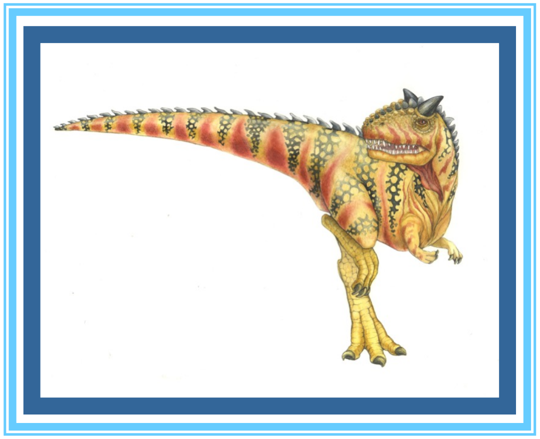
Chapter 5: CPU Scheduling – Part 1

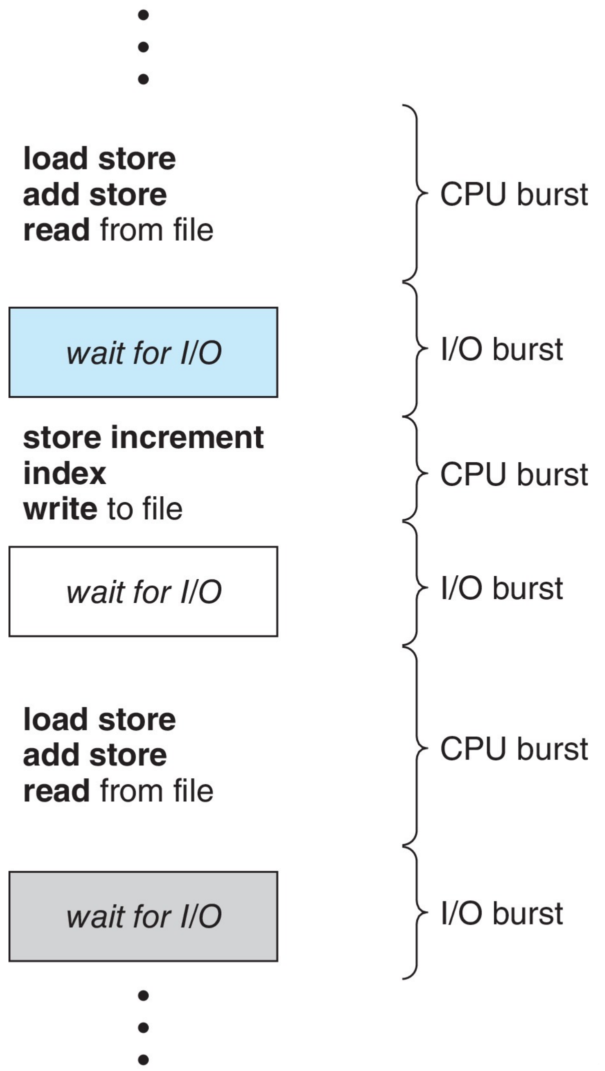


Outline

- **Basic Concepts**
- **Scheduling Criteria**
- **Scheduling Algorithms**
- **Thread Scheduling**
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Example

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – 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



