Chapter 6: CPU Scheduling

Introduction to CPU scheduling

Various CPU scheduling algorithm

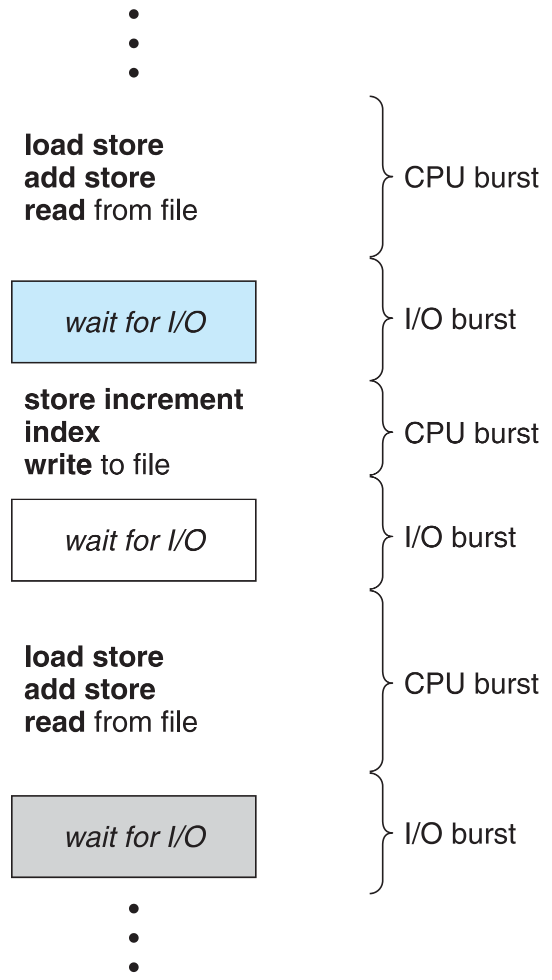
Evaluation criteria for selecting a CPU-scheduling algorithm

scheduling algorithms of several operating systems

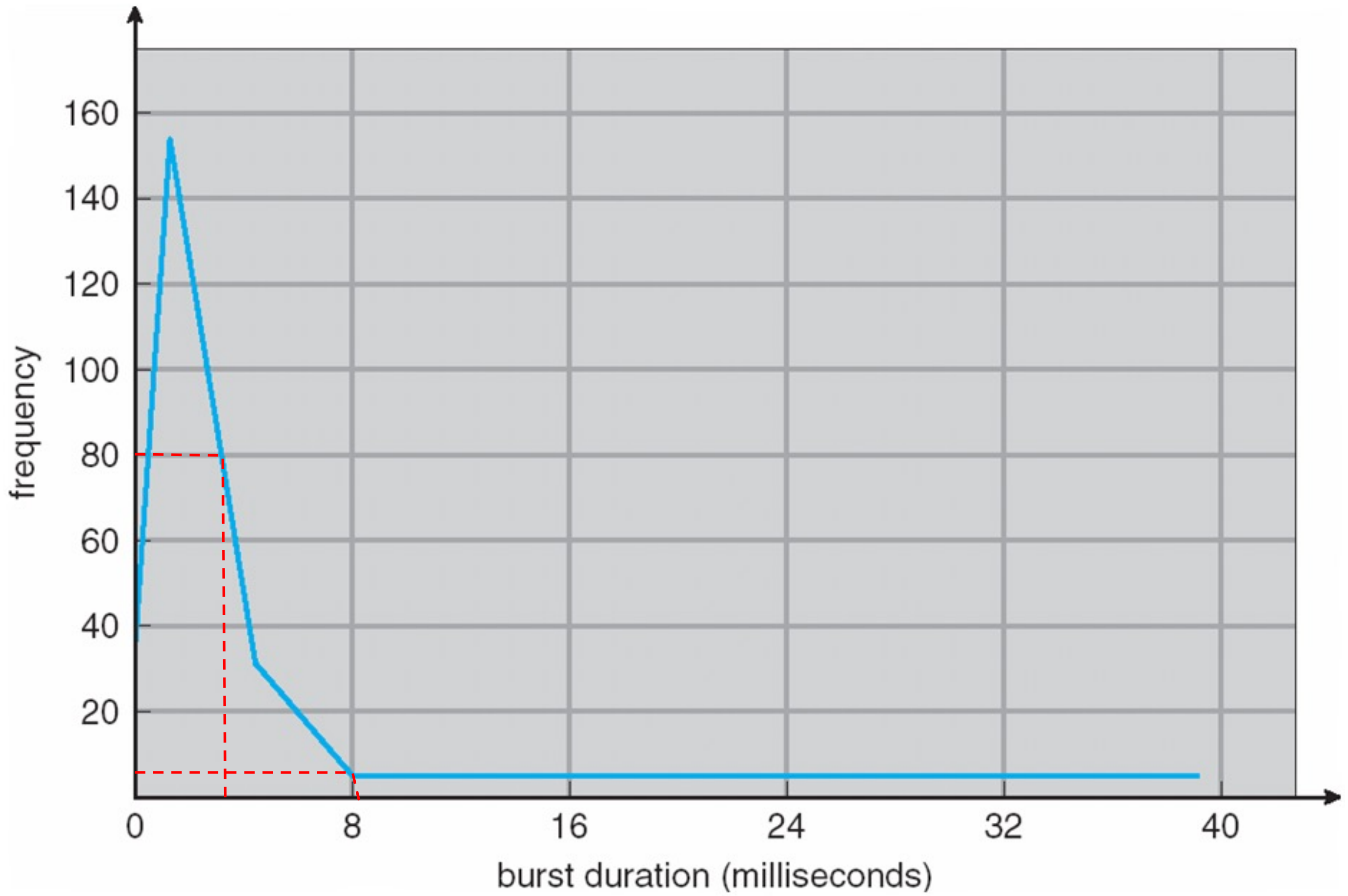
Contents

6.1 Basic Concepts

- CPU burst–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

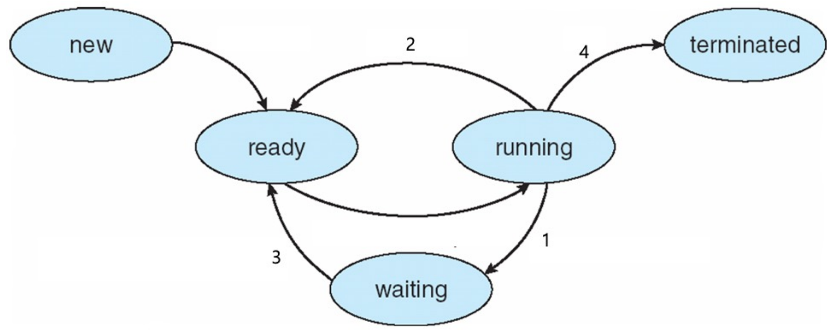


Histogram of CPU-burst Times



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
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



