

CS420: Operating Systems

CPU Scheduling

James Moscola

Department of Engineering & Computer Science

York College of Pennsylvania

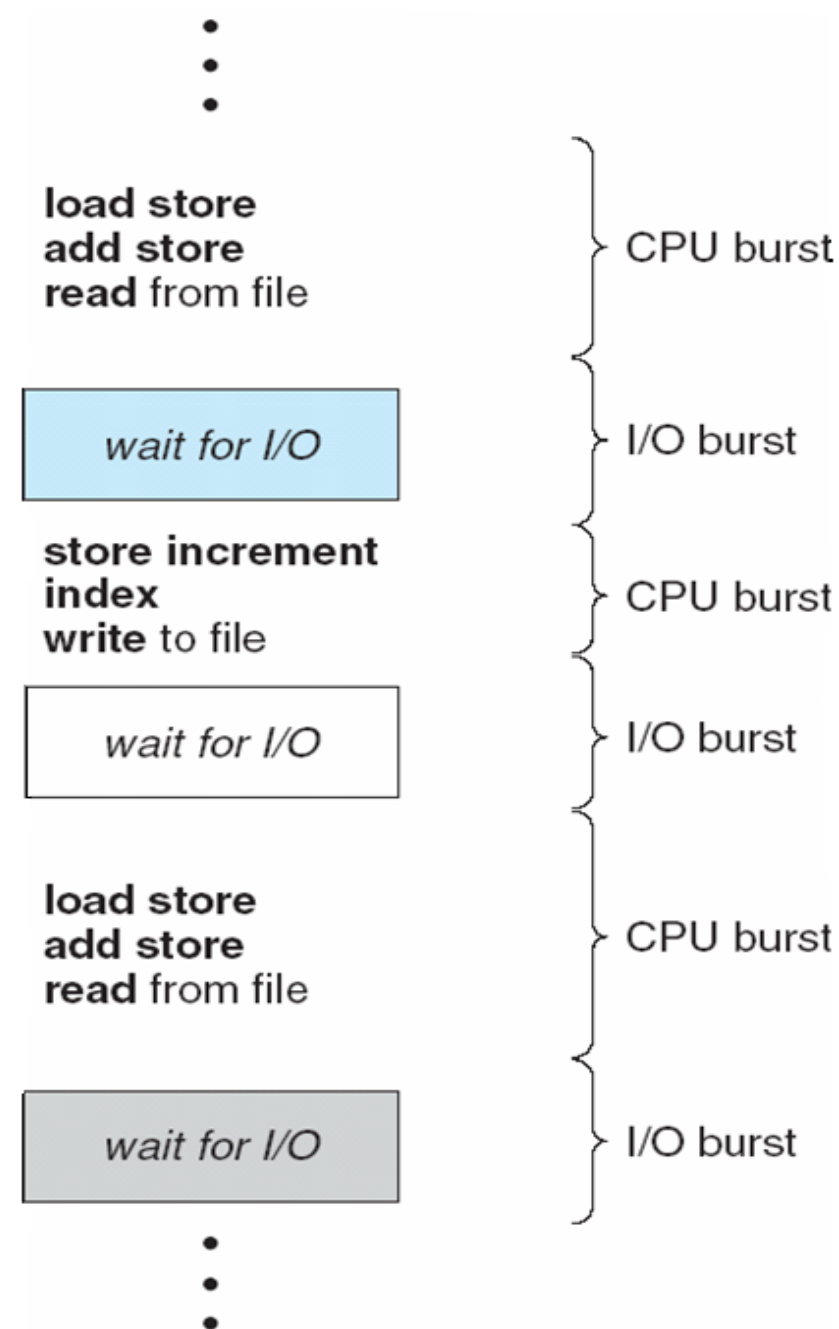


Scheduling Concepts

- **Operating systems schedule kernel-level threads**
- **Maximum CPU utilization is obtained through multiprogramming**
 - Single process cannot keep CPU and I/O devices busy at all times
 - Multiprogramming attempts to ensure that the CPU always has something to execute
 - A process is executed until it must wait for something (typically I/O)
 - When waiting, the OS swaps another process onto the CPU for execution

