6. CPU Scheduling

Sungyoung Lee

College of Engineering
KyungHee University

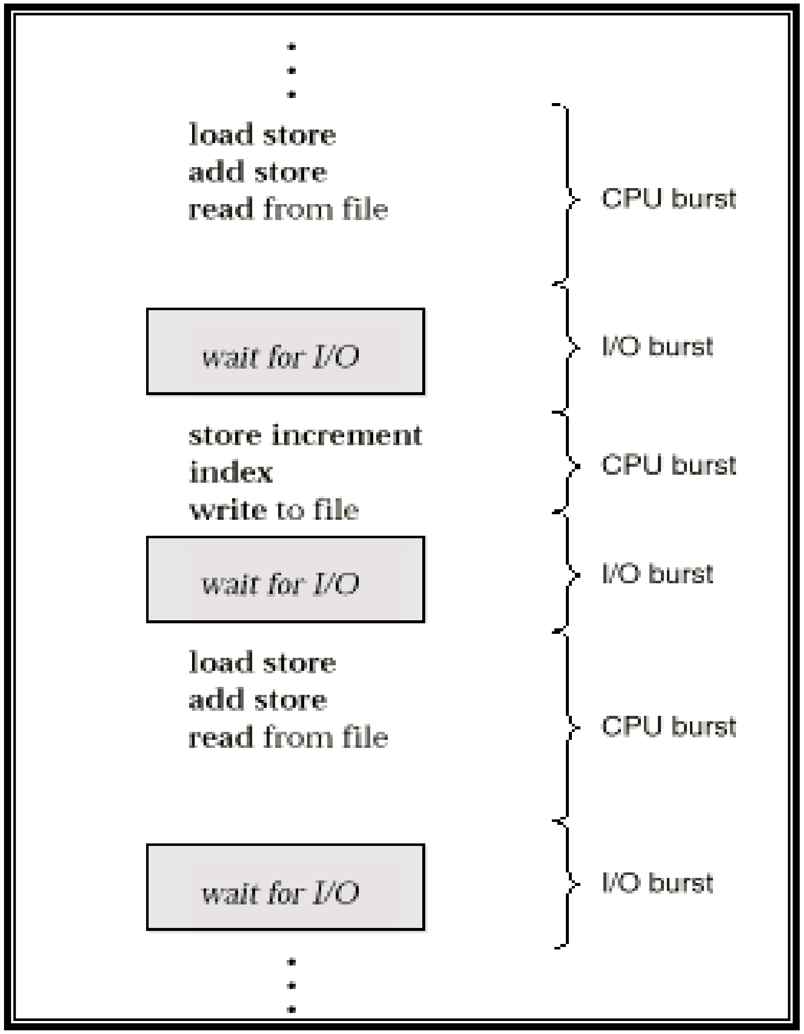
Contents

- *Basic Concepts*
- *Scheduling Criteria*
- *Scheduling Algorithms*
- *Multiple-Processor Scheduling*
- *Real-Time Scheduling*
- *Algorithm Evaluation*

Basic Concepts

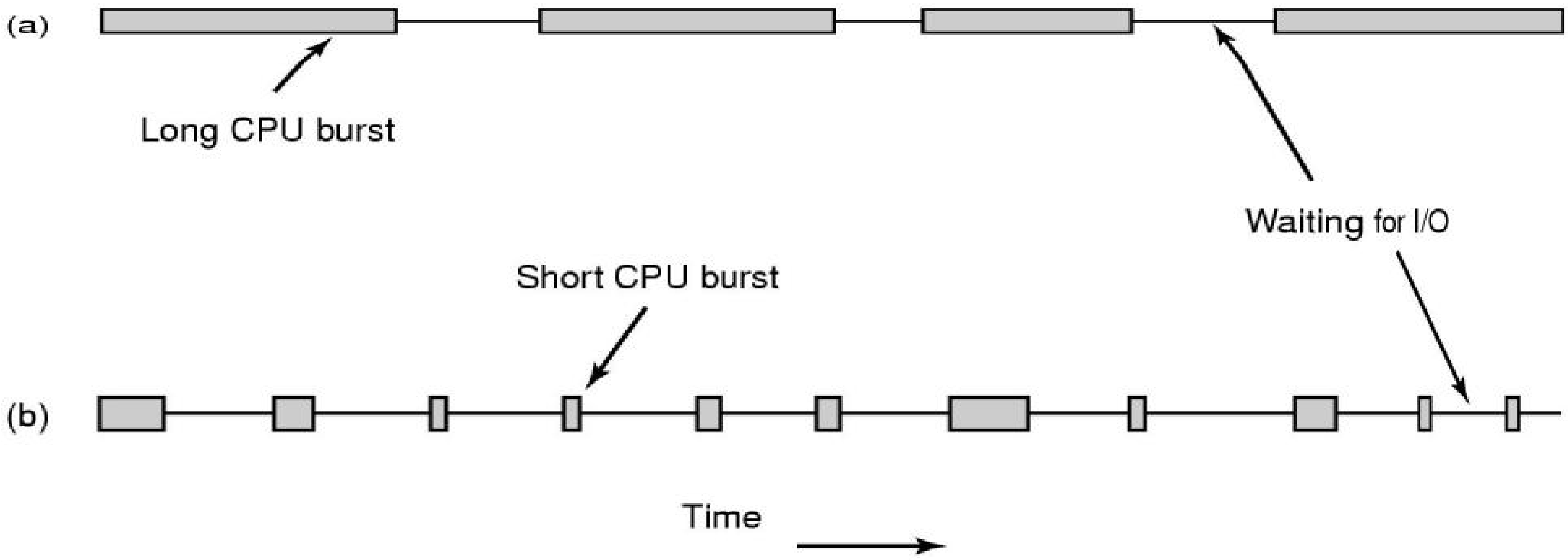
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle
  - Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

