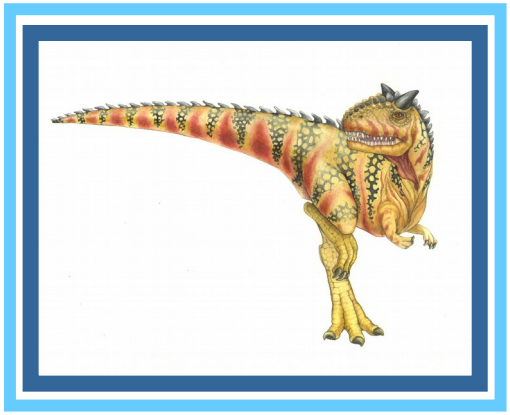
# Chapter 5a: CPU Scheduling



# Chapter 5a: CPU Scheduling

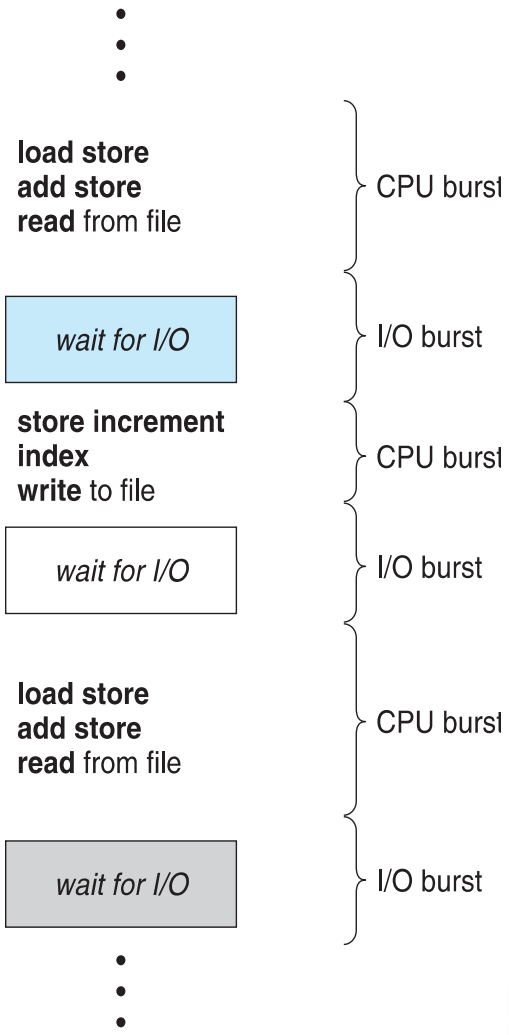
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

# Objectives

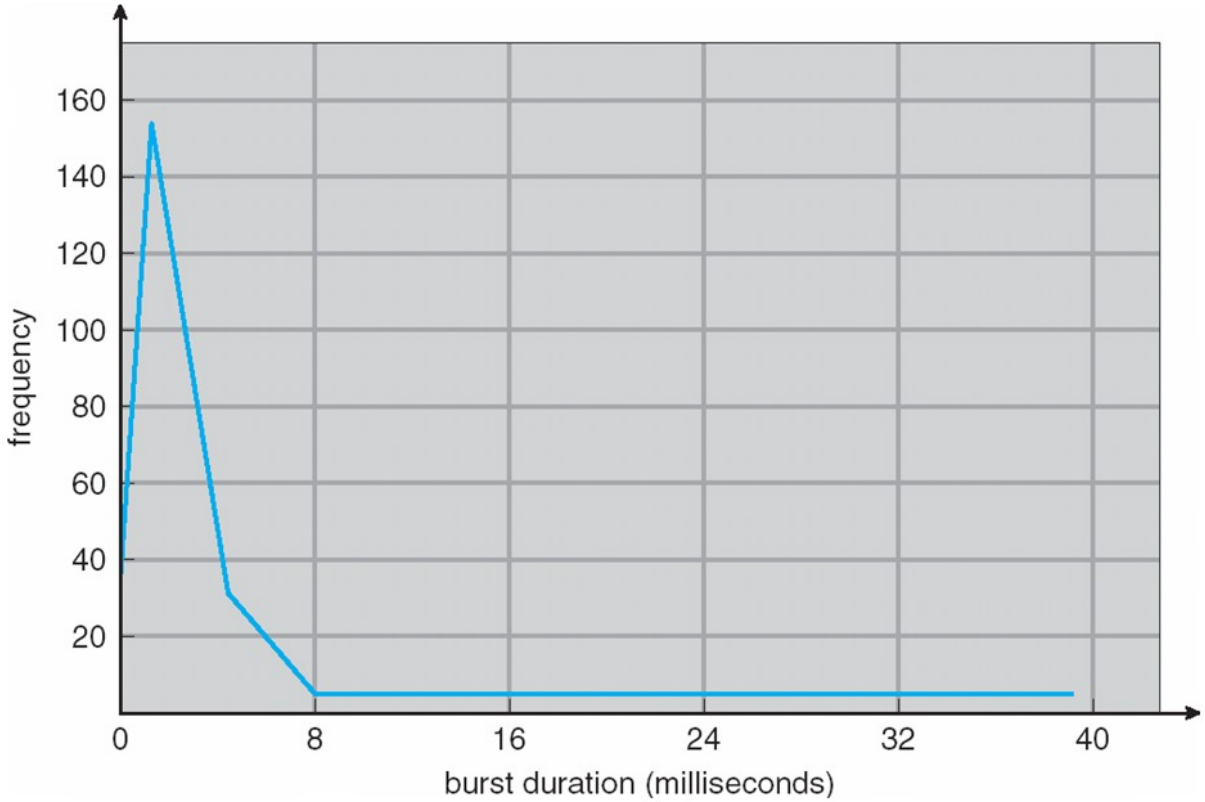
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

# Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- Most processes exhibit the following behavior:
- **CPU burst** followed by **I/O burst**
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- CPU burst distribution is of main concern




# Histogram of CPU-burst Times




# CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **CPU scheduler**.
- The ready queue may be ordered in various ways.
- CPU scheduling decisions may take place when a process:
  1. Switches from running state to waiting state
  2. Switches from running state to ready state
  3. Switches from waiting state to ready state
  4. When a process terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- There is a choice, however, for situations 2 and 3.

# Nonpreemptive Scheduling

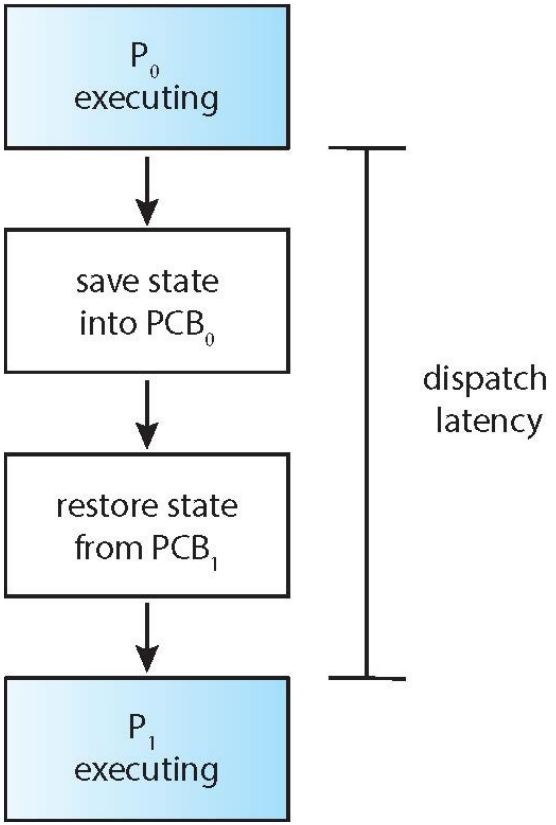
- Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU:
  - either by terminating
  - or by switching to the waiting state.

# Preemptive scheduling

- Preemptive scheduling can result in race conditions when data are shared among several processes.
  - Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities
- Virtually all modern operating systems including Windows, Mac OS X, Linux, and UNIX use preemptive scheduling algorithms.

# Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU 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** – number of processes that complete their execution per time unit (e.g., 5 per second)
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – total 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 (for time-sharing environment)

# Optimization Criteria for Scheduling

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

# Scheduling Algorithm

- First –come, First-serve (FCFS)
- Shortest-Job-First Scheduling (SJF)
- Round-Robin Scheduling (RR)
- Priority Scheduling
- Multilevel Queue Scheduling

# First- Come, First-Served (FCFS) Scheduling

- Consider the following three processes and their burst time

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Suppose that the processes arrive in the order: $P_1$ , $P_2$ , $P_3$
- We use Gantt Chart to illustrate a particular schedule


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

# Shortest-Job-First (SJF)

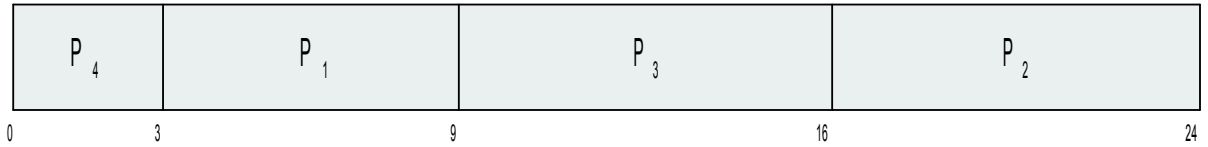
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - How do we know what is  the length of the next CPU request
  - Could ask the user
    - What if the user lies?

# Example of SJF

- Consider the following four processes and their burst time

| 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 (predict) the length – in most cases should be similar to the previous CPU burst
  - Pick the 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. $\alpha, 0 \le \alpha \le 1$
  4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

- Commonly, α set to ½

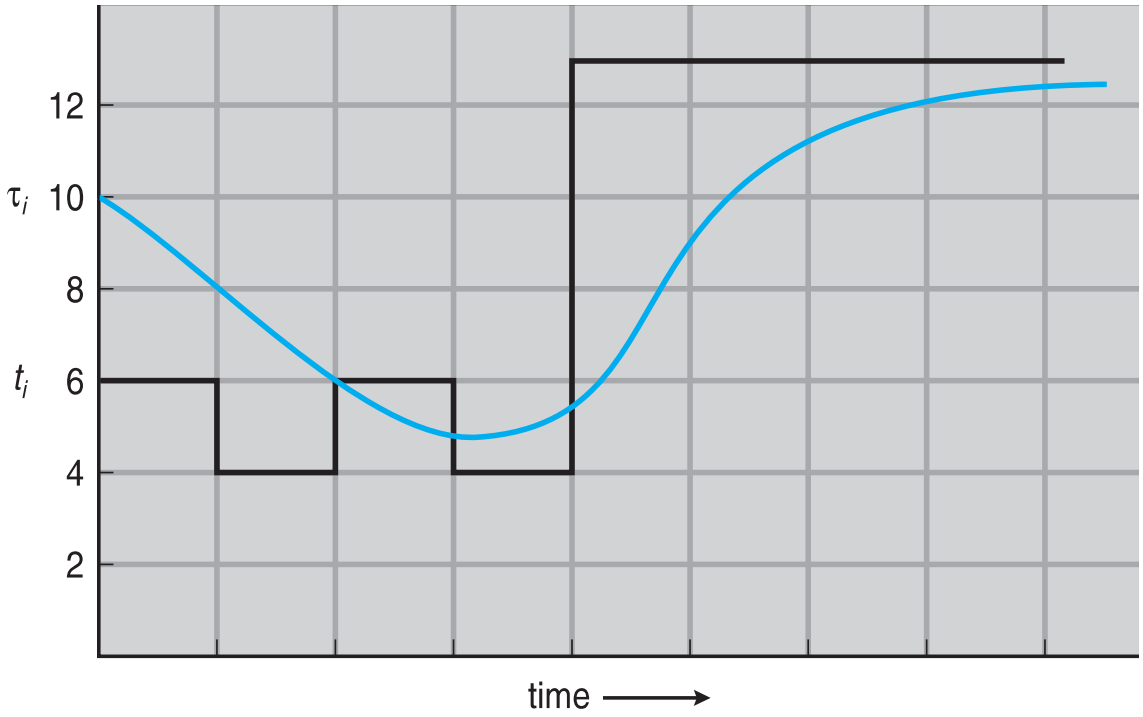
# Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - 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 $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

# Prediction of the Length of the Next CPU Burst



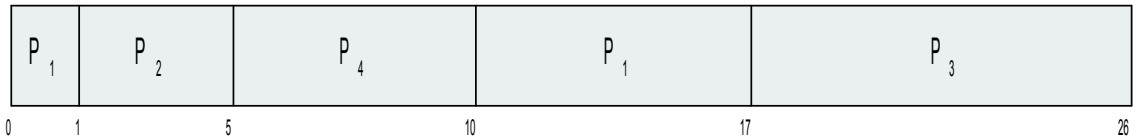

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| CPU burst ($t_i$) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 ... |
| "guess" ($\tau_i$) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 ... |