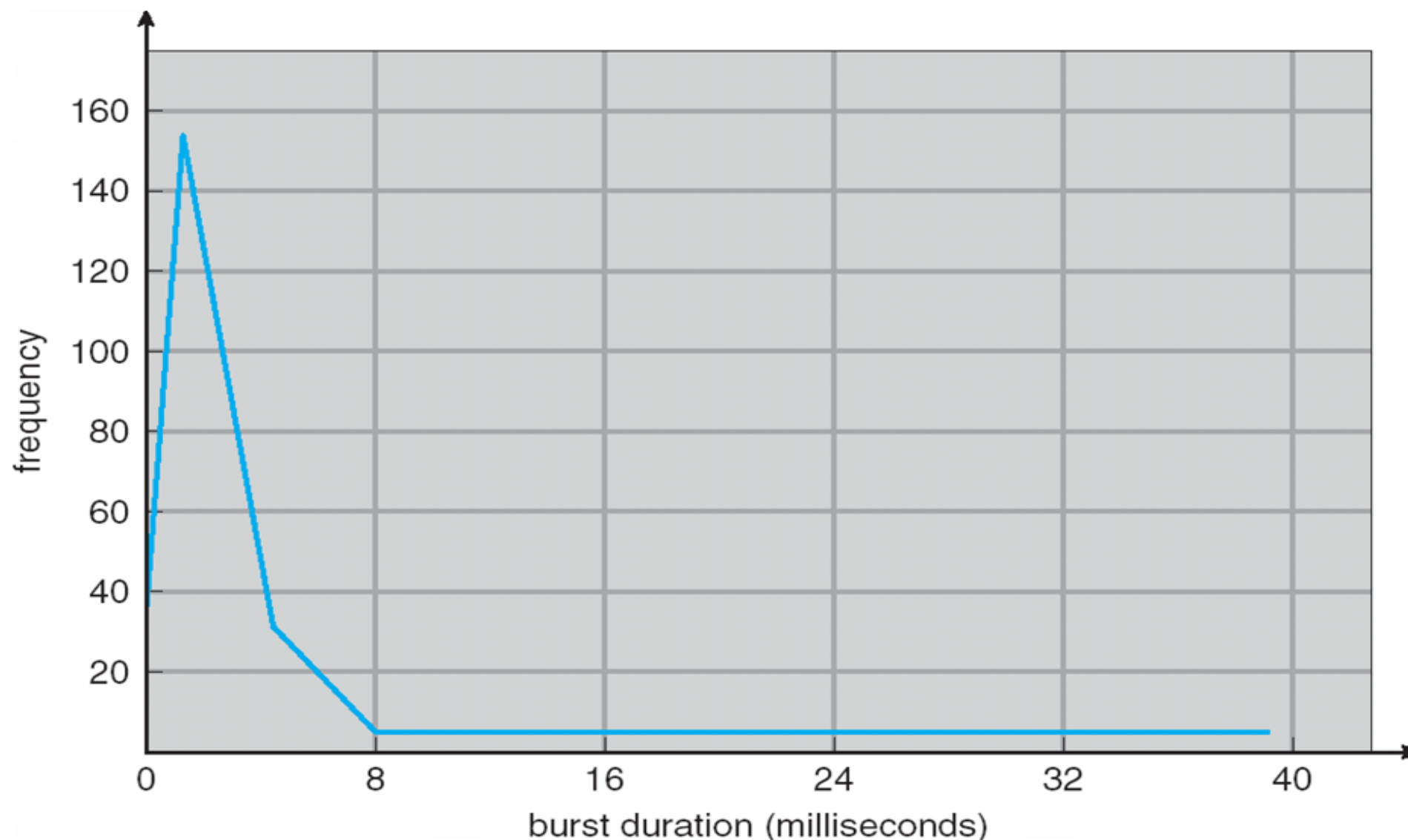
CPU-I/O Burst Cycle

- **CPU scheduling works because of a property known as the CPU-I/O burst cycle**
 - A process cycles between the states of **CPU execution** and **I/O wait**
 - When a process enters the I/O wait state, another process can enter the CPU execution state
- **I/O-bound programs typically have a large number of short CPU bursts**
- **CPU-bound programs typically have only a few long CPU bursts**



Histogram of CPU-burst Times

- **CPU-burst times for a typical program show a large number of short CPU bursts and only a small number of long CPU bursts**



CPU Scheduler (i.e. short-term scheduler)

- **Selects from among the processes in the ready-queue, and allocates the CPU to one of them**
 - Queue may be ordered in various ways:
 - FIFO
 - Priority queue
 - Tree
- **Preemptive scheduling prioritizes processes -- the process with the highest priority should be the process utilizing the CPU**
 - A higher priority process may cause a lower priority process to be removed from the CPU (i.e. the lower priority process goes from the 'running' state back into the ready-queue)
- **In nonpreemptive scheduling, a process can run on the CPU until it voluntarily relinquishes control to another process (also called cooperative scheduling)**

CPU Scheduler (Cont.)

