Chapter 5

CPU Scheduling

Yunmin Go

School of CSEE

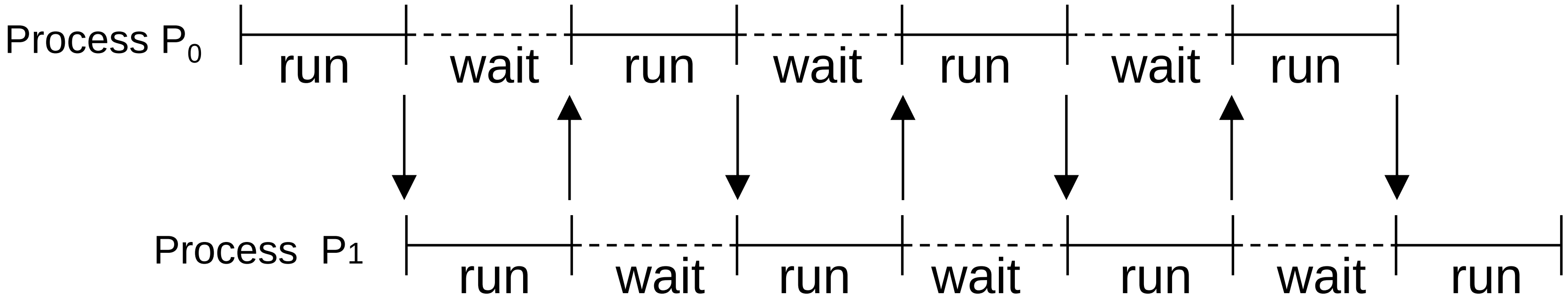
HANDONG GLOBAL UNIVERSITY

Agenda

- Basic Concepts
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Linux Scheduler

Basic Concepts

- Motivation: maximum CPU utilization obtained with multiprogramming and multitasking
  - Resources (including CPU) are shared among processes



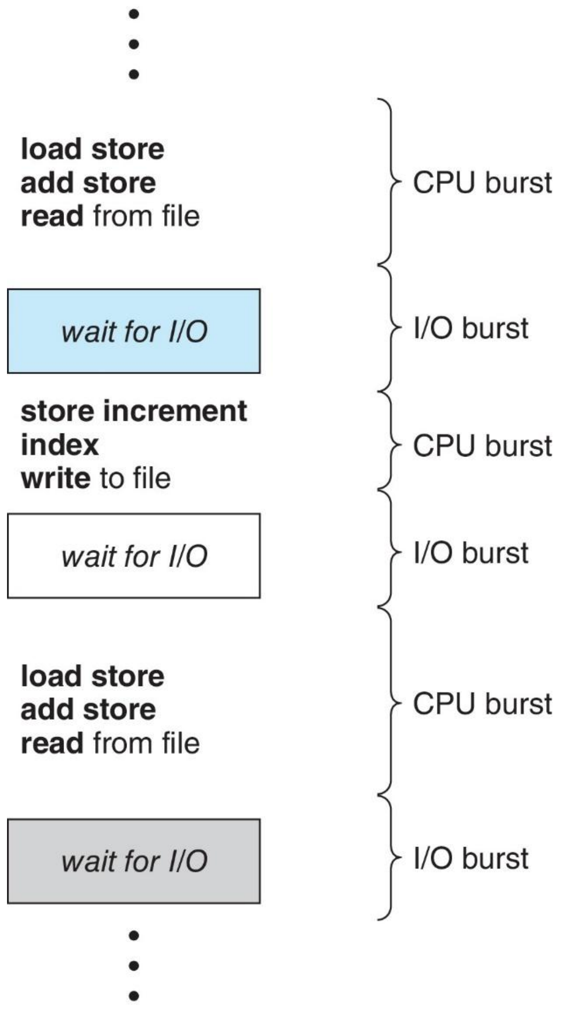
Multiprogramming

CPU-IO Burst Cycle

- Process execution consists of cycle of CPU execution and I/O wait
  - First and last bursts are CPU bursts

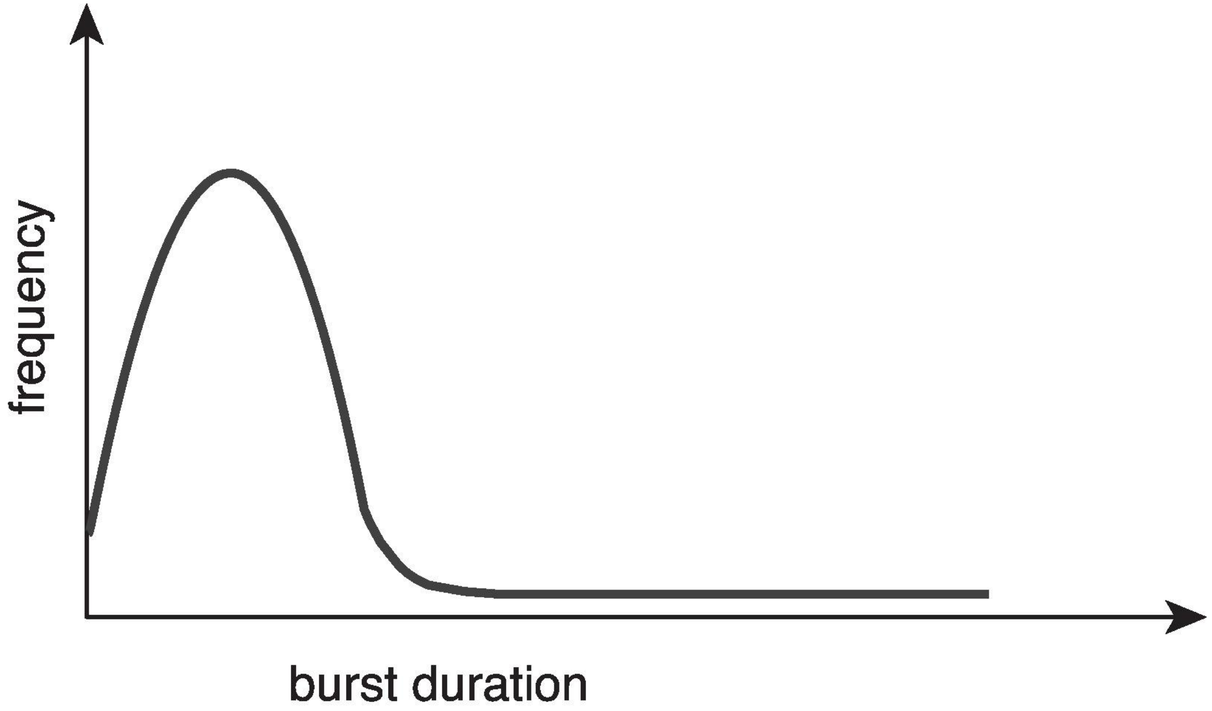
- Types of processes
  - I/O-bound process
    - Consists of many short CPU bursts
  - CPU-bound process
    - Consists of a few long CPU bursts