load store
add store
read from file

wait for I/O

store increment
index
write to file

wait for I/O

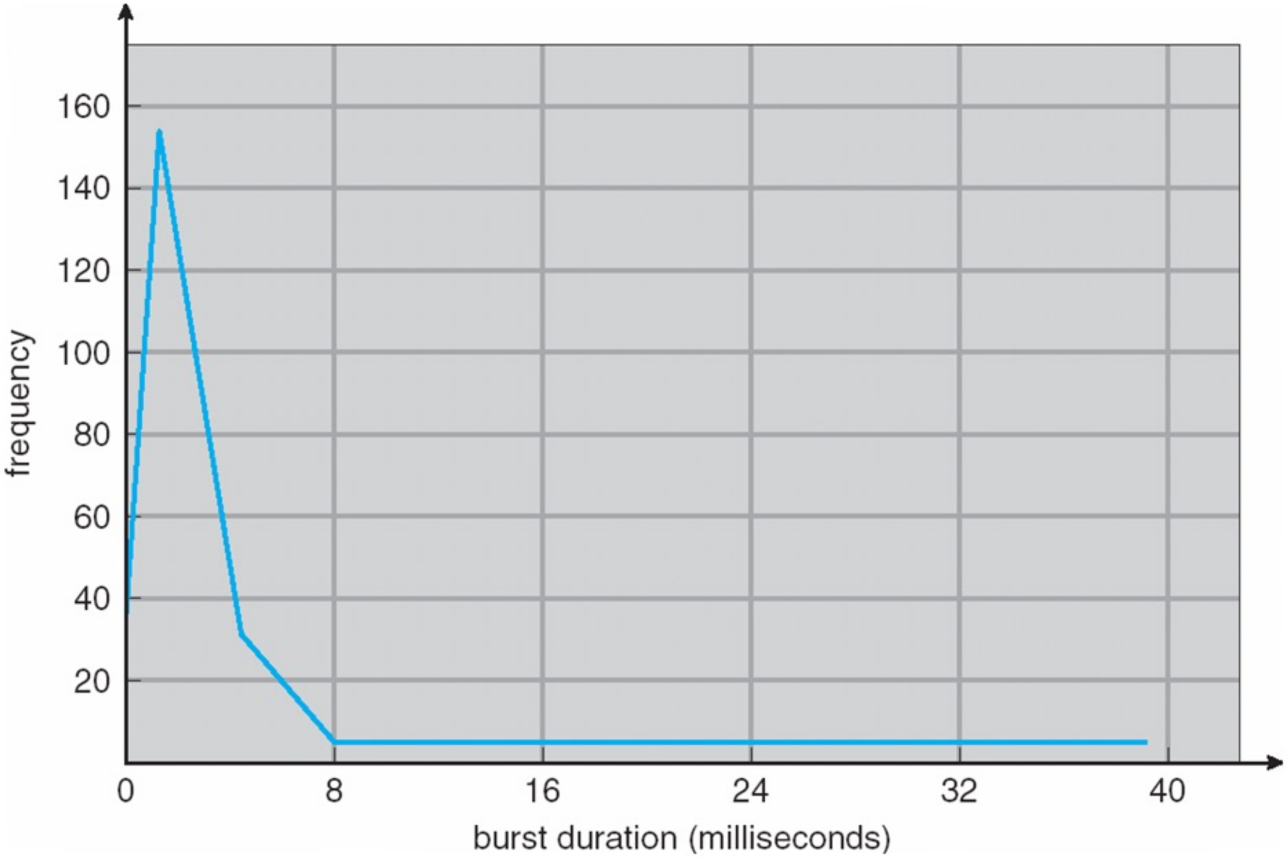
load store
add store
read from file

wait for I/O

Histogram of CPU-burst Times

Large number of short bursts

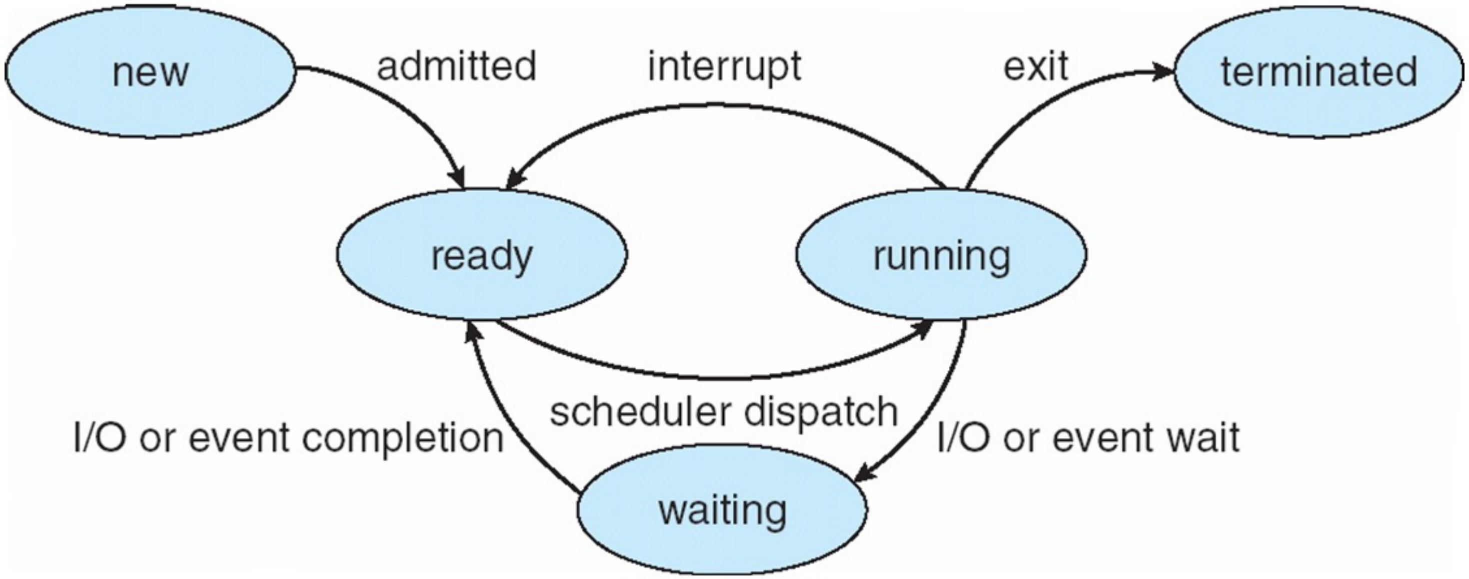
Small number of longer bursts



CPU Scheduler

- The **CPU scheduler** selects from among the processes in **ready queue**, and allocates a CPU core to one of them
  - Queue may be ordered in various ways : FIFO, priority Queue, Tree
- 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
- 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.
- For situations 2 and 3, however, there is a choice.