- In a system with **nonpreemptive (cooperative) scheduling**, scheduling decisions are only made when:
 - (1) A process **voluntarily** switches from a running state to a waiting state
 - (2) A process terminates
- In a system with **preemptive scheduling**, scheduling decisions may take place when:
 - (1) An interrupt occurs that must be handled by another process
 - (2) A process switches from the waiting state and re-enters the ready-queue
 - (3) [Blanket statement] Any change is made to the ready-queue

CPU Scheduler (Cont.)

- **Special considerations when using a preemptive schedule**

- Data shared between processes (i.e. shared memory) may be left in an inconsistent state if one of the processes is preempted while writing data
 - Consider a queue -- the current size is incremented, but the process is preempted before the data can actually be added into the queue
- Low-level data structures (e.g. I/O queues) may be left in an unknown state if a process is preempted while in kernel mode due to a system call
 - If the process that caused the preemption needs to access those same low-level data structures, then bad things happen

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
 - Must be as small as possible since the dispatcher is invoked during every context switch

Scheduling Criteria

- **Different CPU-scheduling algorithms have different properties**
 - Different algorithms may favor one type of process over another
- **Different criteria can be considered when comparing scheduling algorithms**
 - **CPU utilization** – desirable to maximize CPU utilization
 - **Throughput** – number of processes that complete their execution per time unit
 - **Turnaround time** – amount of time to execute a particular process from creation to termination; desirable to minimize turnaround time
 - **Waiting time** – amount of time a process spends 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

- **Desirable to maximize:**

- CPU utilization
- Throughput

- **Desirable to minimize:**

- Turnaround time
- Waiting time
- Response time

Scheduling Algorithms

- **Many different scheduling algorithms exist -- each has its own method of determining which of the processes in the ready-queue will be assigned to the processor next**
 - First-Come, First-Served Scheduling
 - Shortest-Job-First Scheduling
 - Priority Scheduling
 - Round-Robin Scheduling
 - Multilevel Queue Scheduling
 - Multilevel Feedback Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

- **A very simple CPU-scheduling algorithm**
- **The process that requests the CPU first is allocated the CPU first**
- **Easily implemented with a standard FIFO**
 - Add the PCB of a process to the tail of the FIFO when added to the ready-queue
 - When the CPU is available, it is allocated to the process at the head of the FIFO
- **The average waiting time when using FCFS can be long**

FCFS Scheduling (Cont.)

- Consider the following three processes that all arrive at time $t=0$ in the order P_1, P_2, P_3

Process	Burst Time (ms)
P_1	24
P_2	3
P_3	3



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

FCFS Scheduling (Cont.)

- Consider the same three processes that all arrive at time $t=0$ in the order P_2 , P_3 , P_1

Process	Burst Time (ms)
P_2	3
P_3	3
P_1	24



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time = $(6 + 0 + 3) / 3 = 3$ milliseconds

FCFS Scheduling (Cont.)

- **The average waiting time when using FCFS is generally not minimal and can vary substantially**
- **Consider a CPU-bound process followed by many I/O bound processes**
 - CPU-bound process consumes processor time while I/O bound processes wait in the ready-queue
 - If I/O bound processes were allowed to go first, they could be sitting in the I/O queues waiting for I/O instead while the CPU-bound process consumed the CPU
 - If CPU-bound process gets to I/O resources before the I/O bound processes, then the CPU is likely to be left idle
- **FCFS is nonpreemptive which means it is terrible for time-sharing systems as one process may hog CPU resources**