Alternating Sequence of CPU And I/O Bursts

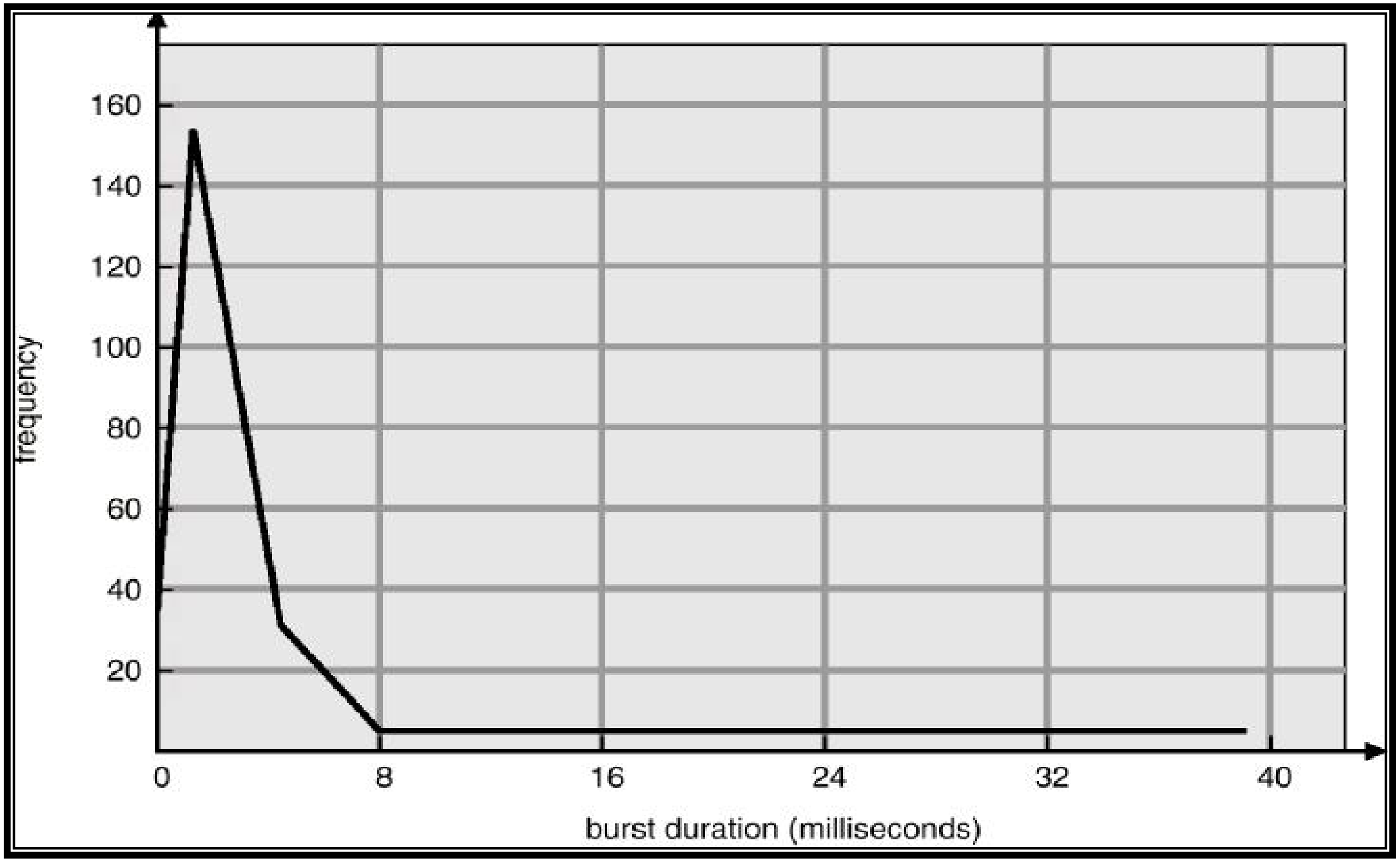


CPU burst vs. I/O burst

- (a) A CPU-bound process
- (b) An I/O-bound process

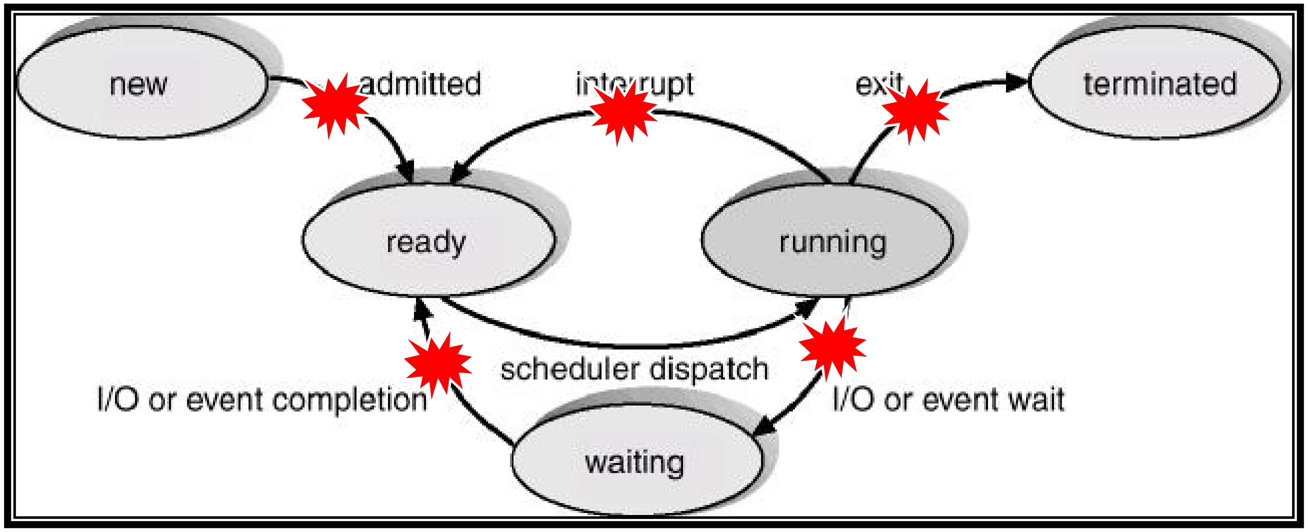


Histogram of CPU-burst Times



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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



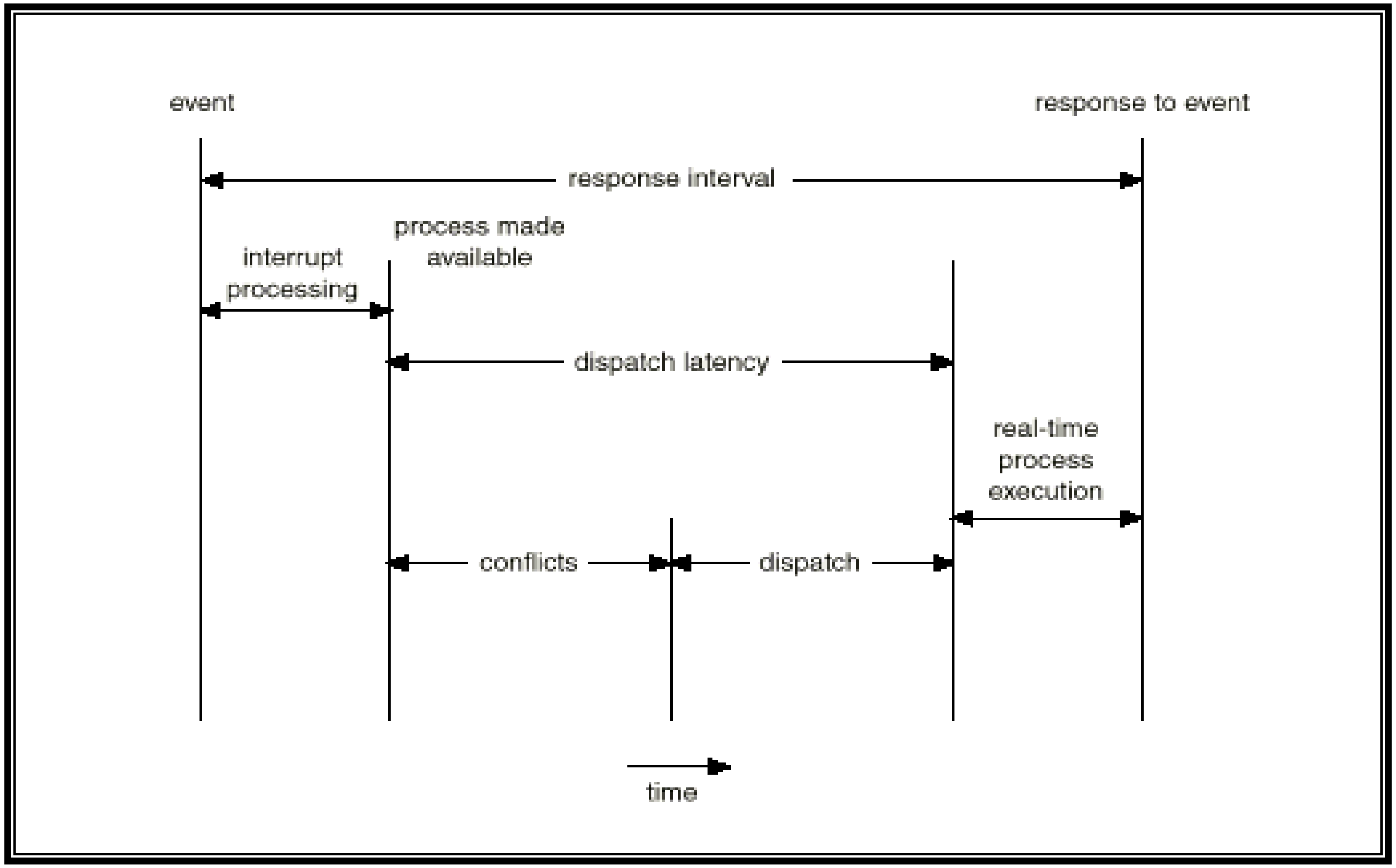
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

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

Dispatch Latency



Preemptive vs. Non-preemptive

- Non-preemptive scheduling
  - The scheduler waits for the running job to explicitly (voluntarily) block
  - Scheduling takes place only when
    - A process switches from running to waiting state
    - A process terminates

- Preemptive scheduling
  - The scheduler can interrupt a job and force a context switch
  - What happens
    - If a process is preempted in the midst of updating the shared data?
    - If a process in system call is preempted?