CPU burst followed by **I/O burst**



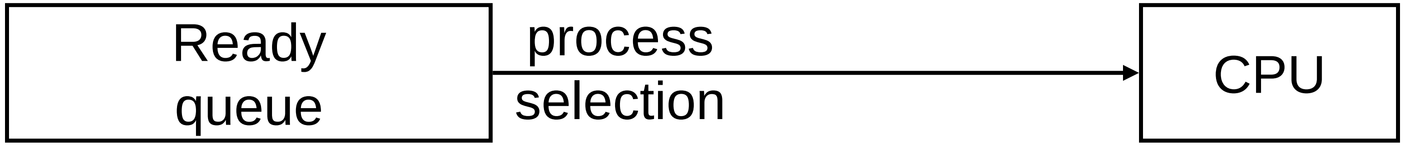
Histogram of CPU-burst Time

- CPU burst distribution is of main concern
  - Large number of short bursts
  - Small number of longer bursts



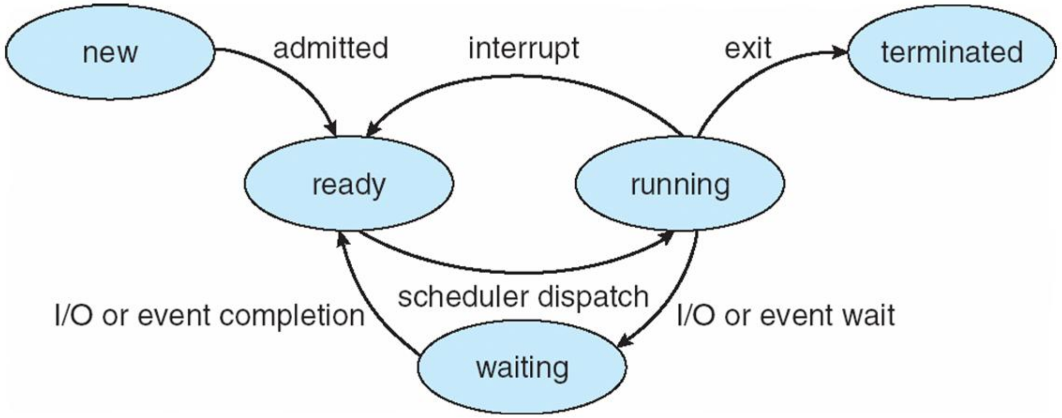
CPU Scheduler

- CPU scheduler (=short term scheduler)
  - CPU scheduler selects a process from ready queue and allocates a CPU core to one of them
- Implementation of ready queue
  - FIFO queue, priority queue, a tree, or simply an ordered linked list
  - Each process is represented by PCB



When Scheduling Occurs?

- 1. Switches from running to waiting state
  - ex) I/O request or invocation of `wait()`
- 2. Switches from running to ready state
  - ex) interrupt
- 3. Switches from waiting to ready
  - ex) completion of I/O
- 4. Terminates



- For situations 1 and 4, there is no choice
- For situations 2 and 3, there is a choice

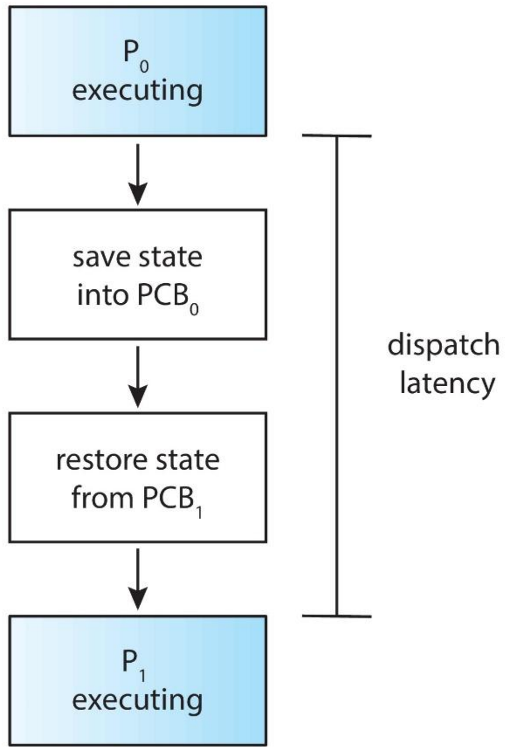
Preemptive Scheduling

- Non-preemptive (or cooperative) scheduling
  - Scheduling can occur at 1, 4 only
  - Running process is not interrupted

- Preemptive scheduling
  - Scheduling can occurs at 1, 2, 3, 4
  - Scheduling may occur while a process is running
    - ex) interrupt, process with higher priority
  - Requires H/W support and shared data handling
  - Preemptive scheduling can result in race conditions when data are shared among several processes

Dispatcher

- Dispatcher: a 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



Scheduling Criteria

- **CPU utilization**: keep the CPU as busy as possible
- **Throughput**: # of processes completed per time unit
- **Turnaround time**: interval from submission of a process to its completion
- **Waiting time**: sum of periods spent waiting in ready queue
- **Response time**: time from submission of a request to first response

→ Importance of each criterion vary with systems

- Measure to optimize
  - Average / minimum / maximum value
  - Variance

Agenda

- Basic Concepts
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Linux Scheduler

Scheduling Algorithms

- First-come, first-served (FCFS) scheduling
- Shortest-job-first (SJF) scheduling
- Priority scheduling
- Round-robin scheduling
- Multilevel queue scheduling
- Multiple feedback-queue scheduling

First-Come, First-Served Scheduling

- Process that requests CPU first, is allocated CPU first
  - Non-preemptive scheduling
  - Simplest scheduling method
- Sometimes average waiting time is quite long
  - CPU, I/O utilities are inefficient
  - Ex) Three processes arrived at time 0

Process	Burst Time	Waiting Time
P_1	24	0
P_2	3	24
P_3	3	27

Average waiting time:
(0 + 24 + 27) / 3 = 17 msec

What if the processes arrive in the order:
P2 , P3 , P1?

