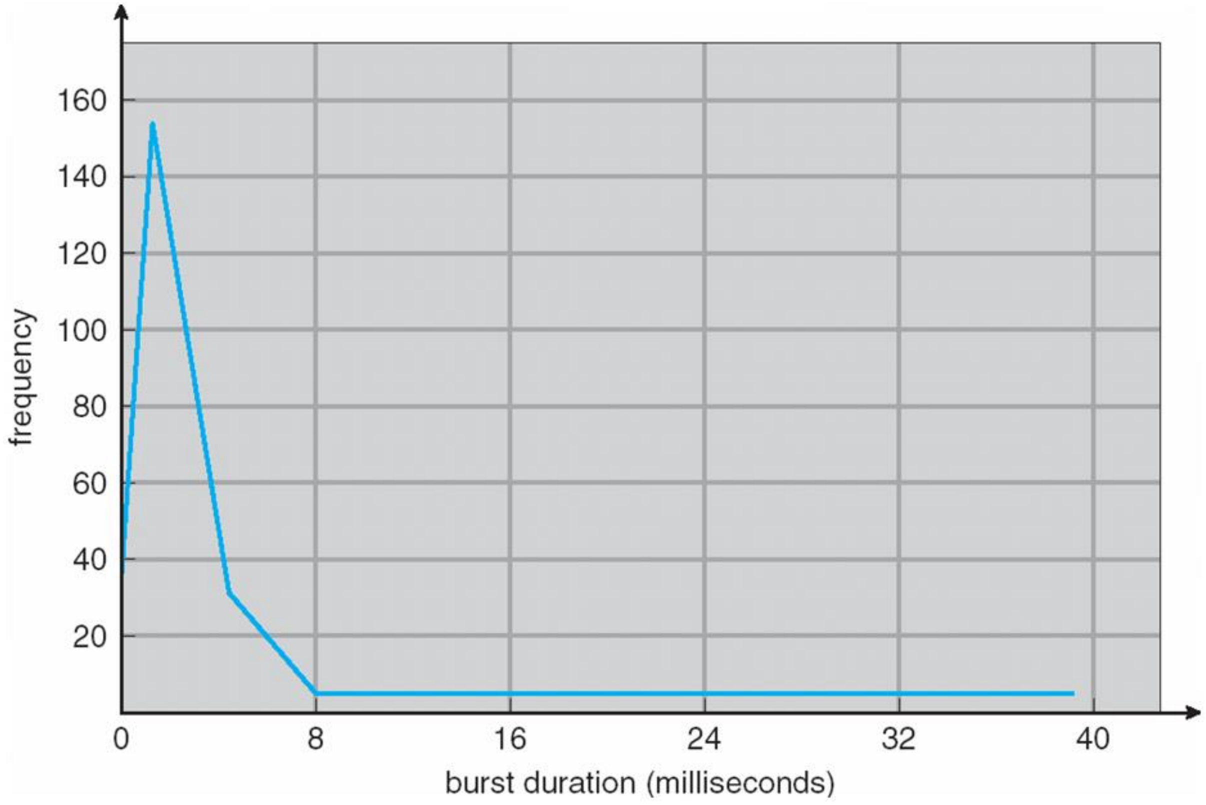
COURSE SUPERVISOR: ANAUM HAMID

CPU–I/O BURST CYCLE

- All processes alternate between two states in a continuing ***cycle;*** A CPU burst of performing calculations, and An I/O burst, waiting for data transfer in or out of the system.
- Process execution consists of a **cycle** of CPU execution and I/O wait.
- **CPU burst** followed by **I/O burst.**
- CPU burst distribution is of main concern



HISTOGRAM OF CPU-BURST TIMES



CPU bursts vary from process to process, and from program to program, but an extensive study shows frequency patterns similar to that shown in Figure.

CPU SCHEDULER

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  2. Switches from waiting to ready.
  3. Terminates.
- Scheduling under 1 and 4 is **Non-Preemptive -** A new process must be selected.
- All other scheduling is **Preemptive -** There is a choice - To either continue running the current process or select a different one.
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

PRE-EMPTIVE VS NON-PREEMPTIVE

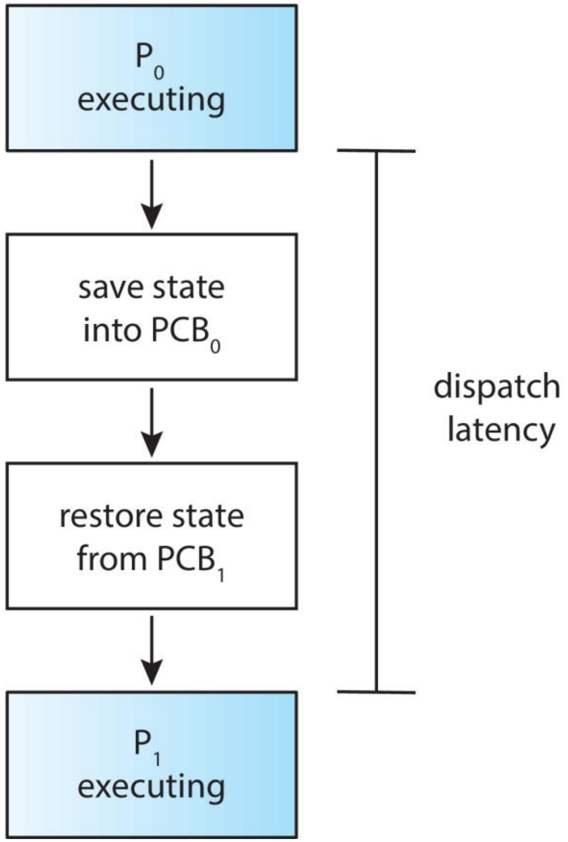
1. The basic difference between preemptive and non-preemptive scheduling is that in preemptive scheduling the CPU is allocated to the processes for the **limited** time. While in Non-preemptive scheduling, the CPU is allocated to the process till it **terminates** or switches to **waiting state**.
2. The executing process in preemptive scheduling is **interrupted** in the middle of execution whereas, the executing process in non-preemptive scheduling is **not interrupted** in the middle of execution.
3. Preemptive Scheduling has the **overhead** of switching the process from ready state to running state, vise-verse, and maintaining the ready queue. On the other hands, non-preemptive scheduling has **no overhead** of switching the process from running state to ready state.
4. In preemptive scheduling, if a process with high priority frequently arrives in the ready queue then the process with low priority have to wait for a long, and it may have to starve. On the other hands, in the non-preemptive scheduling, if CPU is allocated to the process with larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling is quite **flexible** because the critical processes are allowed to access CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is **rigid** as even if a critical process enters the ready queue the process running CPU is not disturbed.

DISPATCHER

Dispatcher module gives control of the CPU to the process selected by the short-term 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 – Ideally the CPU would be busy 100% of the time, to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput –Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.

Turnaround time – Amount of time required for a particular process to complete, from submission time to completion. (Wall clock time.)

Waiting time – Amount of time processes spend in the ready queue waiting their turn to get on the CPU

Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

SCHEDULING ALGORITHM OPTIMIZATION CRITERIA

1. Max CPU utilization
2. Max throughput
3. Min turnaround time
4. Min waiting time
5. 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 - short process behind long process. All other short processes are left behind a big long process to be done by CPU.

- Consider one CPU-bound and many I/O-bound processes

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

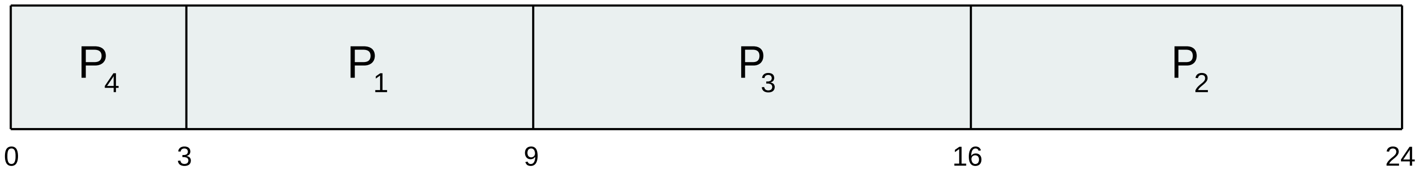
SJF is optimal – gives minimum average waiting time for a given set of processes

- The difficulty is knowing the length of the next CPU request
- Could ask the user

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 – should be similar to the previous one

- Then pick process with shortest predicted next CPU burst

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. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n .$

Commonly, α set to ½

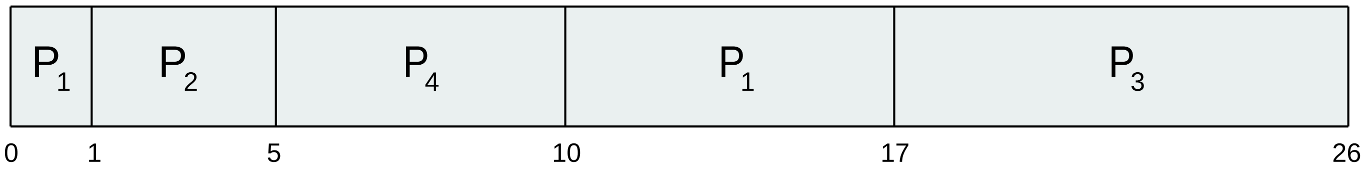
Preemptive version called **shortest-remaining-time-first**

EXAMPLE OF SHORTEST-REMAINING-TIME-FIRST

Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>*Arrival* Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Preemptive SJF Gantt Chart



Waiting Time = (Total waiting time - #of MS executing earlier – Arrival Time)

WT = (10-1-0)+(1-0-1)+(17-0-2)+(5-0-3) = 26/4 = 6.5

Average waiting time = 6.5 msec

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
- Nonpreemptive

SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

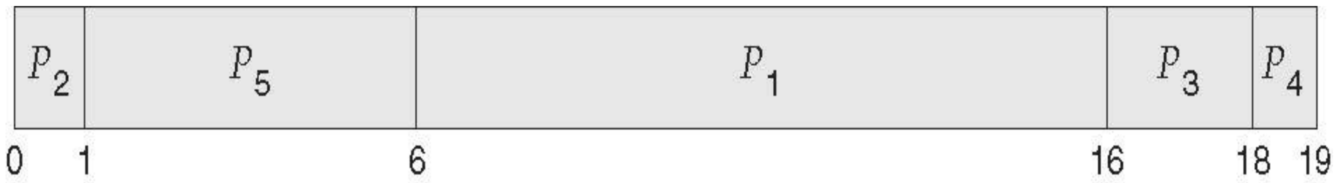
Problem ≡ **Starvation** – low priority processes may never execute

Solution ≡ **Aging** – as time progresses increase the priority of the process

EXAMPLE OF NON-PREEMPTIVE PRIORITY SCHEDULING

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Priority scheduling Gantt Chart



WT = (6+0+16+18+1) = 41/5 = 8.2

Average waiting time = 8.2 msec

EXAMPLE OF PREEMPTIVE PRIORITY SCHEDULING (SEE CLASS LECTURE)

ROUND ROBIN (RR)

Specially designed for timesharing/ time slicing systems.

Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

Timer interrupts every quantum to schedule next process.

Ready queue acts alike circular queue.

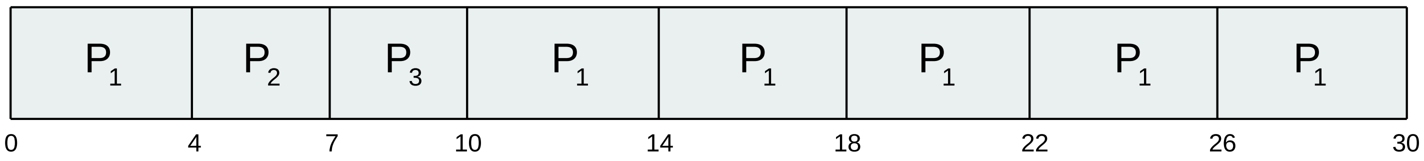
Performance

- q large $\Rightarrow$ FIFO
- q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

EXAMPLE OF RR WITH TIME QUANTUM = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:



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

q should be large compared to context switch time

q usually 10ms to 100ms, context switch < 10 usec

EXAMPLE OF ROUND ROBIN SCHEDULING (SEE CLASS LECTURE)

MULTILEVEL QUEUE

Ready queue is partitioned into separate queues, eg:

- **foreground** (interactive)
- **background** (batch)

Process permanently in each queue

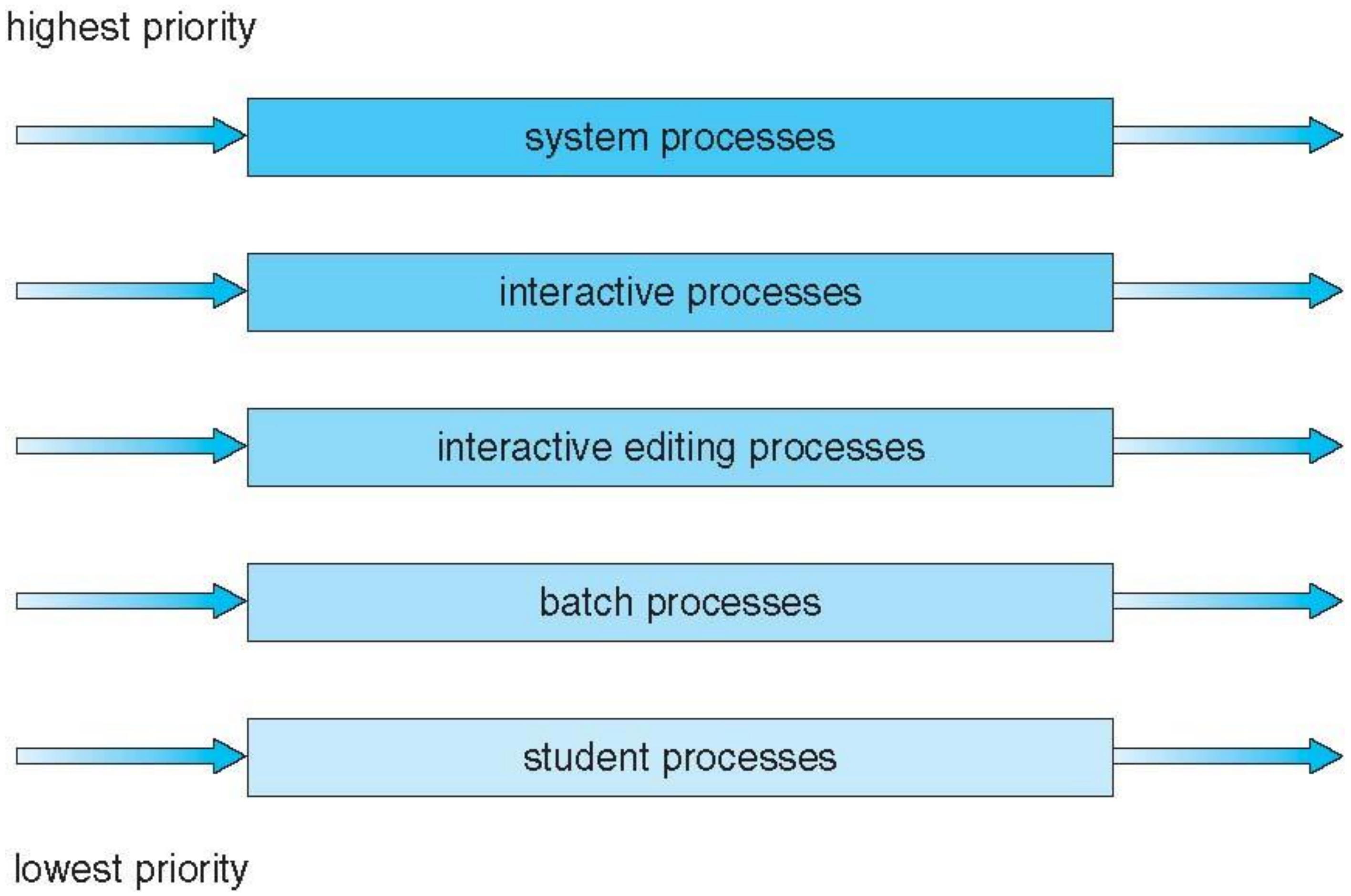
Each queue has its own scheduling algorithm:

- 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



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

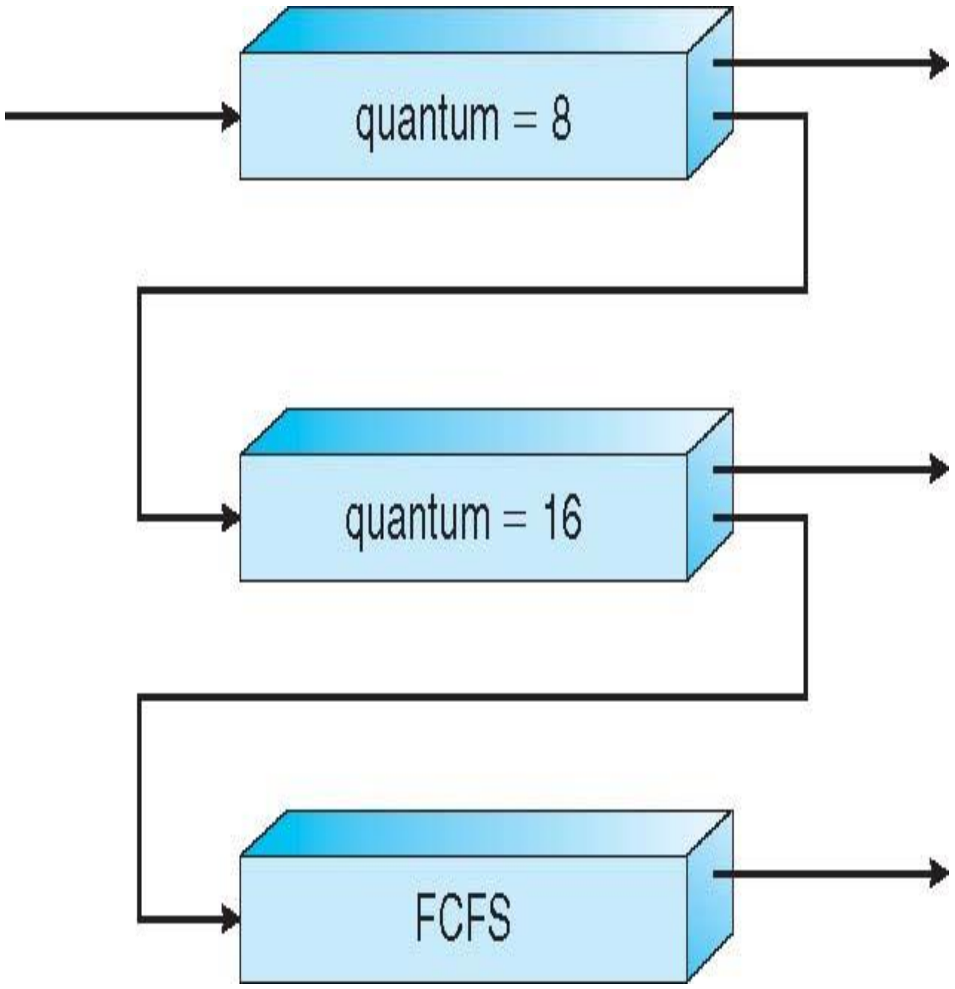
EXAMPLE OF MULTILEVEL FEEDBACK QUEUE

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 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



MULTIPLE-PROCESSOR SCHEDULING

CPU scheduling more complex when multiple CPUs are available

Homogeneous processors within a multiprocessor

Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing

Symmetric multiprocessing (**SMP**) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes

- Currently, most common

PROCESSOR AFFINITY

Processor affinity – process has affinity for processor on which it is currently running

- **soft affinity**
- **hard affinity**
- Variations including **processor sets**

Soft affinity is a mechanism that keeping process on the same CPU if possible otherwise, process is migrated to another processor.

Hard affinity is a mechanism that enables processes can run only on a fixed set of one or more processor(s).

MULTIPLE-PROCESSOR SCHEDULING – LOAD BALANCING

If SMP, need to keep all CPUs loaded for efficiency

Load balancing attempts to keep workload evenly distributed

Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs

Pull migration – idle processors pulls waiting task from busy processor

REAL-TIME CPU SCHEDULING

Can present obvious challenges

Soft real-time systems – no guarantee as to when critical real-time process will be scheduled

Hard real-time systems – task must be serviced by its deadline

Two types of latencies affect performance

1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt.

1. Dispatch latency – time for schedule to take current process off CPU and switch to another