

# CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2025 Lecture 8

Scheduling



## Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

# Forms of Parallelism

- Pipelining: instruction flows through multiple levels
- Multiple issue: Instruction level Parallelism (ILP)
  - Multiple instructions fetched at the same time
  - Static: compiler scheduling of instructions
  - Dynamic: hardware assisted scheduling of operations
    - “Superscalar” processors
    - CPU decides whether to issue 0, 1, 2, ... instructions each cycle
- Thread or task level parallelism (TLP)
  - Multiple processes or threads running at the same time

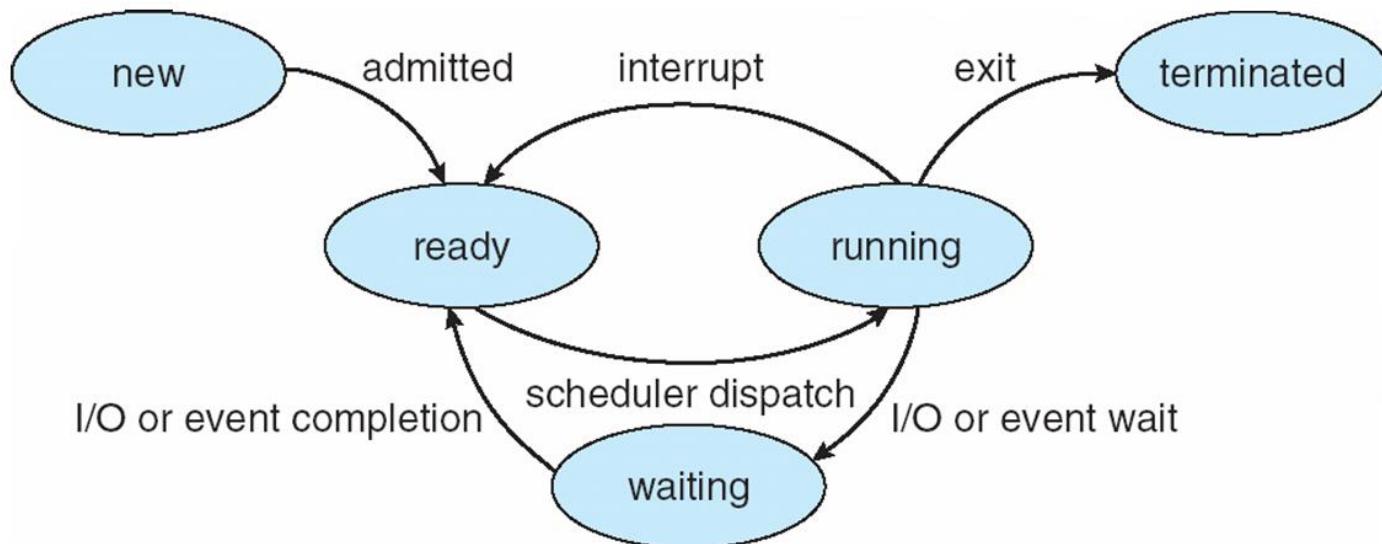
# We have discussed ...

- Multiple threads within a process
- Amdahl's law
- POSIX Pthreads
- Java threads
- Implicit threading including OpenMP
- Signals
- Hyperthreading
- etc