Dispatcher

- Dispatcher module gives control of the CPU to the process se lected 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 p rogram
- **Dispatch latency** – time it takes for the dispatcher to stop on e process and start another running

6.2 Scheduling Criteria

1. **CPU utilization** – keep the CPU as busy as possible
2. **Throughput** – # of processes that complete their execution per time unit
3. **Turnaround time** – amount of tim e to execute a particular process
4. **Waiting time** – amount of time a process has been waiting in the re ady queue
5. **Response time** – amount of time i t takes from when a request was s ubmitted until the first response is produced, not output (for time-sh aring environment)

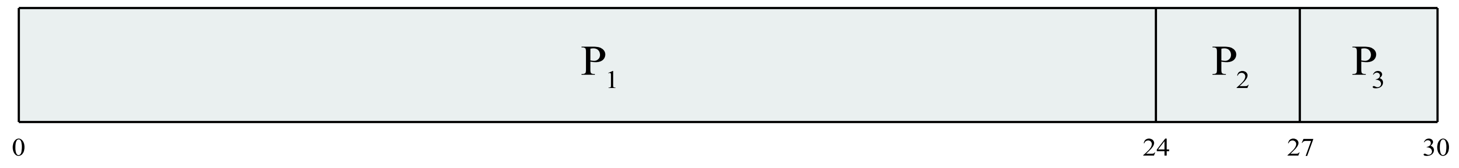
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

6.3 Scheduling Algorithms

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

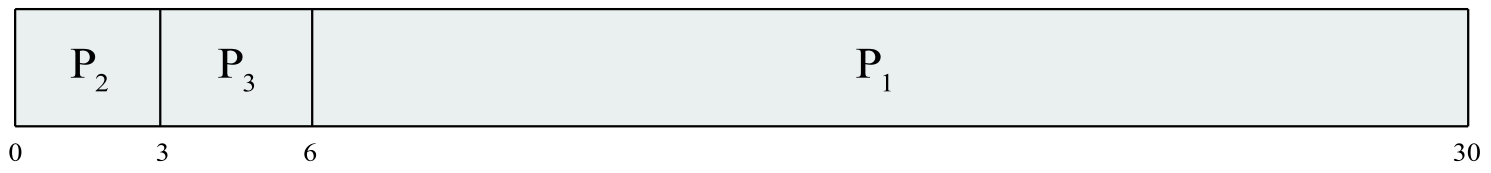
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



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

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order: P_2, P_3, P_1
- The Gantt chart for the schedule is:

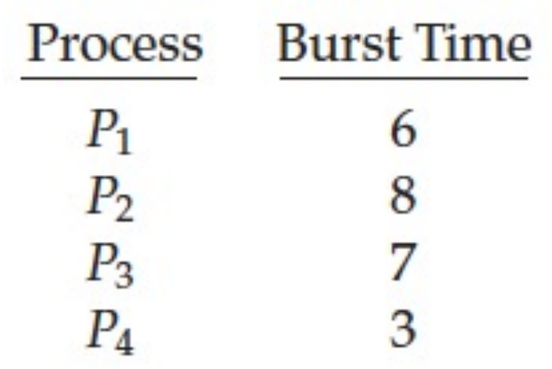


- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Convoy effect - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

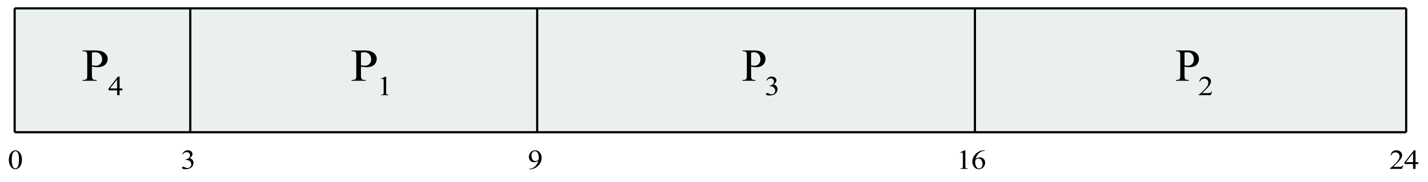
- Associate with each process the length of its next CPU burst
    - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a gi ven set of processes
    - The difficulty is knowing the length of the next CPU request
    - Could ask the user

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

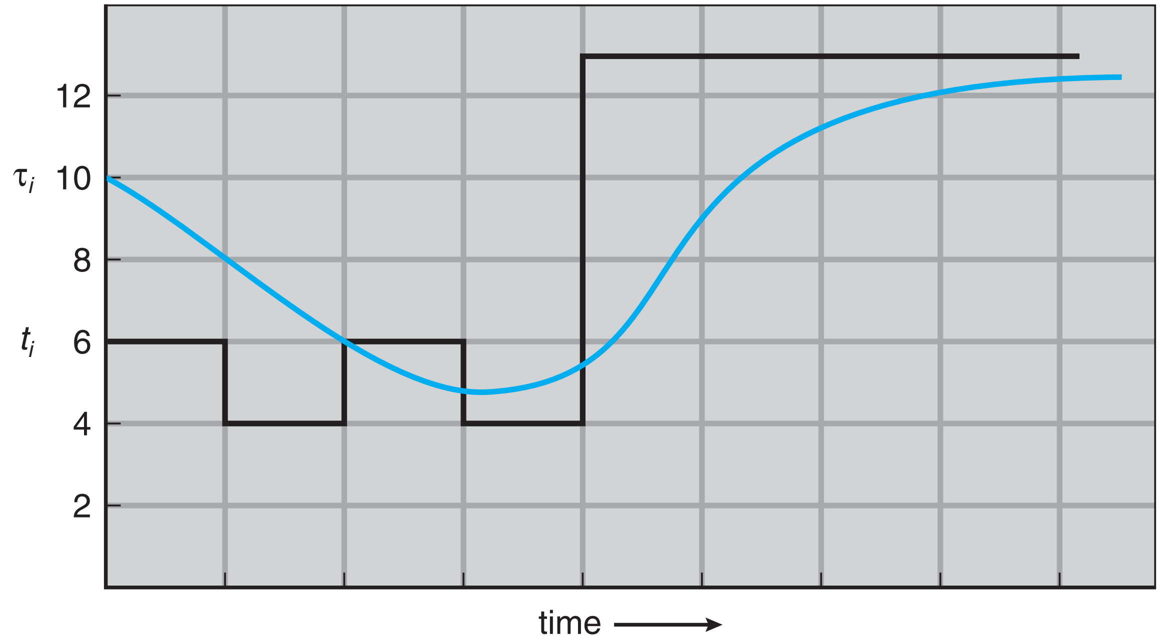
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous o ne
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using ex ponential 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

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
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\ t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

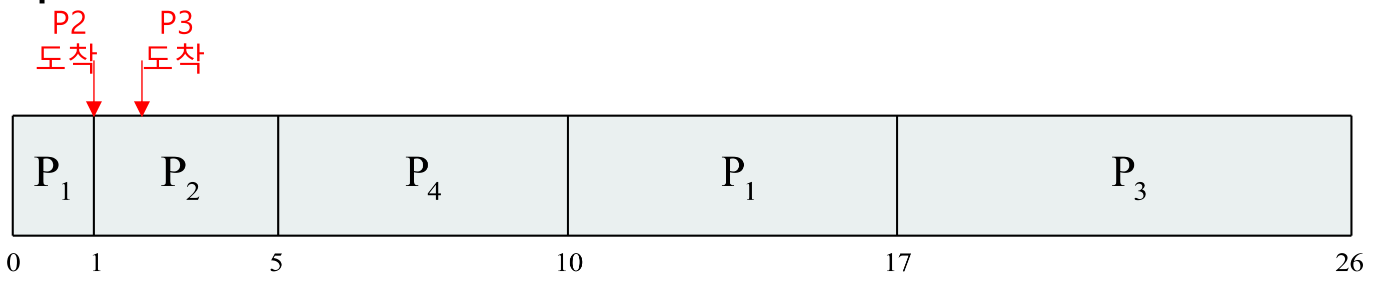
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successi ve term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to th e 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 = $[\underbrace{(0-0)+(10-1)}_{P_1}+\underbrace{(1-1)}_{P_2}+\underbrace{(17-2)}_{P_3}+\underbrace{(5-3)}_{P_4}]/4$ = 26/4 = 6. 5 msec

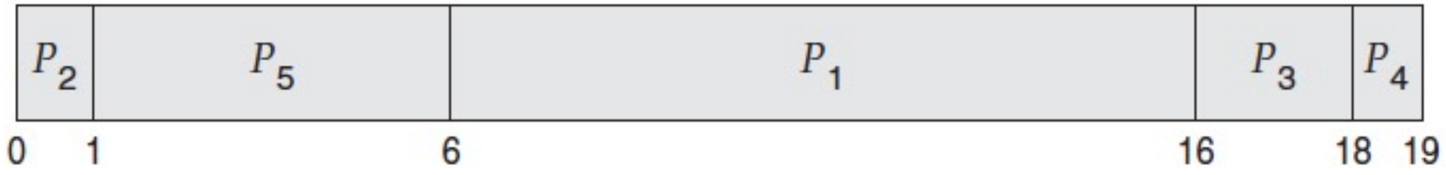
Priority Scheduling

- A priority number (integer) is associated with each process. The CP U is allocated to the process with the highest priority (smallest int eger ≡ highest priority)
- Preemptive vs. 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

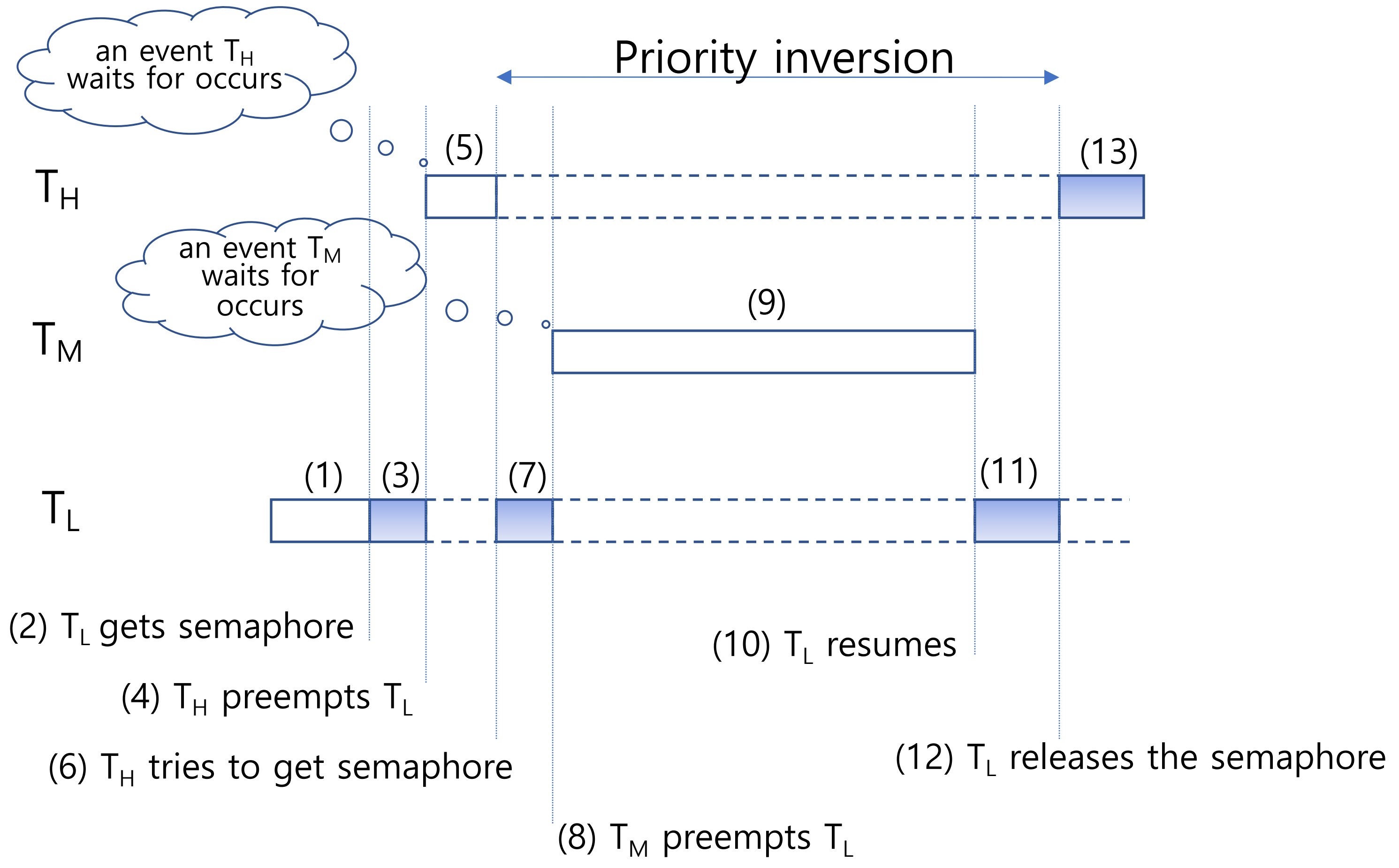
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

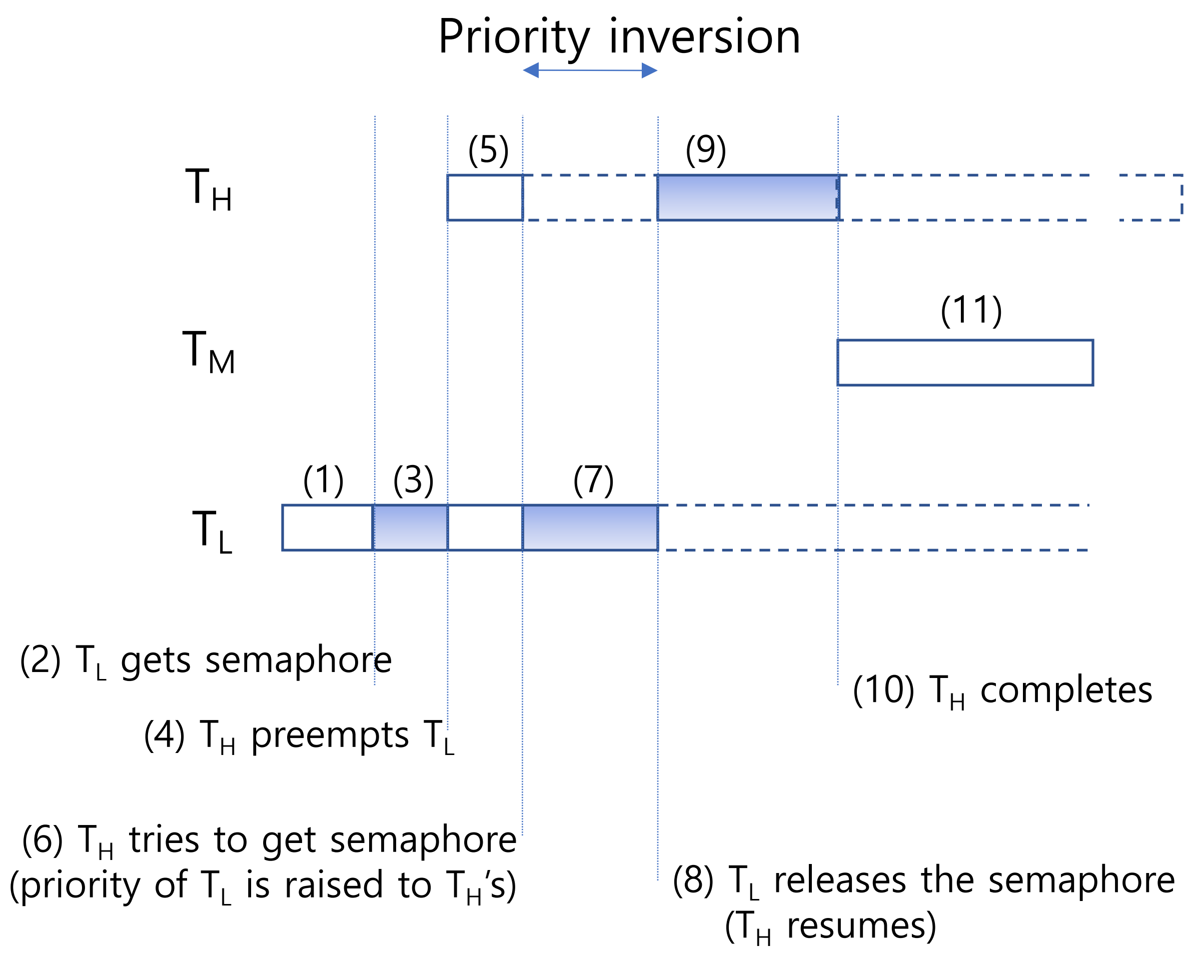


- Average waiting time = 8.2 msec

Priority Inversion



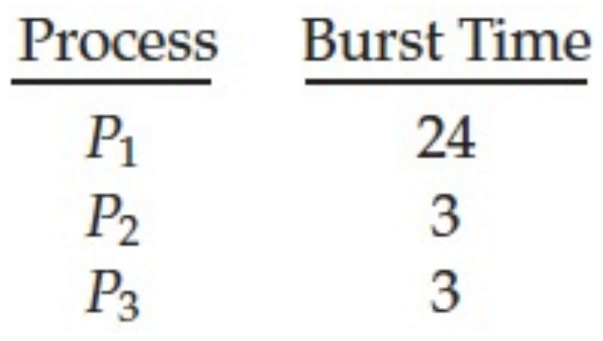
Priority Inheritance (Donation) Protocol


Priority inversion
T_H
T_M
T_L
(1)
(3)
(5)
(7)
(9)
(11)
(2) T_L gets semaphore
(4) T_H preempts T_L
(6) T_H tries to get semaphore (priority of T_L is raised to T_H's)
(8) T_L releases the semaphore (T_H resumes)
(10) T_H completes

Round Robin (RR) Scheduling

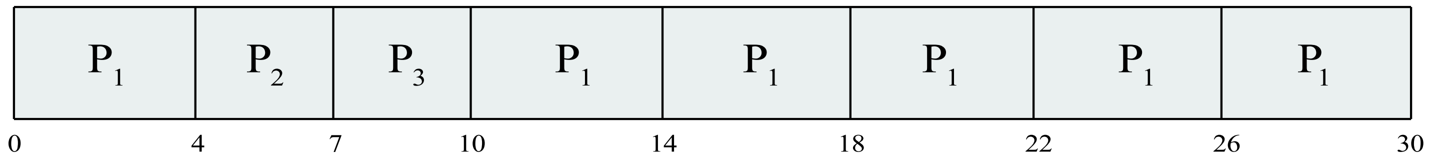
- Each process gets a small unit of CPU time (**time quantum** *q*), usu ally 10-100 milliseconds. After this time has elapsed, the process i s preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (n-1)q tim e 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 overh ead is too high

Example of RR with Time Quantum = 4



Process	Burst Time
P_1	24
P_2	3
P_3	3

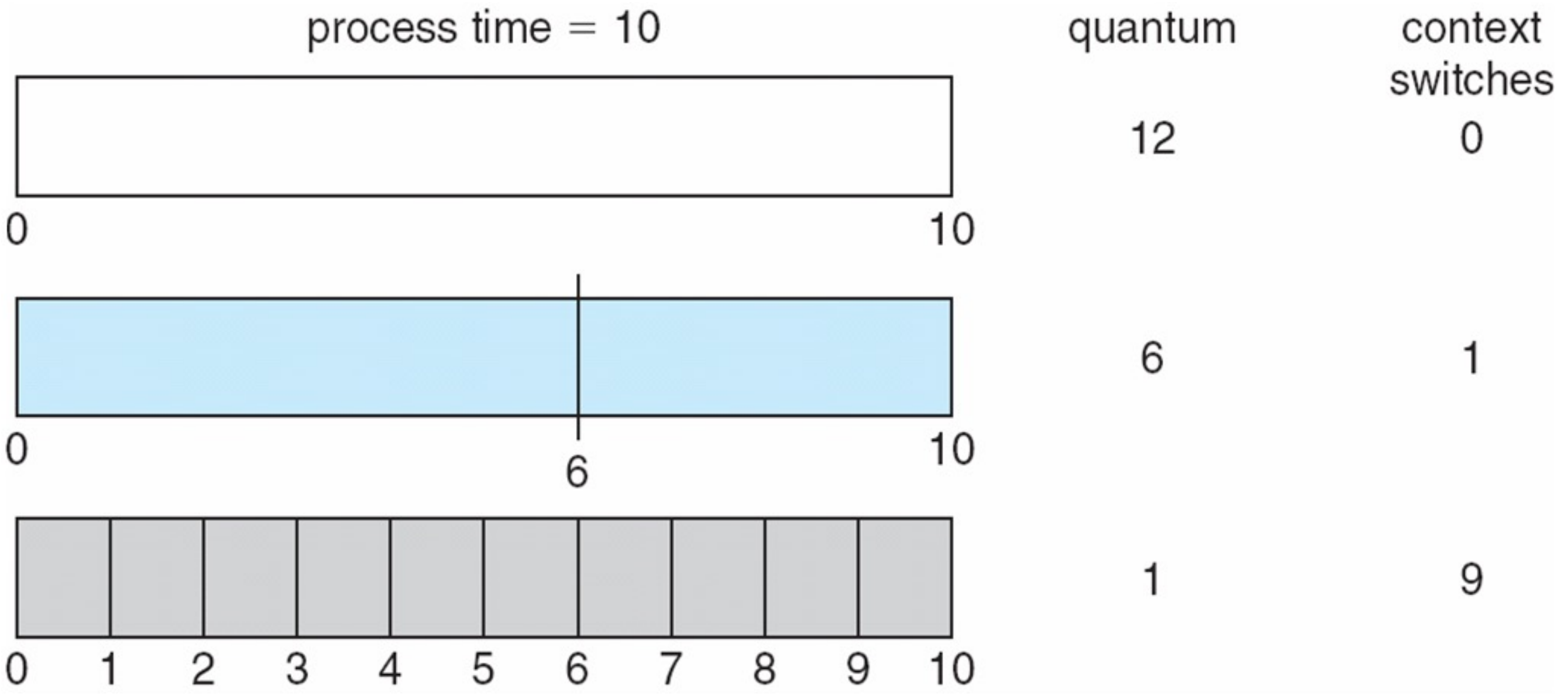
- The Gantt chart is:



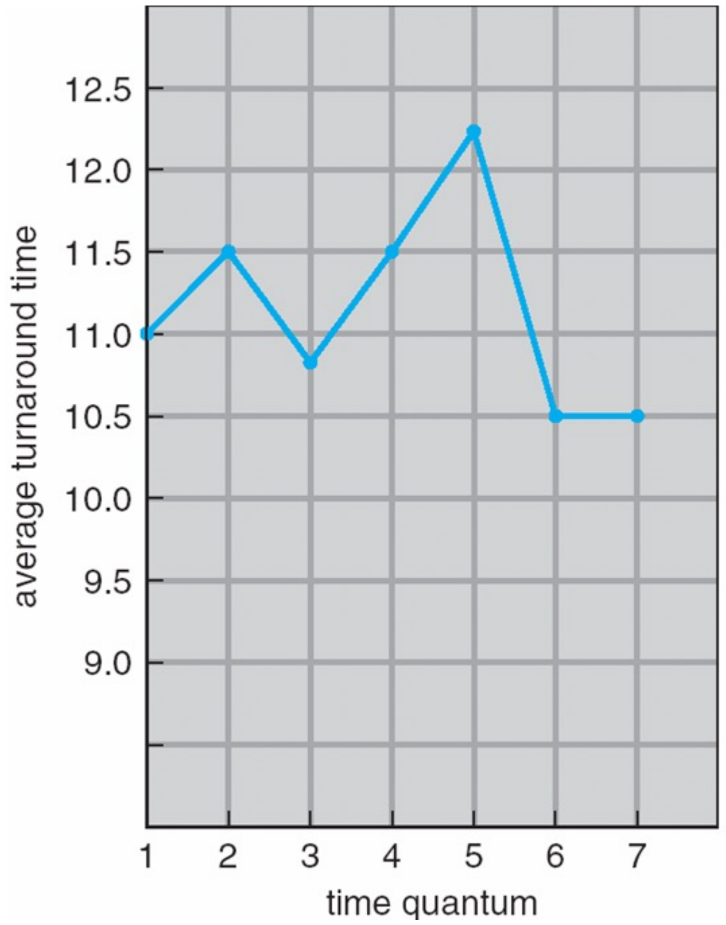
- Typically, higher average turnaround than SJF, but better ***response***

Time Quantum and Context Switch Time

- q should be large compared to context switch time; q usually 10ms to 100ms, context switch < 10 usec



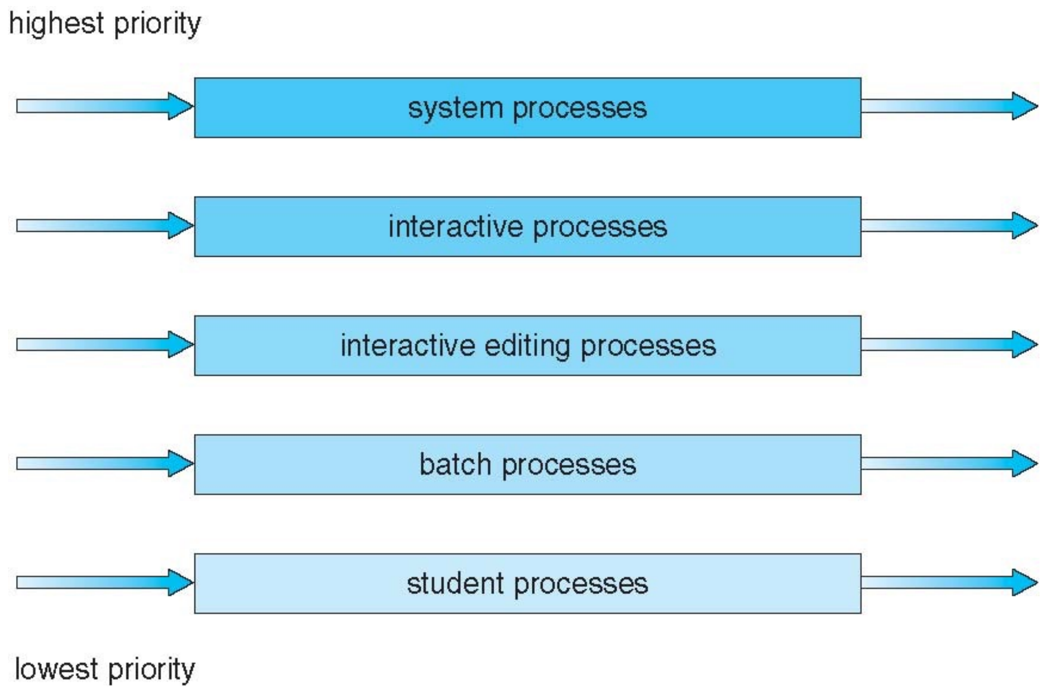
Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

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 and 20% to background in FCFS



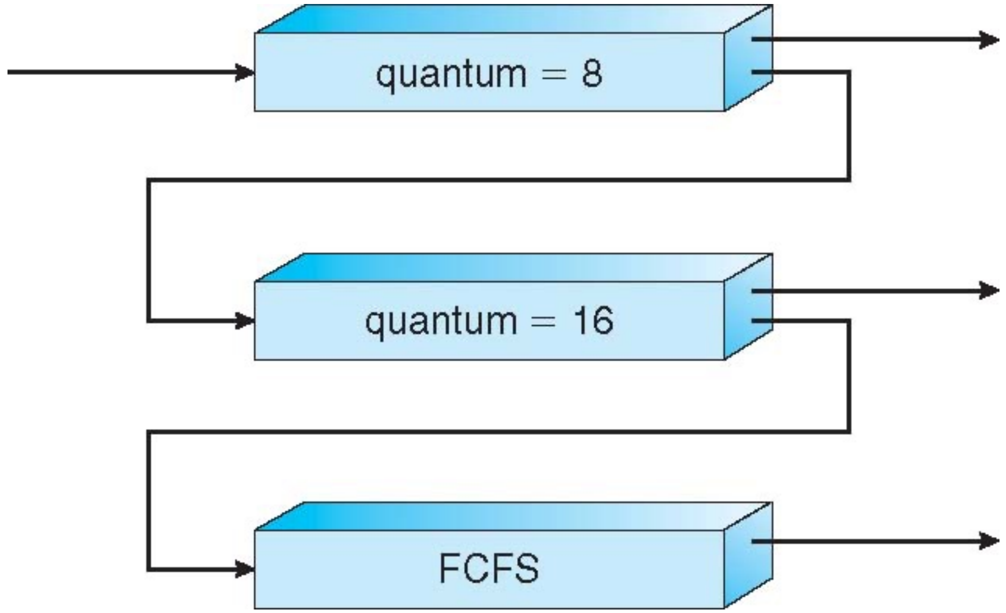
Multilevel Feedback Queue

- A process can move between the various queues; aging can b e 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 t hat process needs service

Example of Multilevel Feedback Queue

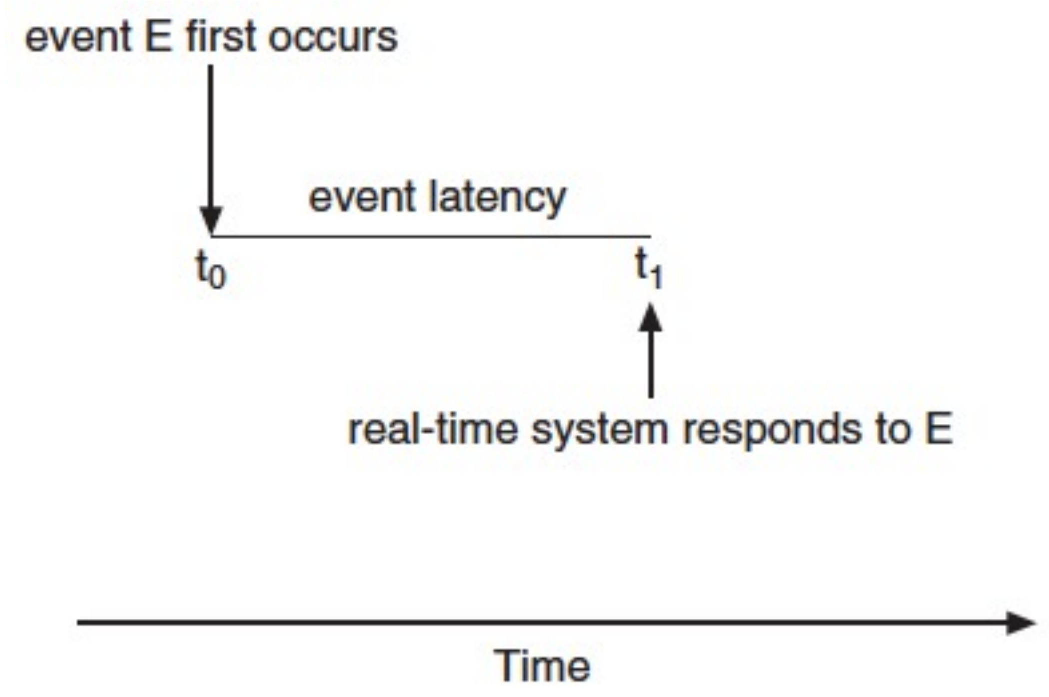
- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- 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 mo ved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 addit ional milliseconds
    - If it still does not complete, it is preempted an d moved to queue Q_2



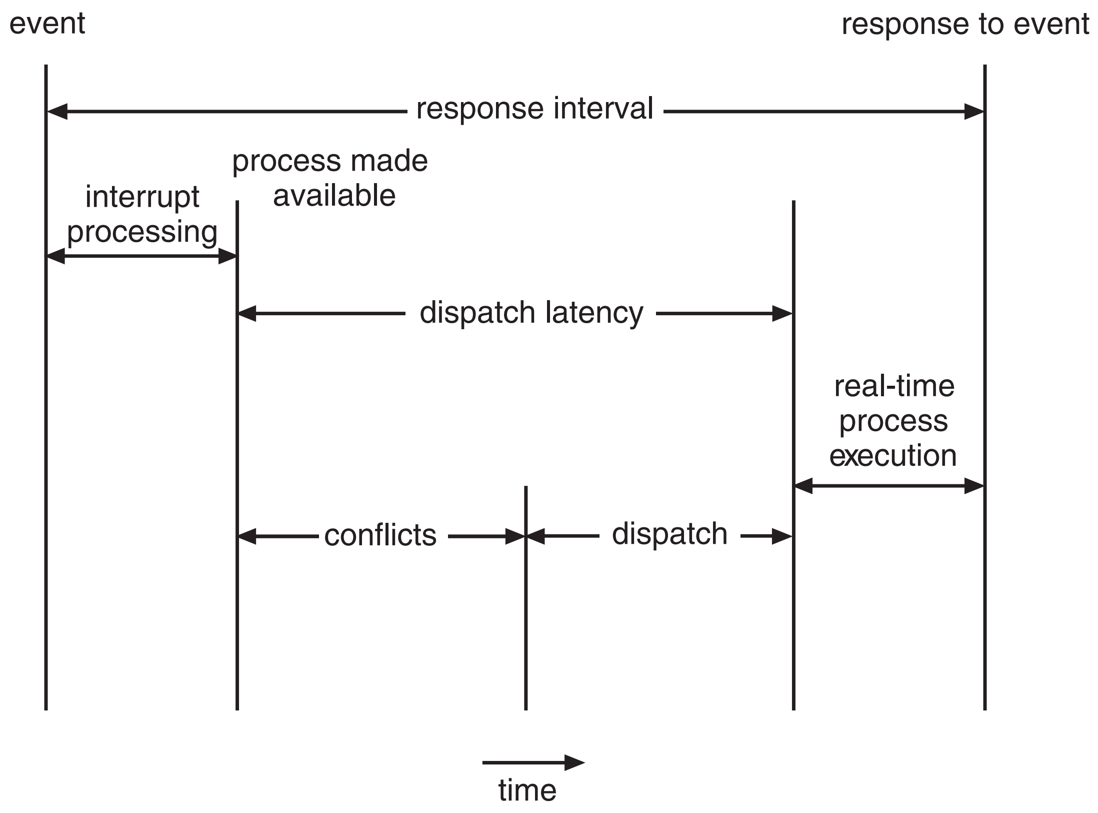
6.6 Real-Time CPU Scheduling

- Real-time system is event-driven
  - **Soft real-time systems** vs. **Hard real-time systems**
- Event latency
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



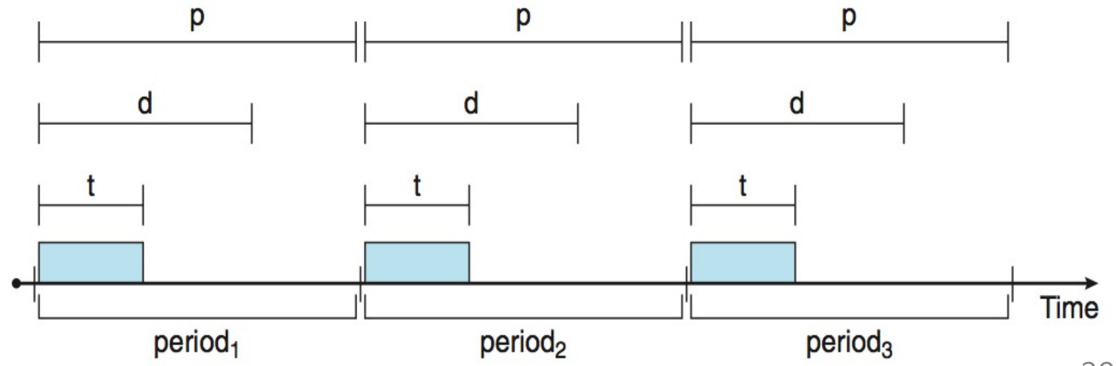
Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



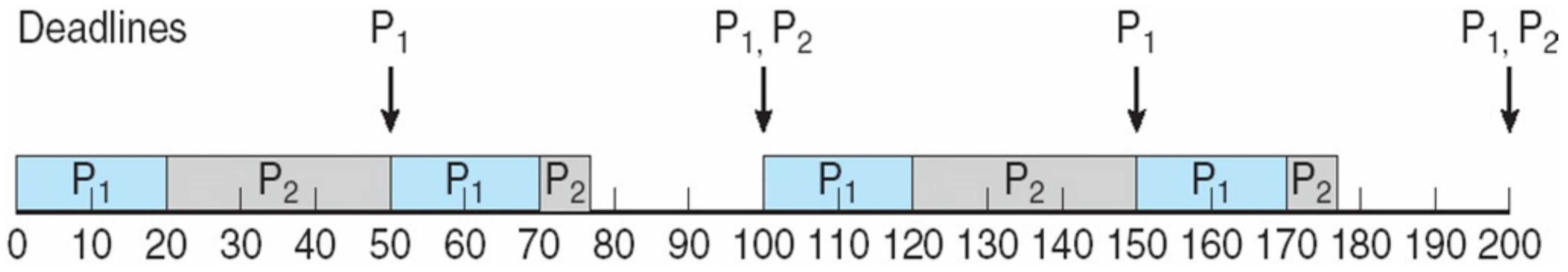
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, prio rity-based scheduling
  - But only guarantee soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at c onstant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$

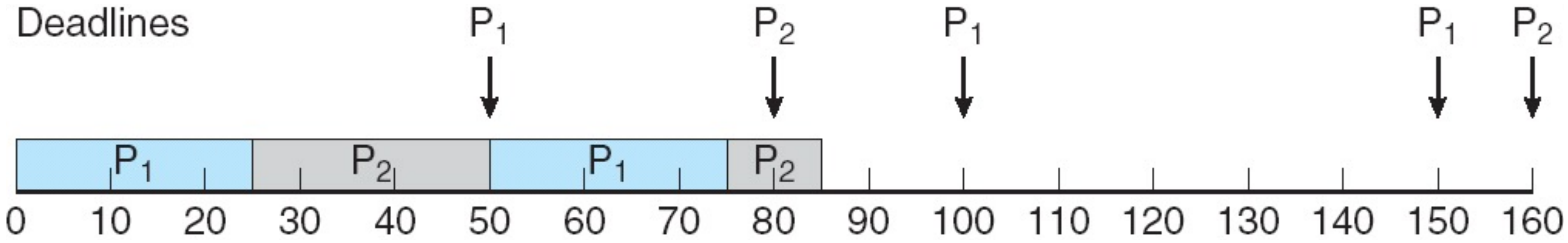


Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority; Longer periods = lower prio rity
- P_1 is assigned a higher priority than P_2.



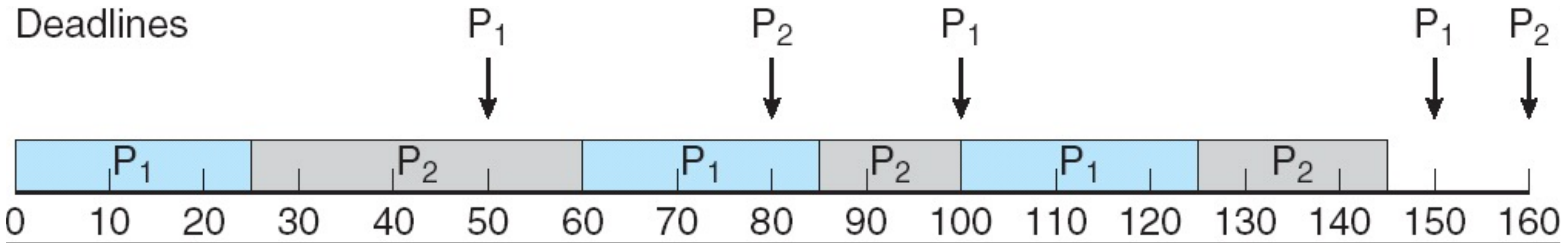
Missed Deadlines with Rate Monotonic Scheduling



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority



Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive $\boldsymbol{N} / T$ of the total pr
ocessor time

6.7 Operating System Examples

- Linux scheduling
- Windows scheduling

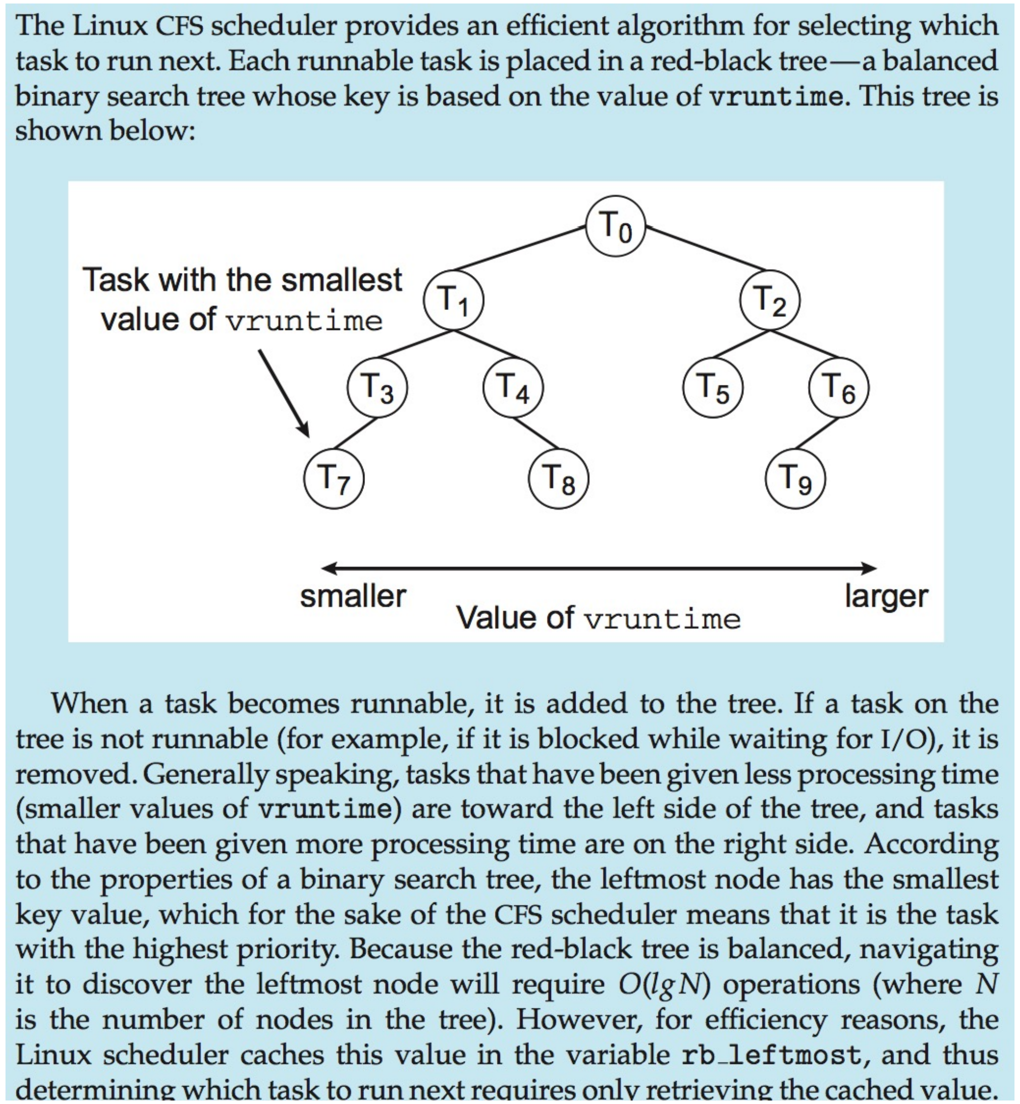
Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduli ng algorithm
- Version 2.5 moved to constant order O(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - Real-time range from 0 to 99 and nice value from 100 to 140
  - Map into global priority with numerically lower values indicating higher pr iority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
  - Worked well, but poor response times for interactive processes

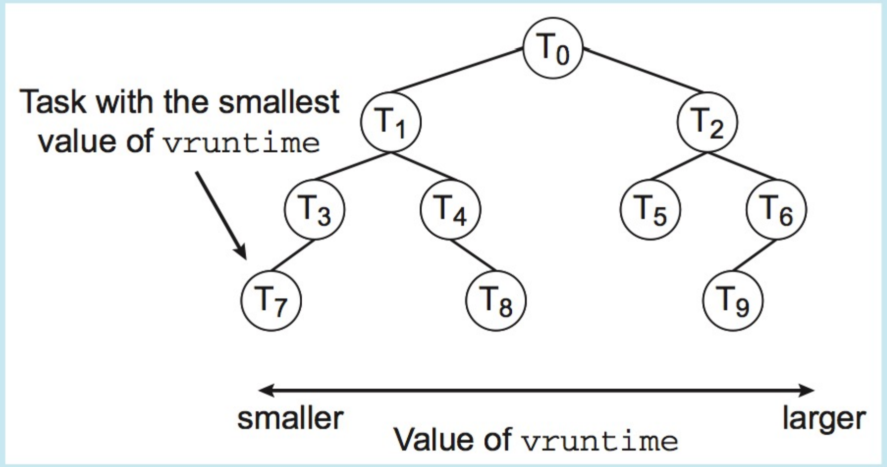
Linux Scheduling in Version 2.6.23 +

- Completely Fair Scheduler (CFS)
- Scheduling classes
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - 2 scheduling classes included, others can be added: default and real-time
- Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - Quantum calculated based on nice value from -20 to +19
  - Lower value is higher priority
  - Calculates target latency – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task virtual run time in variable vruntime
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance



The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- Dispatcher is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by h igher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- Variable class is 1-15, real-time class is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs idle thread

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows 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