Recall: Diagram of Process State



Preemptive and Nonpreemptive Scheduling

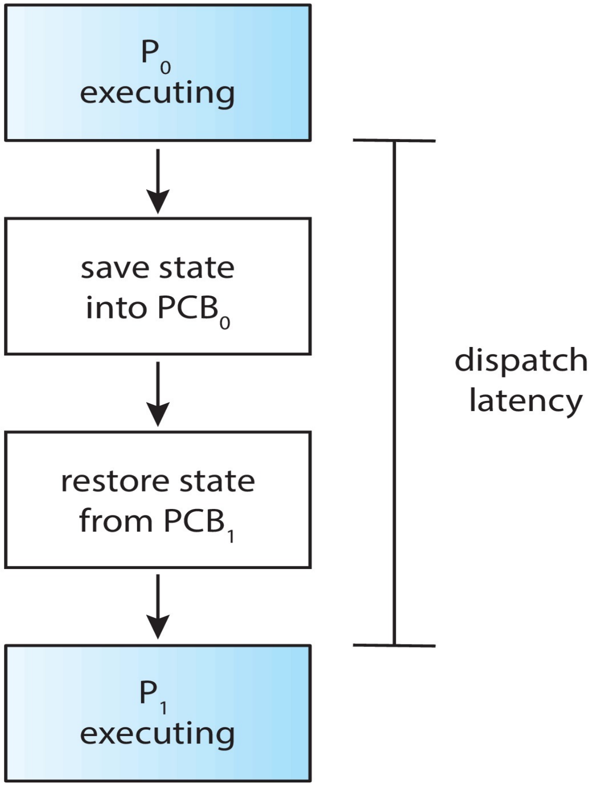
- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Under **Nonpreemptive scheduling**, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

Preemptive Scheduling and Race Conditions

- **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.
- This issue will be explored in detail in Chapter 6.

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** – # 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.

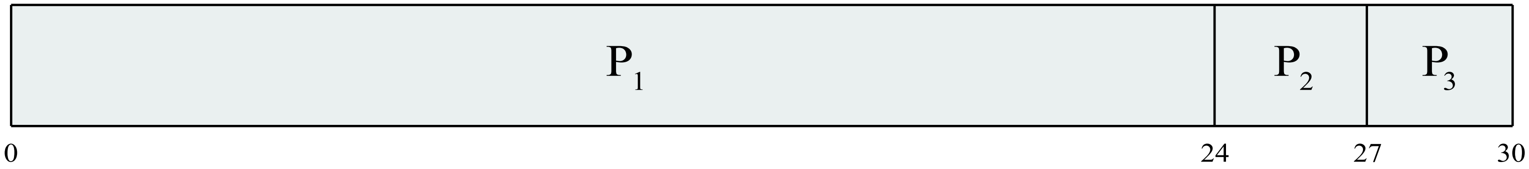
Scheduling Algorithm Optimization Criteria

- 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:



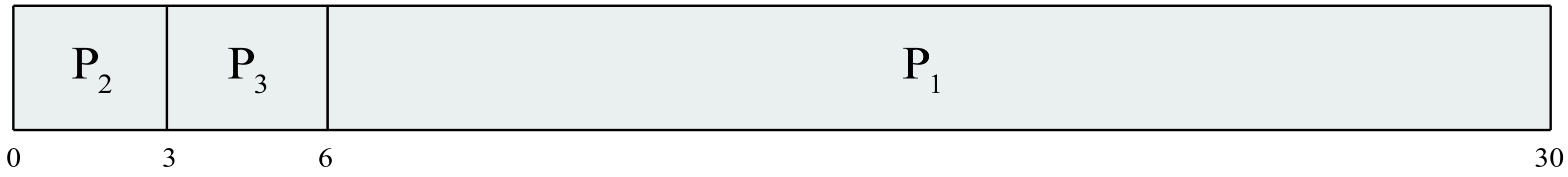
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Turnaround time?
- Response time?

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

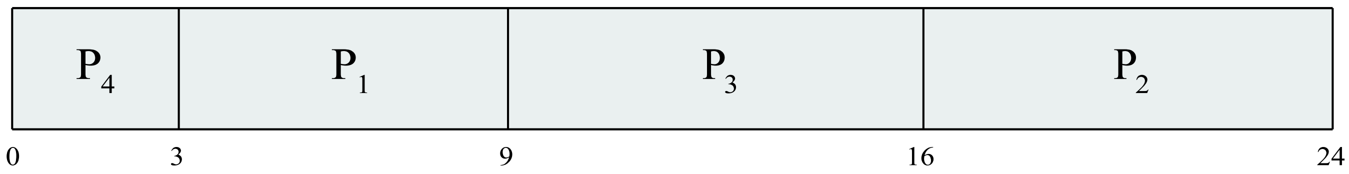
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - **Non-premptive**
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - **Could ask the user**
  - **Estimate**

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
- Average response time?
- Average turnaround time?