# Shortest-remaining-time-first

- Preemptive version of SJF is called **shortest-remaining-time-first**
- Example illustrating the concepts of varying arrival times and preemption.

| Process | *Arrival* Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

- *Preemptive* SJF Gantt Chart




- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

# Round Robin (RR)

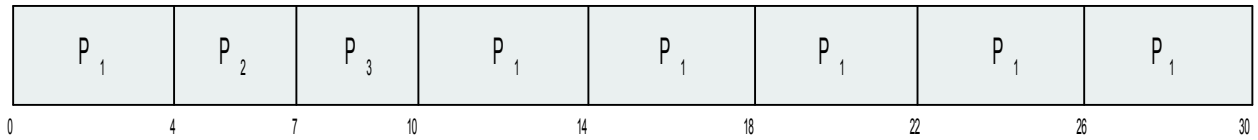
- Each process gets a small unit of CPU time (**time quantum** *q*). After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *N* processes in the ready queue and the time quantum is *q*, then each process gets 1/*N* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*N*-1)**q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

- Consider the following three processes and their burst time

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

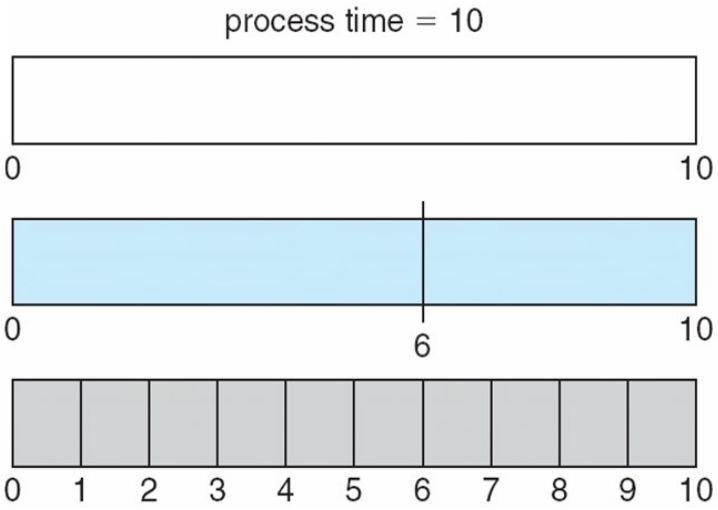
- The Gantt chart is:




- The average waiting time under the RR policy is often longer
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q is usually 10ms to 100ms, context switch < 10 usec

# Time Quantum and Context Switch Time

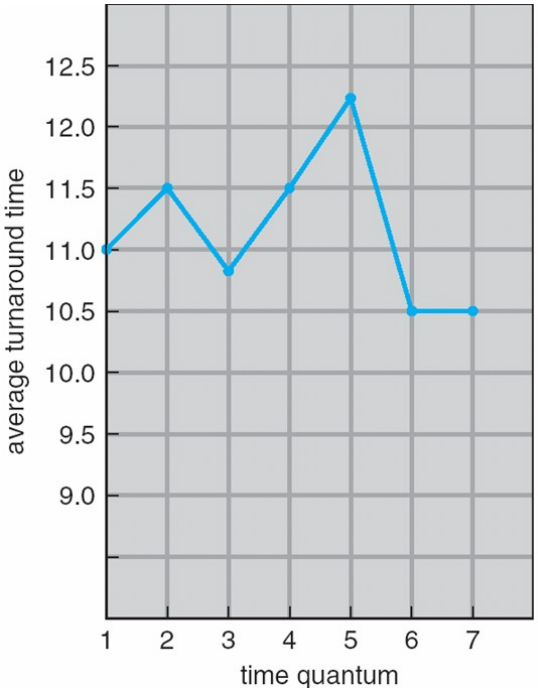
- The performance of the RR algorithm depends on the size of the time quantum. If the time quantum is extremely small (say, 1 millisecond), RR can result in a large number of context switches.




| quantum | context switches |
|---|---|
| 12 | 0 |
| 6 | 1 |
| 1 | 9 |

# Turnaround Time Varies with the Time Quantum

- The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.




| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

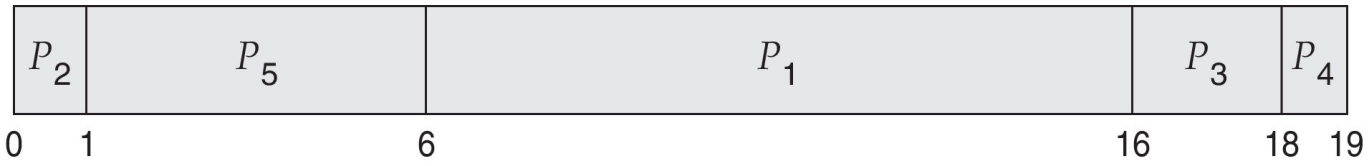
# Priority Scheduling

- 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
- 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 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



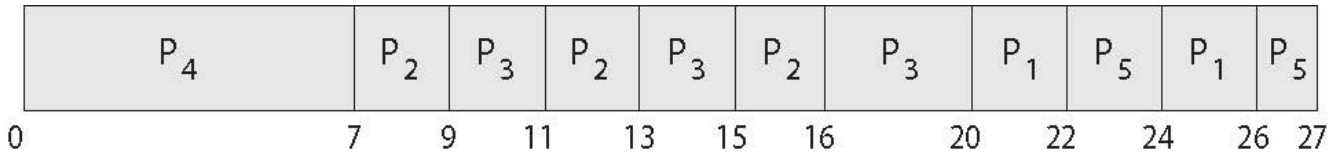

- Average waiting time = 8.2 msec

# Combining Priority Scheduling and RR

- System executes the highest priority process; processes with the same priority will be run using round-robin.
- Consider the following five processes and their burst time

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 4 | 3 |
| $P_2$ | 5 | 2 |
| $P_3$ | 8 | 2 |
| $P_4$ | 7 | 1 |
| $P_5$ | 3 | 3 |

- Priority scheduling Gantt Chart

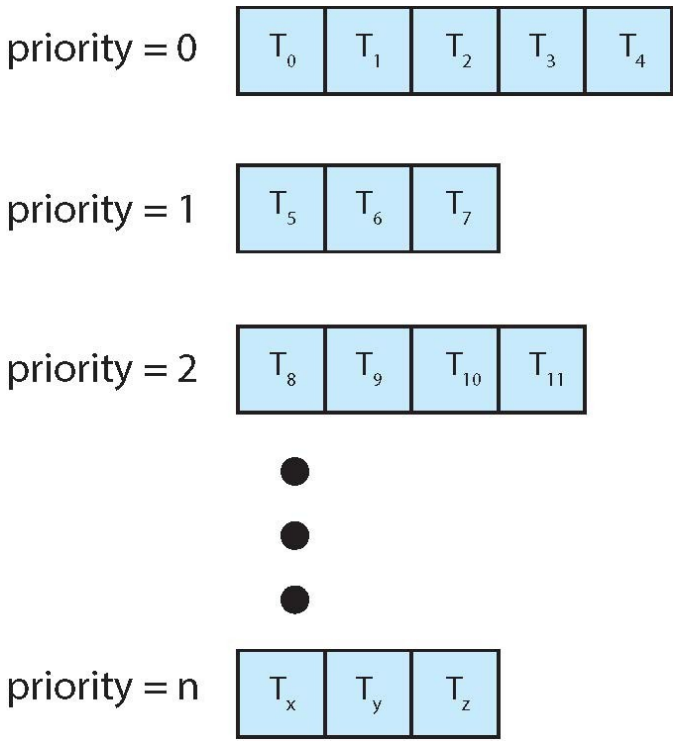



- Average waiting time = 8.2 msec

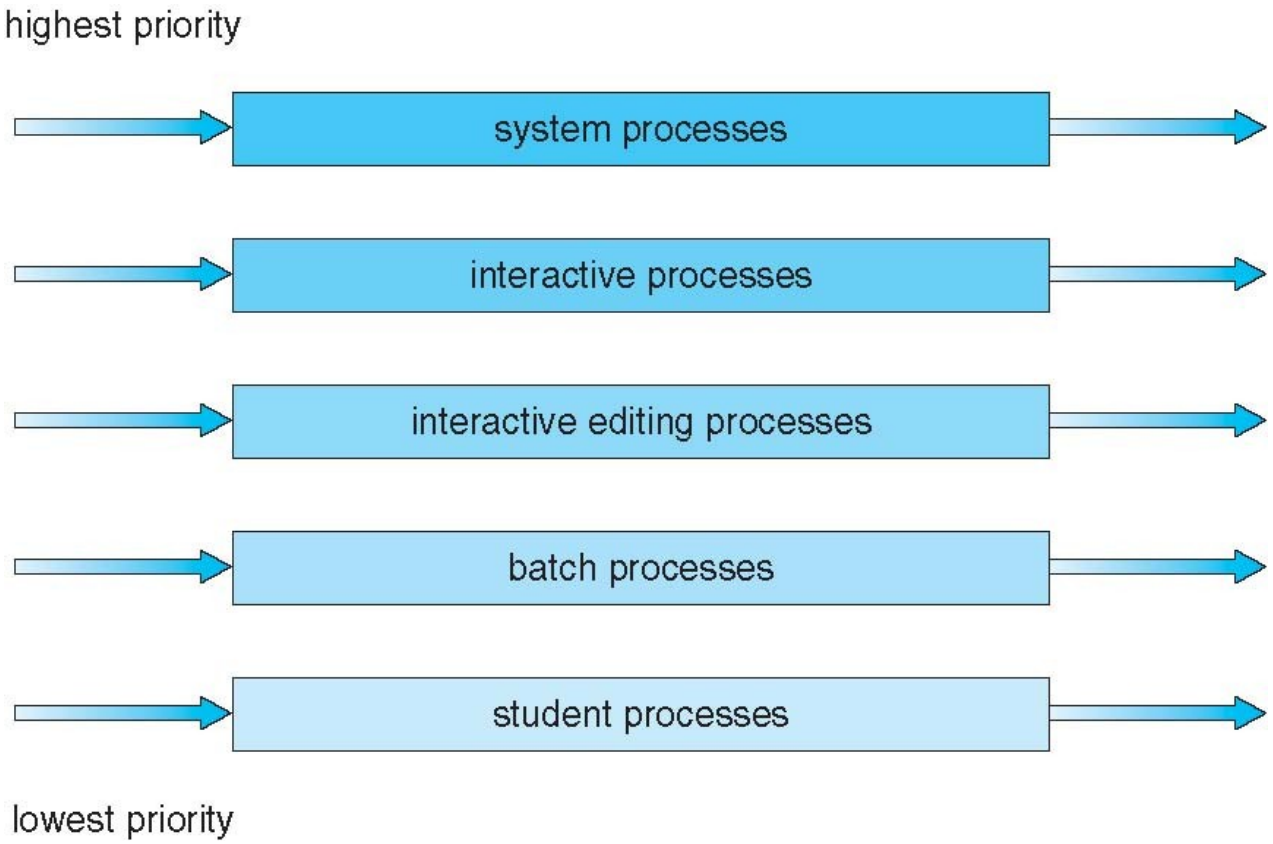
# Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given 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

# Separate Queue For Each Priority




# Multilevel Queue Scheduling

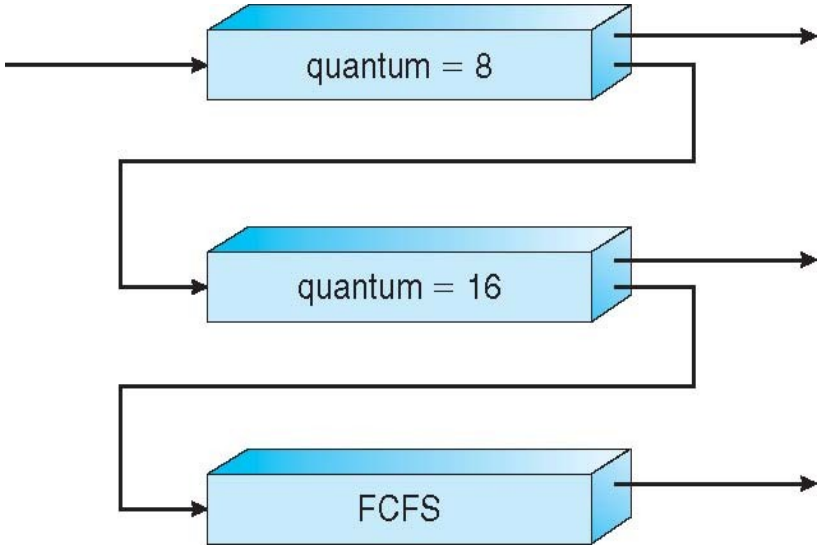



# Multilevel Feedback Queue

- 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 $Q_2$




# End of Chapter 5a