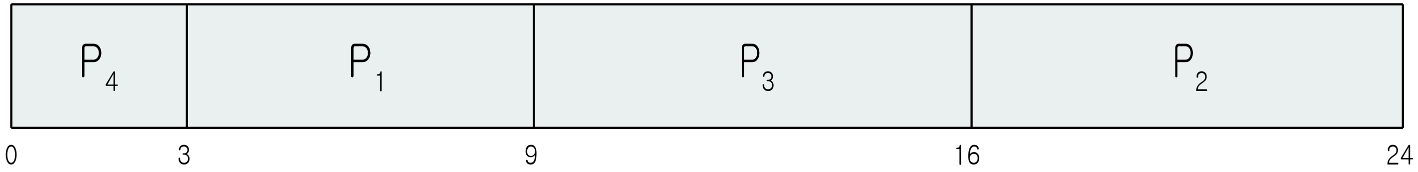
Gantt Chart

Shortest-Job-First Scheduling

- Assign to the process with the smallest next CPU burst
  - Ex) Four processes arrived at time 0

Process	Burst Time	Waiting Time
P_1	6	3
P_2	8	16
P_3	7	9
P_4	3	0

Average waiting time:
(3 + 16 + 9 + 0) / 4 = 7 msec



  - SJF algorithm is optimal in minimum waiting time
  - Problem: difficult to know length of next CPU burst

Shortest-Job-First Scheduling

- Predicting next CPU burst from history
  - Exponential averaging

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

    - t_n : actual length of n-th CPU burst
    - τ_n : prediction for n-th CPU burst
    - α : a coefficient between 0 and 1
      - $\alpha = 0$: recent history has no effect
      - $\alpha = 1$: only recent history matters
      - Usually, $\alpha = 0.5$
    - If we expand the formula, we get:

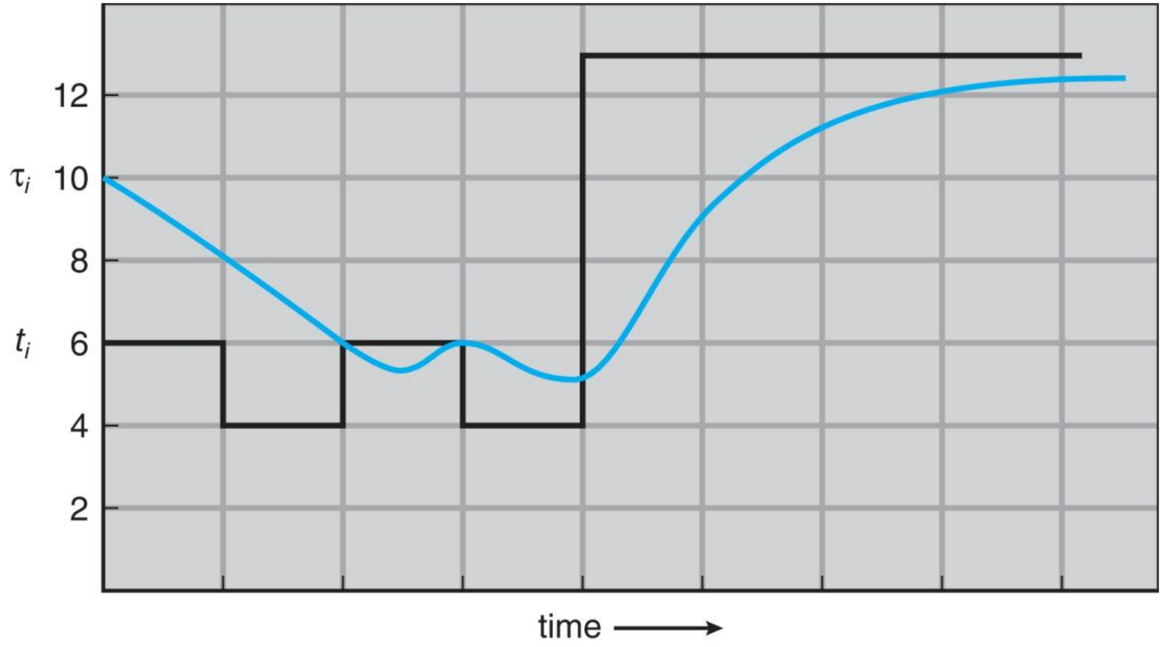
$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$$

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

Shortest-Job-First Scheduling

- Prediction of CPU Burst

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



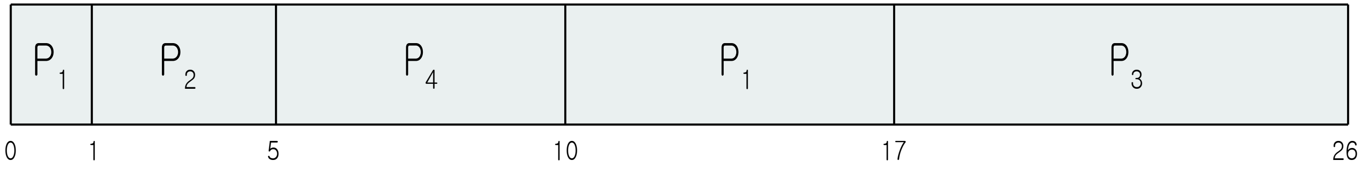
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Shortest-Job-First Scheduling

- Preemptive version of SJF scheduling
  - Shortest-remaining-time-first scheduling

Process	Arrival Time	Burst Time	Waiting Time
P_1	0	8	9 (= 10 – 1)
P_2	1	4	0 (= 1 – 1)
P_3	2	9	15 (= 17 – 2)
P_4	3	5	2 = (5 – 3)

Average waiting time:
(9 + 0 + 15 + 2) / 4 = 26 / 4 = 6.5



Round-Robin Scheduling

- Similar to FCFS, but it's preemptive
  - Designed for time-sharing systems
  - CPU time is divided into time quantum (or time slice)
    - A time quantum is 10~100 msec.

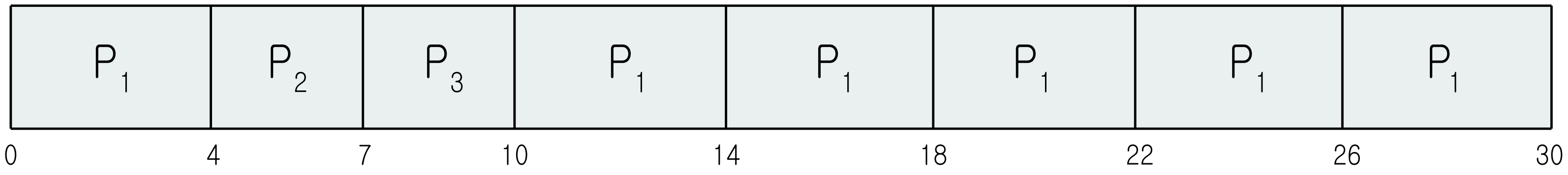
    Cf. switching latency: 10 μsec
  - Ready queue is treated as **circular queue**
  - CPU scheduler goes around the ready queue and allocate CPU time up to 1 time quantum

Round-Robin Scheduling

- Example of RR scheduling with time quantum = 4
  - Ex) Three processes arrived at time 0