# Chapter 5: CPU Scheduling

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

# Diagram of Process State



Ready to Running: scheduled by scheduler

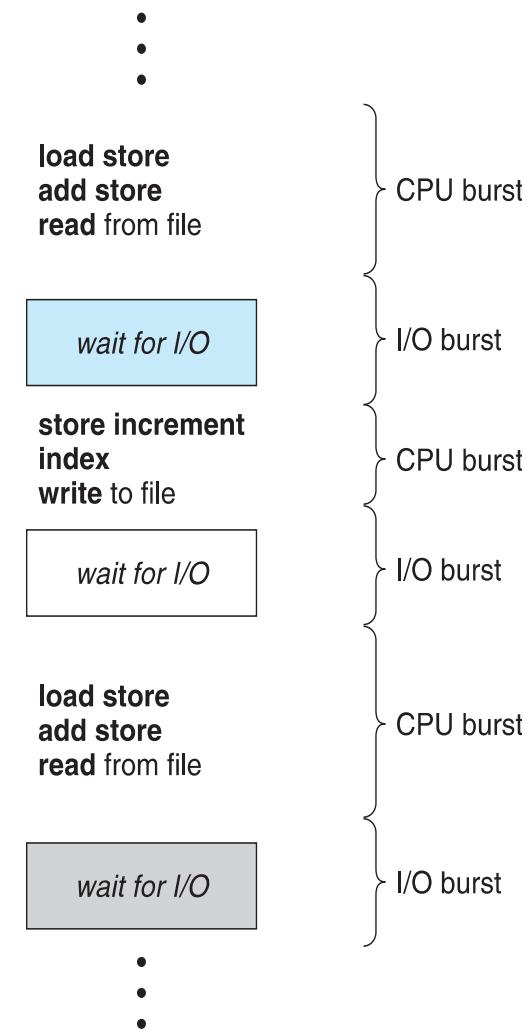
Running to Ready: scheduler picks another process, back in ready queue

Running to Waiting (Blocked) : process blocks for input/output

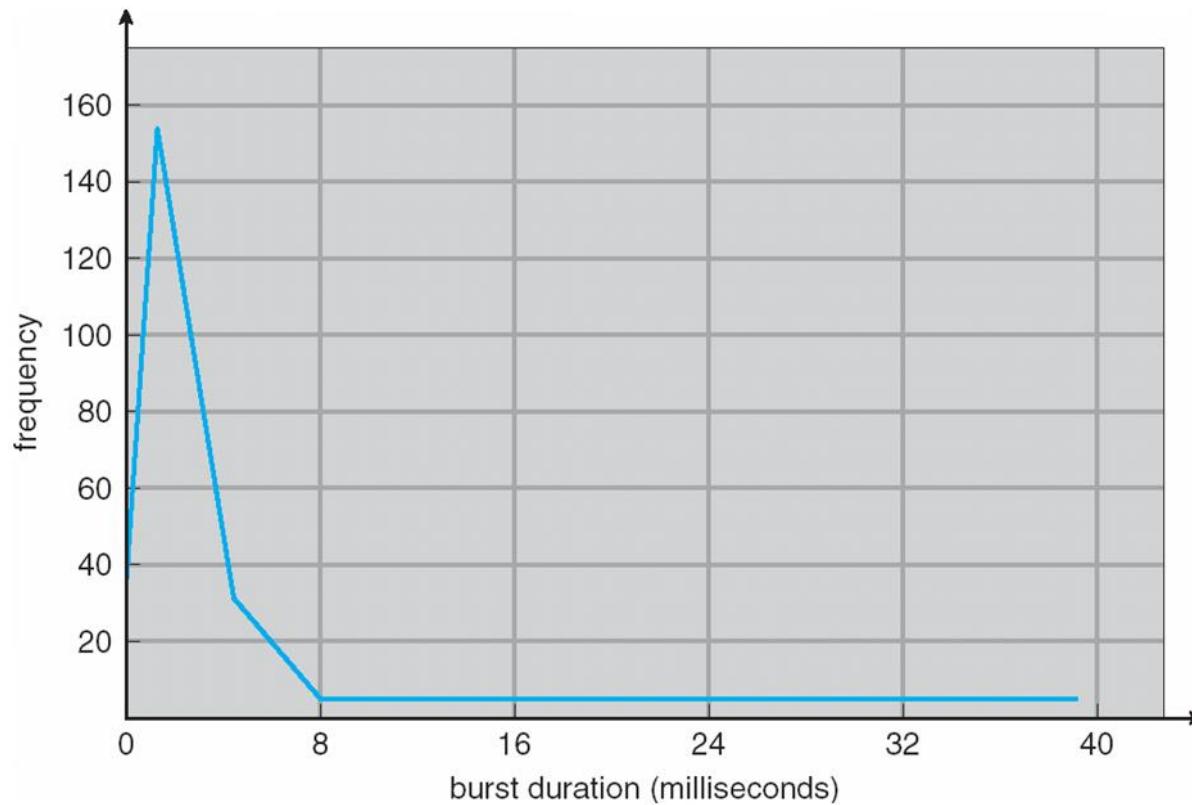
Waiting to Ready: Input available

# 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



# Histogram of CPU-burst Times



Typical distribution of CPU bursts. Most CPU bursts are just a few ms.

# 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
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive. These need to be considered**
  - access to shared data by multiple processes
  - preemption while in kernel mode
  - interrupts occurring during crucial OS activities

Not  
Controlled by  
the process

# 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

# The Dispatcher

(dentist's office)

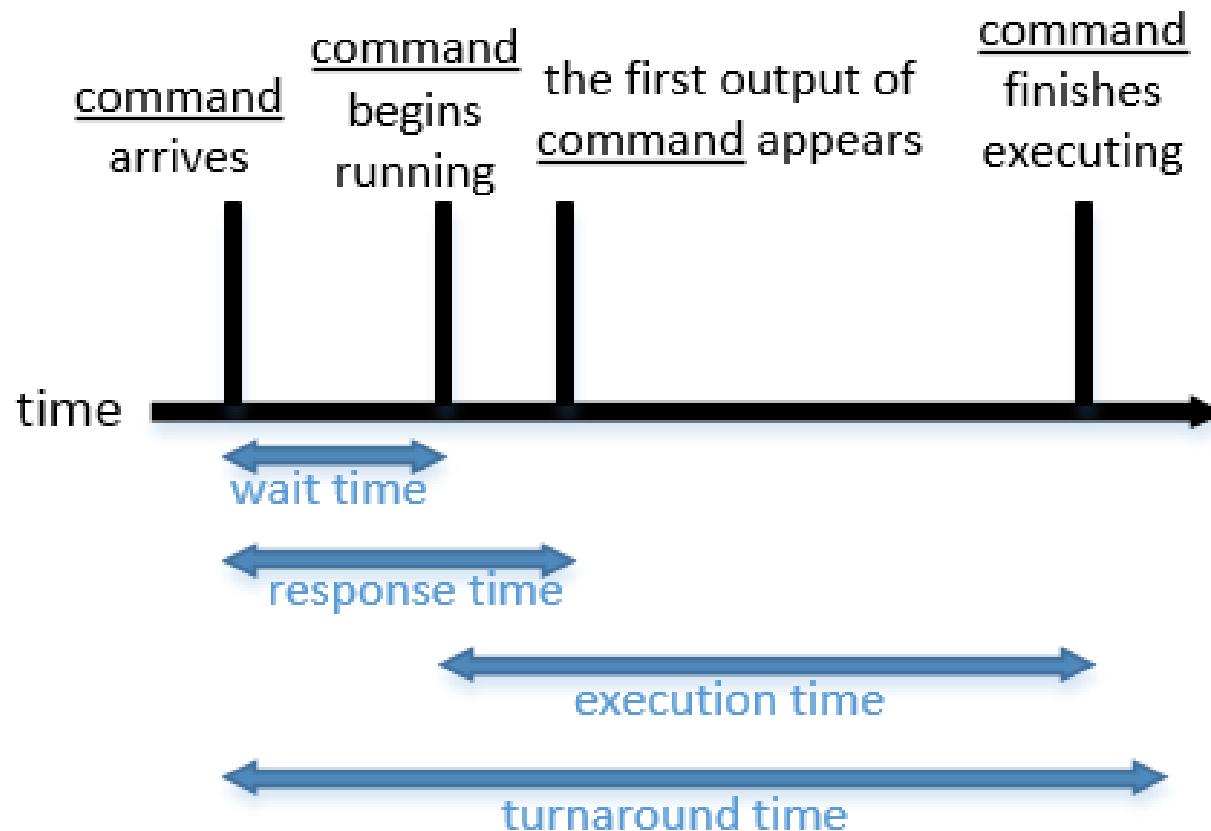


Colorado State University

# Scheduling Criteria

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

# Terms for a single process



# Scheduling Algorithms

We will now examine several major scheduling approaches

- Decide which process in the ready queue is allocated the CPU
- Could be preemptive or nonpreemptive
  - preemptive: remove in middle of execution (“forced”)
- Optimize ***measure*** of interest
  - We will use **Gantt charts** to illustrate *schedules*
  - Bar chart with start and finish times for processes

Involuntary  
deboarding!

# Non-preemptive vs Preemptive scheduling

- **Non-preemptive:** Process keeps CPU until it relinquishes it when
  - It terminates
  - It switches to the waiting state
  - Used by initial versions of OSs like Windows 3.x
- **Preemptive** scheduling
  - Pick a process and let it run for a maximum of some fixed time
  - If it is still running at the end of time interval
    - Suspend it and pick another process to run
- A **clock interrupt** at the end of the time interval to give control back of CPU back to scheduler

# Scheduling Algorithms

## Basic algorithms

- First- Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
  - Shortest-remaining-time-first
- Priority Scheduling
- Round Robin (RR) with time quantum

## Advanced algorithms

- Multilevel Queue
  - Multilevel Feedback Queue
- “Completely fair”

## Comparing Performance

- Average waiting time etc.

Some simplifying assumptions used for clarity.

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

- Process requesting CPU first, gets it first
- Managed with a FIFO queue
  - When process **enters** ready queue
    - PCB is tacked to the **tail** of the queue
  - When CPU is **free**
    - It is allocated to process at the **head** of the queue
- Simple to write and understand

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

Henry Gantt,  
1910s

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$  *but almost the same time.*  
The **Gantt Chart** for the schedule is:



- Waiting time for  $P_1 =$  ;  $P_2 =$  ;  $P_3 =$
- Average waiting time:  $( + + ) / =$
- Throughput:  $/ =$  per unit time

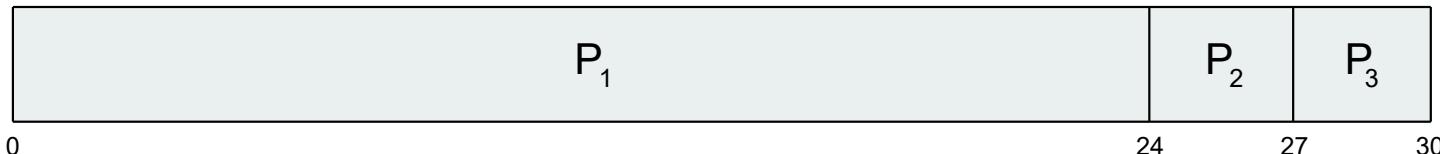
Pause for students to do the computation

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

Henry Gantt,  
1910s

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$  *but almost the same time.*  
The **Gantt Chart** for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$
- Throughput:  $3/30 = 0.1$  per unit 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
- But note -Throughput:  $3/30 = 0.1$  per unit **same**
- **Convoy effect** - short processes behind a long process
  - Consider one CPU-bound and many I/O-bound processes

The Convoy Effect, visualized



# 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
- Reduction in waiting time for short process *GREATER THAN* Increase in waiting time for long process
- 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
  - Estimate or could ask the user



# Example of SJF

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

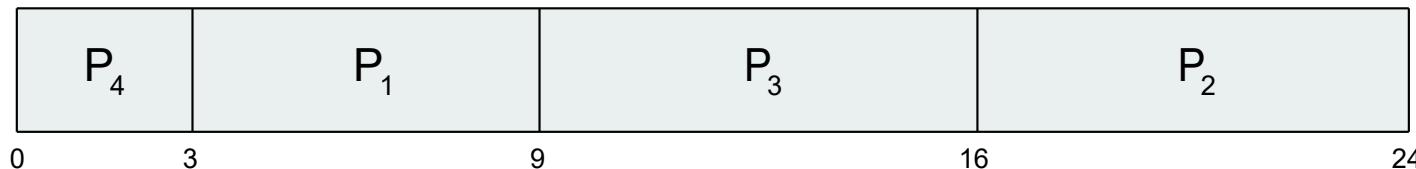
- All arrive at time 0.
- SJF scheduling chart: Draw it here.
- Average waiting time for  $P_1, P_2, P_3, P_4 = ( \quad + \quad + \quad + \quad ) / \quad =$

Pause for students to do the computation

# Example of SJF

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- All arrive at time 0.
- SJF scheduling chart

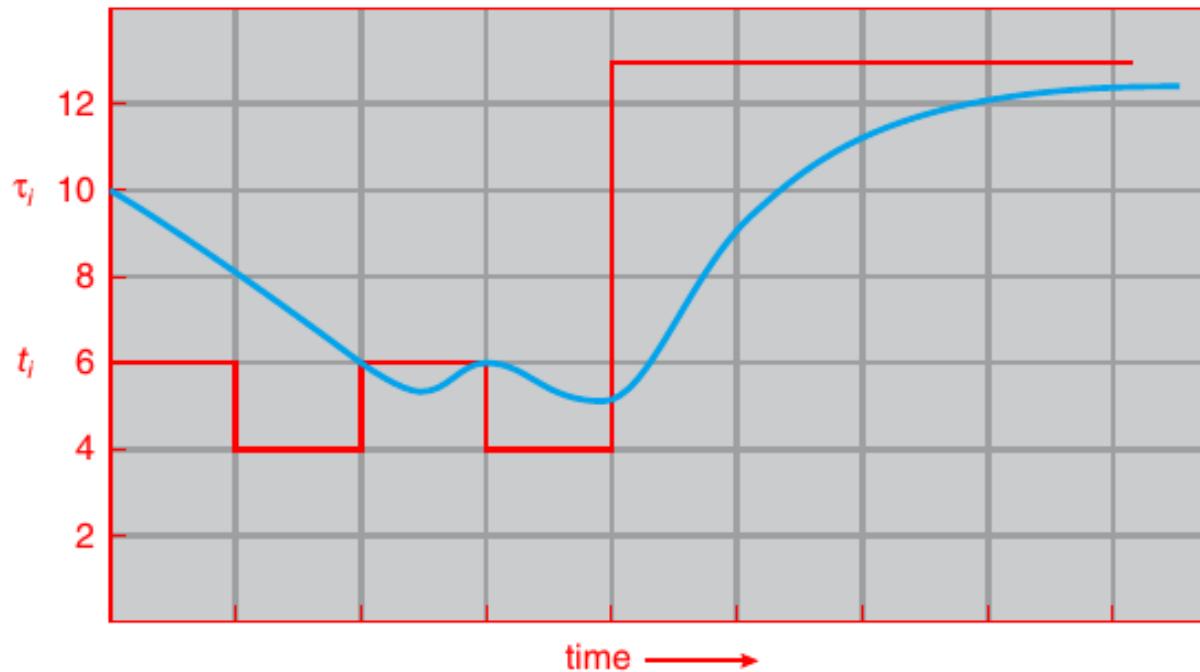


- Average waiting time for  $P_1, P_2, P_3, P_4 = (3 + 16 + 9 + 0) / 4 = 7$

# Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the recent bursts
  - 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,  $\alpha$  set to  $\frac{1}{2}$

# 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	...

# 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
- $$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$
- If we expand the formula, substituting for  $\tau_n$ , we get:
$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

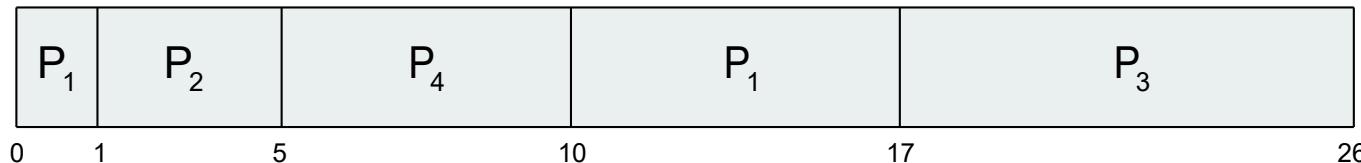
Widely used for predicting stock-market etc

# Shortest-remaining-time-first (preemptive SJF)

- Preemptive version called **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 (will preempt because 4<7)
$P_3$	2	9 (will not preempt)
$P_4$	3	5

- *Preemptive SJF Gantt Chart*



- Average waiting time for P1,P2,P3,P4  
$$= [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 \text{ msec}$$

# Priority Scheduling

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



MIT had a low priority job waiting from 1967 to 1973 on IBM 7094! ☺

# Ex Priority Scheduling

non-preemptive

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

- P1, P2, P3, P4, P5 all arrive at time 0.
- Priority scheduling Gantt Chart



- Average waiting time for P1, .. P5:  $(6+0+16+18+1)/5 = 8.2$  msec

Variation: Priority scheduling with preemption

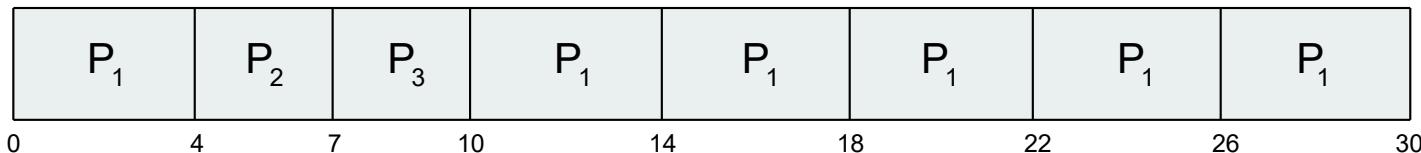
# Round Robin (RR) with time quantum

- Each process gets a small unit of CPU time (**time quantum  $q$** ), usually **10-100** milliseconds. After this, the process is preempted, 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  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high (**overhead typically in 0.5% range**)

# Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- Arrive a time 0 in order  $P_1, P_2, P_3$ : The Gantt chart is:



- Waiting times:  $P_1:10-4 = 6$ ,  $P_2:4$ ,  $P_3:7$ , average  $17/3 = 5.66$  units
- 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  $\mu$ sec

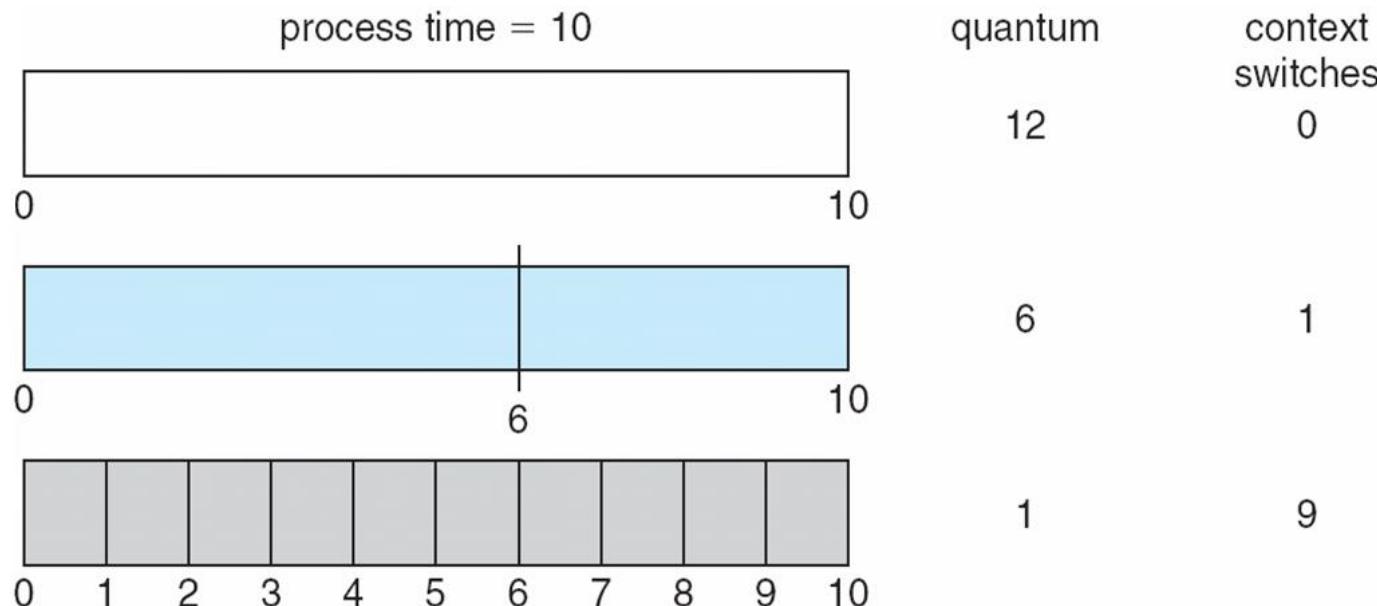
Response time: Arrival to beginning of execution  
Turnaround time: Arrival to finish of execution

# RR: different arrival times

Process at the head of the Ready Queue is scheduled first. You must track the Ready Queue.

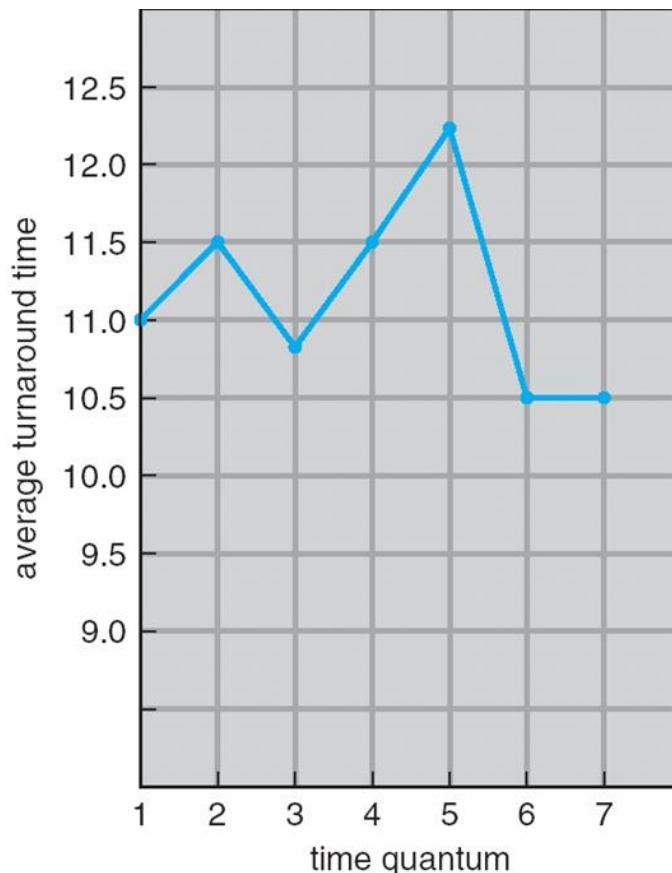
- When a process is switched out, it gets into the Ready Queue.
- When a new process arrives, it gets into the Ready Queue.
- When a process A gets switched out and a new process B arrives at the same time, which one gets into the Ready Queue first?
  - Assume the new process is placed first in the ready queue.

# Time Quantum and Context Switch Time



Much smaller quantum compared to burst: many switches

# Turnaround Time Varies With The Time Quantum



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

**Rule of thumb:** 80% of CPU bursts should be shorter than  $q$

Illustration

Consider  $q=7$ :

$P_1, P_2, P_3, P_4$ : all arrive at time 0 in this order.

Turnaround times for  $P_1, P_2, P_3, P_4$ :  
6, 9, 10, 17    $av = 10.5$

Similarly for  $q = 1, \dots, 6$  (verify yourself)

Students: Repeat for  $q = 1, \dots, 6$  at home to verify the plot.

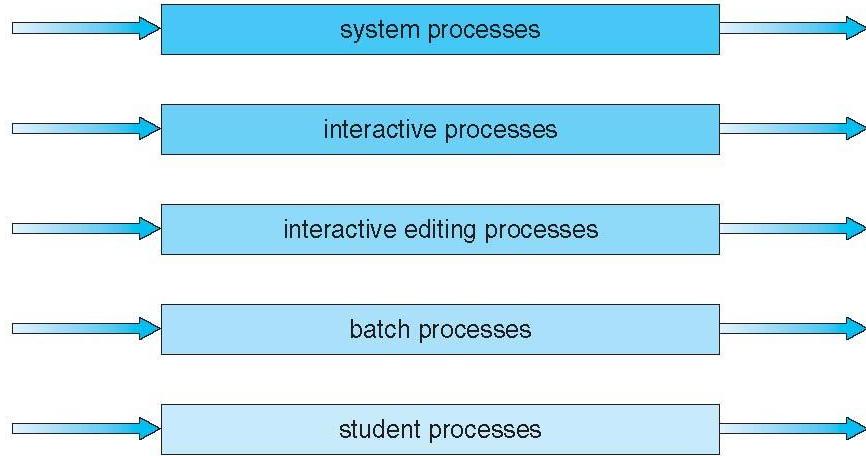
Turnaround time: Arrival to finish of execution

# Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm, e.g.:
  - 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. Or
  - 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

highest priority



lowest priority

Real-time processes may have the highest priority.



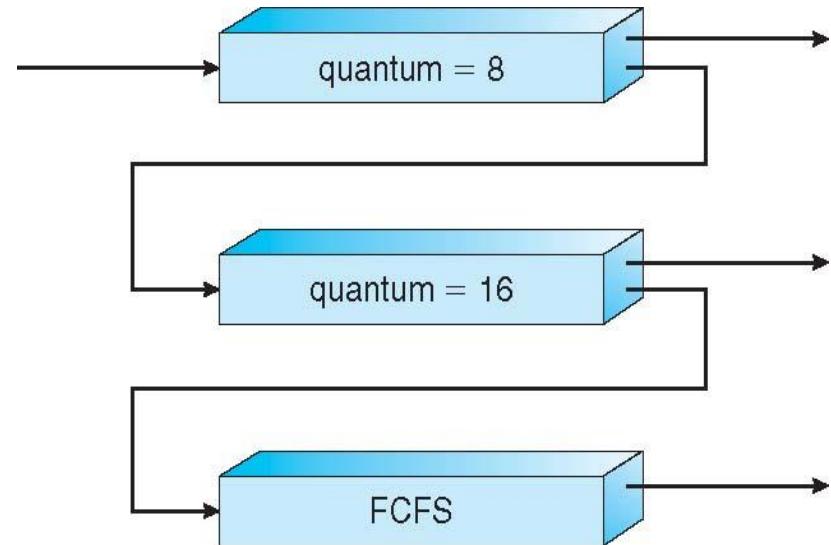
# Multilevel *Feedback* Queue

- 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
  - [Details at ARPACI-DUSSEAU](#)

Inventor FJ Corbató won the Touring award!

# 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 (no time quantum limit)
- 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  $Q_2$



Upgrading may be based on aging. Periodically processes may be moved to the top level.

Variations of the scheme were used in earlier versions of Linux.

# Completely fair scheduler Linux 2.6.23

Goal: fairness in dividing processor time to tasks ([Con Kolivas, Anaesthetist](#))

- Variable time-slice based on number and priority of the tasks in the queue.
  - Maximum execution time based on waiting processes ( $Q/n$ ).
  - Fewer processes waiting, they get more time each
- Queue ordered in terms of “virtual run time”
  - execution time on CPU added to value
  - smallest value picked for using CPU
  - small values: tasks have received less time on CPU
  - I/O bound tasks (shorter CPU bursts) will have smaller values
- *Balanced (red-black) tree* to implement a ready queue;
  - Efficient.  $O(\log n)$  insert or delete time
- Priorities (*niceness*) cause different decays of values: higher priority processes get to run for longer time
  - virtual run time is the weighted run-time

Scheduling schemes have continued to evolve with continuing research. [A comparison.](#)