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)

Optimization Criteria

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

Scheduling Goals

- All systems
  - Fairness: giving each process a fair share of the CPU
  - Balance: keeping all parts of the system busy

- Batch systems
  - Throughput: maximize jobs per hour
  - Turnaround time: minimize time between submission and termination
  - CPU utilization: keep the CPU busy all the time

Scheduling Goals (Cont’d)

- Interactive systems
  - Response time: minimize average time spent on ready queue
  - Waiting time: minimize average time spent on wait queue
  - Proportionality: meet users’ expectations

- Real-time systems
  - Meeting deadlines: avoid losing data
  - Predictability: avoid quality degradation in multimedia systems

Scheduling Non-goals

- Starvation
  - A situation where a process is prevented from making progress because another process has the resource it requires.
    - Resource could be the CPU or a lock
  - A poor scheduling policy can cause starvation
    - If a high-priority process always prevents a low-priority process from running on the CPU
  - Synchronization can also cause starvation
    - One thread always beats another when acquiring a lock
    - Constant supply of readers always blocks out writers

FCFS/FIFO

- First-Come, First-Served
  - Jobs are scheduled in order that they arrive
  - “Real-world” scheduling of people in lines
    - e.g. supermarket, bank tellers, McDonalds, etc.
  - Typically, non-preemptive
  - Jobs are treated equally: no starvation

- Problems
  - Average waiting time can be large if small jobs wait behind long ones
    - Basket vs. cart
  - May lead to poor overlap of I/O and CPU

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: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont’d)

- 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

SJF

- Shortest Job First
  - Choose the job with the smallest expected CPU burst
  - Can prove that SJF has optimal min. average waiting time
    - Only when all jobs are available simultaneously
  - Non-preemptive

- Problems
  - Impossible to know size of future CPU burst
  - Can you make a reasonable guess?
  - Can potentially starve

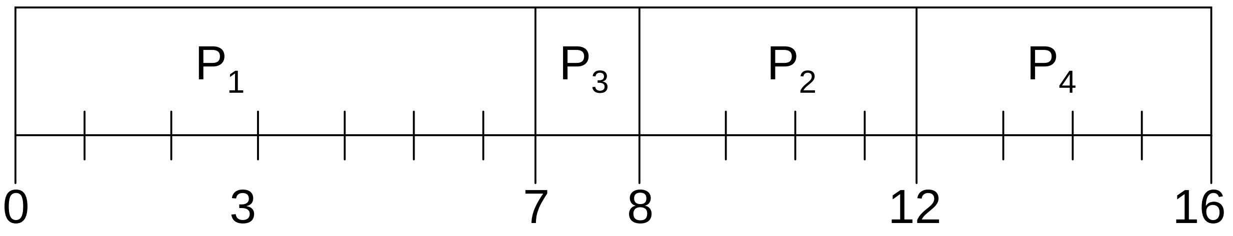
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use these 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
  - Preemptive
    - If a new process arrives with CPU burst length less than remaining time of current executing process, preempt
    - This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- 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)

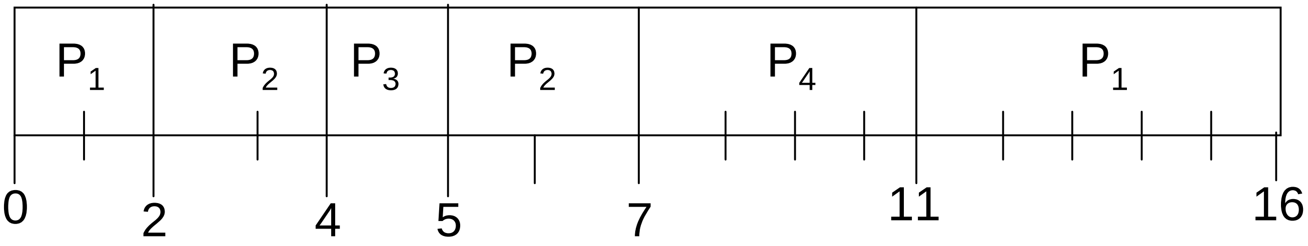


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

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) (= SRTF)



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

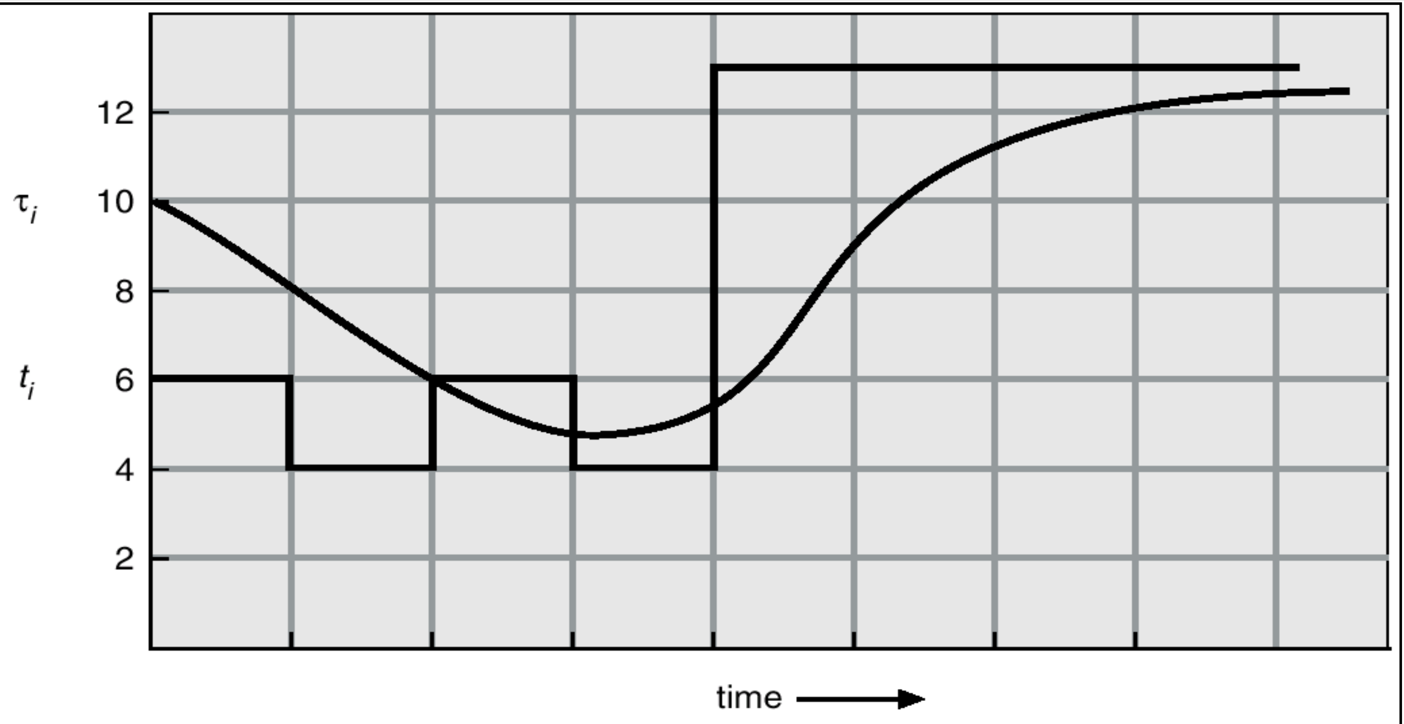
Determining Length of Next CPU Burst

- 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. $t_{n+1} =$ predicted value for the next CPU burst
3. $a, 0 \leq a \leq 1$
4. Define :

$$t_{n=1} = a\ t_n + (1 - a\ ) t_n .$$

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	. . .

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts

- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_n - 1 + \ldots$$
$$+ (1 - \alpha)^j\, \alpha\, t_n - 1 + \ldots$$
$$+ (1 - \alpha)^{n=1}\, t_n\, \tau_0$$

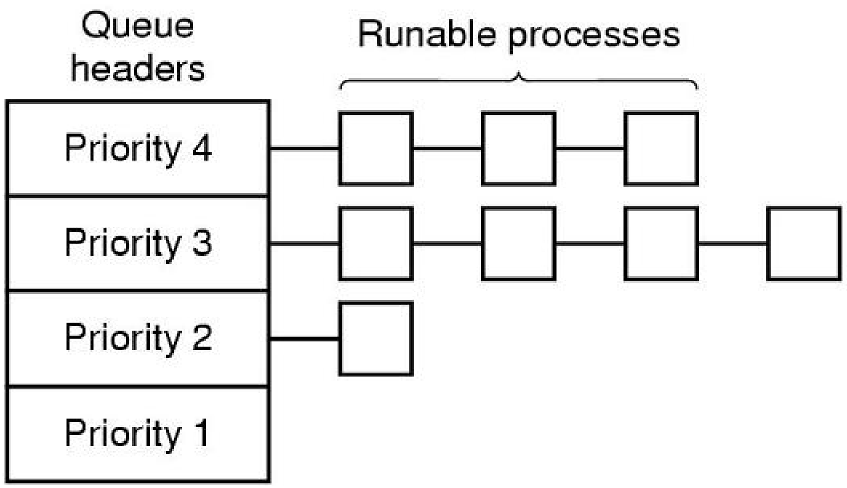
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