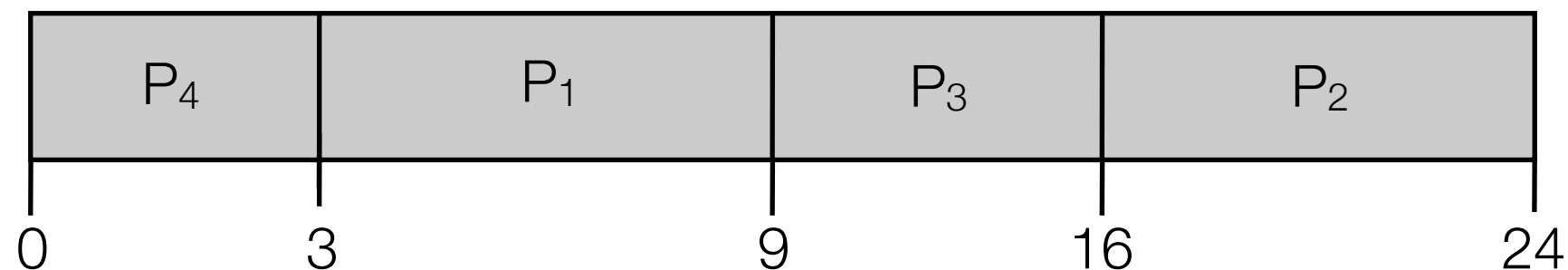
Shortest-Job-First (SJF) Scheduling

- **Associate with each process the length of its next CPU burst**
 - Use these lengths to schedule the process with the shortest burst time next
 - Note that the process is selected based on the length of its *next* CPU burst only, NOT the length of the entire process
 - If multiple processes have the same burst length, the tie is broken using FCFS
- **SJF is provably optimal – gives minimum average waiting time for a given set of processes (only when all jobs are available simultaneously)**
 - The difficulty is knowing the length of the next CPU request
 - Could ask the user
 - May be able to predict the length of the next CPU burst

SJF Scheduling (Cont.)

- Consider the following four processes that all arrive at time $t=0$ in the order P_1, P_2, P_3, P_4

<u>Process</u>	<u>Burst Time (ms)</u>
P_1	6
P_2	8
P_3	7
P_4	3



- Average waiting time** = $(3 + 16 + 9 + 0) / 4 = 7$ (provably minimal)

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
- **Generally predicted as an exponential average of the measured lengths of previous CPU bursts**