Determining Length of Next CPU Burst

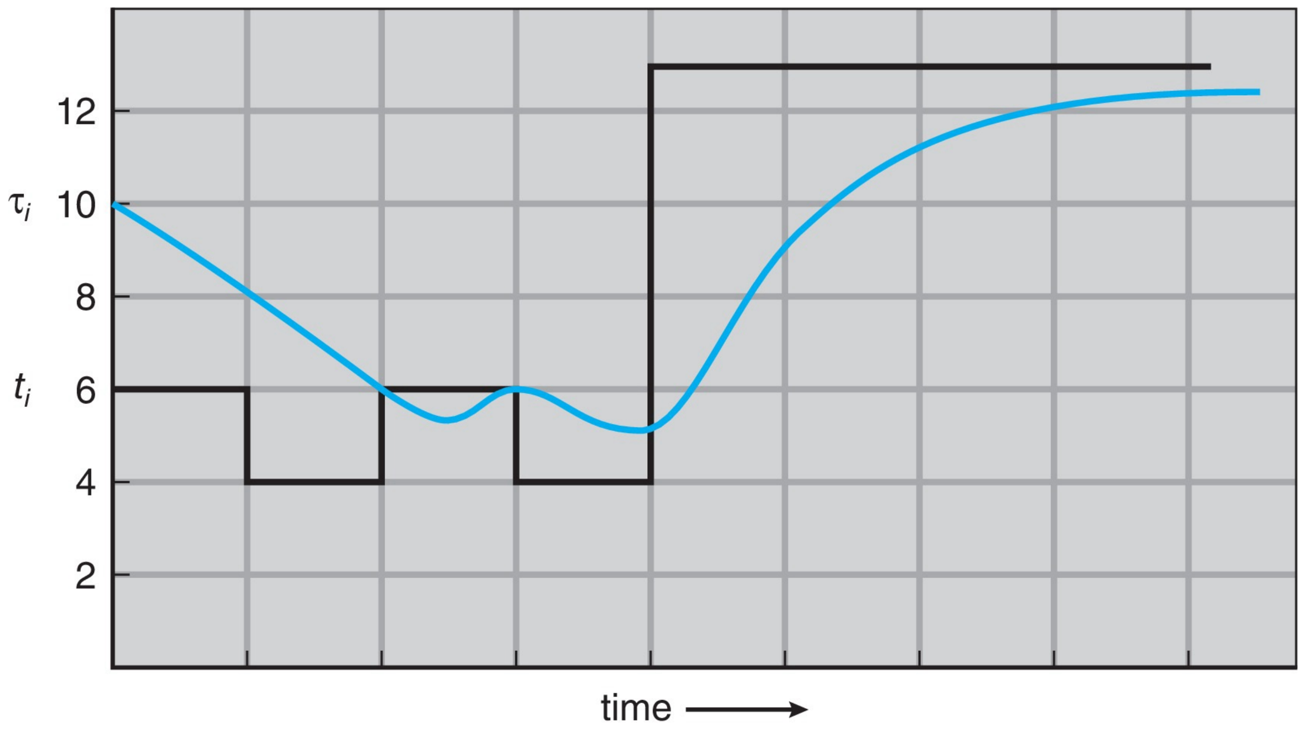
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define:

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

- Commonly, α set to ½

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

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

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - 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:

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

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successor predecessor term has less weight than its predecessor

Shortest Remaining Time First Scheduling

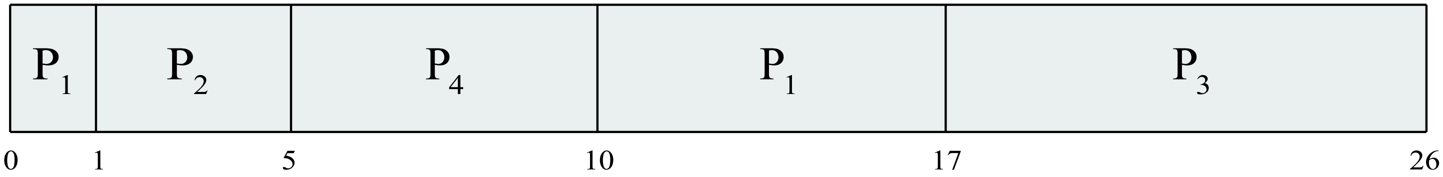
- Preemptive version of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
- Is SRT more “optimal” than SJF in terms of the minimum average waiting time for a given set of processes?

Example of Shortest-remaining-time-first

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

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

Let’s trace SRTF

Round Robin (RR)

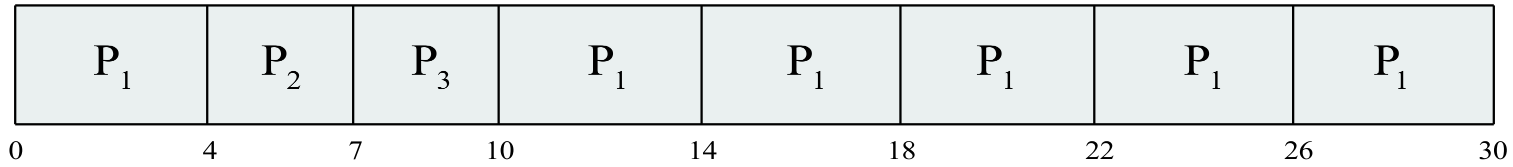
- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are ***n*** processes in the ready queue and the time quantum is ***q***, then each process gets **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 (FCFS)
  - *q* small ⇒ RR
- Note that q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Suppose P1, P2, and P3 arrive at the same time

