Process and Thread Scheduling

Raju Pandey
Department of Computer Sciences
University of California, Davis
Winter 2005

Scheduling

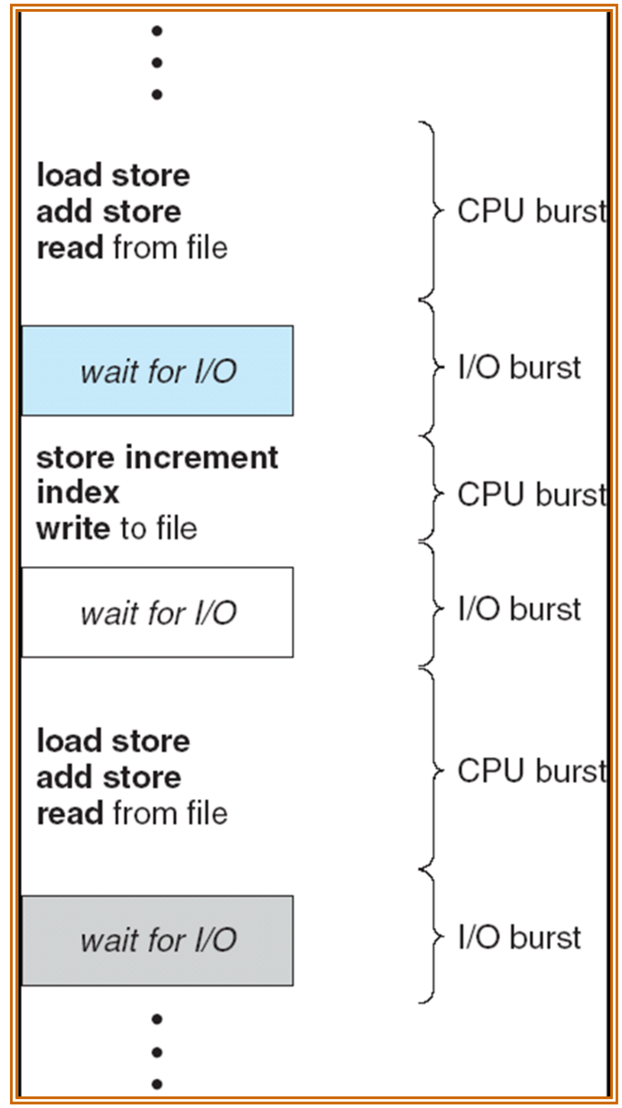
- Context switching
  - an interrupt occurs (device completion, timer interrupt)
  - a thread causes a trap or exception
  - may need to choose a different thread/process to run
- We glossed over the choice of which process or thread is chosen to be run next
  - “some thread from the ready queue”
- This decision is called scheduling
  - o scheduling is a policy
  - o context switching is a mechanism

Objectives

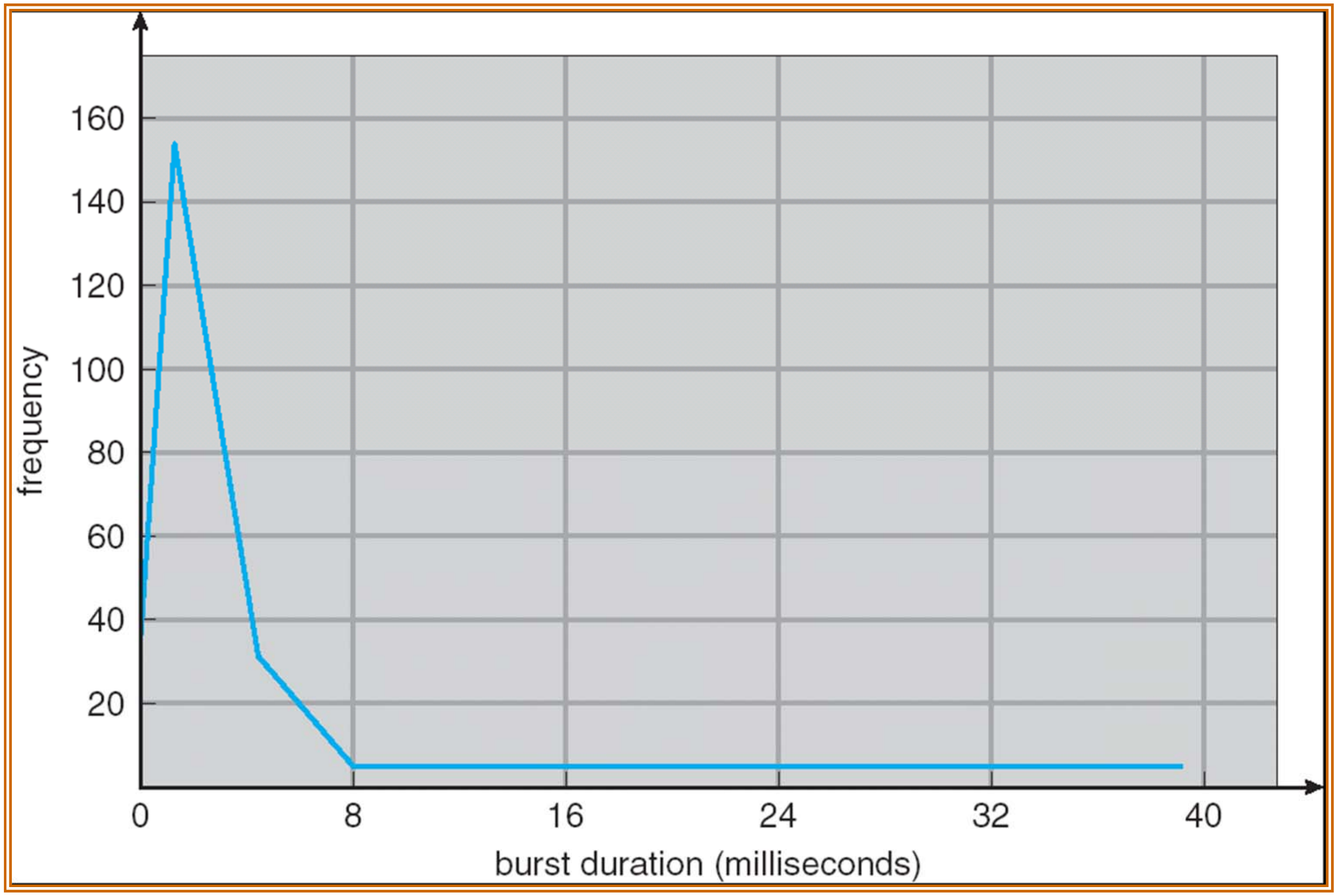
- After this lecture, you should understand:
  - the goals of scheduling.
  - preemptive vs. non-preemptive scheduling.
  - the role of priorities in scheduling.
  - scheduling criteria.
  - common scheduling algorithms.