Process	Burst Time	Waiting Time
P_1	24	6
P_2	3	4
P_3	3	7

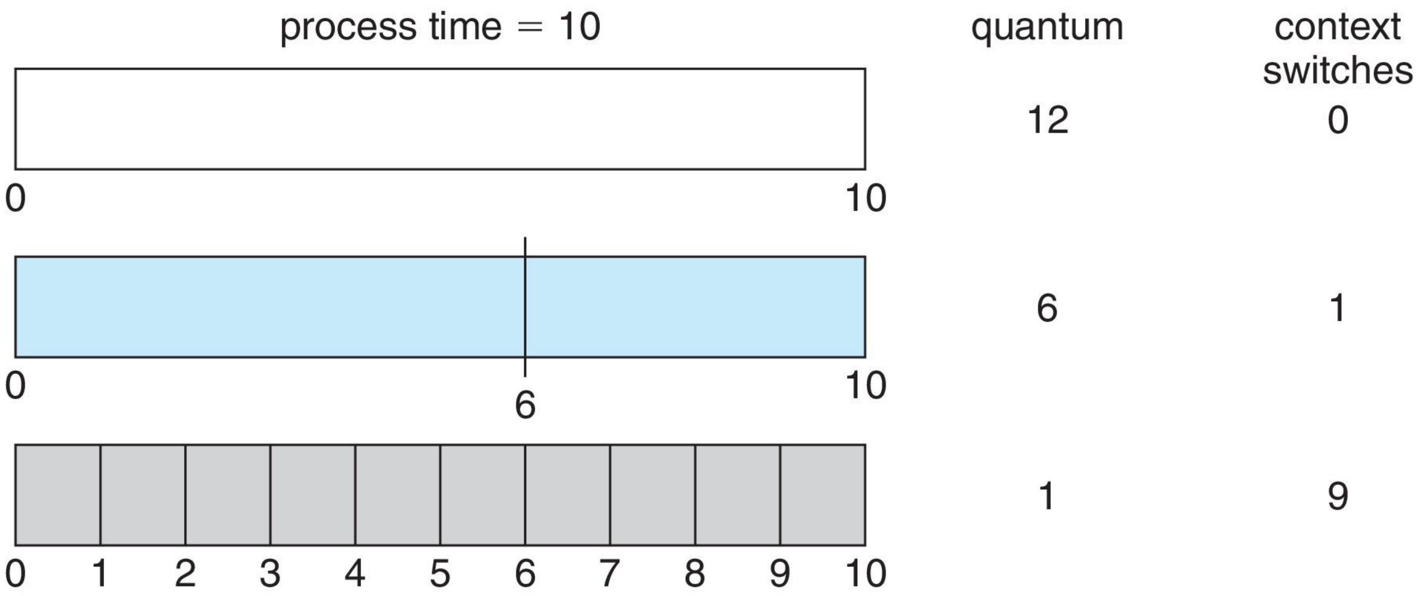
Average waiting time:
(6 + 4 + 7) / 3 = 5.66



HANDONG GLOBAL UNIVERSITY

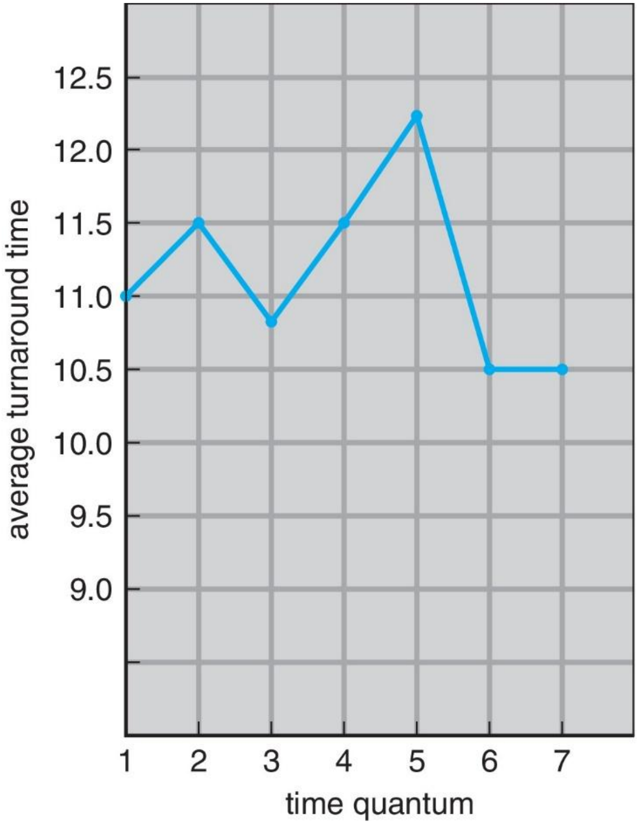
Round-Robin Scheduling

- Performance of RR scheduling heavily depends on size of time quantum
  - Time quantum is small: processor sharing
  - Time quantum is large: FCFS
- Context switching overhead depends on size of time quantum



Round-Robin Scheduling

- Turnaround time also depends on size of time quantum
  - Average turnaround time is not proportional nor inverse-proportional to size of time quantum
  - Average turnaround time is improved if most processes finish their next CPU burst in a single time quantum
  - However, too long time quantum is not desirable
  - A rule of thumb: about 80% of CPU burst should be shorter than time quantum



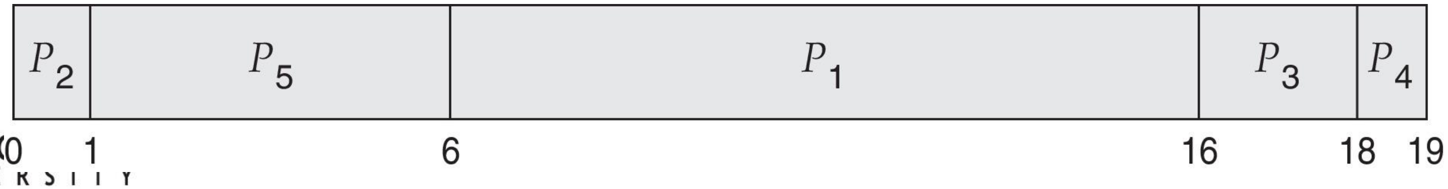
process	time
P_1	6
P_2	3
P_3	1
P_4	7

Priority Scheduling

- CPU is allocated the process with the highest priority
  - Each process has its priority
    - In this text, lower number means higher priority
  - Equal-priority processes: FCFS
  - Ex) Five processes arrived at time 0

Process	Burst Time	Priority	Waiting Time
P_1	10	3	6
P_2	1	1	0
P_3	2	4	16
P_4	1	5	18
P_5	5	2	1

Average waiting time:
(6 + 0 + 16 + 18 + 1) / 5 = 8.2



Priority Scheduling

- Priority can be assigned internally and externally
  - Internally: determined by measurable quantity or qualities
    - Time limit, memory requirement, # of open files, ratio of I/O burst and CPU burst, …
  - Externally: importance, political factors
- Priority scheduling can be either preemptive or non-preemptive.
- Major problems
  - **Indefinite blocking** (= **starvation**) of processes with lower priorities

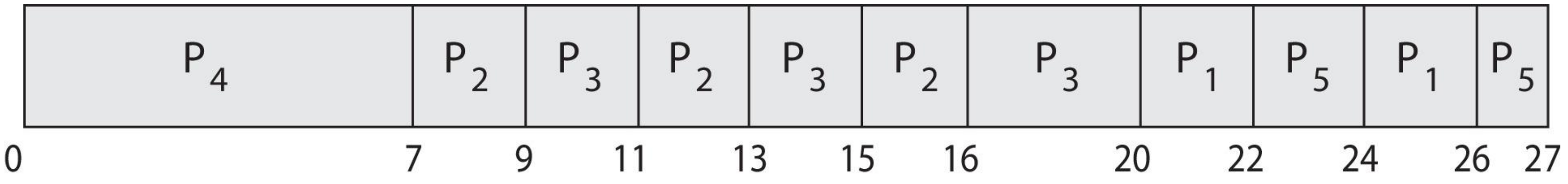
  → Solution: **aging** (gradually increase priority of processes waiting for long time)

Priority Scheduling

- Priority scheduling with Round-Robin scheduling
  - Run the process with the highest priority
  - Processes with the same priority run round-robin
  - Ex) 2 msec time quantum for RR

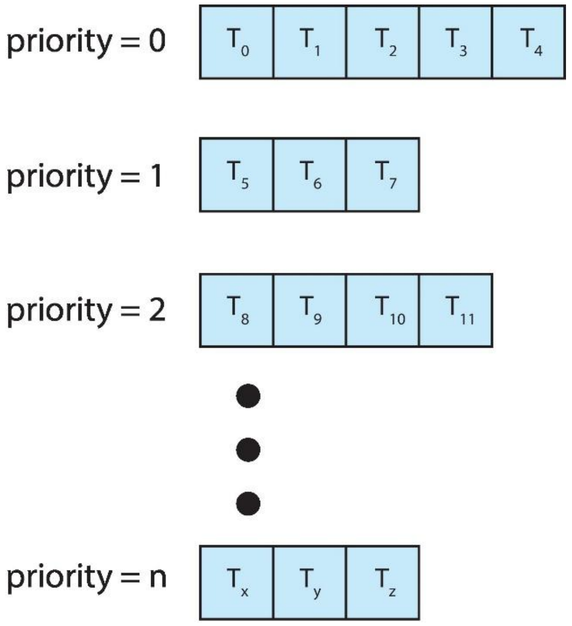
Process	Burst Time	Priority	Waiting Time
P_1	4	3	22 (=20+2)
P_2	5	2	11 (=7+2+2)
P_3	8	2	12 (=9+2+1)
P_4	7	1	0
P_5	3	3	24 (=22+2)

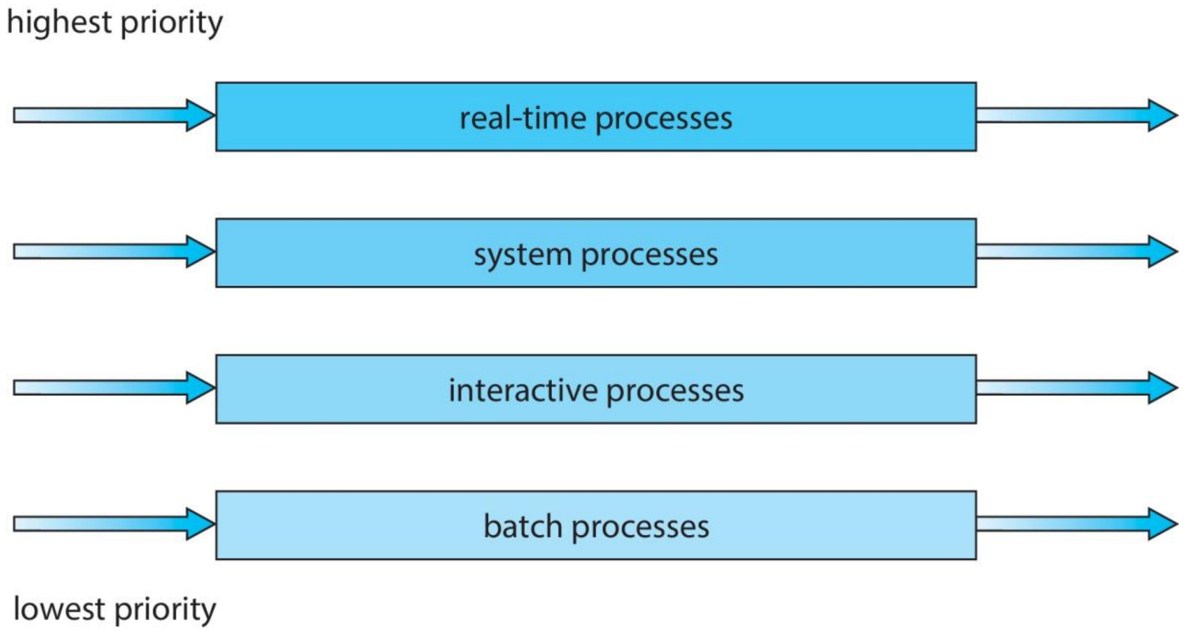
Average waiting time:
(22 + 11 + 12 + 0 + 24) / 5 = 13.8



Multilevel Queue Scheduling

- Classify processes into different groups and apply different scheduling
  - Memory requirement, priority, process type, …
- Partition ready queue into several separate queues





Multilevel Queue Scheduling

- Each queue has its own scheduling algorithm
- Scheduling among queues
  - Fixed-priority preemptive scheduling
    - A process in lower priority queue can run only when all of higher priority queues all empty
  - Time-slice among queues

    Ex) foreground queue (interactive processes): 80% of the CPU time for RR

    background queue (batch processes): 20% of the CPU time for FCFS

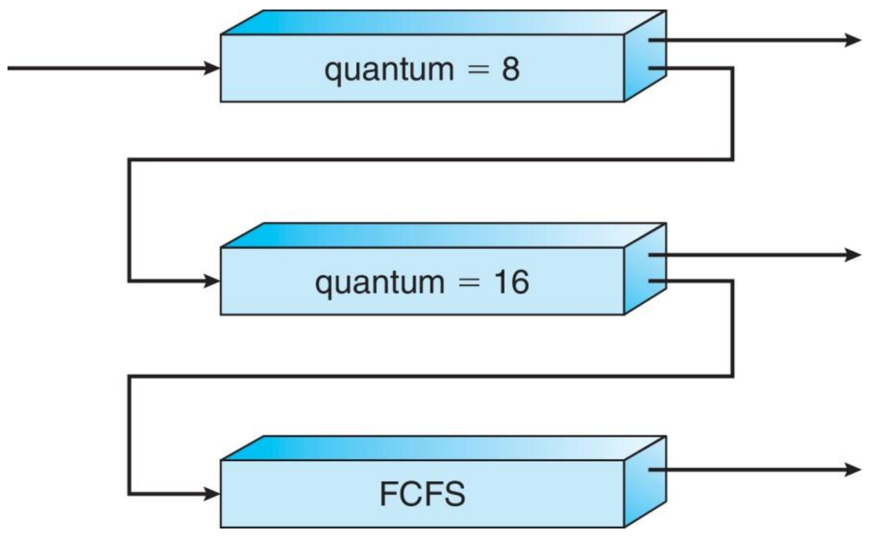
Multilevel Feedback-Queue Scheduling

- Similar to multilevel queue scheduling, but a process can move between queues
- Idea: separate processes according to characteristics of their CPU bursts.
  - If a process uses too much CPU time, move it to lower priority queue
  - I/O-bound, interactive processes are in higher priority queues

Multilevel Feedback-Queue Scheduling

- Example of multilevel feedback queue
  - Ready queue consists of three queues (0~2)
  - Q_0 : RR with time quantum 8 msec
  - Q_1 : RR with time quantum 16 msec
  - Q_2 : FCFS

→ A new process is put in Q_0.
If it exceeds time limit, it moves to lower priority queue



Multilevel Feedback-Queue Scheduling

- Parameters to define a multilevel feedback-queue scheduler
  - # of queues
  - Scheduling algorithm for each queue
  - Method to determine when to upgrade a process to higher priority queue
  - Method to determine when to demote a process to lower priority queue
  - Method to determine which queue a process will enter when it needs service

→ The most complex algorithm

Agenda

- Basic Concepts
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Operating System Example

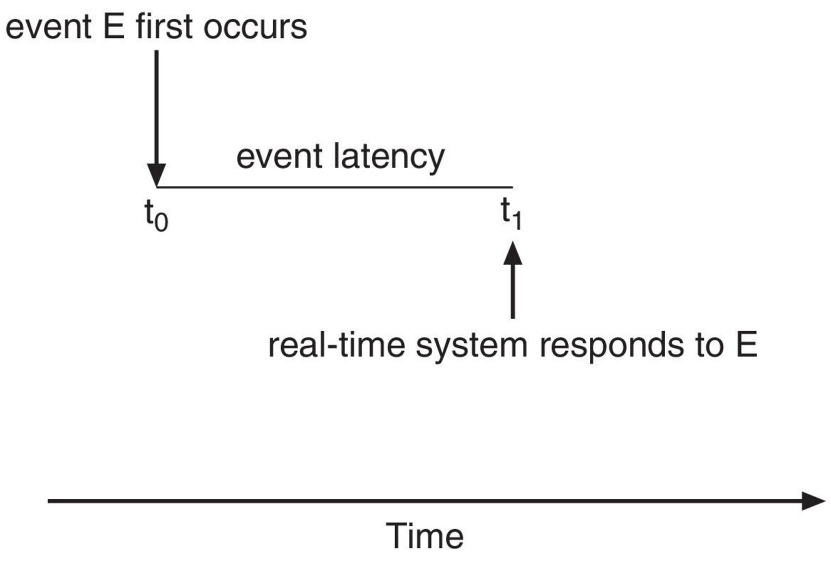
Real-time CPU Scheduling

- Real-time Operating Systems (RTOS)
  - OS intended to serve real-time systems

  Ex) Antirock brake system (ABS) requires latency of 3~5 msec
- Soft real-time systems
  - No guarantee as to when a critical real-time process will be scheduled.
  - Guarantee only that the process will be given preference over noncritical processes.
- Hard real-time systems
  - A task must be serviced by its deadline
  - Service after the deadline has expired is the same as no service at all.

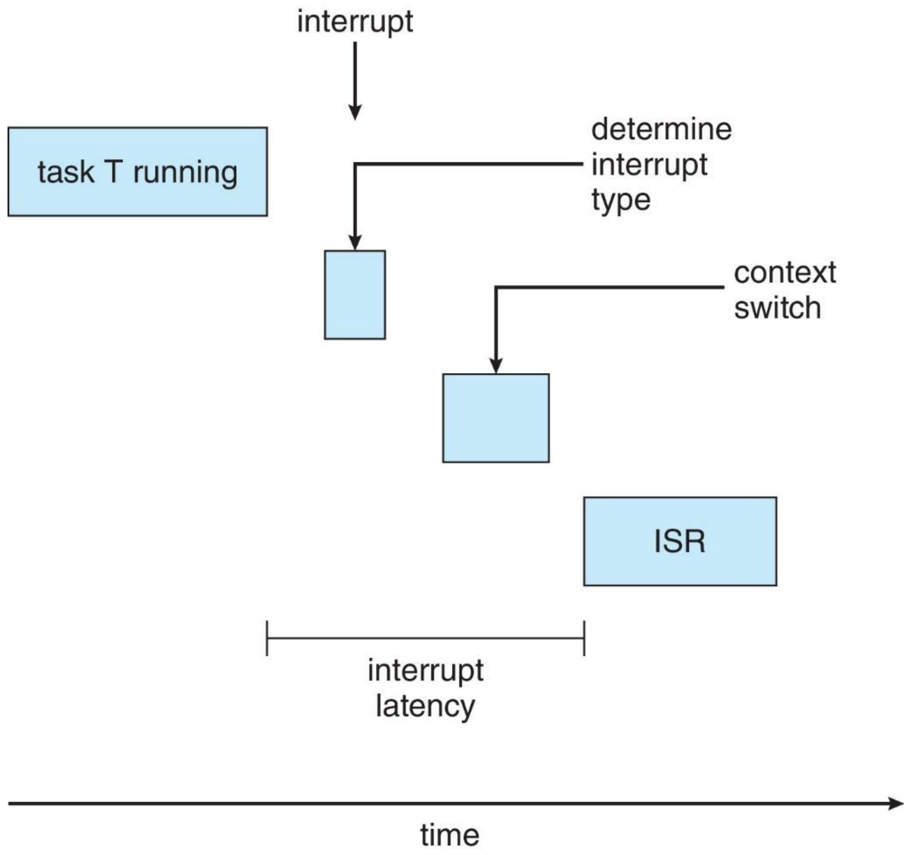
Minimizing Latency

- Most real-time systems are event-driven system
  - When an event occurs, the system must respond to and service it as quickly as possible
- **Event latency**: the amount of time that elapses from when an event occurs to when it is serviced
  - Interrupt latency, dispatch latency



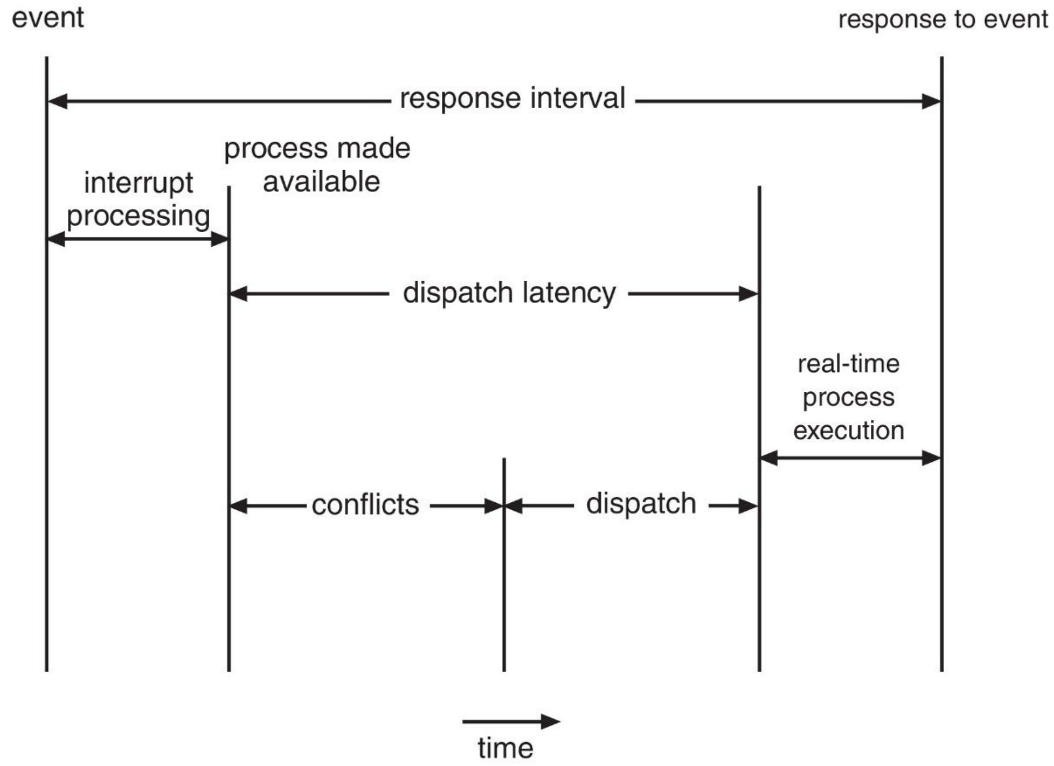
Interrupt Latency

- **Interrupt latency**: the period of time from the arrival of interrupt at the CPU to the start of the routine that services the interrupt.



Dispatch Latency

- **Dispatch latency**: the amount of time required for the scheduling dispatcher to stop one process and start another
  - Preemptive kernels are the most effective technique to keep dispatch latency low
- 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



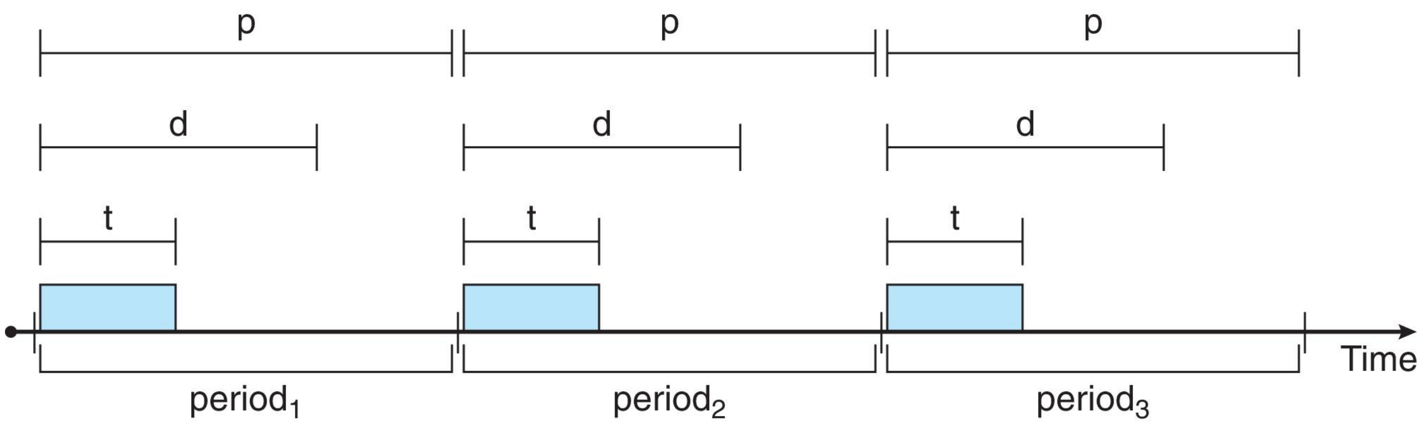
Priority-based Scheduling

- The most important feature of RTOS is to respond immediately to a real-time process
  - The scheduler should support priority-based preemptive algorithm
  - Most OS assign highest priority to real-time processes
  - Preemptive, priority-based scheduler only guarantees soft real-time functionality.
- Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements
  - Requires additional scheduling features.

Periodic Process

- Periodic processes require the CPU at constant intervals (periods).
  - Processing time: t
  - Deadline: d (by which it must be serviced by the CPU)
  - Period: p
  - Rate of a periodic task: $1/p$

$$0 \leq t \leq d \leq p$$



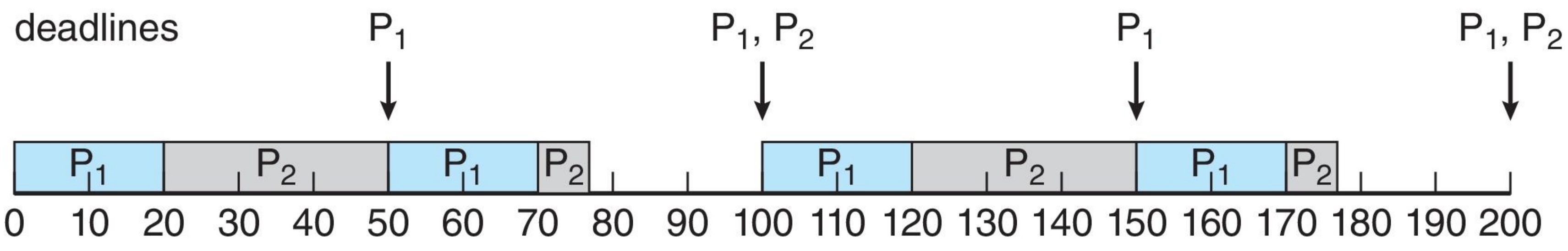
Scheduling with Deadline Requirement

- A process may have to announce its deadline requirements to the scheduler
- Using an admission-control algorithm, the scheduler does one of two things:
  - **Admits the process**, guaranteeing that the process will complete on time, or
  - **Rejects the request** as impossible if it cannot guarantee that the task will be serviced by its deadline.

Rate-Monotonic Scheduling

- Rate-monotonic scheduling algorithm
  - Schedules periodic tasks using **a static priority policy with preemption**
  - Each periodic task is assigned a priority **inversely based on its period**
    - Shorter periods → high priority
    - Longer periods → lower priority
  - The processing time of a periodic process is assumed the same for each CPU burst

Process P_1: $p_1 = 50$, $t_1 = 20$, Process P_2: $p_2 = 100$, $t_2 = 35$

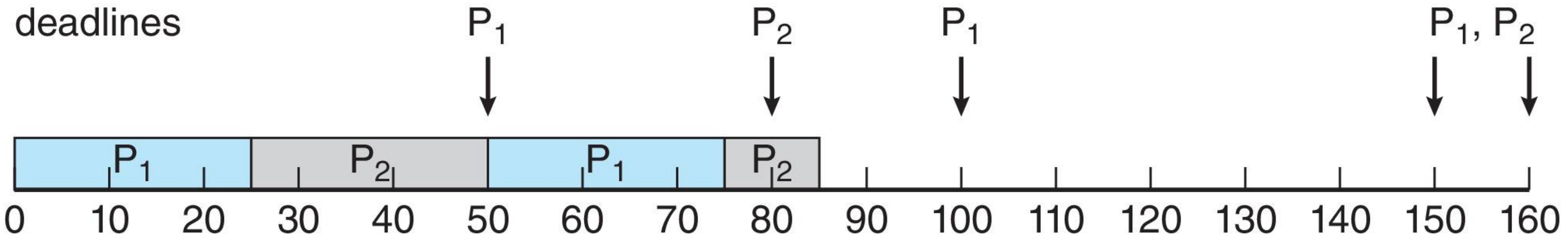


Rate-Monotonic Scheduling

- Rate-monotonic scheduling is considered optimal
  - If a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.
- Limitation
  - CPU utilization is bounded, and it is not always possible to maximize CPU resources fully.
  - Worst-case CPU utilization: $N(2^{1/N}-1)$
    - N: # of processes

Process P_1: $p_1 = 50$, $t_1 = 25$, Process P_2: $p_2 = 80$, $t_2 = 35$

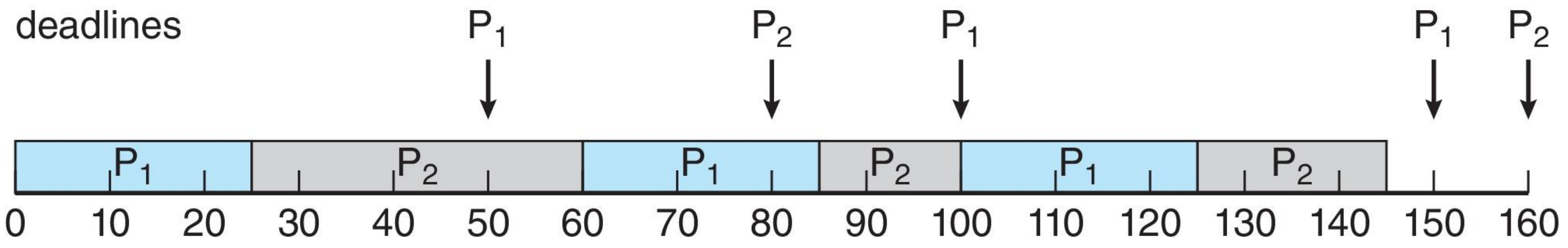
Process P2 misses finishing its deadline at time 80



Earliest-Deadline-First Scheduling

- Earliest-deadline-first (EDF) scheduling
  - Priorities are assigned according to deadline
    - The earlier the deadline, the higher the priority
    - The later the deadline, the lower priority

Process P_1: $p_1 = 50$, $t_1 = 25$, Process P_2: $p_2 = 80$, $t_2 = 35$



Earliest-Deadline-First Scheduling

- Requirements
  - Does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst
  - When a process becomes runnable, it must announce its deadline requirements to the system
- Theoretically optimal
  - Theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent.

Proportional Share Scheduling

- Proportional share schedulers
  - Allocates T shares among all applications.
  - An application can receive N shares of time, thus ensuring that the application will have $N\,/\,T$ of the total processor time
  - Must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time

Agenda

- Basic Concepts
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Linux Scheduler

Linux Scheduler

- History
  - Traditional UNIX scheduling algorithm (ver. < 2.5)
    - Not adequate support for SMP system
    - Not scale well as # of tasks increases
  - O(1) scheduling algorithm (ver. ≥ 2.5)
    - O(1) complexity
    - Increased support for SMP
    - Poor response time for the interactive processes
  - Completely Fair Scheduler (CFS) (ver. ≥ 2.6)

Linux Scheduler

- Based on scheduling classes
  - Each class is assigned a specific priority
    - Default scheduling class (CFS algorithm)
      - Static priority 100 ~ 139
    - Real-time scheduling class (SCHED_FIFO, SCHED_RR)
      - Static priority 0 ~ 99
    - If necessary, new scheduling classes can be added
  - Allows different scheduling algorithms based on the needs of the system (e.g. server vs. mobile)
  - Scheduler selects highest-priority task belonging to highest-priority class

Linux CFS Scheduler

- CFS scheduler assigns a proportion of CPU processing time to each task
  - The portion is calculated from nice value
    - Nice value: relative priority in [-20, +19], default is 0
      - Higher value represents lower priority ('nice' to other tasks)
  - Proportions of CPU time are allocated from the value of targeted latency
    - Targeted latency: interval of time during which every runnable task should run at least once
      - Can increase if the number of active tasks in the system grows beyond a certain threshold

Linux CFS Scheduler

- Records how long each task has run
  - per task variable **vruntime** (virtual run time)
  - Normal priority tasks: vruntime = <actual physical run time>
- vruntime of each task is associated with a decay factor
  - Lower priority task: higher decay factor
  - Higher priority task: lower decay factor

  ➔ Higher priority tasks are likely to have smaller vruntime
- The scheduler simply selects the task that has **the smallest vruntime value**
- Higher-priority task can preempt a lower-priority task.

Linux CFS Scheduler

- I/O-bound tasks: run only for short periods before blocking for additional I/O

  → vruntime will be lower

- CPU-bound tasks: exhaust its time period whenever it has an opportunity to run on a processor

  → vruntime will be higher

→ Gives I/O-bound tasks higher priority than CPU-bound tasks