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

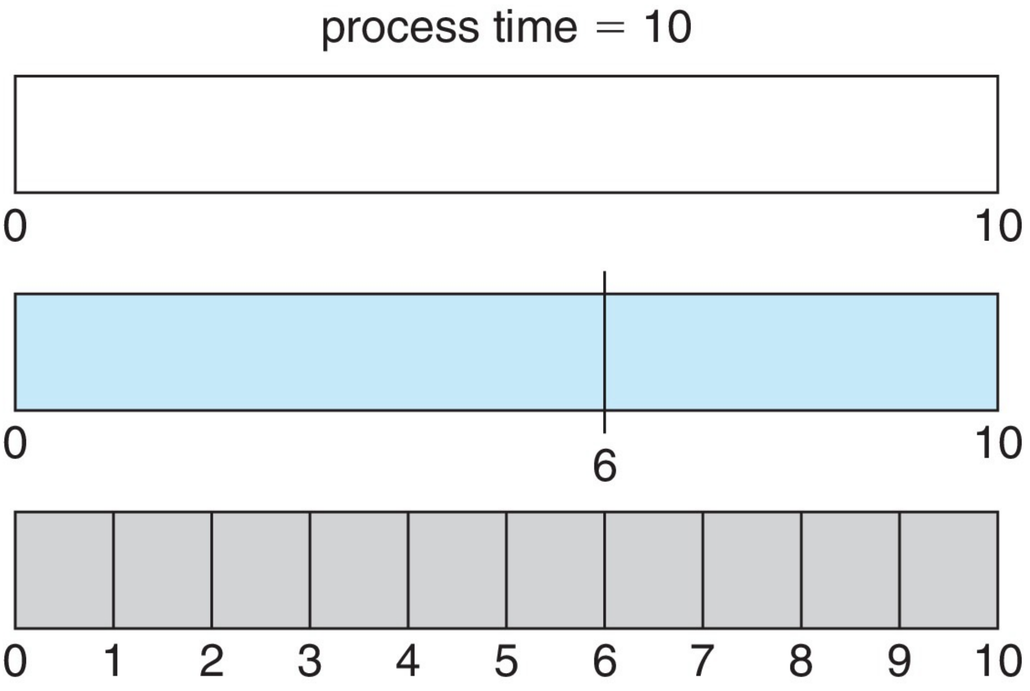


- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

Let’s Trace RR

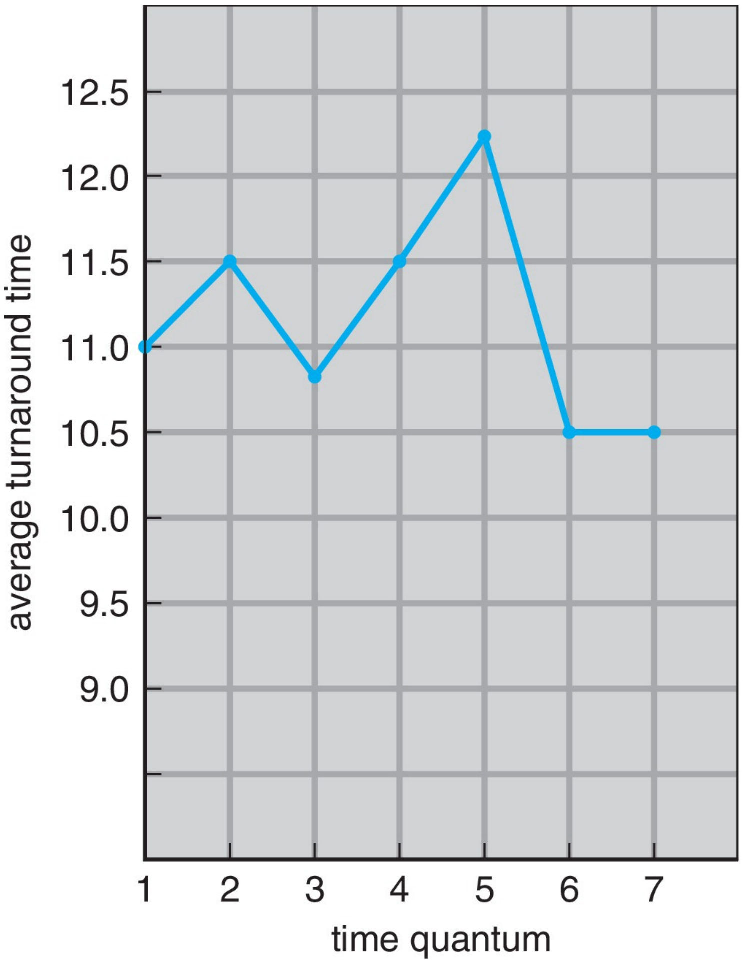
Time Quantum and Context Switch Time

How a smaller time quantum increases context switches.