t_n = actual length of n^{th} CPU burst

τ_{n+1} = predicted value for next CPU burst

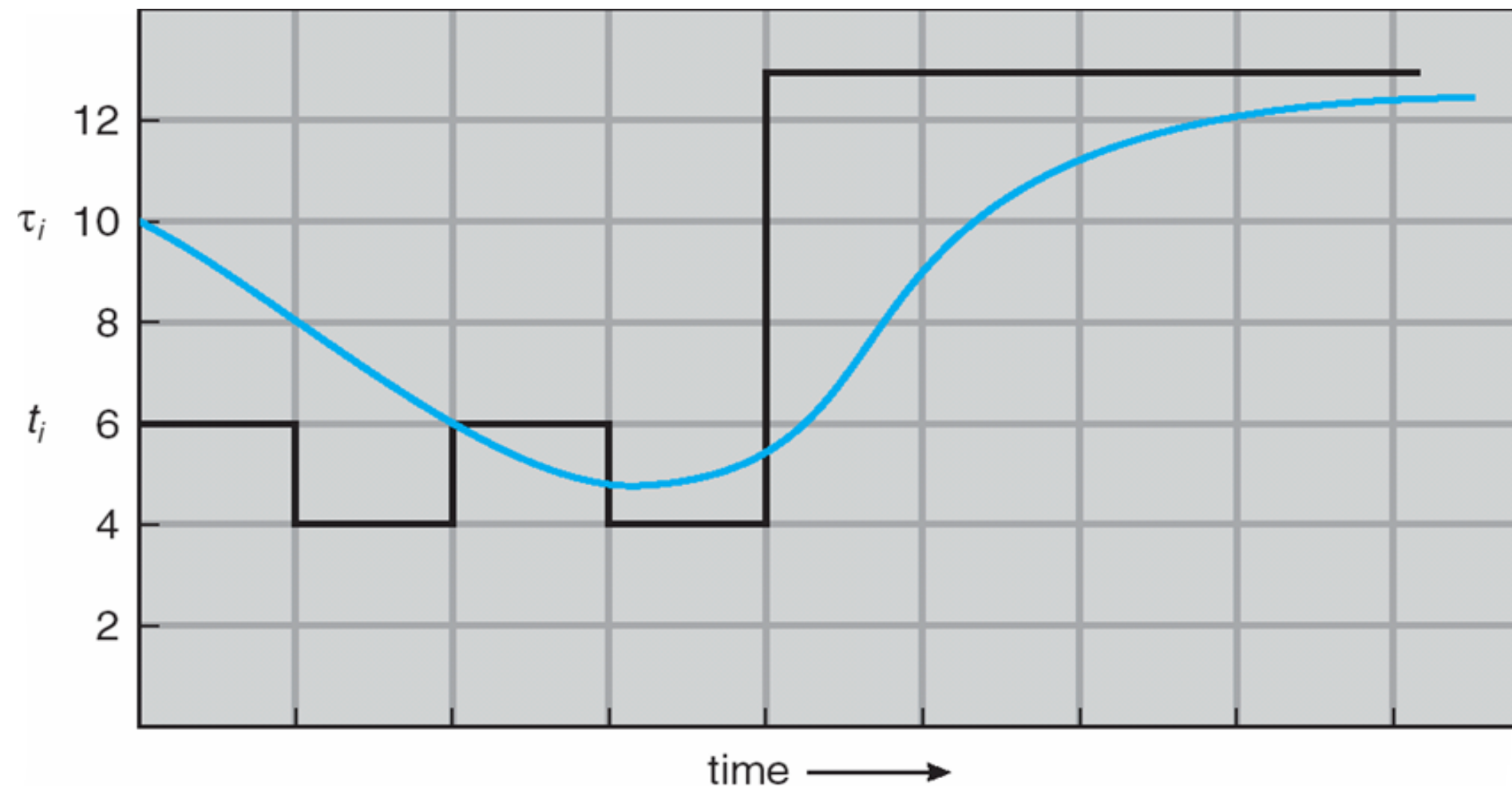
$\alpha, 0 \leq \alpha \leq 1$

Define:

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

- **Commonly, α set to $\frac{1}{2}$ -- determines the relative weight of recent and past history**
 - If $\alpha=0$, then recent history has no effect
 - If $\alpha=1$, then only the most recent CPU bursts matter

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	5	6	9	11	12	...

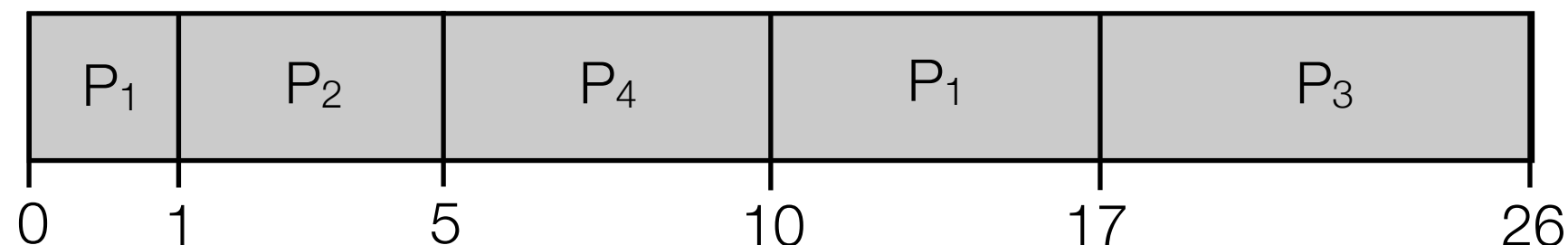
SJF Scheduling

- **SJF scheduling can be either nonpreemptive or preemptive**
 - If a new process enters the ready-queue while a previous process is still executing, should the new process preempt the currently running process if its next CPU burst is shorter than the currently executing process?
 - A nonpreemptive version will allow the currently executing process to finish
 - A preemptive version will allow the new process with the shorter CPU burst to preempt the currently executing process
- **Preemptive version is typically called shortest-remaining-time-first**

Example of Shortest-Remaining-Time-First

- Consider the following four processes with the specified arrival times and burst times

Process	Arrival Time	Burst Time (ms)
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



- Average waiting time** = $[(10-1)+(1-1)+(17-2)+(5-3)] / 4 = 26/4 = 6.5$ milliseconds
 - P₁ arrived at $t=0$ and ran immediately, then it waited from $t=1$ to $t=10$;
 - P₂ arrived at $t=1$ and never waited;
 - P₃ arrived at $t=2$ and waited until $t=17$ to run;
 - P₄ arrived at $t=3$ and ran at $t=5$