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 (or Indefinite blocking)
  - low priority processes may never execute
- Solution ≡ Aging
  - as time progresses increase the priority of the process

Priority Scheduling

- Abstractly modeled as multiple “priority queues”
  - Put ready job on Q associated with its priority



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

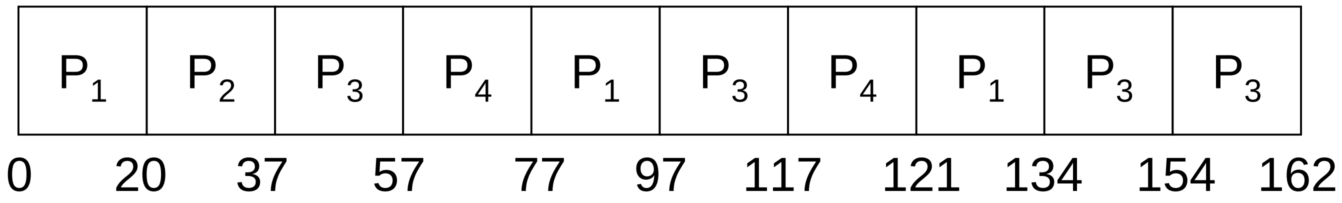
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once
  - No process waits more than $(n-1)q$ time units

- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

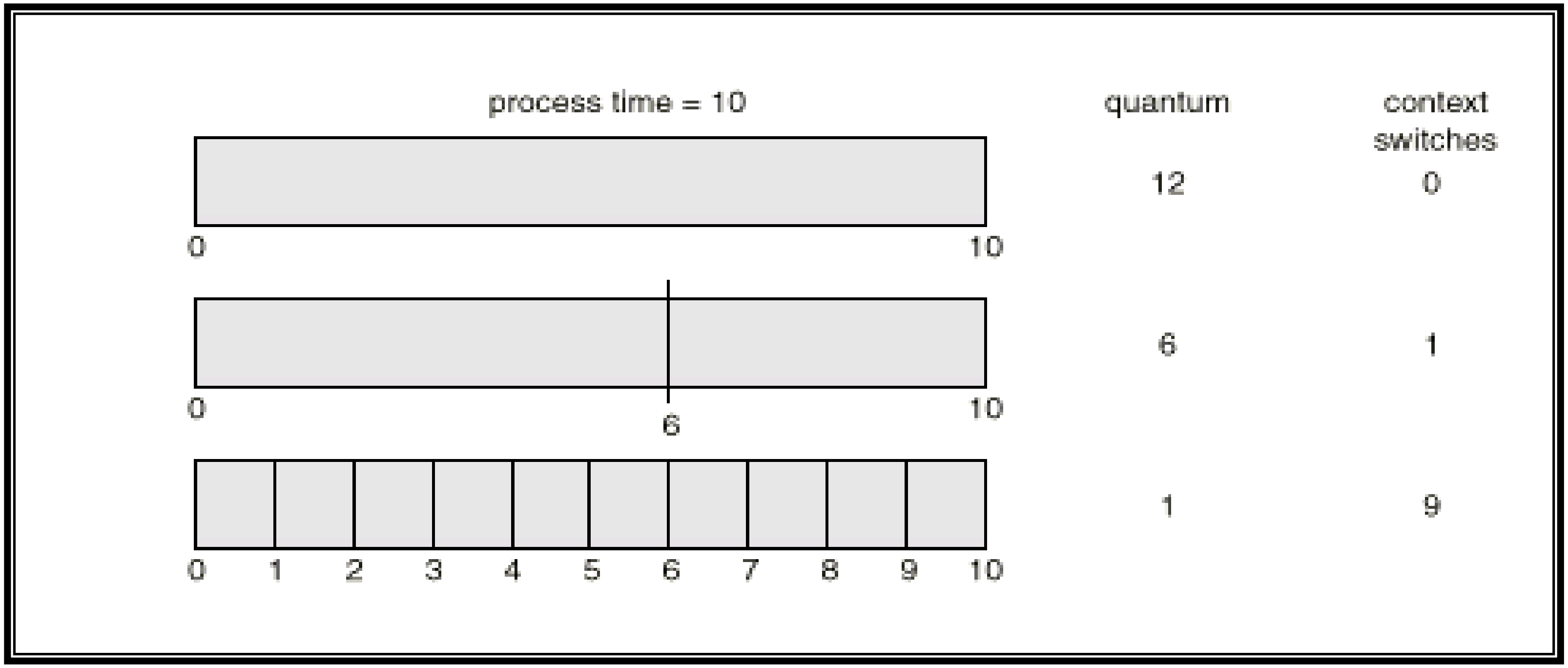
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

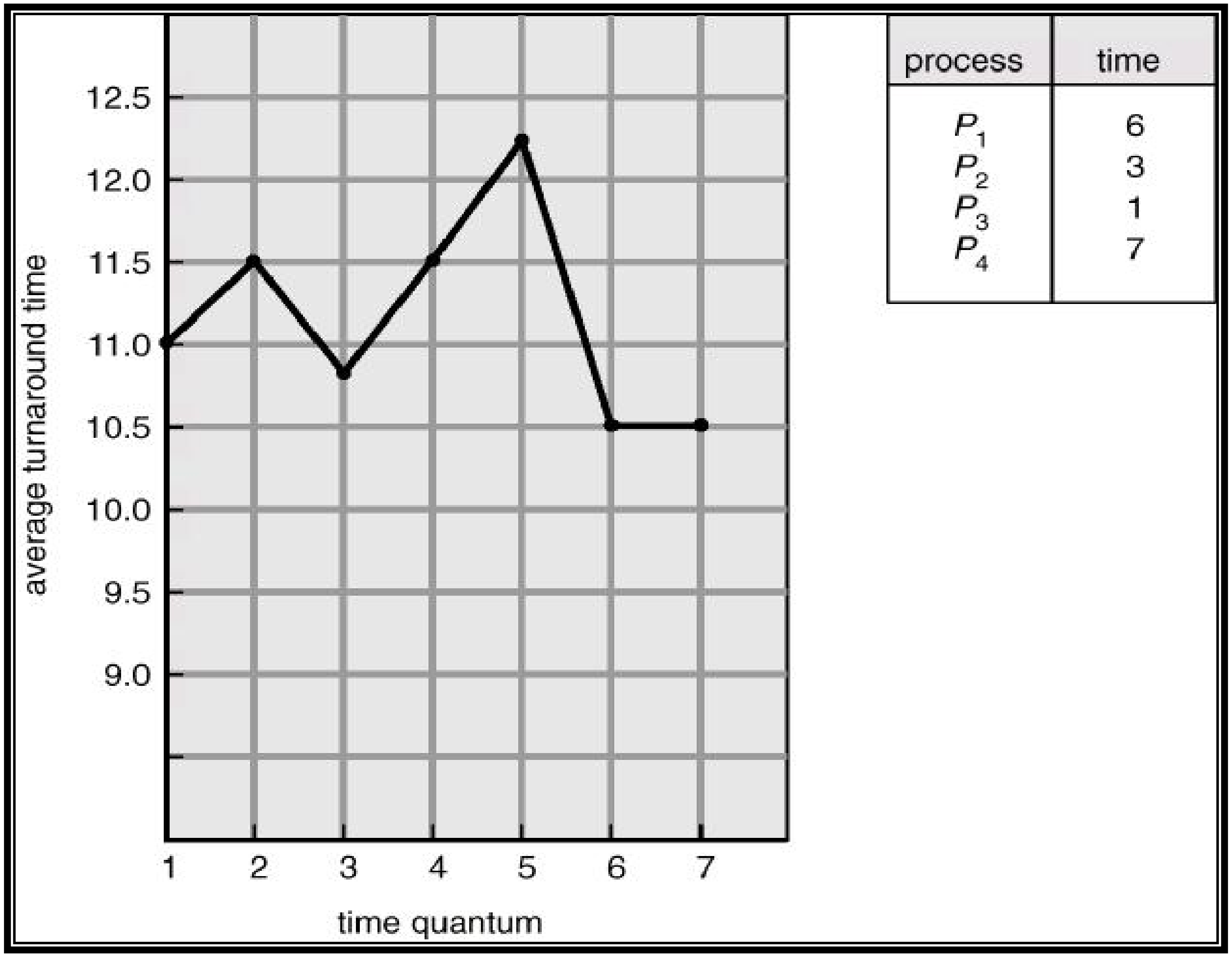


- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Problems of RR

- What do you set the quantum to be?
  - quantum → ∞ : FIFO

    quantum → 0 : processor sharing
  - If small, then context switches are frequent incurring high overhead (CPU utilization drops)
  - If large, then response time drops
  - A rule of thumb: 80% of the CPU bursts should be shorter than the time quantum
- Treats all jobs equally
  - Multiple background jobs?

Combining Algorithms

- Scheduling algorithms can be combined in practice
    - Have multiple queues
    - Pick a different algorithm for each queue
    - Have a mechanism to schedule among queues
    - And maybe, move processes between queues

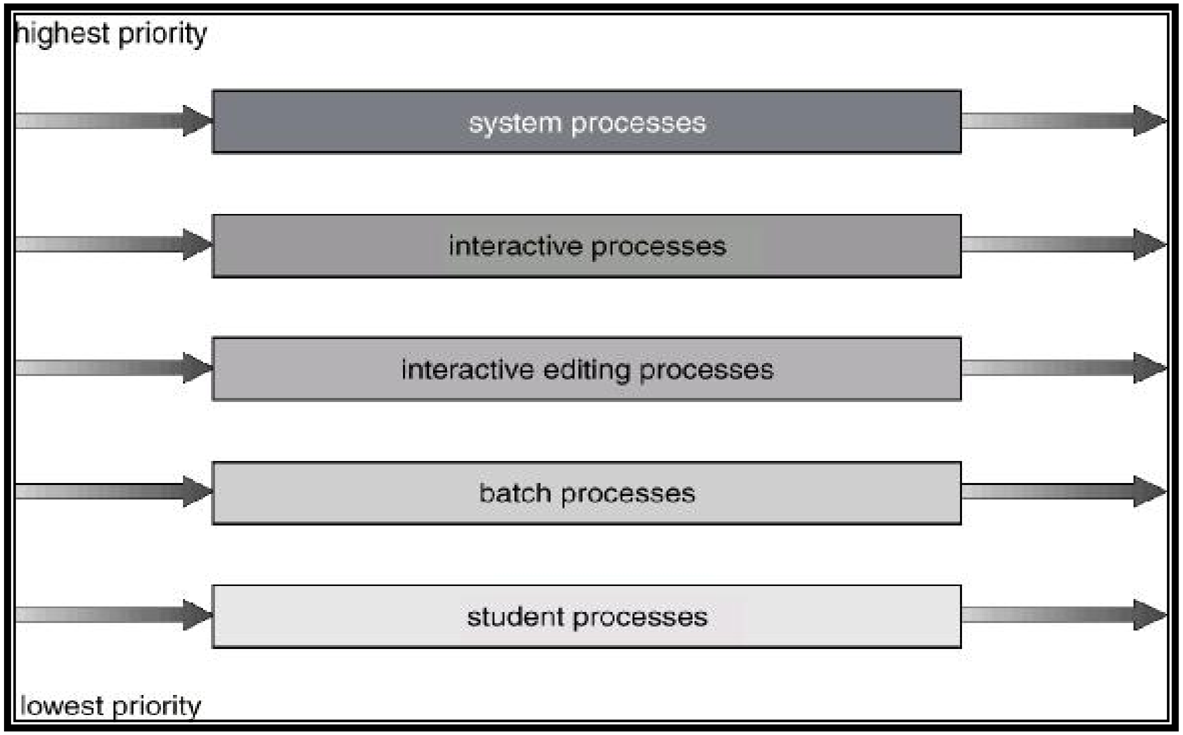
Multilevel Queue

- 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

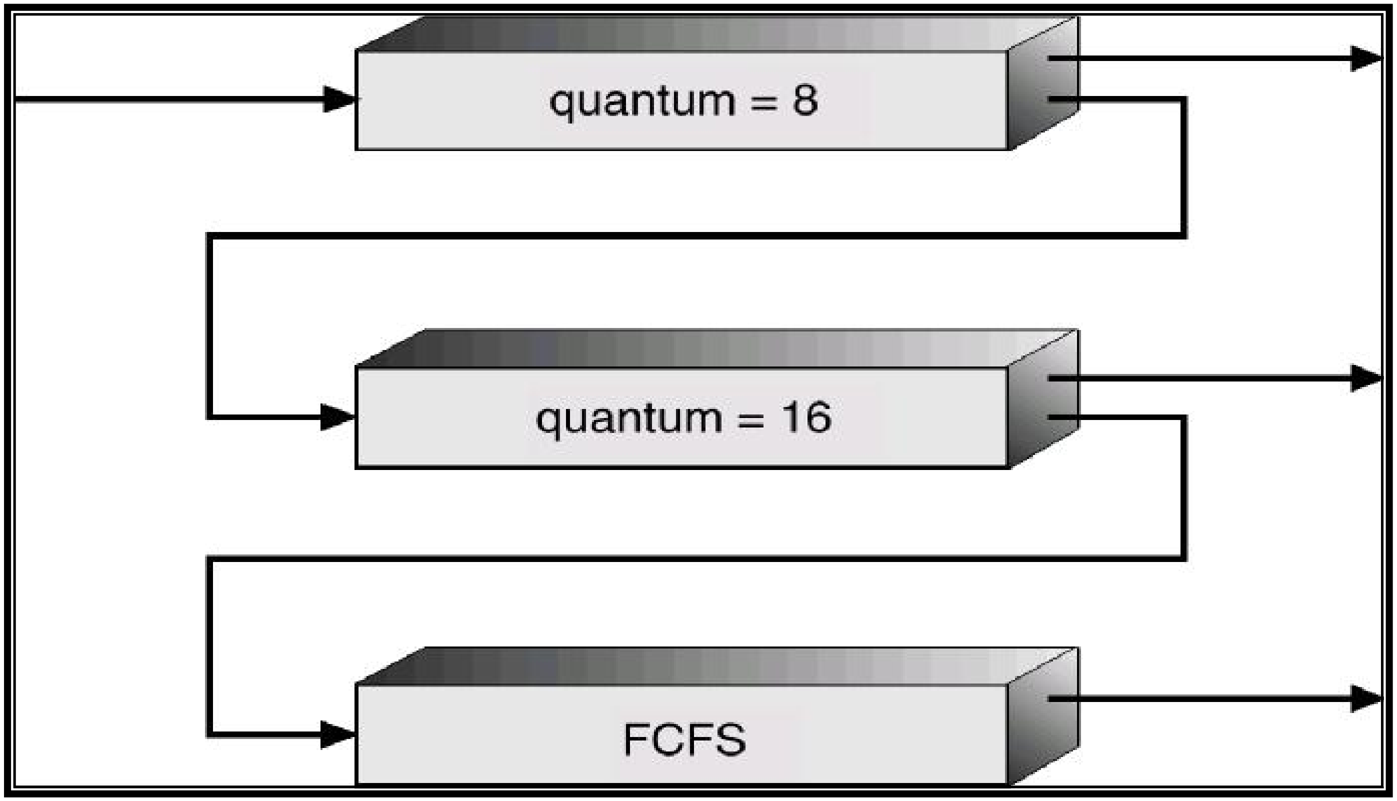
- Multilevel-feedback-queue scheduler defined by the following parameters:
    - number of queues
    - scheduling algorithms for each queue
    - method used to determine when to upgrade a process
    - method used to determine when to demote a process
    - method used to determine which queue a process 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



UNIX Scheduler

- The canonical UNIX scheduler uses a MLFQ
  - 3 – 4 classes spanning ~170 priority levels
    - Timeshare, System, Real-time, Interrupt (Solaris 2)
  - Priority scheduling across queues, RR within a queue
    - The process with the highest priority always runs
    - Processes with the same priority are scheduled RR
  - Processes dynamically change priority
    - Increases over time if process blocks before end of quantum
    - Decreases over time if process uses entire quantum

UNIX Scheduler (Cont’d)

- Motivation
  - The idea behind the UNIX scheduler is to reward interactive processes over CPU hogs
  - Interactive processes typically run using short CPU bursts
    - They do not finish quantum before waiting for more input
  - Want to minimize response time
    - Time from keystroke (putting process on ready queue) to executing the handler (process running)
    - Don’t want editor to wait until CPU hog finishes quantum
  - This policy delays execution of CPU-bound jobs

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available

- *Homogeneous processors* within a multiprocessor
  - UMA (Uniform Memory Access)

- *Load sharing*

- *Asymmetric multiprocessing*
  - Only one processor accesses the system data structures, alleviating the need for data sharing
  - Not efficient

Real-Time Scheduling

- *Hard real-time* systems
  - required to complete a critical task within a guaranteed amount of time

- *Soft real-time* computing
  - requires that critical processes receive priority over less fortunate ones

- *Static vs. Dynamic priority scheduling*
  - Static: Rate-Monotonic algorithm
  - Dynamic: EDF (Earliest Deadline First) algorithm

Real-Time Scheduling

- Hard real-time
  - Must complete a critical task within a guaranteed amount of time
  - Resource reservation
    - A process is submitted along with its resource requirements
  - Requires worst-case timing analysis
    - Minimize unavoidable and unforeseeable variation in the amount of time to execute a particular process
    - Very difficult in a system with secondary storage or virtual memory
  - Typically composed of special-purpose software running on dedicated hardware with limited functionality

Real-Time Scheduling (Cont'd)

- Soft real-time
  - Less restrictive
    - Multimedia, high-speed interactive graphics, etc.
  - May cause an unfair allocation of resources and may result in longer delays, or even starvations, for some processes
  - Requirements
    - The system must have priority scheduling, and real-time processes must have the highest priority

      (The priority of real-time processes must not degrade over time)
    - Dispatch latency must be small

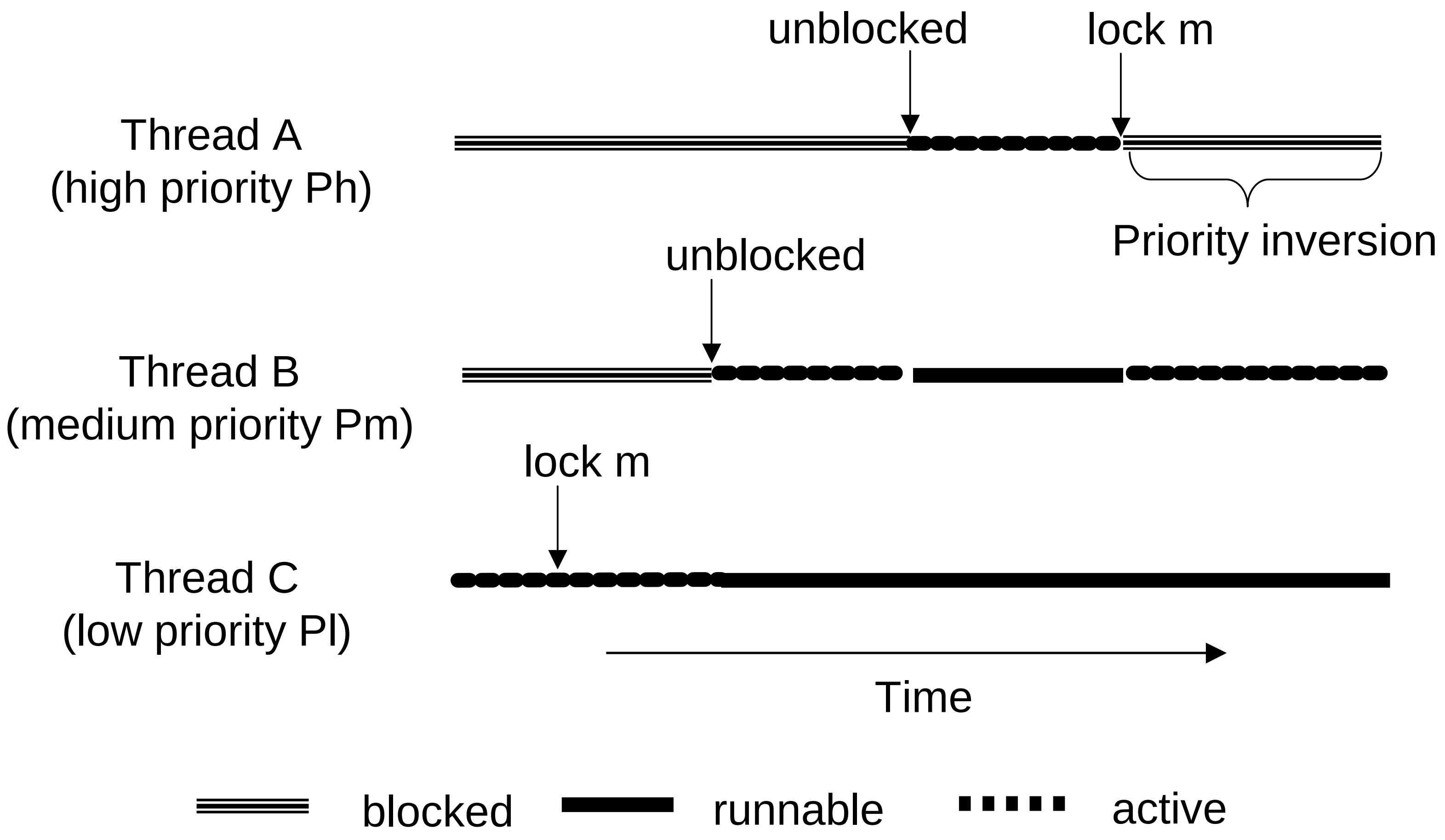
Real-Time Scheduling (Cont'd)

- Problem
  - Most versions of UNIX are forced to wait either for a system call to complete or for an I/O block to take place before doing a context switch

- Preempting system calls
  - Insert preemption points
    - Still dispatch latency can be large
  - Make the entire kernel preemptible.
    - All kernel data structures must be protected
    - Solaris 2

Real-Time Scheduling (Cont’d)

- Priority inversion problem



unblocked

lock m

Thread A
(high priority Ph)

Priority inversion

unblocked

Thread B
(medium priority Pm)

lock m

Thread C
(low priority Pl)

Time

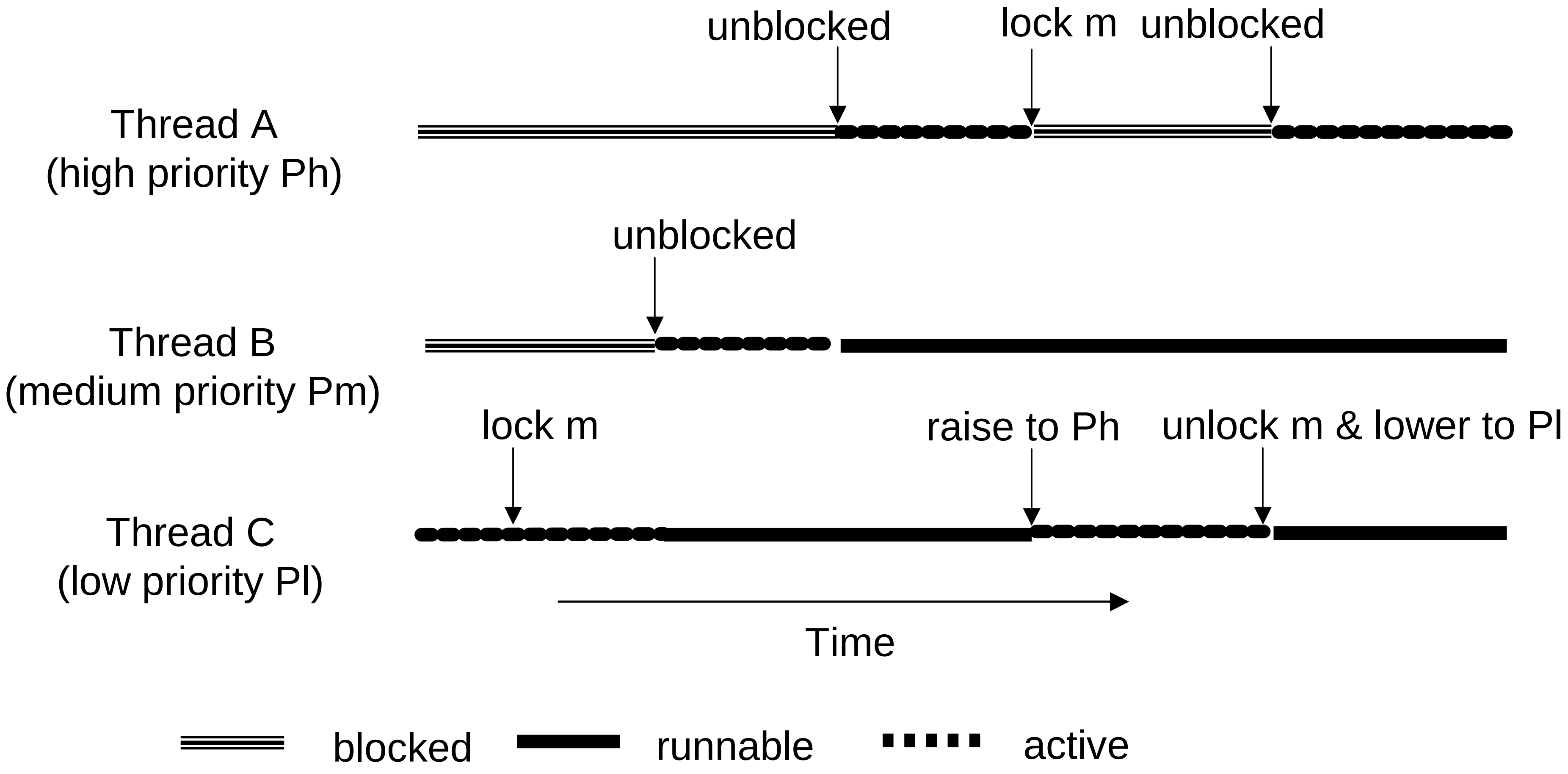
blocked

runnable

active

Real-Time Scheduling (Cont’d)

- Priority inheritance protocol



unblocked
lock m
unblocked

Thread A
(high priority Ph)

unblocked

Thread B
(medium priority Pm)

lock m
raise to Ph
unlock m & lower to Pl

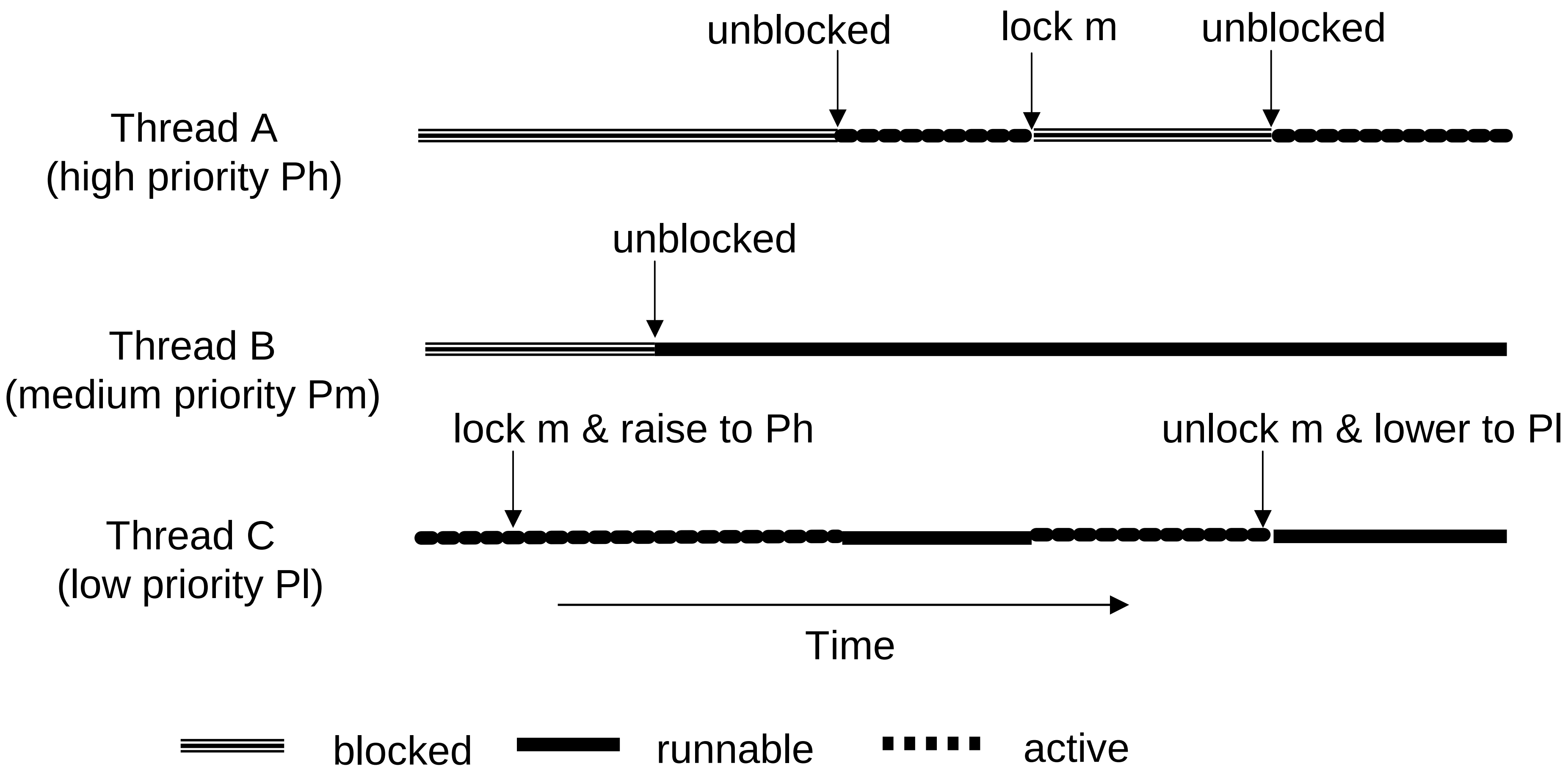
Thread C
(low priority Pl)

Time

═ blocked
▬ runnable
▪▪▪▪▪ active

Real-Time Scheduling (Cont'd)

- Priority ceiling protocol



unblocked

lock m

unblocked

Thread A
(high priority Ph)

unblocked

Thread B
(medium priority Pm)

lock m & raise to Ph

unlock m & lower to Pl

Thread C
(low priority Pl)

Time

═ blocked ▬ runnable ▪▪▪▪▪ active

Algorithm Evaluation

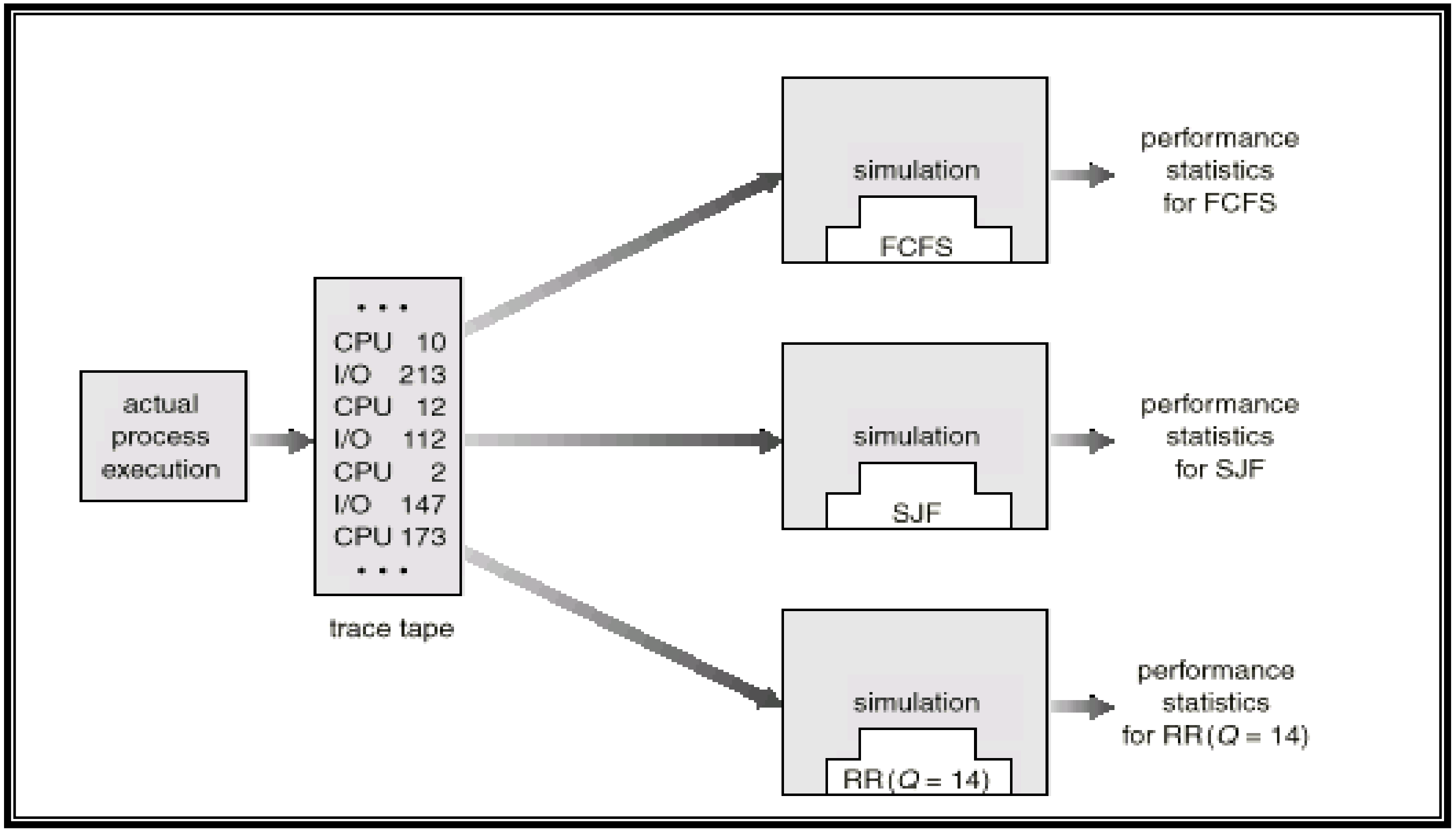
- Deterministic modeling
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

- Queueing models
  - Mathematical models used to compute expected system parameters

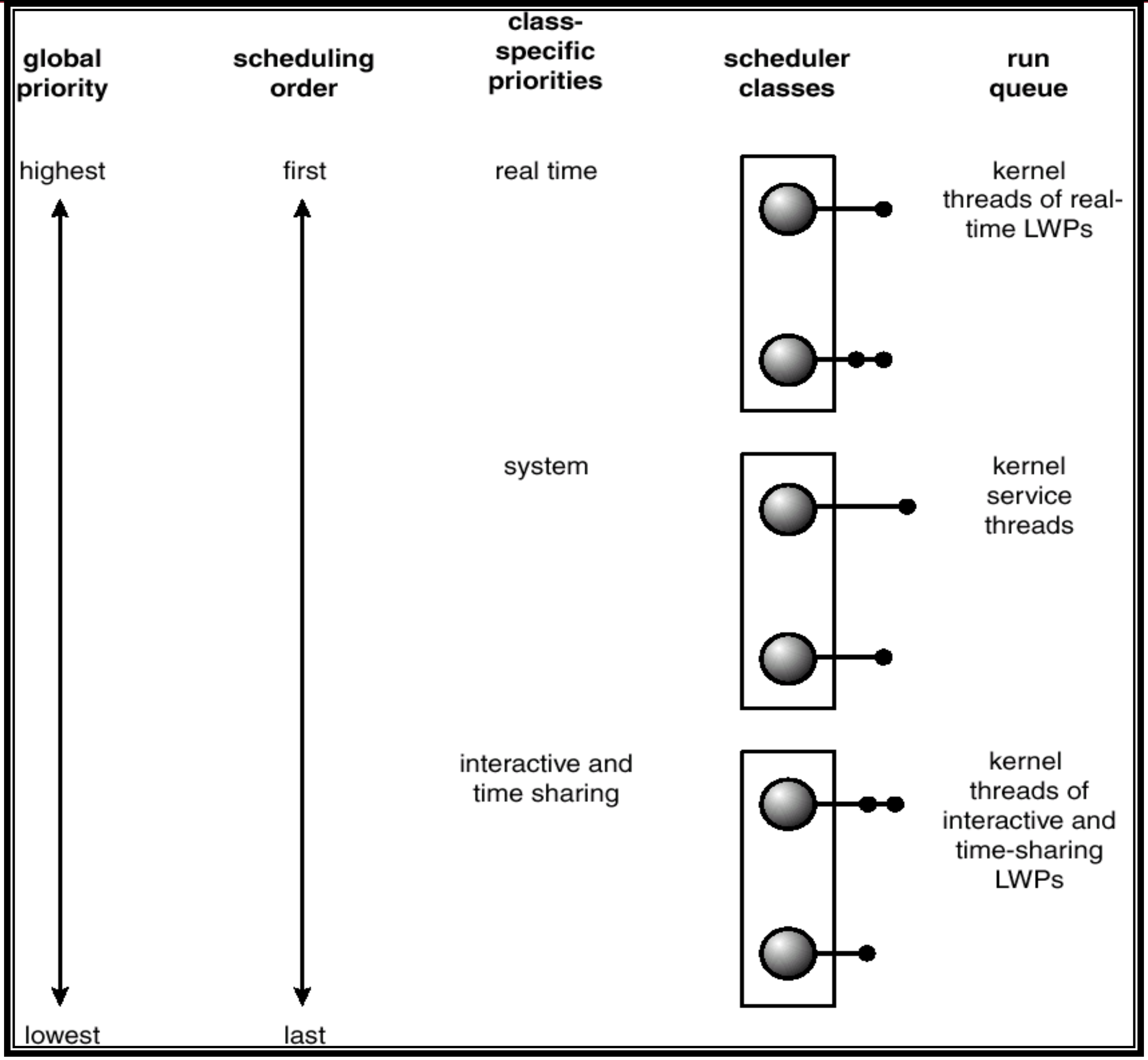
- Simulation
  - Algorithmic models which simulate a simplified version of a system using statistical input
  - Trace tape (or trace data)
  - *Cf) Emulation*

- Implementation
  - Direct implementation of the system under test, with appropriate benchmarks

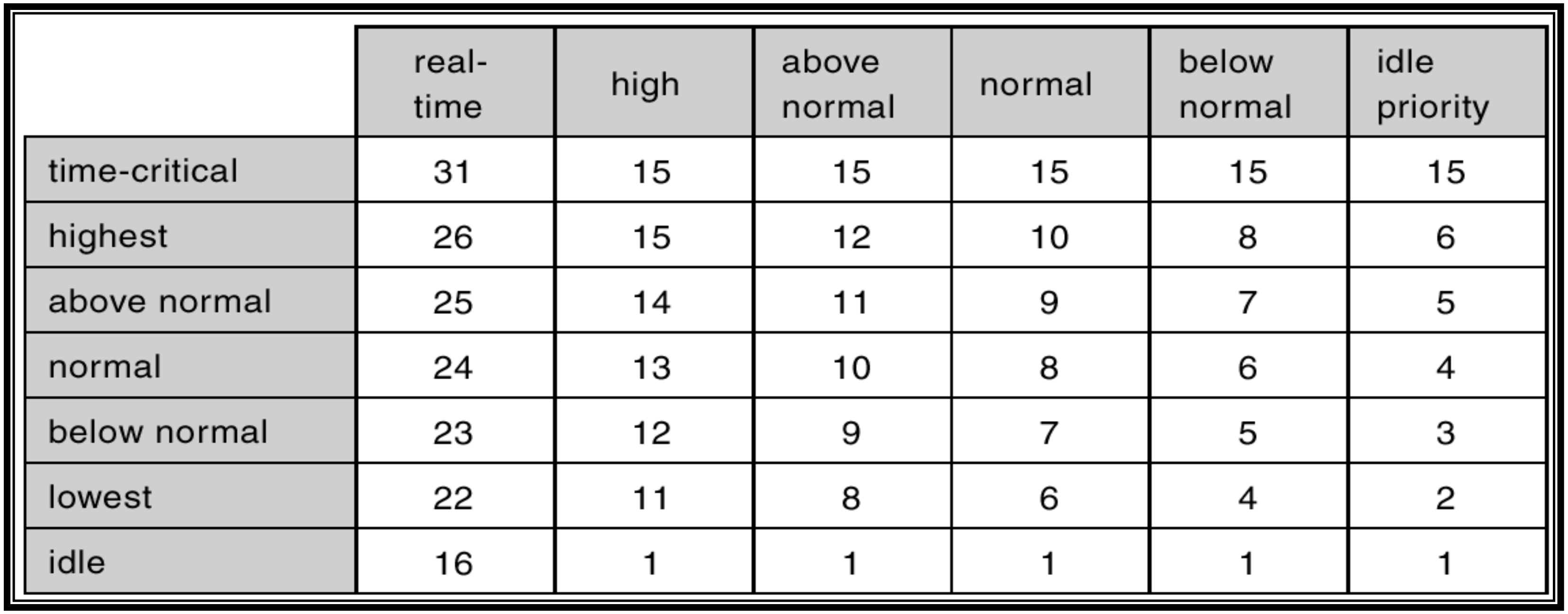
Evaluation of CPU Schedulers by Simulation



Solaris 2 Scheduling



Windows 2000 Priorities



	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1