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Priority Scheduling

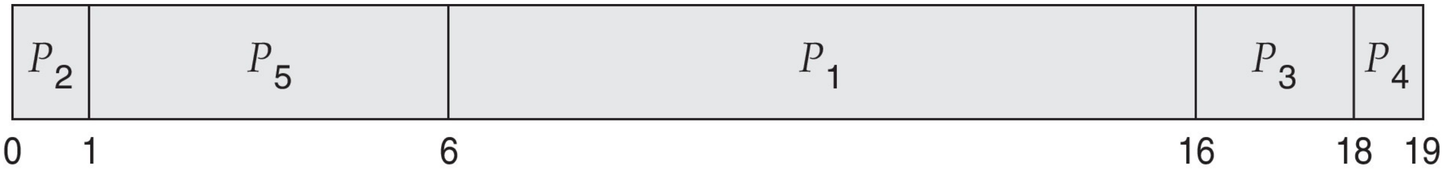
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (e.g smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

smallest integer ≡ highest priority

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

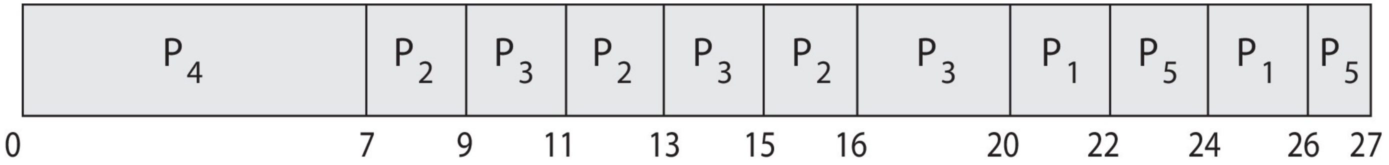
Let’s Trace Priority Scheduling

Priority Scheduling w/ Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

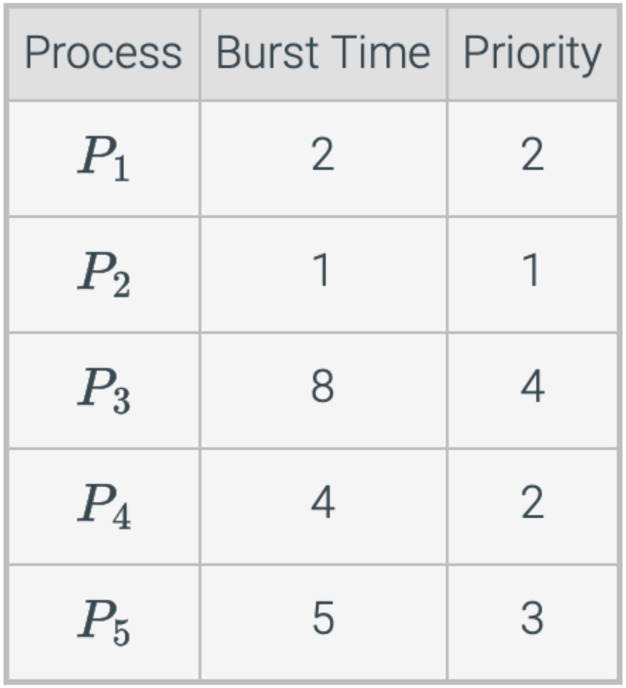
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Gantt Chart with time quantum = 2



Check your understanding

- Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

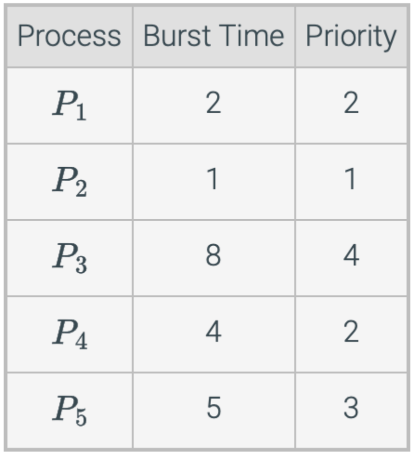


Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5 all at time 0.

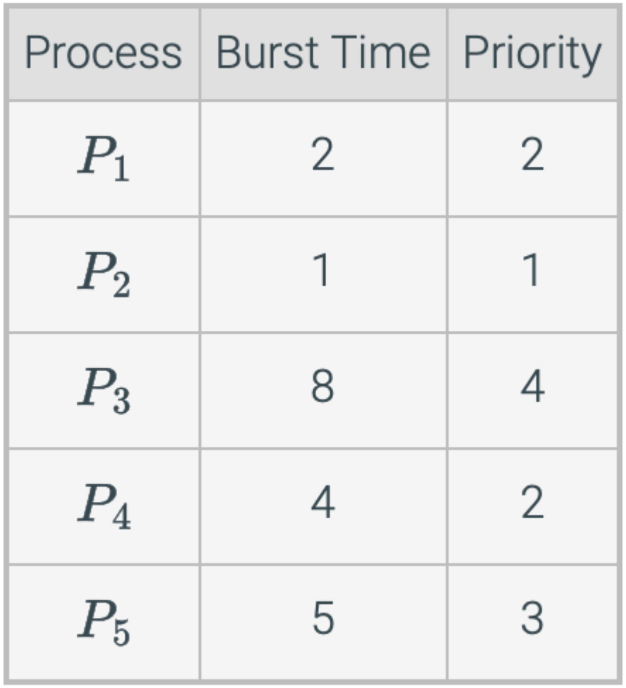
Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (**a larger priority number implies a higher priority**), and RR (quantum = 2)