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 distribution



Histogram of CPU-burst Times
Exponential/HyperExponential



I/O bound: Many short cpu bursts
CPU bound: few very long cpu bursts

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, **not** output (for time-sharing environment)

Scheduling Objectives

- Different objectives depending on system
  - Maximize throughput
  - Maximize number of interactive processes receiving acceptable response times
  - Minimize resource utilization
  - Avoid indefinite postponement
  - Enforce priorities
  - Minimize overhead
  - Ensure predictability
- Several goals common to most schedulers
  - Fairness
  - Predictability
  - Scalability

Scheduling Objectives: Fairness

- No single, compelling definition of “fair”
  - How to measure fairness?
    - o Equal CPU consumption? (over what time scale?)
  - Fair per-user? per-process? per-thread?
  - What if one process is CPU bound and one is I/O bound?
- Sometimes the goal is to be unfair:
  - Explicitly favor some particular class of requests (priority system), but…
  - avoid starvation (be sure everyone gets at least some service)

Preemptive vs. Nonpreemptive Scheduling

- Preemptive processes
  - Can be removed from their current processor
  - Can lead to improved response times
  - Important for interactive environments
  - Preempted processes remain in memory
- Nonpreemptive processes
  - Run until completion or until they yield control of a processor
  - Unimportant processes can block important ones indefinitely

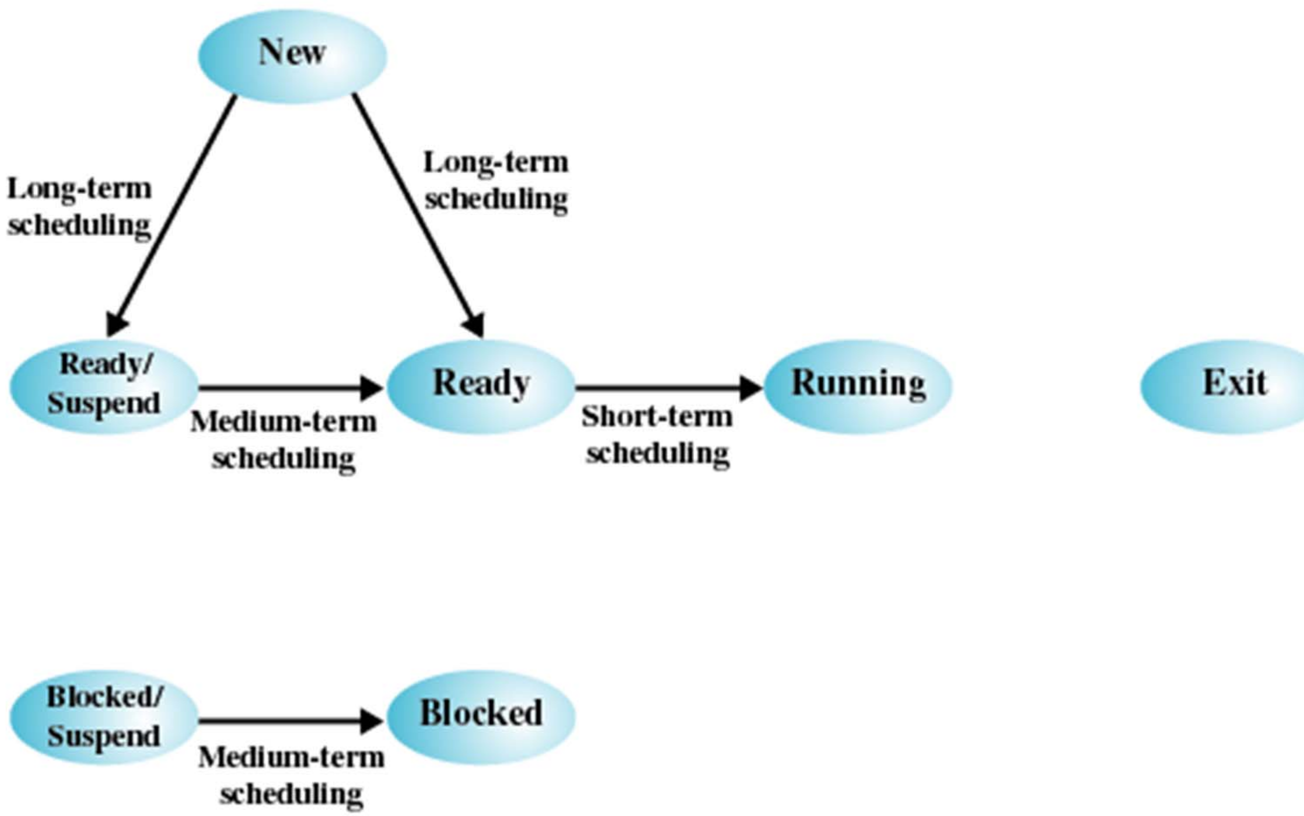
Priorities

- Static priorities
  - Priority assigned to a process does not change
  - Easy to implement
  - Low overhead
  - Not responsive to changes in environment
- Dynamic priorities
  - Responsive to change
  - Promote smooth interactivity
  - Incur more overhead than static priorities
    - Justified by increased responsiveness