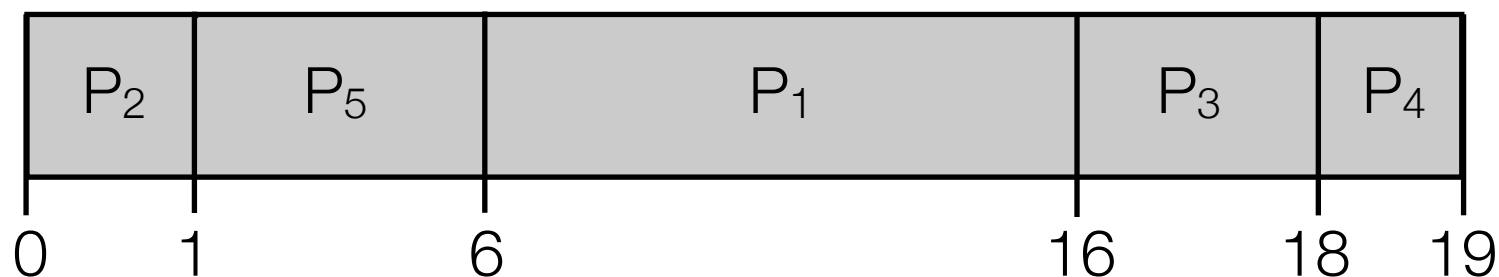
Priority Scheduling

- A **priority number** (integer) is associated with each process
- The **CPU** is allocated to the process with the highest priority (smallest integer usually the highest priority)
 - Can be **preemptive** or **nonpreemptive**
- **SJF** is a method of priority scheduling where the priority is the inverse of next predicted **CPU burst time**

Example of Priority Scheduling

- Consider the following five processes that all arrive at time $t=0$ in the order P_1, P_2, P_3, P_4, P_5 -- (nonpreemptive version)

<u>Process</u>	<u>Burst Time (ms)</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



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

Priority Scheduling

- **One problem with priority scheduling is something called starvation**
 - A low-priority process **never** gets allocated the CPU because there are always higher priority processes in the ready-queue
- **One possible solution to the problem of starvation is called aging**
 - As time progresses the priority of a process increases
 - A low-priority process gradually becomes a higher priority process
 - The rate at which a process ages can be tuned

Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (called a **time quantum** or **time slice**), usually 10-100 milliseconds
- Processes are stored in the ready queue which is implemented as a simple FIFO
- When the CPU is available, it is allocated to the process at the head of the ready queue
- The process is run for some period of time (the **time quantum**) after which it is preempted by a timer, added back to the tail of the ready queue and the next process in the ready queue is dispatched

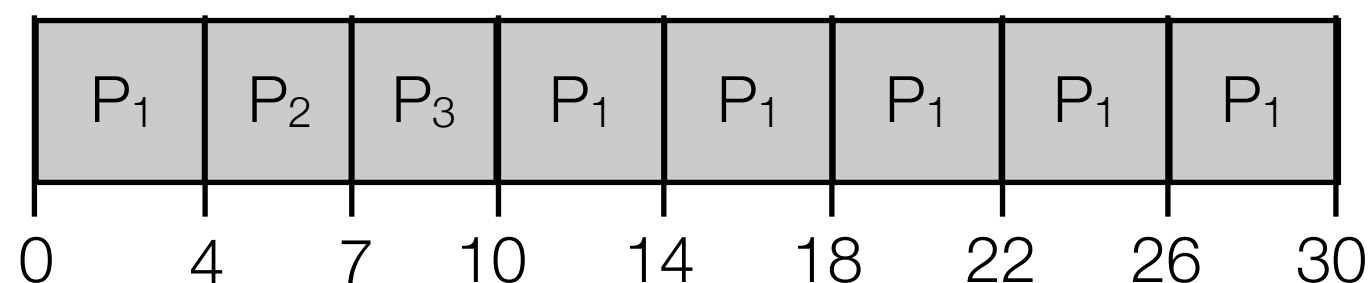
RR Scheduling (Cont.)

- **Designed especially for time-sharing systems**
- **Similar to FCFS, but with preemption**
 - If time quantum is very large, then RR scheduling behaves identically to FCFS
- **If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once.**
 - No process waits more than $(n-1)*q$ time units between CPU bursts
 - No process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process)

Example of RR with Time Quantum = 4

- Consider the following three processes that all arrive at time $t=0$ in the order P_1, P_2, P_3 -- the time quantum = 4 milliseconds

Process	Burst Time (ms)
P_1	24
P_2	3
P_3	3



- Average waiting time = $(6 + 4 + 7) / 3 = 5.67$ milliseconds

Performance of RR Scheduling

- **Typically, higher average turnaround time than SJF, but better response**
 - Good response time makes it well-suited for time-sharing systems
- **Size of time quantum can dramatically impact the performance of RR scheduling**
 - If time quantum is too large, then RR starts to look like FCFS
 - If time quantum is too small and context switching occurs too often, then there is a high overhead for context switching

Multilevel Queue Scheduling

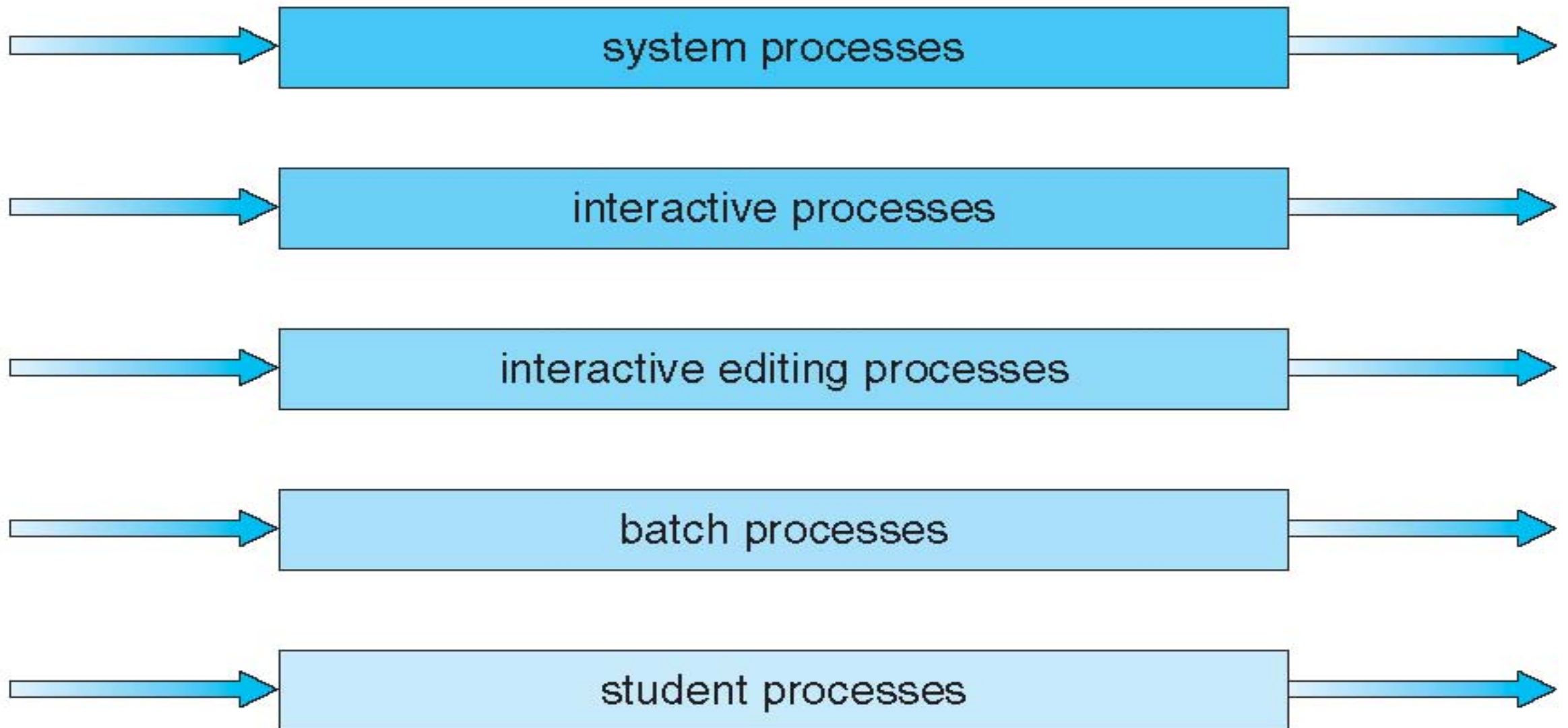
- **Created for situations in which processes are easily classified into different groups (i.e. interactive processes / background processes)**
 - Different types of processes may have different scheduling requirements
- **Ready queue is partitioned into separate queues for different types of processes**
- **A process is permanently assigned to a specific queue**
- **Each queue has its own scheduling algorithm**
 - Interactive processes may be scheduled using RR scheduling
 - Background processes may be scheduled using FCFS scheduling

Multilevel Queue Scheduling

- **Scheduling must also be done between the various queues**
 - One method uses **fixed priority preemptive scheduling**
 - Serve all processes from interactive process queue, then serve processes from the background process queue
 - Possibility of starvation for background processes
 - Another method assigns **time slices** to each queue
 - Each queue gets a certain amount of CPU time which it can schedule amongst its processes
 - 80% of time slice may go to interactive processes which are scheduled in RR fashion
 - 20% of time slice may go to background processes which are scheduled in FCFS fashion

Multilevel Queue Scheduling

highest priority



lowest priority

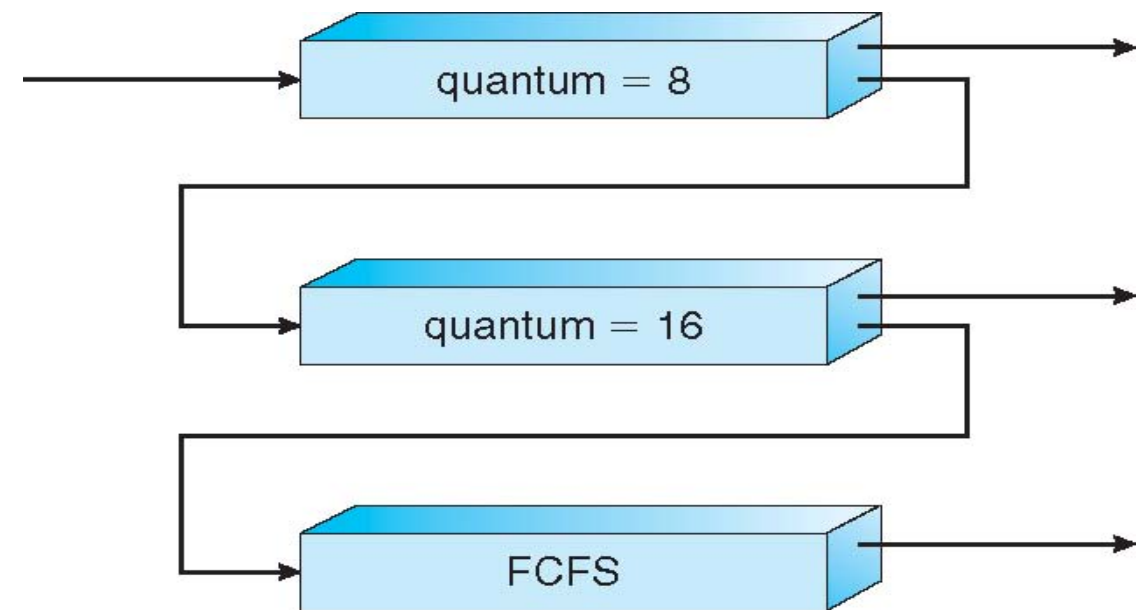
Multilevel Feedback Queue Scheduling

- **Similar to Multilevel Queue Scheduling, but processes may move between the various queues**
- **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 to a higher priority queue
 - Method used to determine when to demote a process to a lower priority queue
 - Method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue Scheduling

- **Three queues:**

- Q0 – RR with time quantum 8 milliseconds
- Q1 – RR time quantum 16 milliseconds
- Q2 – FCFS



- **Scheduling**

- A new process enters queue Q0 which is served with RR scheduling
 - When it gains the CPU, the process receives 8 milliseconds
 - If it does not finish in 8 milliseconds, it is preempted and moved to queue Q1
- At Q1 the process is again served with RR scheduling and receives 16 additional milliseconds (only if Q0 is empty)
 - If it still does not complete, it is preempted and moved to queue Q2
- Once in Q2, a process is able to utilize whatever CPU cycles are left over from Q0 and Q1