Gantt Chart



Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

Check your understanding



Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

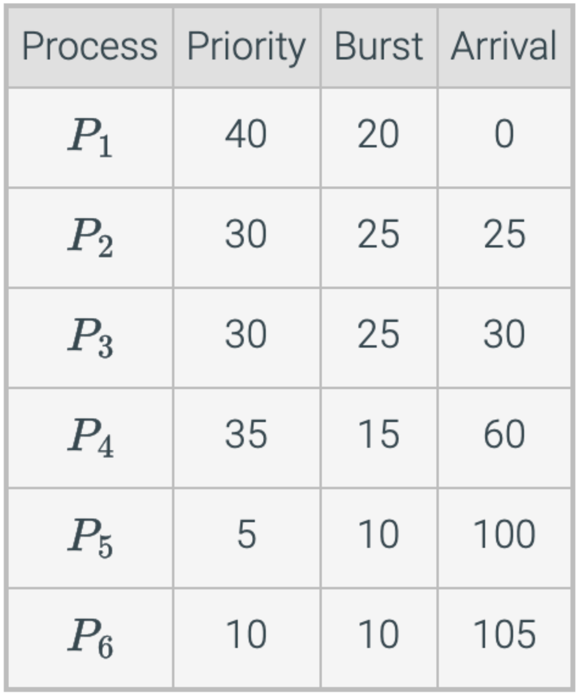
What is the turnaround time of each process for each of the scheduling algorithms in part 1?

What is the waiting time of each process for each of these scheduling algorithms?

Which of the algorithms results in the minimum average waiting time (over all processes)?

Another Practice Exercise

- The following processes are being scheduled using **a preemptive**, **round-robin** scheduling algorithm.



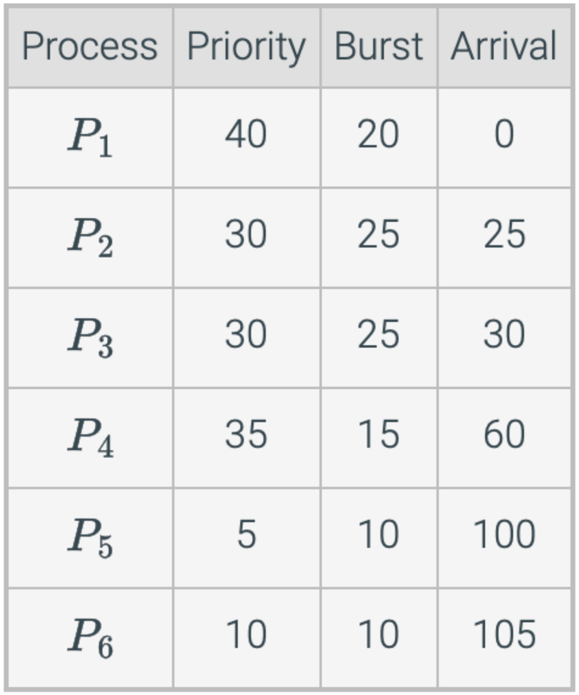
Process	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an **idle task** (which consumes no CPU resources and is identified as . This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a **time quantum is 10 units**. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

- **Show the scheduling order of the processes using a Gantt chart.**

Another Practice Exercise (Cont.)

- The following processes are being scheduled using a preemptive, round-robin scheduling algorithm.



Process	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

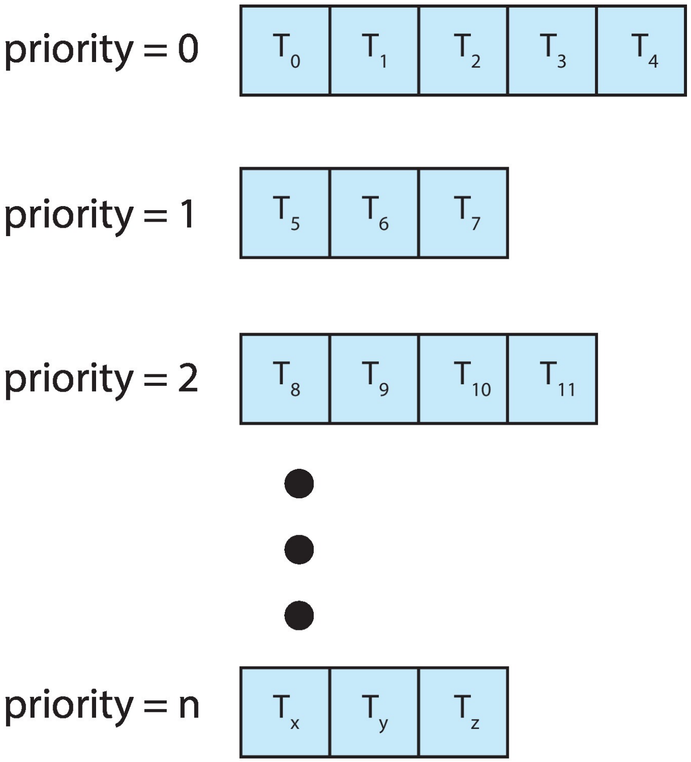
- **What is the turnaround time for each process?**
- **What is the waiting time for each process?**
- **What is the CPU utilization rate?**

Multilevel Queue Scheduling

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues

Multilevel Queue Scheduling (Cont.)