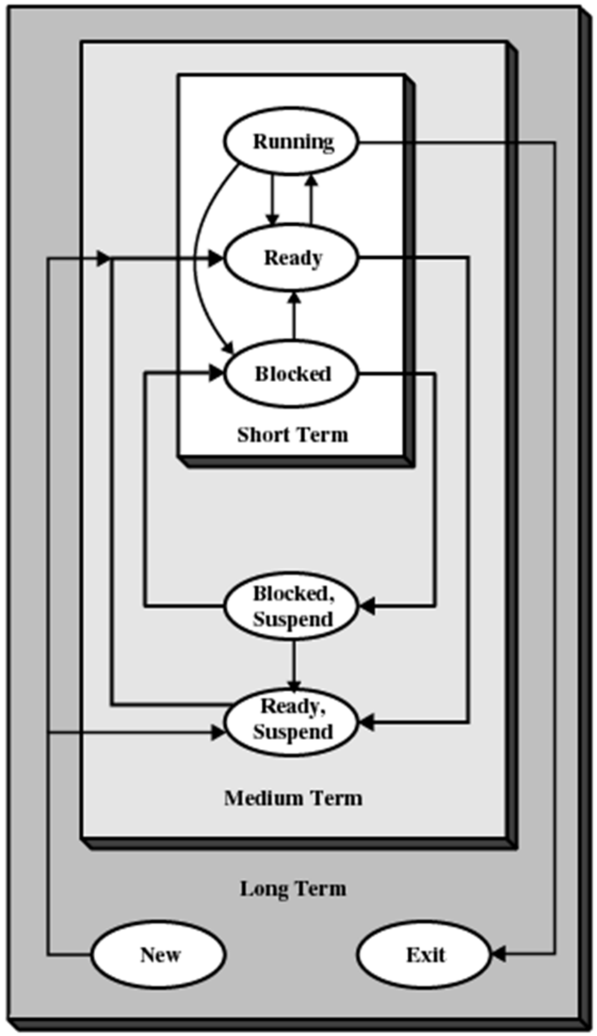
Multiple levels of scheduling decisions

- Long term
  - Should a new “job” be “initiated,” or should it be held?
    - typical of batch systems
    - what might cause you to make a “hold” decision?
- Medium term
  - Should a running program be temporarily marked as non-runnable (e.g., swapped out)?
- Short term
  - Which thread should be given the CPU next? For how long?
  - Which I/O operation should be sent to the disk next?
  - On a multiprocessor:
    - should we attempt to coordinate the running of threads from the same address space in some way?
    - should we worry about cache state (processor affinity)?

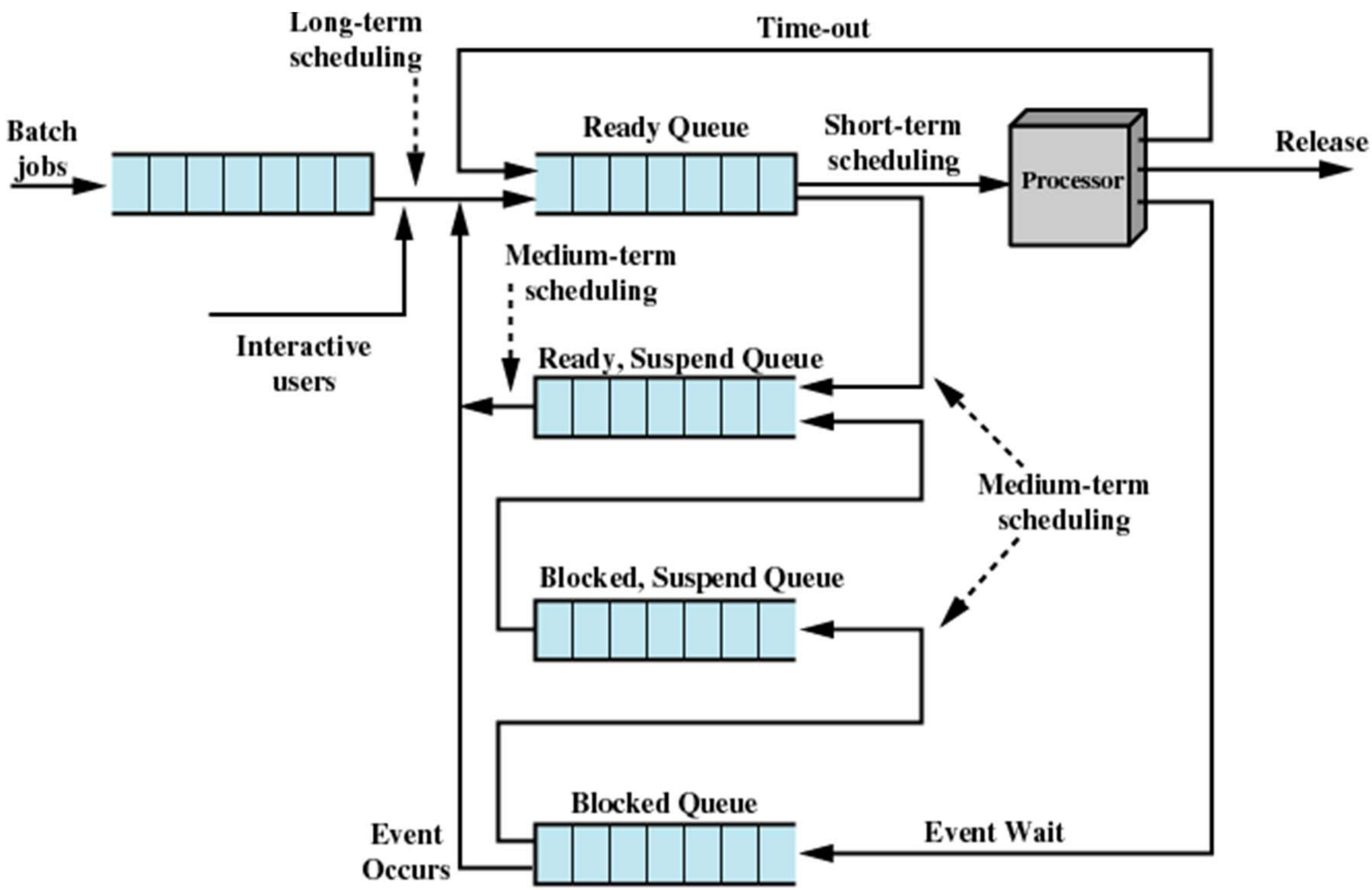
Scheduling and Process State Transition



Levels of scheduling



Queuing Diagram for Scheduling



Scheduling levels

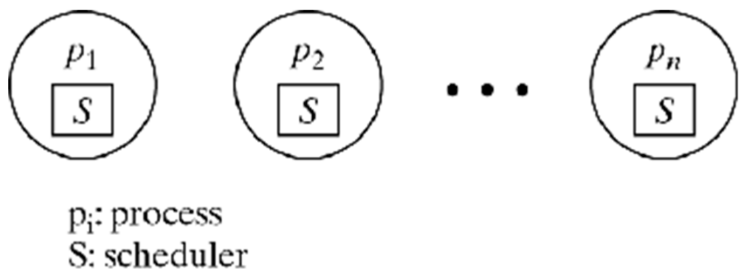
- Long term scheduling:
  - Determines which programs are admitted to the system for processing
  - Controls the degree of multiprogramming
  - More processes, smaller percentage of time each process is executed
- Midterm scheduling:
  - Part of the swapping function
  - Based on the need to manage the degree of multiprogramming
- Short term scheduling:
  - Known as the dispatcher
  - Executes most frequently
  - Invoked when an event occurs
    - Clock interrupts
    - I/O interrupts
    - Operating system calls
    - Signals

Dispatcher

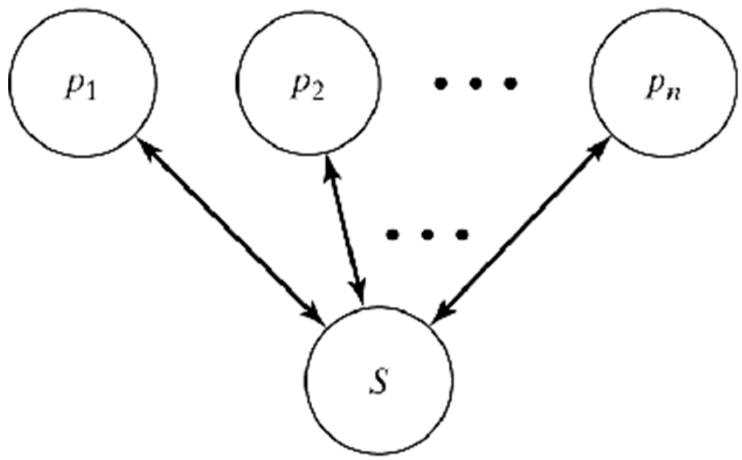
- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

Organization of Schedulers

- Embedded
  - Called as function at end of kernel call
  - Runs as part of calling process
- Autonomous
  - Separate process
  - May have dedicated CPU on a multiprocessor
  - On single-processor, run at every quantum: scheduler and other processes alternate



(a)



(b)

Framework for Scheduling

- **When** is scheduler invoked?
  - **Decision mode**
    - o **Preemptive**: scheduler called periodically (quantum-oriented) or when system state changes
    - o **Nonpreemptive**: scheduler called when process terminates or blocks
- **How** does it select highest priority process?
  - **Priority function**:

    P = Priority(p)
  - **Arbitration rule**: break ties
    - o Random
    - o Chronological (First In First Out = FIFO)
    - o Cyclic (Round Robin = RR)

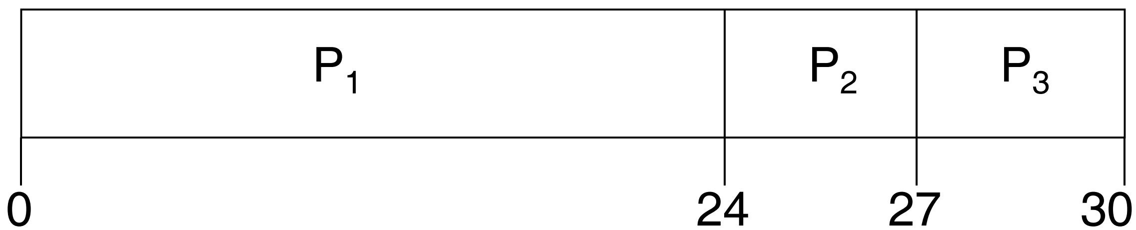
Priority function

- Different ways to determine priority
- Possible attributes of processes used to define priority:
  - Attained service time (***a***): amount of CPU time allocated
  - Real time in system (***r***): attained time + waiting time
  - Total service time (***t***): total time between arrival and departure
  - Periodicity (***d***): repetition of a computation
  - Deadline (explicit or implied by periodicity): Point in real-time by which process must be completed
  - External priority (***e***)
  - Memory requirements (mostly for batch)
  - System load (not process-specific)

First-Come, First-Served (FCFS) Scheduling
First In Firtst Out

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:



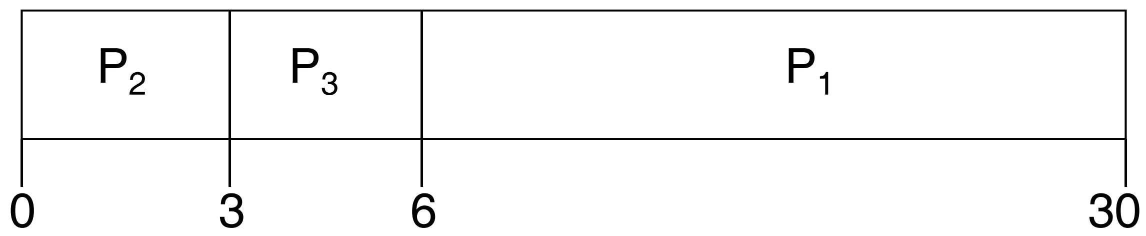
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Priority function = r (r = arrival time)
- Decision mode: non-preemptive

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

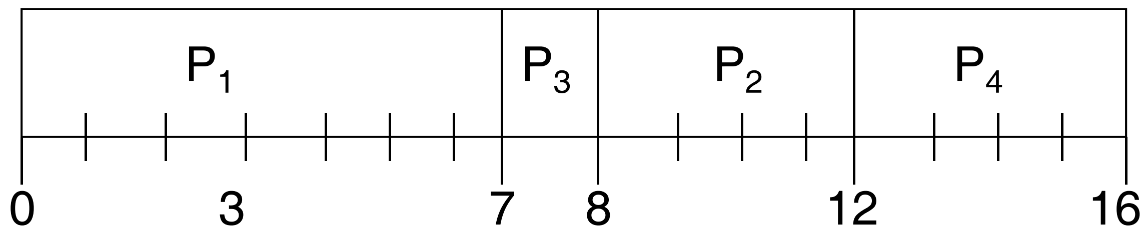
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
    - Priority Function = -total service time
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
    - Priority function = -(t-a), t = total service time; a = total attained service time
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

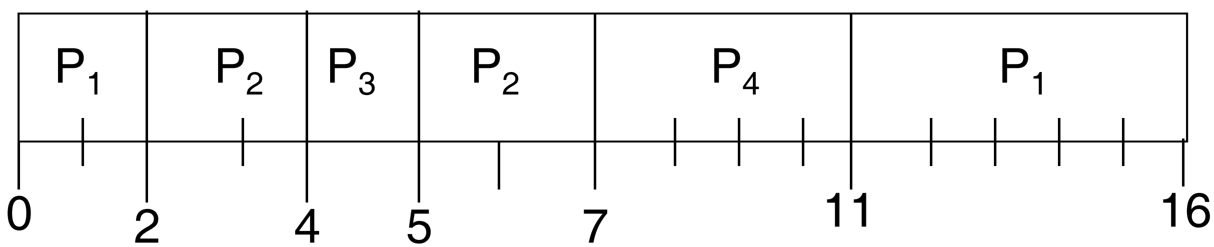


- Average waiting time = (0 + 6 + 3 + 7)/4 - 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



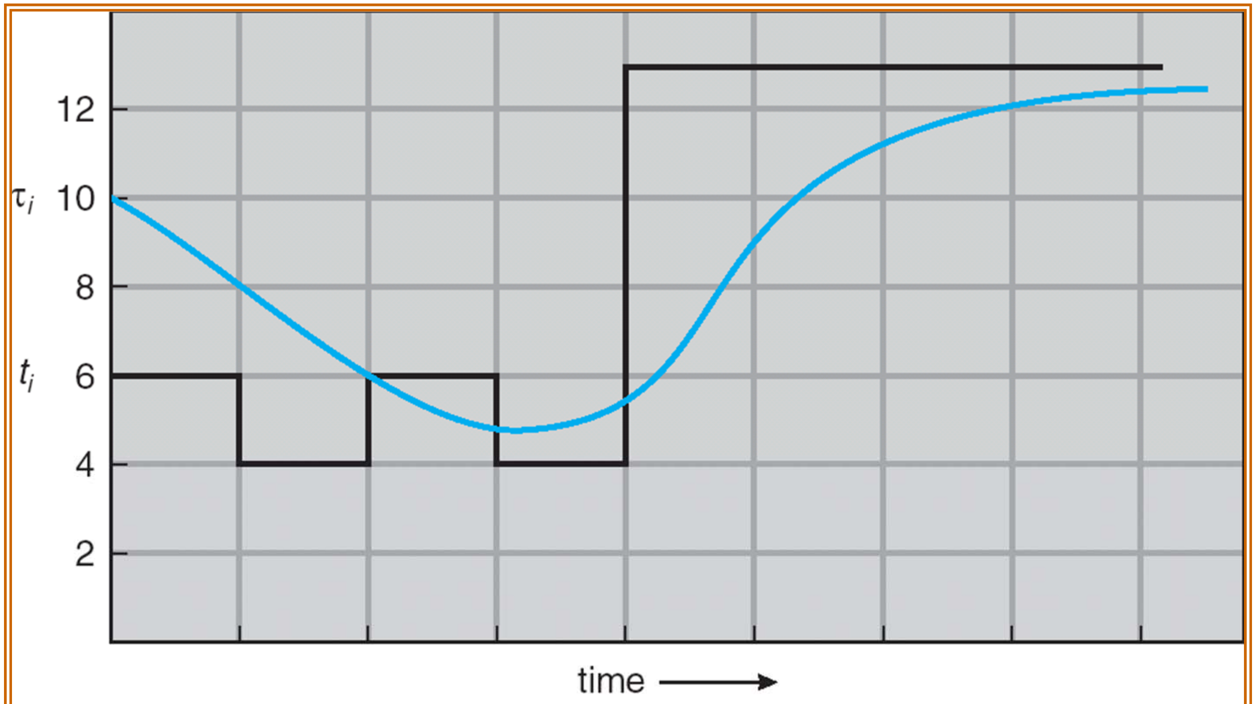
- Average waiting time = (9 + 1 + 0 +2)/4 - 3

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

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	…

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 a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), 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.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high
- Priority function
  - All processes have same priority

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

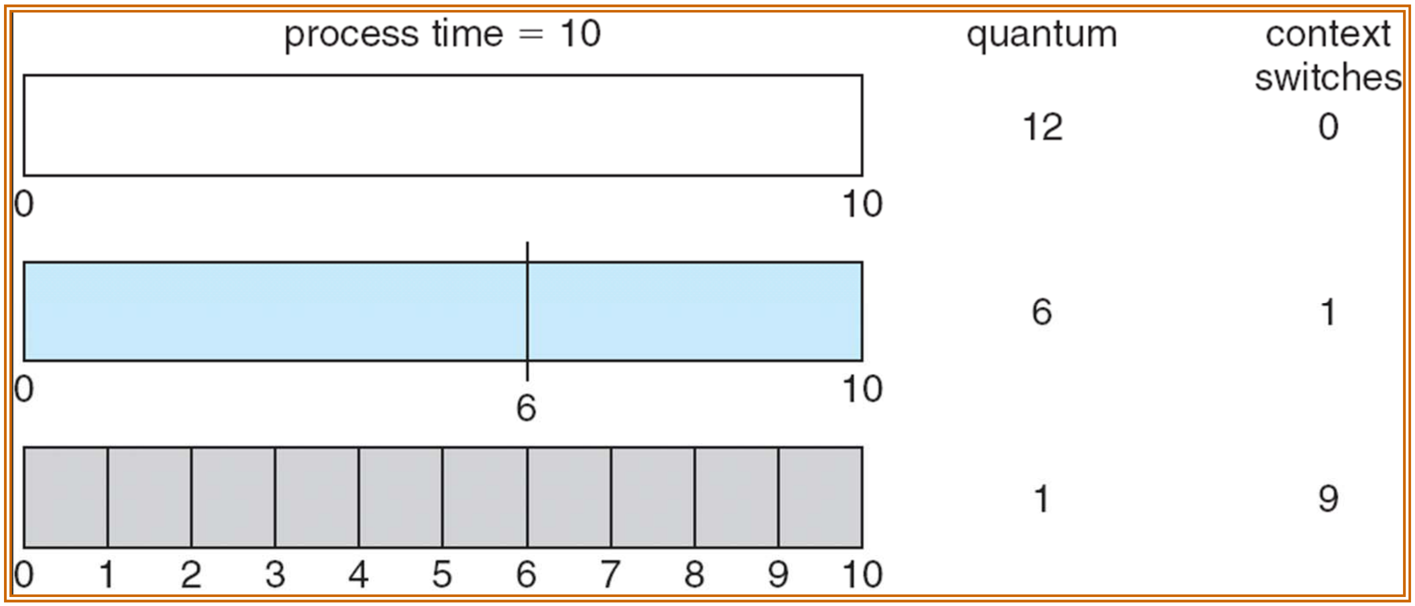
- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

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

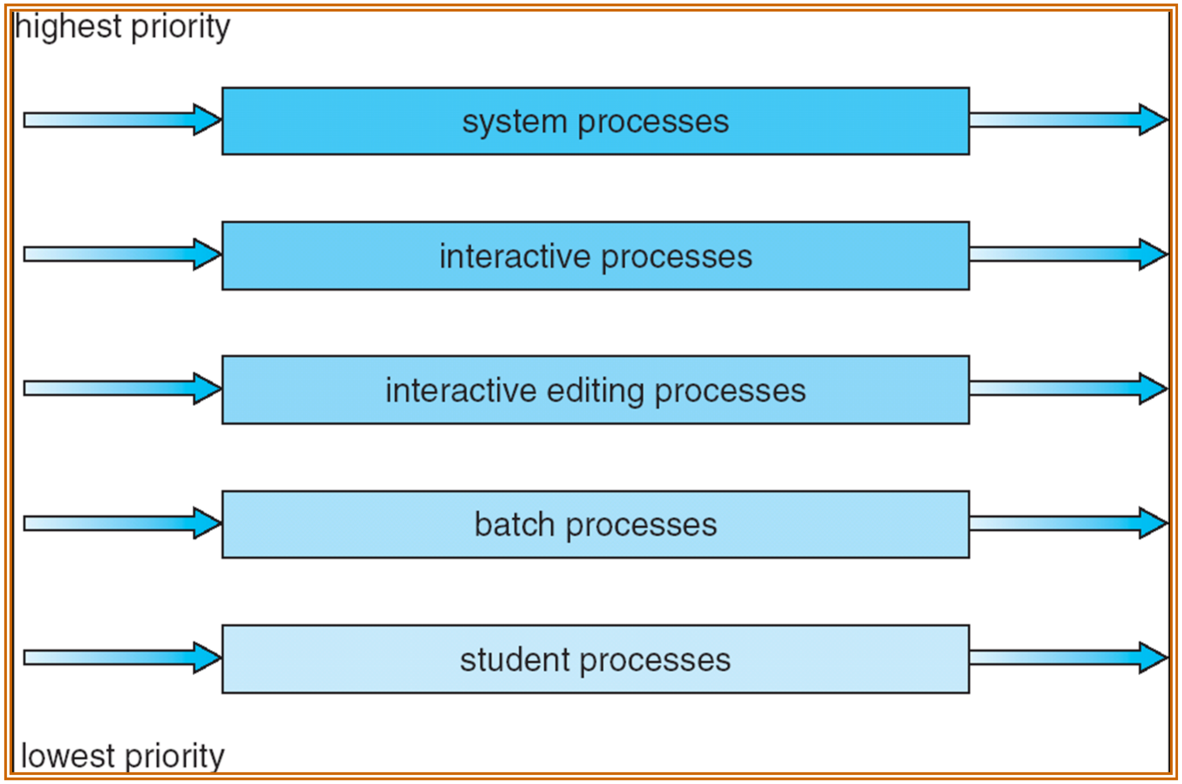
Time Quantum and Context Switch Time



Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- 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



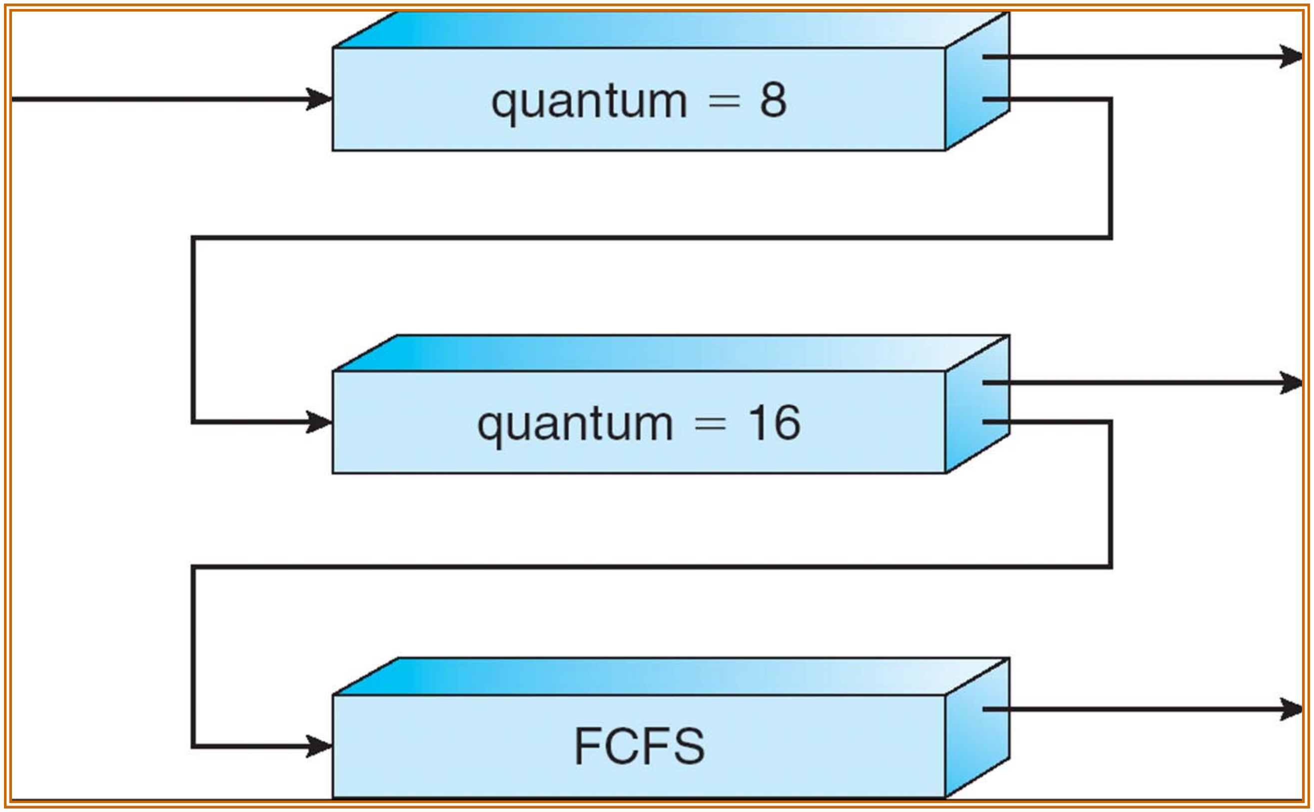
Multilevel Feedback Queue

- 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 – time quantum 8 milliseconds
  - Q_1 – 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.

Multilevel Feedback Queues



Scheduling Algorithms: Real-time systems

- **Real-time systems:** Periodic in nature so that computations are repeated at fixed intervals
- Typically: a process has a period of d, it is activated every d seconds, and its computation (total service time) must be completed before start of the next period
- **Rate Monotonic (RM):**
  - d:periodicity
  - Preemptive
  - Highest priority: shortest period: $P = -d$

Scheduling Algorithms: Real-time systems

- **Earliest Deadline First (EDF):**
  - Intended for periodic (real-time) processes
  - Preemptive
  - r = time since the process first entered the system
  - d = Periodicity
  - Highest priority: shortest time to next deadline
    - o $r \div d$ — number of completed periods
    - o $r \% d$ — time in current period
    - o $d - r \% d$ — time remaining in current period
    - o $P = -(d - r \% d)$

Scheduling algorithms

Name	Decision Mode	Priority	Arbitration
FIFO	Nonpreemptive	$P = r$	random
Shortest Job First (SJF)	Nonpreemptive	$P = -t$	
Shortest Remaining Time (SRT)	Preemptive	$P = -(t-a)$	Chronological random
Round Robin (RR)	Preemptive	$P=0$	Cyclic
Multi Level Priority (MLF)	Preemptive	$P = e$	cyclic
	Non-preemptive	$P = e$	chronological
Rate Monotonic	Pre-emptive	$-d$	Chronological Random
Earliest Deadline First	Pre-emptive	$-(d-r\%d)$	Chronological Random

Comparison of Methods

- FIFO, SJF, SRT: Primarily for batch systems
  - FIFO simplest, SJF & SRT have better average turnaround times: ***(r1+r2+…+rn)/n***
- Time-sharing systems
  - Response time is critical
  - RR or MLF with RR within each queue are suitable
  - Choice of quantum determines overhead
    - o When q $\rightarrow \infty$, RR approaches FIFO
    - o When q $\rightarrow 0$, context switch overhead $\rightarrow 100\%$
    - o When q >> context switch overhead, n processes run concurrently at 1/n CPU speed

Priority Inversion Problem

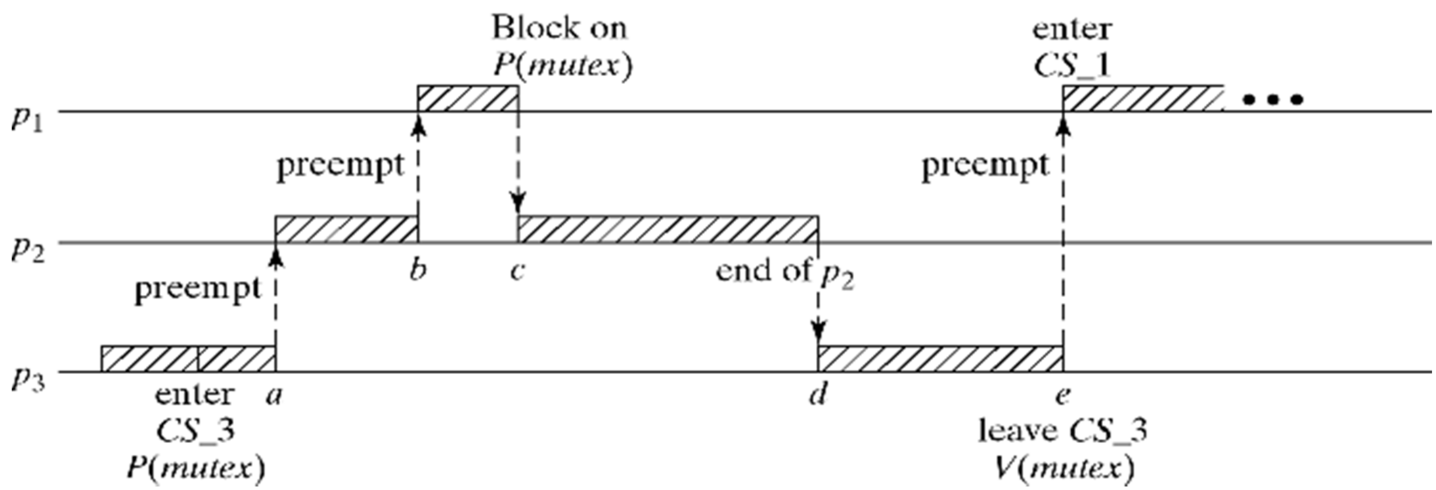


Figure 5-10

- Assume priority order ***p1>p2>p3***
- (Unrelated) ***p2*** may delay ***p1*** indefinitely.
- Naïve "solution": Always run CS at priority of highest process that shares the CS.
  Problem: ***p1*** cannot interrupt lower-priority process inside CS -- a different form of priority inversion.

Priority Inversion Problem

- Solution: Dynamic Priority Inheritance
  - ***p3*** is in its CS
  - ***p1*** attempts to enter its CS
  - ***p3*** inherits ***p1***'s (higher) priority for the duration of CS

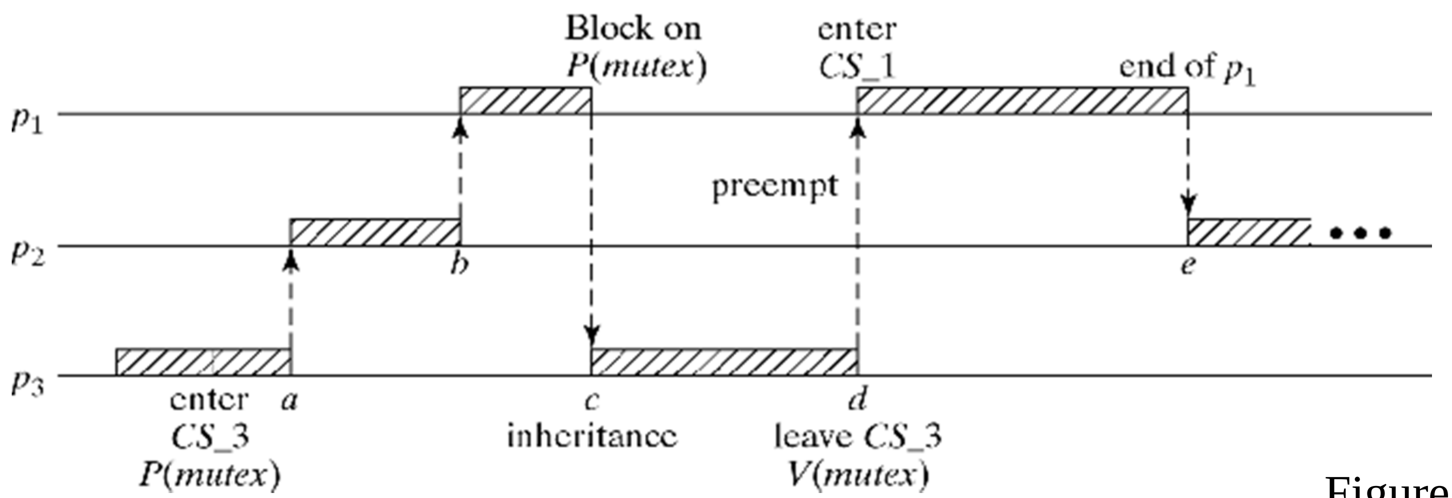


Figure 5-11

Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
  - this difference increases with the variability in service requirements
- Multiple goals, sometimes conflicting
- There are many “pure” algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids that exploit observed program behavior