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



priority = 0: T_0 T_1 T_2 T_3 T_4

priority = 1: T_5 T_6 T_7

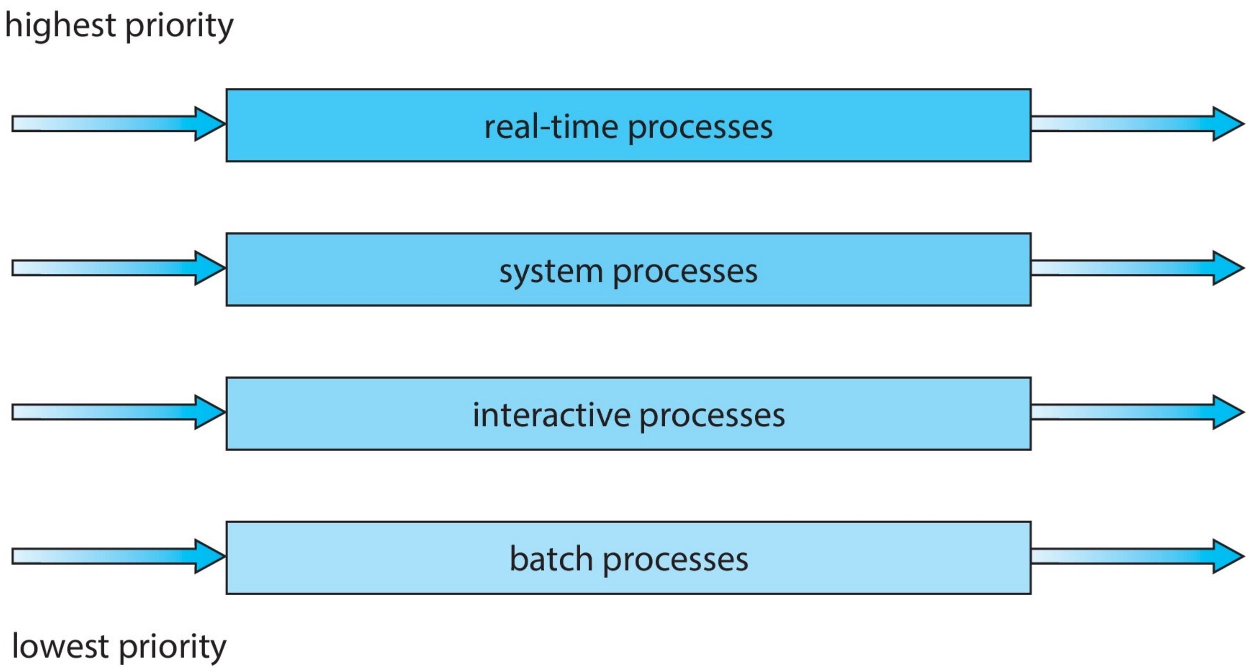
priority = 2: T_8 T_9 T_{10} T_{11}

●
●
●

priority = n: T_x T_y T_z

Multilevel Queue Scheduling (Cont.)

- Prioritization based upon process type



highest priority

real-time processes

system processes

interactive processes

batch processes

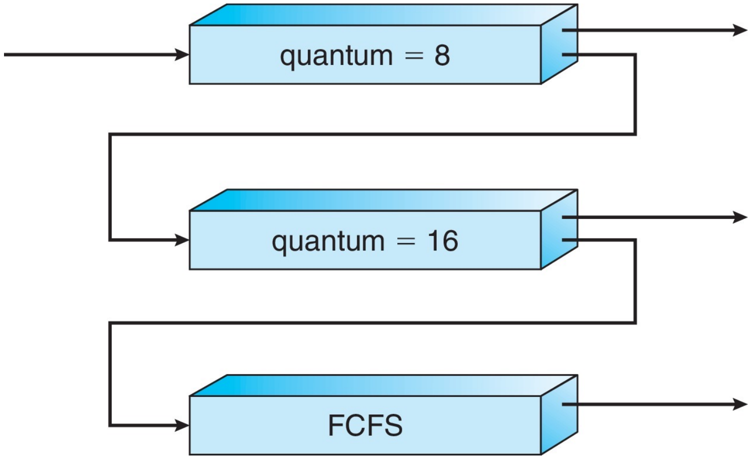
lowest priority

Multilevel Feedback Queue Scheduling

- A process can 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
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue Scheduling

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Recap So Far ..

- Process Scheduling:
  - Non preemptive
  - Preemptive
- General scheduling algorithms:
  - First Come First Served (FCFS)
  - Shortest Job First (SJF)
  - Shortest Remaining Time First
  - Round Robin
  - Priority Scheduling
  - Priority Scheduling with Round Robin
  - **Multilevel Queue Scheduling**
  - **Multilevel Feedback Queue Scheduling**

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP (Light Weight Process)
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM