고려대학교
KOREA UNIVERSITY

KU-The Future

Operating Systems (10th Ed., by A. Silberschatz)

Chapter 5 CPU Scheduling

Heonchang Yu

Distributed and Cloud Computing Lab.

Contents

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

Objectives

- Describe various CPU scheduling algorithms.
- Assess CPU scheduling algorithms based on scheduling criteria.
- Explain the issues related to multiprocessor and multicore scheduling.
- Describe various real-time scheduling algorithms.
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems.
- Apply modeling and simulations to evaluate CPU scheduling algorithms.
- Design a program that implements several different CPU scheduling algorithms.

Basic Concepts

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait (Figure 5.1)
- CPU burst distribution – Figure 5.2
  - ✓ A large number of short CPU bursts and a small number of long CPU bursts
  - ✓ I/O-bound program has many short CPU bursts
  - ✓ CPU-bound program might have a few long CPU bursts

Alternating Sequence of CPU And I/O Bursts

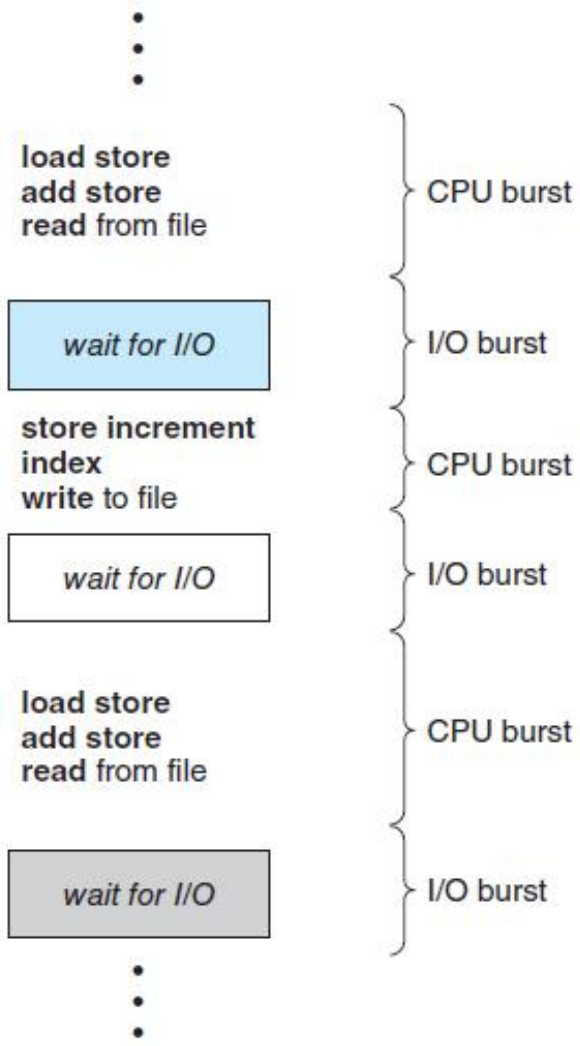


Figure 5.1 Alternating sequence of CPU and I/O bursts.

Histogram of CPU-burst Times

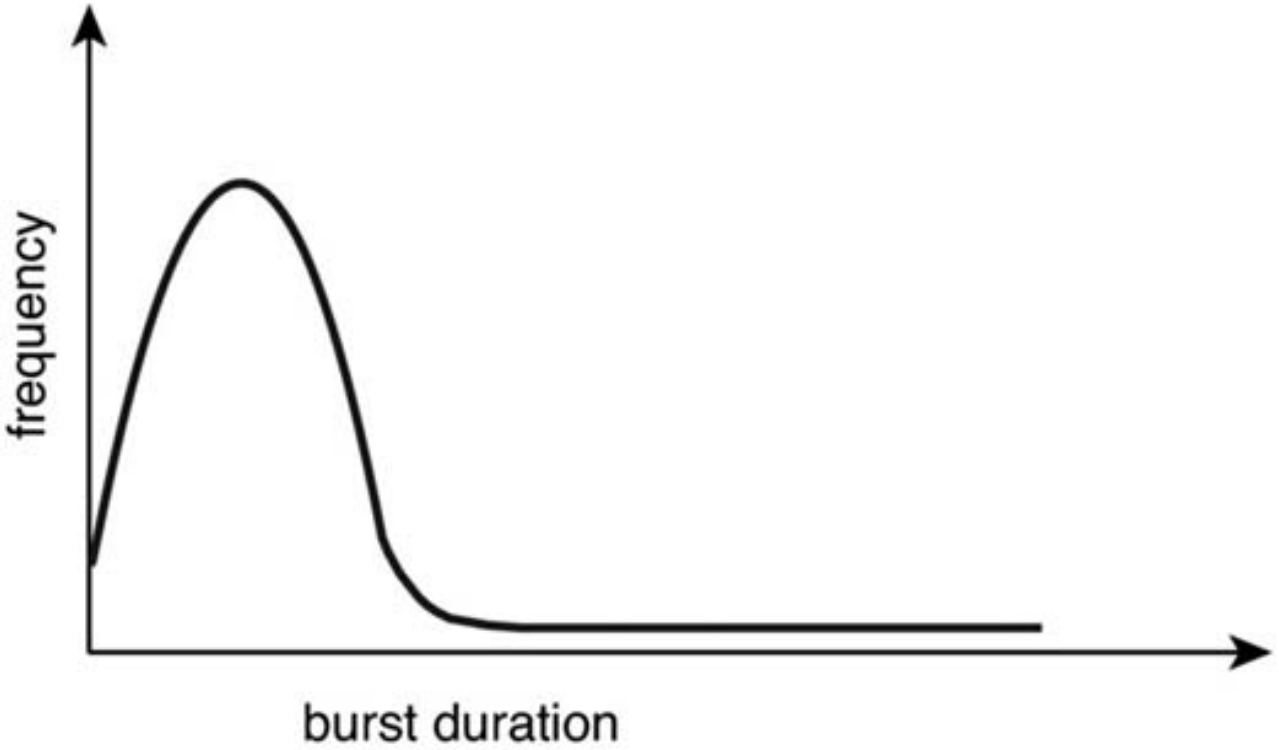


Figure 5.2 Histogram of CPU-burst durations.

CPU Scheduler

- Selects a process from the processes in memory that are ready to execute and allocates the CPU to that process
- CPU-scheduling decisions may take place when a process:
  - ✓ Switches from the running state to the waiting state
  - ✓ Switches from the running state to the ready state
  - ✓ Switches from the waiting state to the ready state
  - ✓ Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU's core to the process selected by the CPU scheduler; This function involves:
  - ✓ Switching context
  - ✓ Switching to user mode
  - ✓ Jumping to the proper location in the user program to resume that program
- *Dispatch latency* – the time it takes for the dispatcher to stop one process and start another running

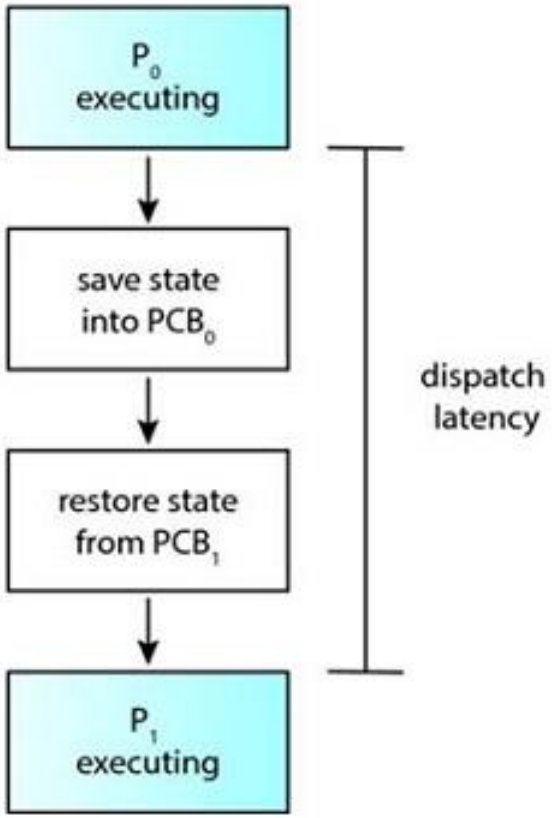


Figure 5.3 The role of the dispatcher.

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – the number of processes that are completed per time unit
- Turnaround time – the interval from the time of submission of a process to the time of completion
- Waiting time – the sum of the periods spent waiting in the ready queue
- Response time – the time from the submission of a request until the first response is produced, **not** the time it takes to output the response
- To maximize CPU utilization and throughput
- To minimize turnaround time, waiting time, and response time

First-Come, First-Served (FCFS) Scheduling

- The process that requests the CPU first is allocated the CPU first.
- Is managed with a FIFO queue

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
  - ✓ Turnaround time for P_1 = 24; P_2 = 27; P_3 = 30
    - ❖ Average turnaround time: (24 + 27 + 30)/3 = 27

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

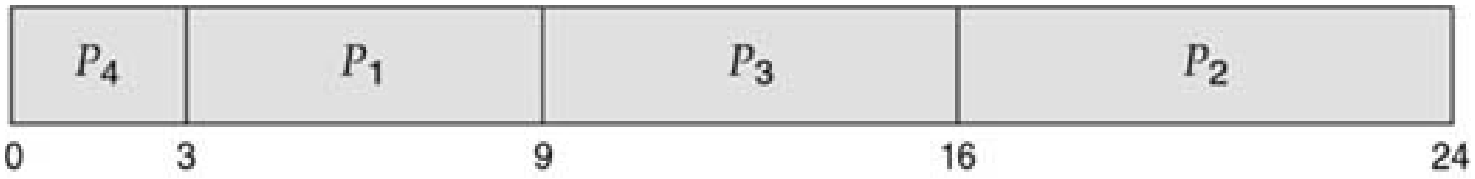
  - ✓ The Gantt chart for the schedule is:

  - ✓ Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
    - ❖ Average waiting time: (6 + 0 + 3)/3 = 3
  - ✓ Turnaround time for $P_1 = 30; P_2 = 3; P_3 = 6$
    - ❖ Average turnaround time: (30 + 3 + 6)/3 = 13
  - ✓ Much better than previous case
  - ✓ **Convoy effect** : All the other processes wait for the one big process to get off the CPU.

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of the process's next CPU burst.
  - ✓ It is assigned to the process that has the smallest next CPU burst
    ⇒ shortest-next-CPU-burst algorithm

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



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

Shortest-Job-First (SJF) Scheduling

- Two schemes:
  - ✓ nonpreemptive – allow the currently running process to finish its CPU burst
  - ✓ preemptive – preempt the currently executing process
    - ❖ Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives the minimum average waiting time for a given set of processes
- Difficulty : knowing the length of the next CPU request
  - ✓ We can use as the length the process time limit that a user specifies when he submits the job
  - ✓ We may be able to predict its value.

Determining Length of Next CPU Burst

- The next CPU burst is predicted as an exponential average of the measured length of previous CPU bursts
  1. t_n : actual length of n^{th} CPU burst
  2. τ_{n+1} : predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq 1$
  4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\tau_n$

Examples of Exponential Averaging

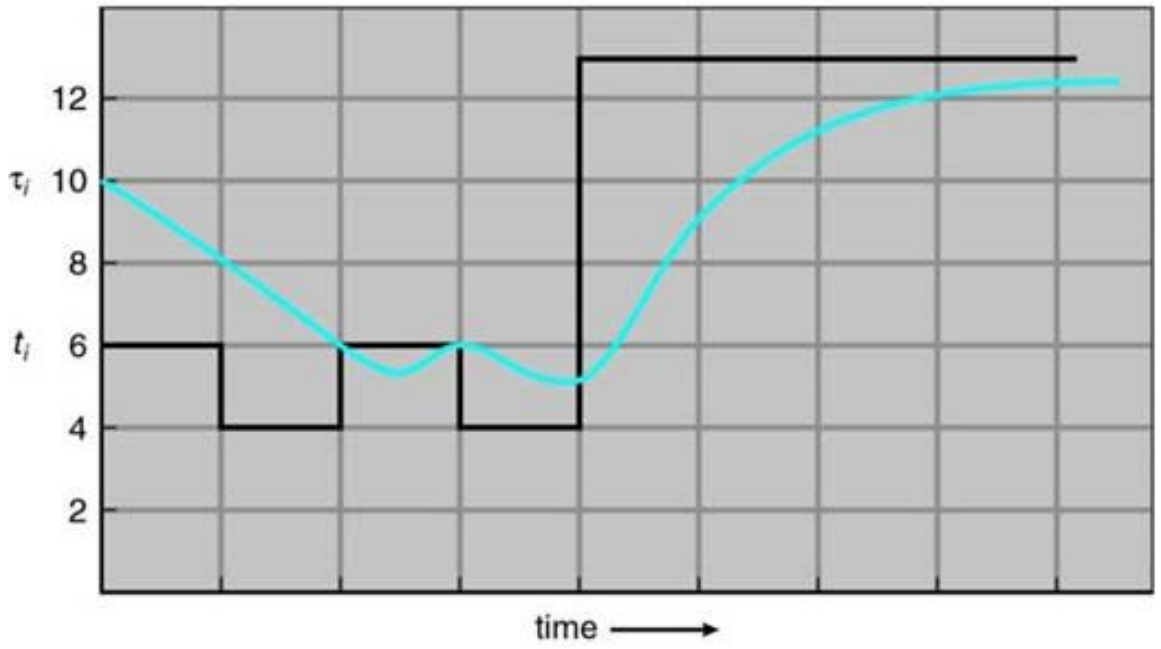
- $\alpha = 0$
  - ✓ $\tau_{n+1} = \tau_n$
  - ✓ Recent history has no effect
- $\alpha = 1$
  - ✓ $\tau_{n+1} = t_n$
  - ✓ Only the most recent CPU burst matters
- $\alpha = 1/2$
  - ✓ Recent history and past history are equally weighted
- If we expand the formula for τ_{n+1} by substituting for τ_n, we get:

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

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

Prediction of the Length of the Next CPU Burst

- An exponential average with α = 1/2 and τ_0=10



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

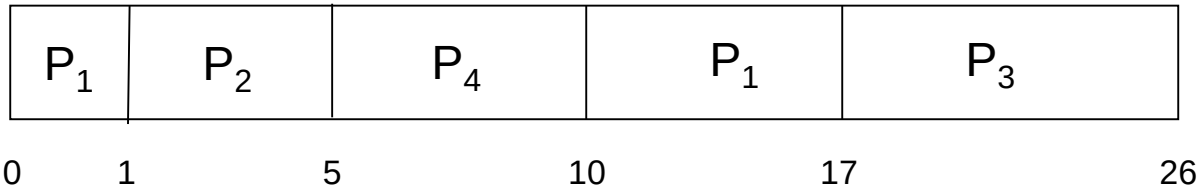
Figure 5.4 Prediction of the length of the next CPU burst.

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the 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

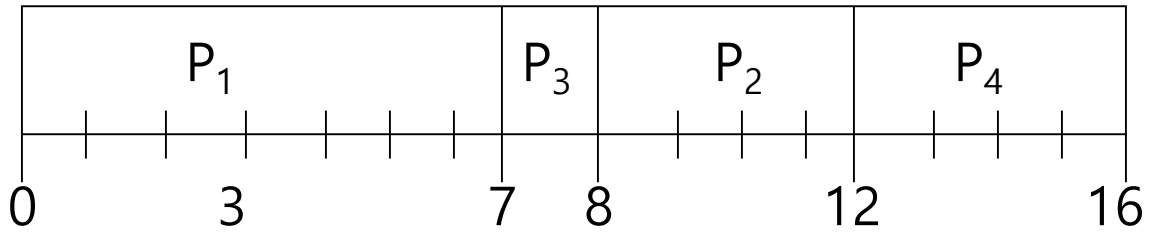


✓ Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 ms

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)

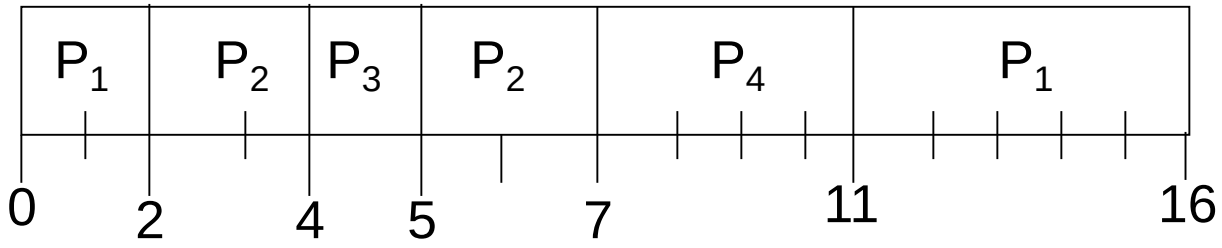


✓ Average waiting time = (0 + 6 + 3 + 7)/4 = 4

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

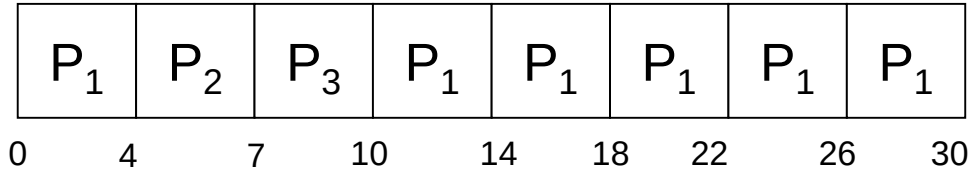
Round Robin (RR)

- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- 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.
  - ✓ Each process must wait no longer than $(n-1)q$ time units until its next time quantum.
- Performance of the RR algorithm
  - ✓ If the time quantum is extremely large ⇒ FCFS
  - ✓ If the time quantum is extremely small ⇒ a large number of context switches

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



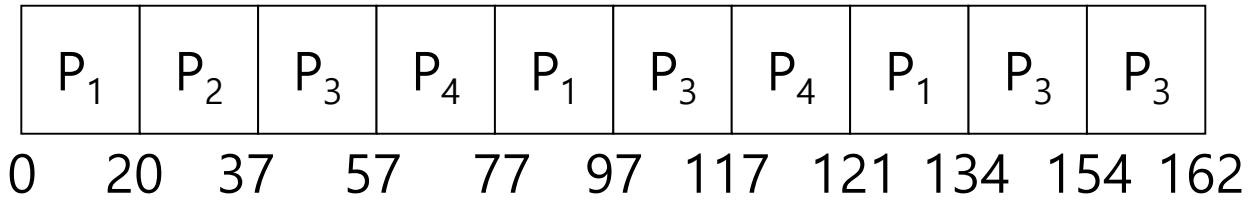
✓ Average waiting time = [(10-4) + 4 + 7]/3 = 17/3 = 5.66

✓ Average turnaround time = (30 + 7 + 10)/3 = 47/3 = 15.67

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

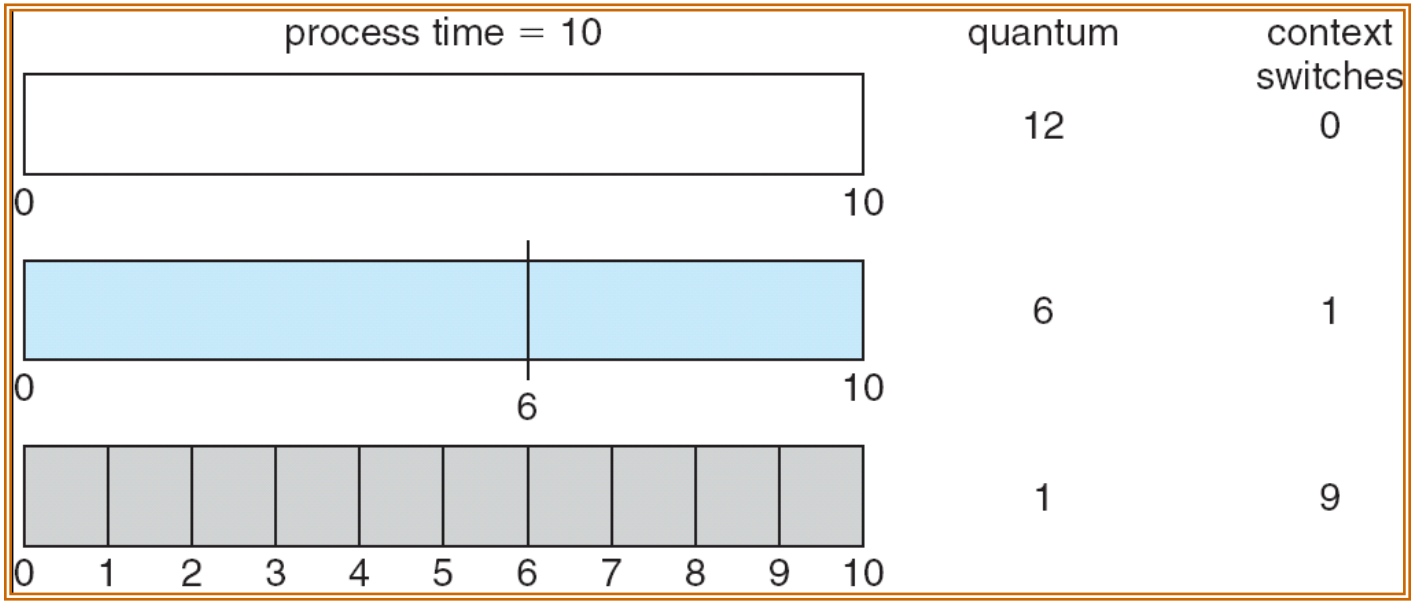
- The Gantt chart is:



✓ Average waiting time ?

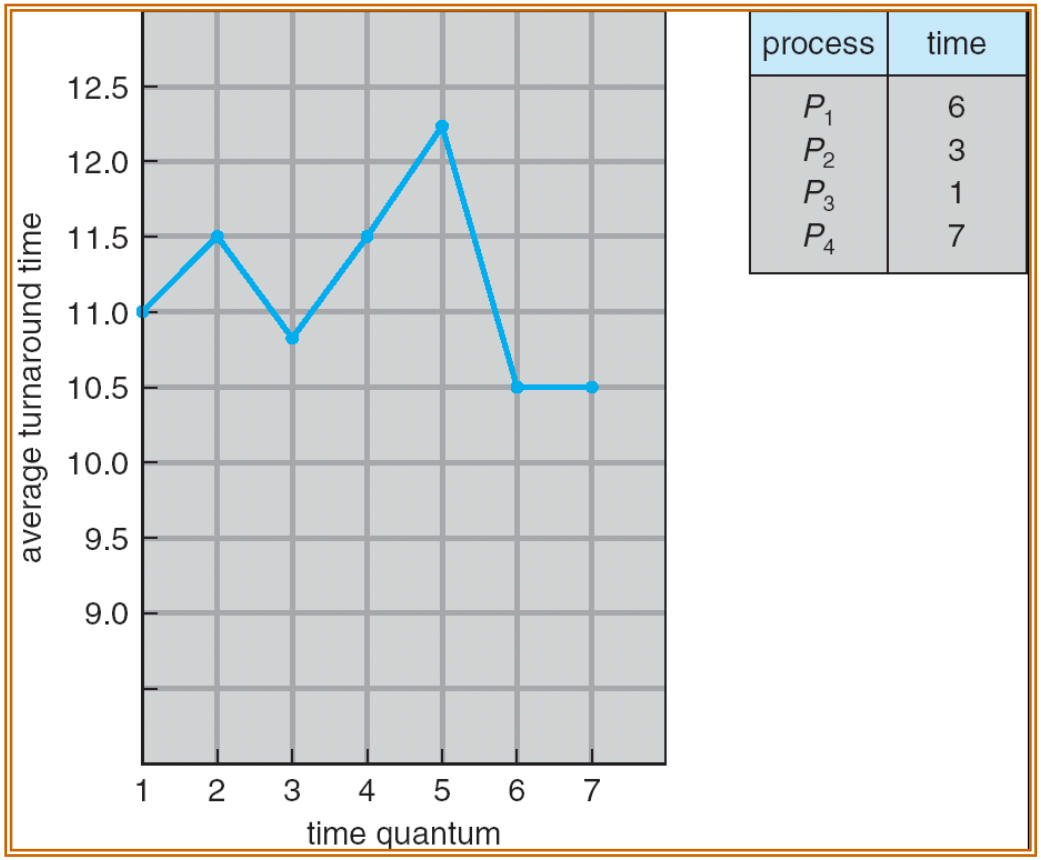
✓ Average turnaround time ?

Time Quantum and Context Switch Time



- We want the time quantum to be large with respect to the context-switch time, otherwise overhead is too high.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- The average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - ✓ Equal-priority processes are scheduled in FCFS order.
  - ✓ An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- Priorities can be defined either internally or externally
  - ✓ Internally defined priorities – use some measurable quantity or quantities to compute the priority of a process
    - ❖ Time limits, memory requirements, the number of open files, the ratio of average I/O burst to average CPU burst
  - ✓ External priorities – set by criteria outside the operating system
    - ❖ Importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, other political factors

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

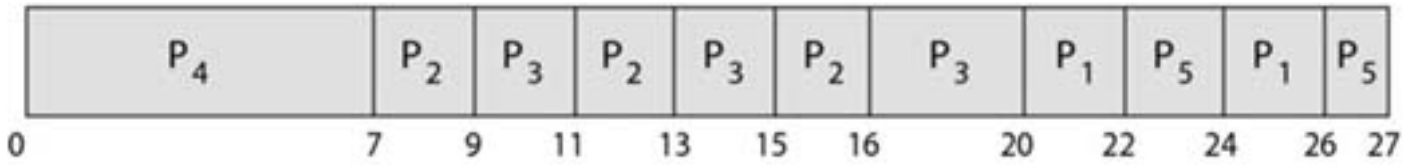
✓ Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 41/5 = 8.2

Priority Scheduling

- Preemptive or nonpreemptive
  - ✓ A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem – Starvation (indefinite blocking)
  - ✓ Can leave some low-priority processes waiting indefinitely
  - ✓ In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- Solution – Aging
  - ✓ A technique of gradually increasing the priority of the processes that wait in the system for a long time

Priority Scheduling

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3



- ✓ Using priority scheduling with round-robin for processes with equal priority
- ✓ Average waiting time = [(20+2) + (7+2+2) + (9+2+1) + 0 + (22+2)]/5
  = (22+11+12+0+24)/5 = 69/5 = 13.8

Multilevel Queue Scheduling

- Depending on how the queues are managed, an $O(n)$ search may be necessary to determine the highest-priority process.
  - ✓ Is often easier to have separate queues for each distinct priority

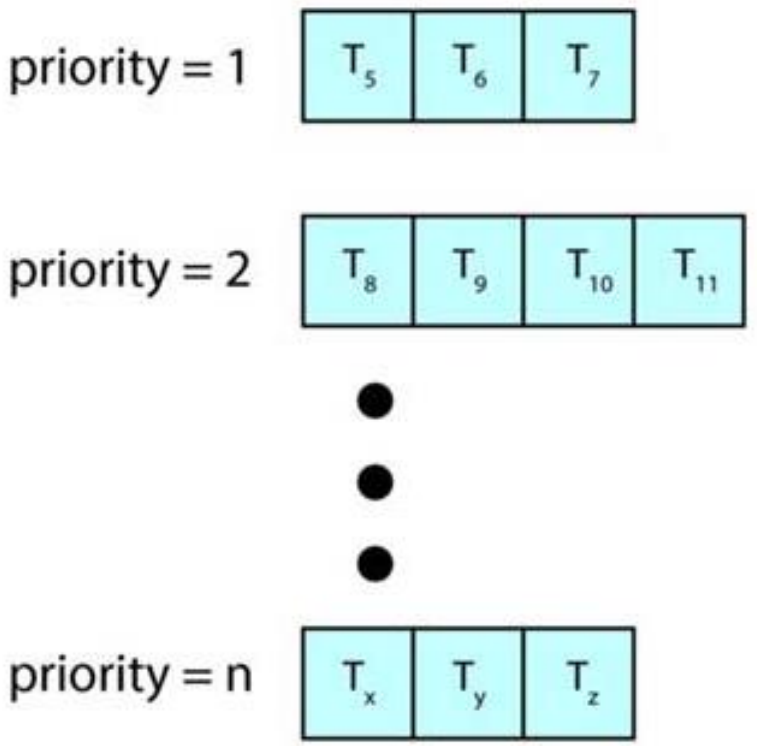


Figure 5.7 Separate queues for each priority.

Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type.
  - ✓ foreground (interactive), background (batch)
- Each queue has its own scheduling algorithm
  - ✓ foreground – RR
  - ✓ background – FCFS
- There must be scheduling among the queues.
  - ✓ Each queue has absolute priority over lower-priority queues.
    - ❖ Serve all from foreground then from background
    - ❖ Possibility of starvation.
  - ✓ Time slice – each queue gets a certain portion of the CPU time, which it can schedule amongst its various processes
    - ❖ Foreground queue – 80% of the CPU time for RR
    - ❖ Background queue – 20% of the CPU time for FCFS

Multilevel Queue Scheduling

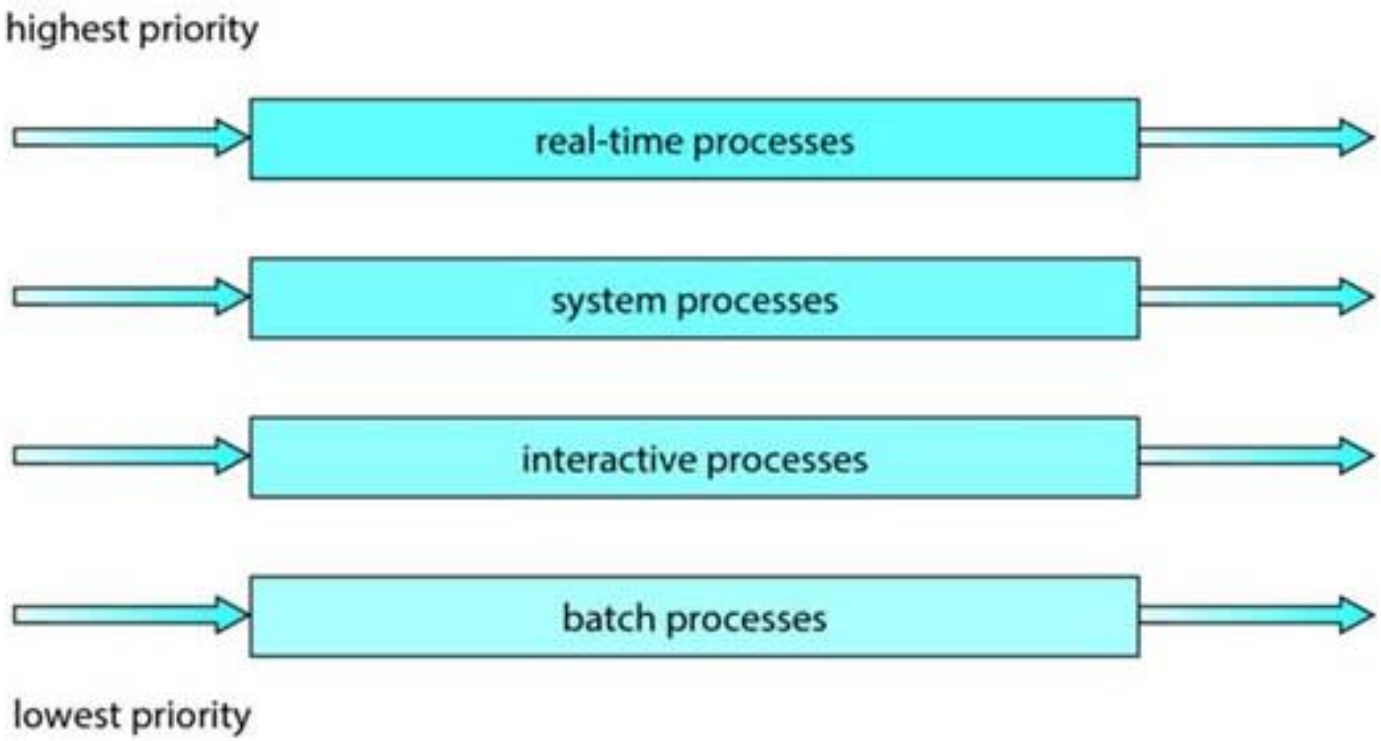


Figure 5.8 Multilevel queue scheduling.

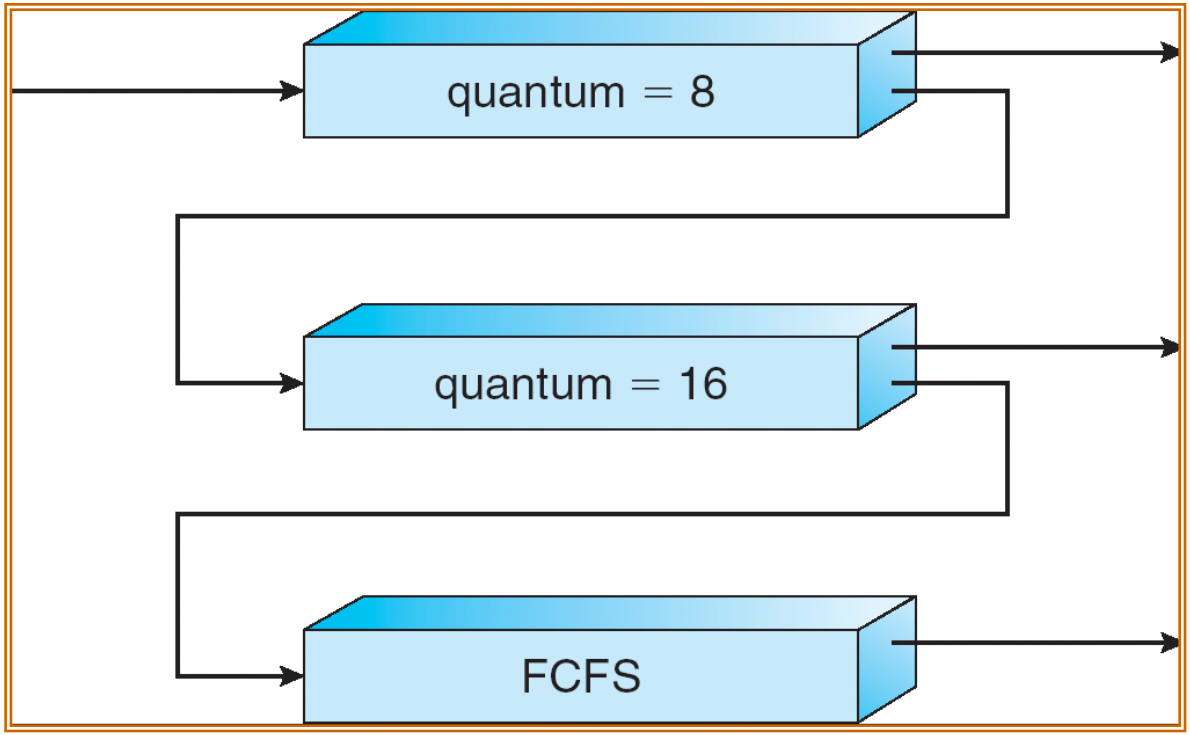
Multilevel Feedback Queue Scheduling

- When the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
- Allows a process to move between queues
- This form of aging prevents starvation.
  - ✓ If a process uses too much CPU time, it will be moved to a lower-priority queue.
  - ✓ A process that waits too long in a lower-priority queue may be moved to a higher-priority queue

Example of Multilevel Feedback Queue

- Three queues:
  - ✓ Q_0 – RR with time quantum 8 milliseconds
  - ✓ Q_1 – RR with time quantum 16 milliseconds
  - ✓ Q_2 – FCFS
- Scheduling
  - ✓ A new job enters queue Q_0 which is served in 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 in FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queue Scheduling



Multilevel Feedback Queue Scheduling

- Multilevel-feedback-queue scheduler is defined by the following parameters:
  - ✓ The number of queues
  - ✓ The scheduling algorithm for each queue
  - ✓ The method used to determine when to upgrade a process
  - ✓ The method used to determine when to demote a process
  - ✓ The method used to determine which queue a process will enter when that process needs service

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor 동질적인 코어들의 collection이다.
- Asymmetric multiprocessing Master - Slave 노드 구조로 구성
  - ✓ Handled by a single processor – master server
  - ✓ Only one processor accesses the system data structures, reducing the need for data sharing
- Symmetric multiprocessing 프로세서마다 각자 알아서 구성
  - ✓ Each processor is self-scheduling
  - ✓ Common ready queue or its own private queue

코어마다 레디 큐를 따로 갖고 있다

Multiple-Processor Scheduling

- Common ready queue
  - ✓ possible race condition on the shared ready queue
  - ✓ use locking to protect the common ready queue from this race condition
- Its own private queue
  - ✓ workloads of varying sizes
  - ✓ balancing algorithms to equalize workloads among all processors

Private Ready Queue
스케줄링 정책에 따라 다르기 때문에
환경의 변화에 따라 기법이 어떻게 적용되는지 알고 있으면 됨

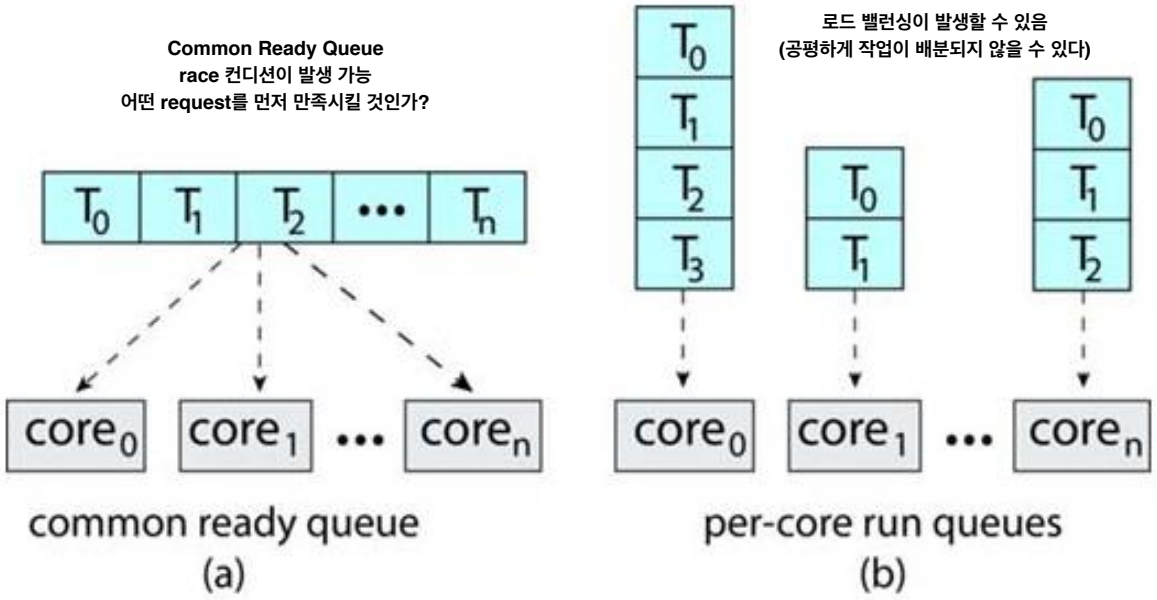


Figure 5.11 Organization of ready queues.

Multiple-Processor Scheduling

- Multicore processor – placing multiple computing cores on the same physical chip
  - ✓ Is faster and consume less power than systems in which each CPU has its own physical chip
- Memory stall – When a processor accesses memory, it spends a significant amount of time waiting for the data to become available.
  - ✓ The processor can spend up to 50 percent of its time waiting for data to become available from memory

코어를 바쁘게 만드는 것이 중요한데, 효율적이지 못한 상황을 지적하는 것이 memory stall

메모리 접근 전까지 대기하는 시간은 어쩔 수 없이 발생하지만,
이를 좀 더 활용할 수 있는 방법이 없겠느냐?
이 방법을 고민한 게 multi-thread process

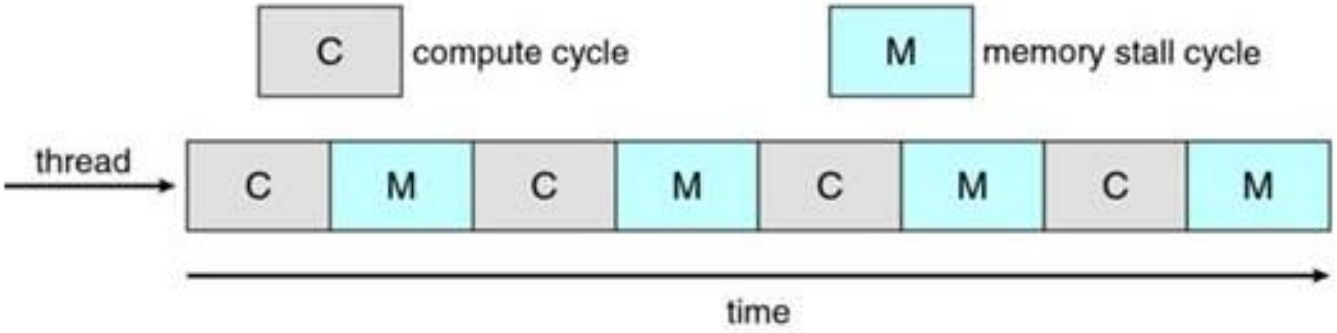


Figure 5.12 Memory stall.

Multiple-Processor Scheduling

- Multithreaded processing cores
  - ✓ Two or more hardware threads are assigned to each core
  - ✓ Figure 5.13
    - ❖ Dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved

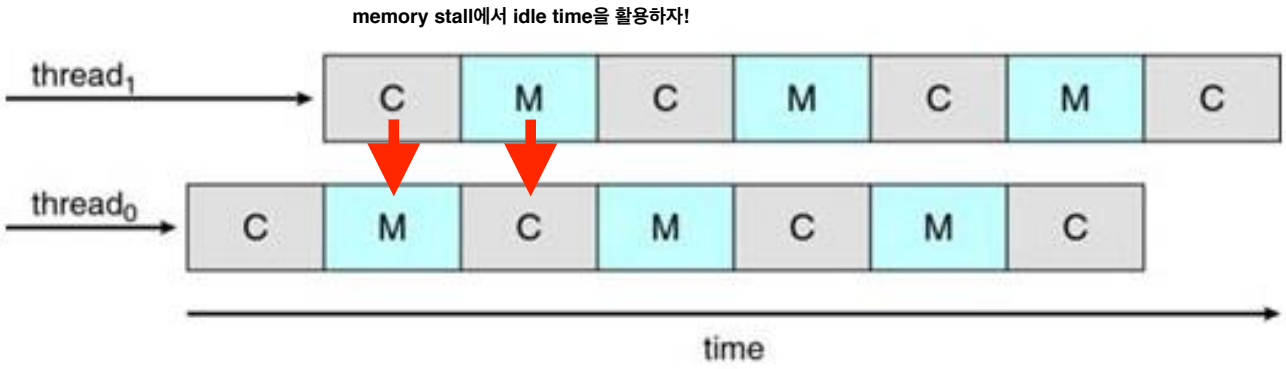


Figure 5.13 Multithreaded multicore system.

Multiple-Processor Scheduling

- Multithreaded processing cores
  - ✓ Figure 5.14
    - ❖ Each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread.
    - ❖ Chip multithreading (CMT)
    - ❖ Four computing cores, with each core containing two hardware threads – eight logical CPUs

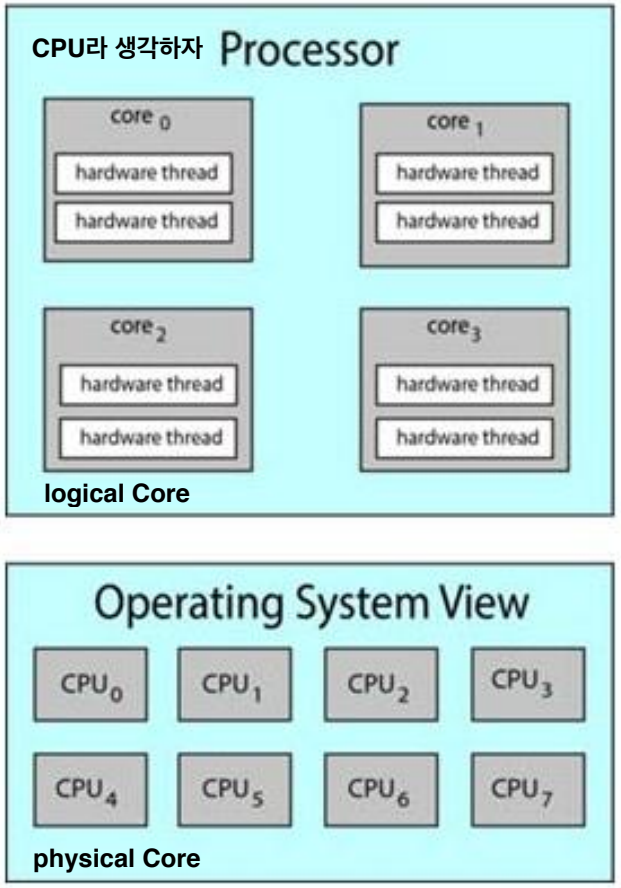


Figure 5.14 Chip multithreading.

Multiple-Processor Scheduling

- Intel processors use the term hyper-threading (also known as simultaneous multithreading or SMT)
- Two ways to multithread : 여기서 알고리즘을 소개하는 것은 아니다
  - ✓ Coarse-grained multithreading : a thread executes on a core until a long-latency event such as a memory stall
  - ✓ Fine-grained (or interleaved) multithreading : switches between threads at a much finer level of granularity – typically at the boundary of an instruction cycle occurs

    괭장히 세세한 레벨까지 관리
- Figure 5.15
  - ✓ the resources of the physical core (such as caches and pipelines) must be shared among its hardware threads
  - ✓ a processing core can only execute one hardware thread at a time

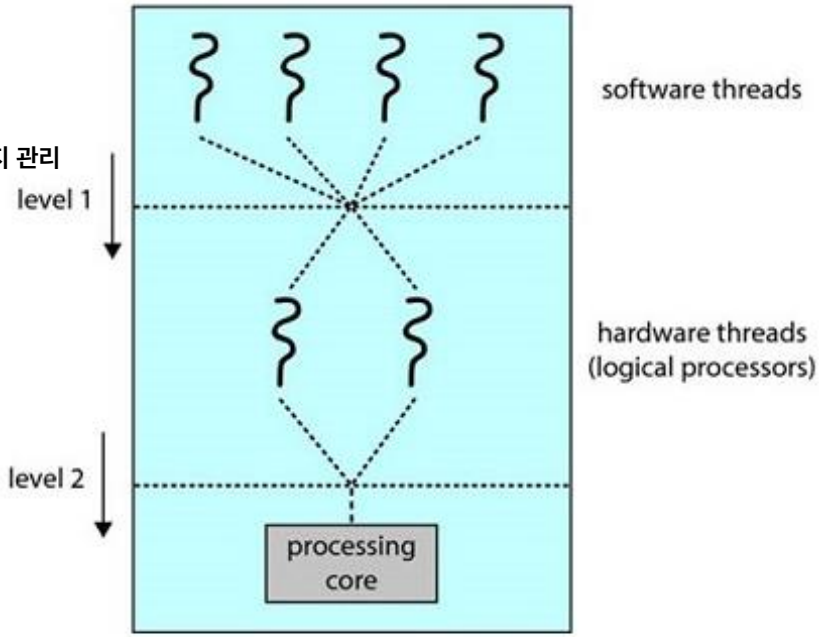
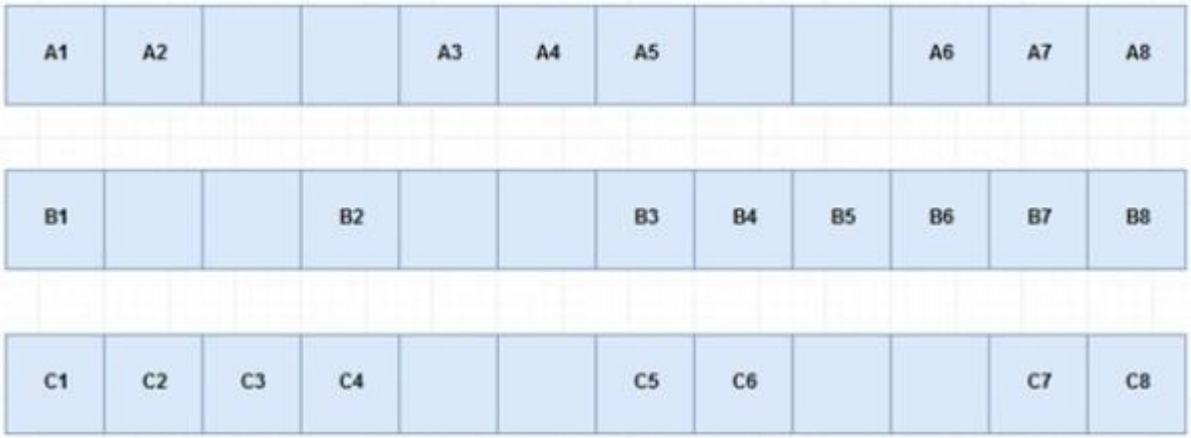


Figure 5.15 Two levels of scheduling.

Multiple-Processor Scheduling

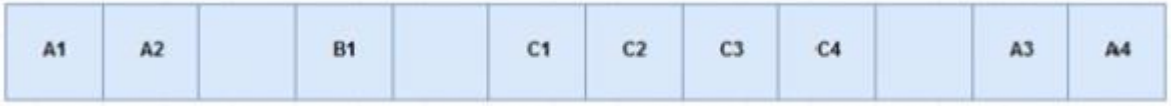
- A, B, C are three threads. 12 cycles of those threads are as follows.



공백 부분이 memory stall

- Fine-grained multithreading 이론적인 측면에서 표현된 감이 있다

- Coarse-grained multithreading



✓ Wasting a clock cycle due to stalling

Multiple-Processor Scheduling

- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system
  - ✓ Each processor has its own private ready queue.
- Two approaches to load balancing
  - ✓ Push migration 많은 쪽에서 적은 쪽으로 옮겨주는 것
    - ❖ A specific task periodically checks the load on each processor
    - ❖ Distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors
  - ✓ Pull migration
    - ❖ When an idle processor pulls a waiting task from a busy processor

Multiple-Processor Scheduling

- What happens if the thread migrates to another processor
  - ✓ The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated
  - ✓ High cost of invalidating and repopulating caches 발생하는 일들이 여러가지로 복잡할 때, High Cost라는 표현을 쓴다.
- Processor affinity 친밀도
  - ✓ Attempt to keep a thread running on the same processor
  - ✓ Soft affinity 좀 더 유연하게 적용됨
    - ❖ When an operating system has a policy of attempting to keep a process running on the same processor – but not guaranteeing that it will do so
  - ✓ Hard affinity 옮기지 못하게 좀 더 strict함
    - ❖ Allowing a process to specify a subset of processors on which it can run
  - ✓ Load balancing counteracts the benefits of processor affinity.

로드 밸런싱을 한다는 것이 옮기는 것인데,
이게 Processor Affinity의 이점을 줄이는 행위를 하는 것임

Multiple-Processor Scheduling

- Figure 5.16 – NUMA and CPU scheduling
  - ✓ An architecture featuring non-uniform memory access (NUMA) where there are two physical processor chips each with their own CPU and local memory
  - ✓ A CPU has faster access to its local memory than to memory local to another CPU

레디큐 작업을 옮기는 것 자체가 CPU간의 작업을 옮기는 것이다

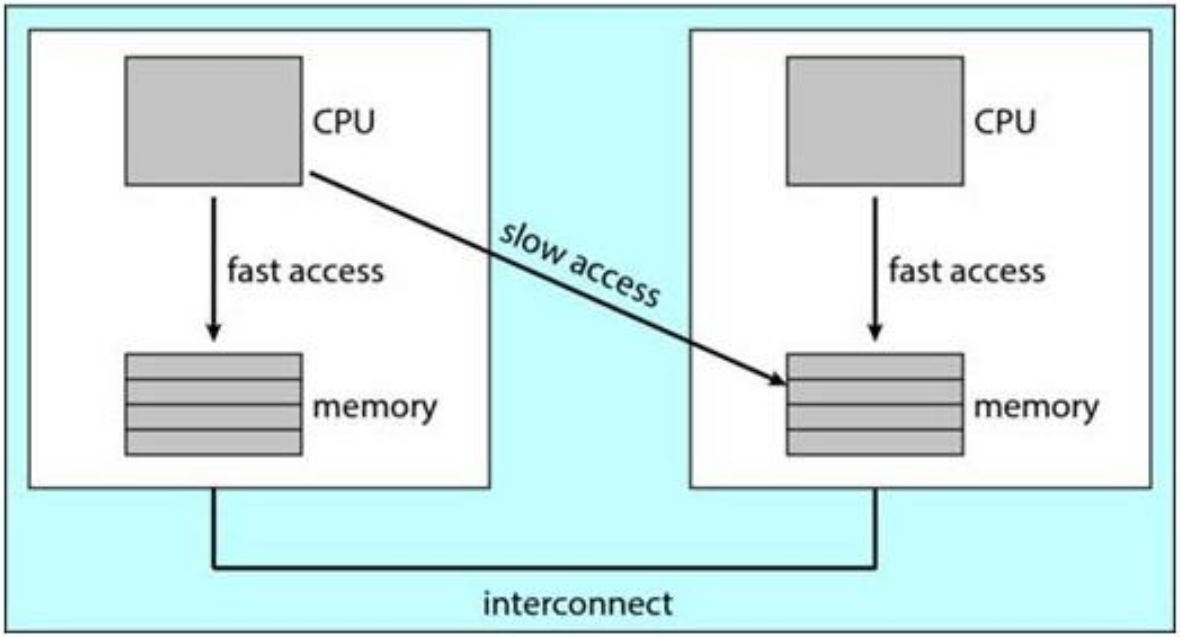


Figure 5.16 NUMA and CPU scheduling.

Real-Time CPU Scheduling

기존의 스케줄링을 작업하는 방식과 다르다

- CPU scheduling for real-time OS involves special issues.
  - ✓ **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled.
  - ✓ **Hard real-time systems** – a task must be serviced by its deadline.

여기서는 deadline을 더 엄격하게 지켜야한다가 hard

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced

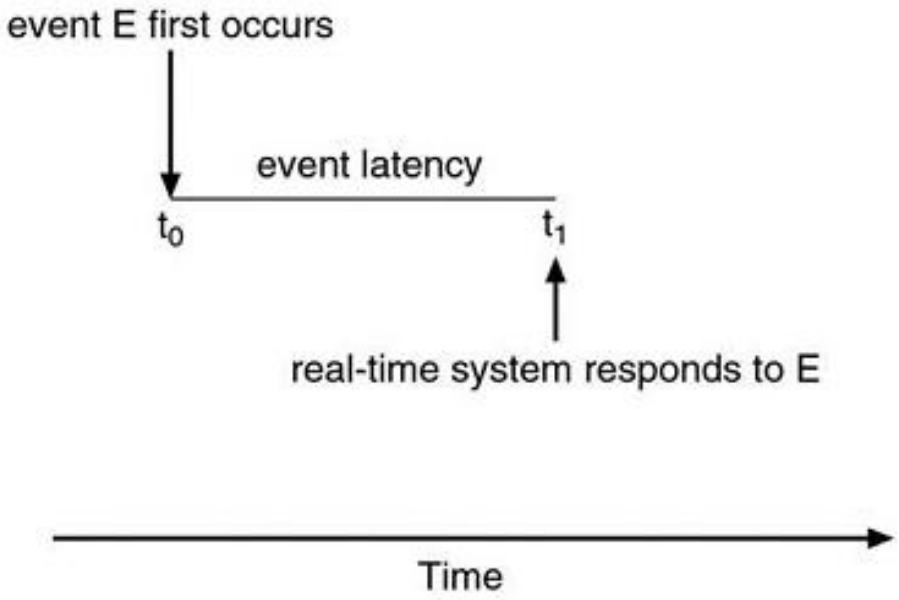


Figure 5.17 Event latency.

Real-Time CPU Scheduling

참고로 알아둘 것

- Two types of latencies affect performance
  - ✓ Interrupt latency – time from arrival of interrupt at the CPU to start of routine that services the interrupt
  - ✓ Dispatch latency – the amount of time required for the scheduling dispatcher to stop one process and start another

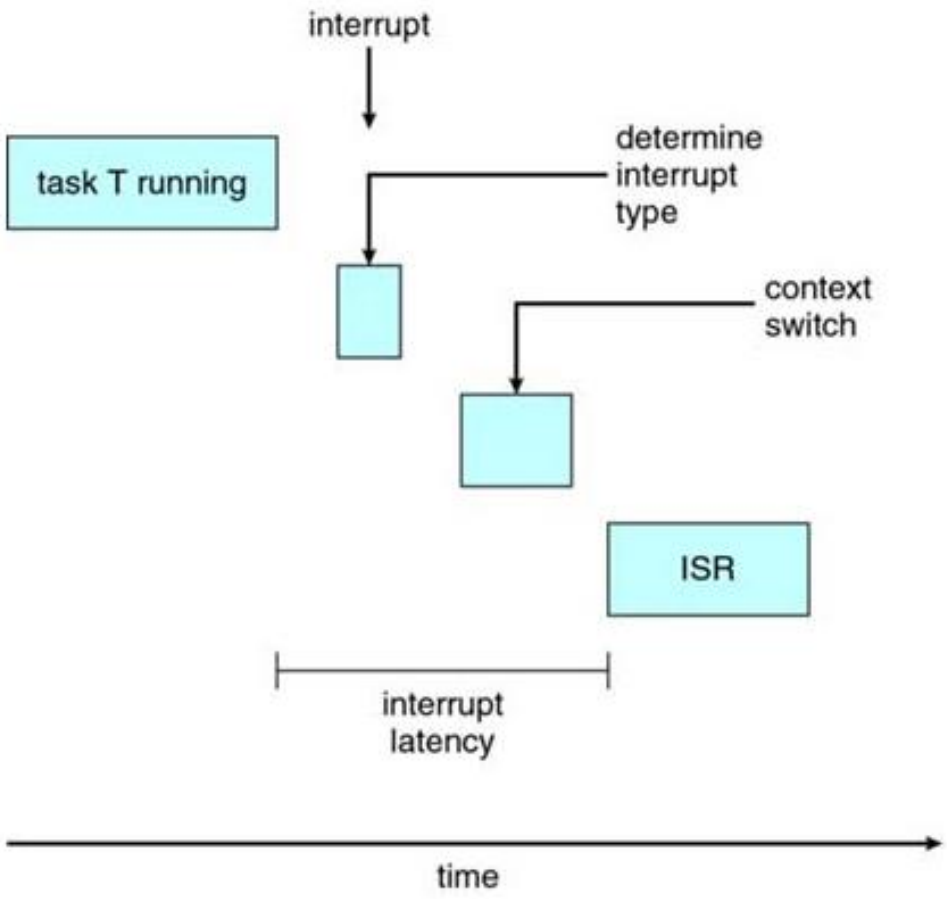


Figure 5.18 Interrupt latency.

Real-Time CPU Scheduling (Cont.)

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

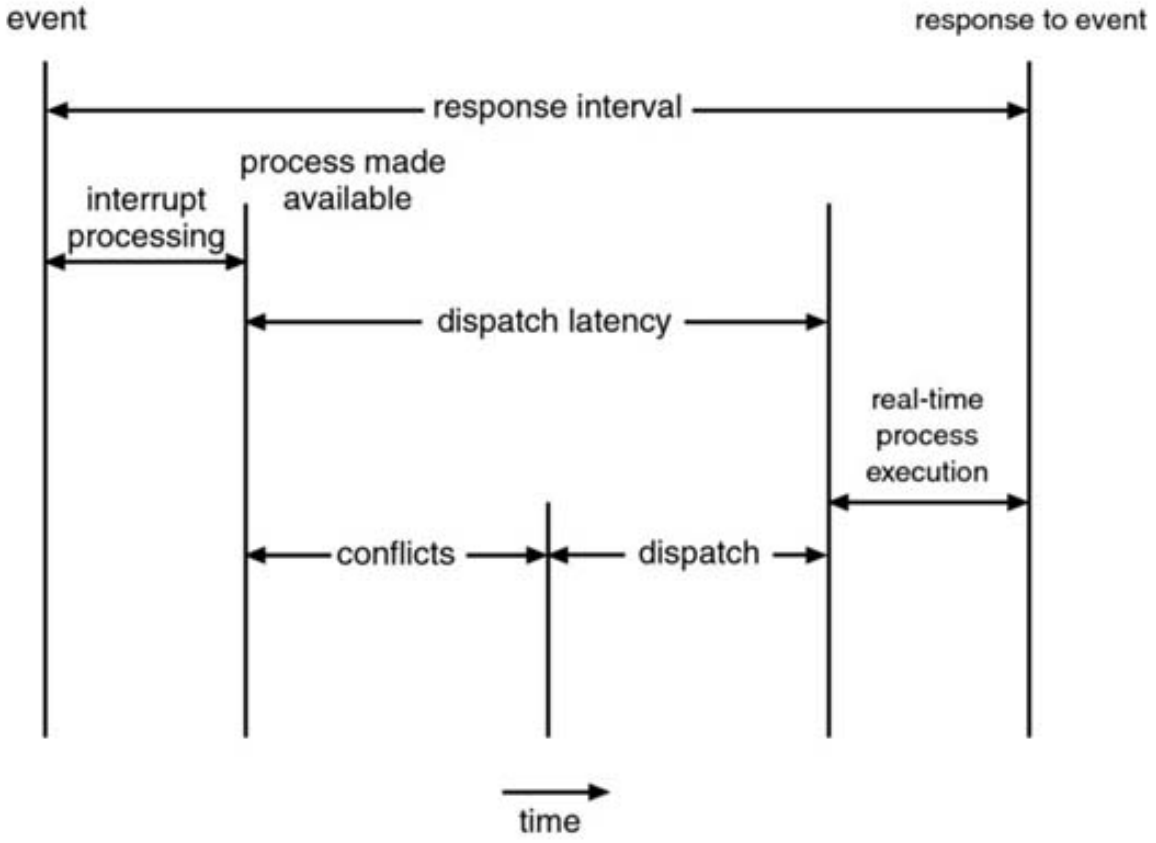


Figure 5.19 Dispatch latency.

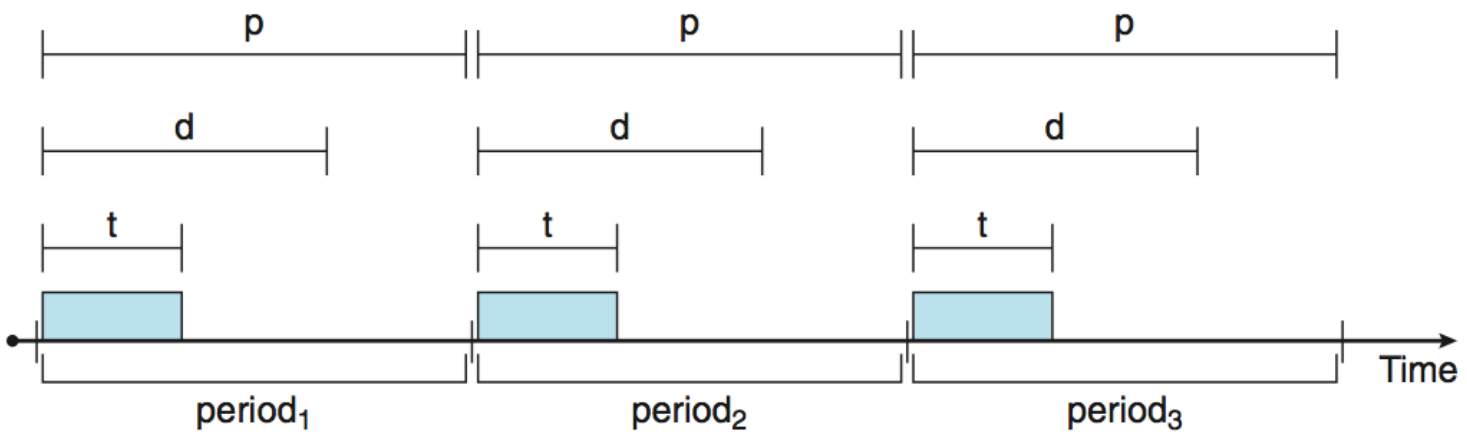
Priority-based Scheduling

- The scheduler for a real-time operating system must support a priority-based algorithm with preemption.
  - ✓ But only guarantees soft real-time functionality
- Hard real-time systems must guarantee that real-time tasks will be serviced in accord with their deadline requirements.
- Define certain characteristics of the processes that are to be scheduled: **periodic** ones require CPU at constant intervals
  - ✓ Once a periodic process has acquired the CPU, it has a fixed processing time t, a deadline d by which it must be serviced by the CPU, and a period p.
    - ❖ $0 \leq t \leq d \leq p$
  - ✓ **Rate** of periodic task is $1/p$

Priority-based Scheduling

- Figure 5.20
  - ✓ The execution of a periodic process over time

p,d,t가 모두 같을 수도 있다.



Rate-Monotonic Scheduling

이 방식에 문제가 있어서 이후에 EDF 방식으로 개선됨

- The 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.
  - ✓ The shorter the period, the higher the priority 급박한 작업은 우선순위를 높게
  - ✓ The longer the period, the lower the priority
- The periods of two processes P_1 and P_2 are 50 and 100, that is p_1=50 and p_2=100. The processing times are t_1=20 for P_1 and t_2=35 for P_2. 대문자 P는 process, 소문자 p는 period
  - ✓ Whether it is possible to schedule these tasks so that each meets its deadlines.
  - ✓ If we measure the CPU utilization of a process P_i as the ratio of its burst to its period – t_i / p_i – the CPU utilization of P_1 is 20/50=0.40 and that of P_2 is 35/100=0.35, for a total CPU utilization of 75%.
  - ✓ Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

Rate-Monotonic Scheduling

- Suppose we assign P_2 a higher priority than P_1.
  - ✓ P_2 starts execution first and completes at time 35. At this point, P_1 starts; it completes its CPU burst at time 55.
  - ✓ However, the first deadline for P_1 was at time 50, so the scheduler has caused P_1 to miss its deadline.

p1 : t = 20 / p = 50
p2 : t = 35 / p = 100
그런데 P2가 우선순위가 높다고 가정하고 진행

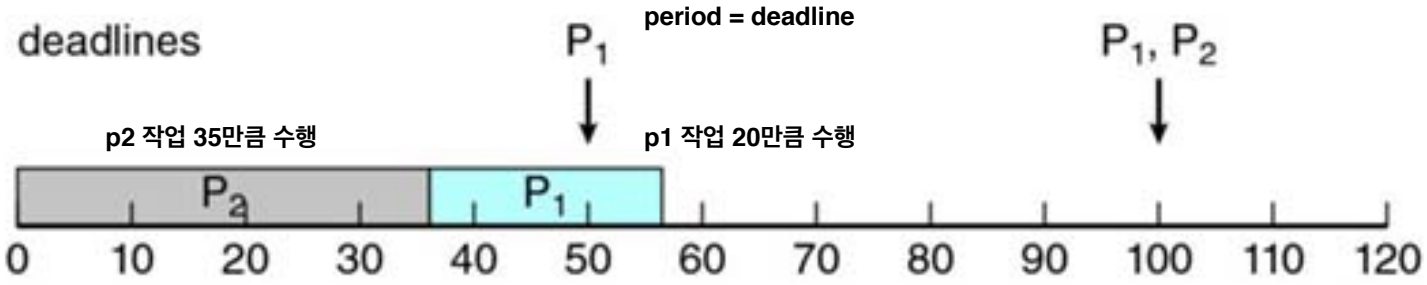


Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1.

위의 예제 상황
p2가 먼저 실행되면서 p1의 마감기한 (p = 50)까지 작업이 마무리 안되면서 스케줄링이 잘못 설정되어 있음

Rate-Monotonic Scheduling

- Suppose we use rate-monotonic scheduling, in which we assign P_1 a higher priority than P_2 because the period of P_1 is shorter than that of P_2.
  - ✓ P_1 starts first and completes its CPU burst at time 20.
  - ✓ At time 50, P_2 is preempted by P_1, although it still has 5ms remaining in its CPU burst. P_2 completes its CPU burst at time 75

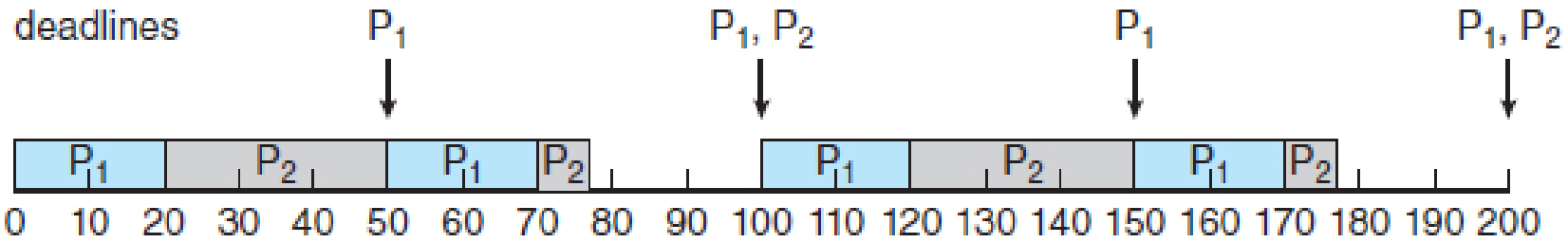


Figure 5.22 Rate-monotonic scheduling.

Missing Deadlines with Rate-Monotonic Scheduling

- Assume that process P_1 has a period of p_1=50 and a CPU burst of t_1=25. For P_2, the corresponding values are p_2=80 and t_2=35.
  - ✓ Assign process P_1 a higher priority
  - ✓ P_1 runs until it completes its CPU burst at time 25. P_2 begins running and runs until time 50, when it is preempted by P_1.
  - ✓ P_2 finishes its burst at time 85, after the deadline for completion of its CPU burst at time 80.

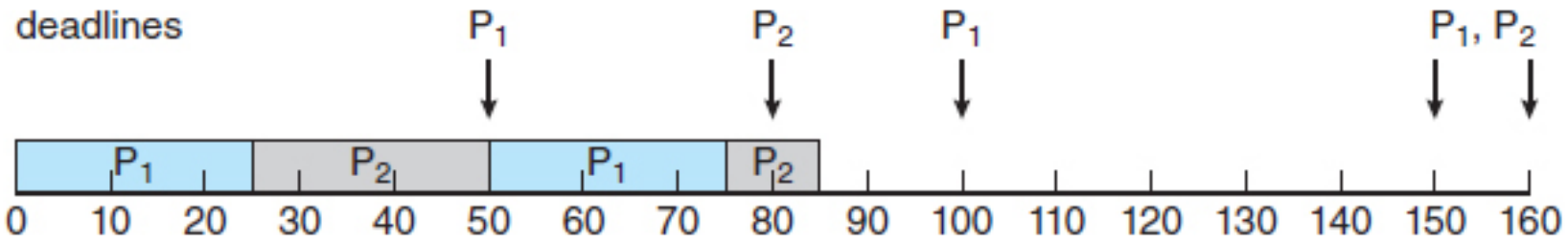


Figure 5.23 Missing deadlines with rate-monotonic scheduling.

Rate-Monotonic Scheduling

- Rate-Monotonic scheduling has a limitation.
  - ✓ CPU utilization is bounded, and it is not always possible to maximize CPU resources fully.
  - ✓ The worst-case CPU utilization for scheduling N processes
    - ❖ $N(2^{1/N} - 1)$
    - ❖ With one process in the system, CPU utilization is 100 percent, but it falls to approximately 69 percent as the number of processes approaches infinity.
    - ❖ With two processes, CPU utilization is bounded at about 83 percent.

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

- P_1 has values of p_1=50 and t_1=25 and P_2 has values of p_2=80 and t_2=35.
  - ✓ Process P_1 has the earliest deadline, so its initial priority is higher than that of process P_2.
  - ✓ Whereas rate-monotonic scheduling allows P_1 to preempt P_2 at the beginning of its next period at time 50, EDF scheduling allows process P_2 to continue running.

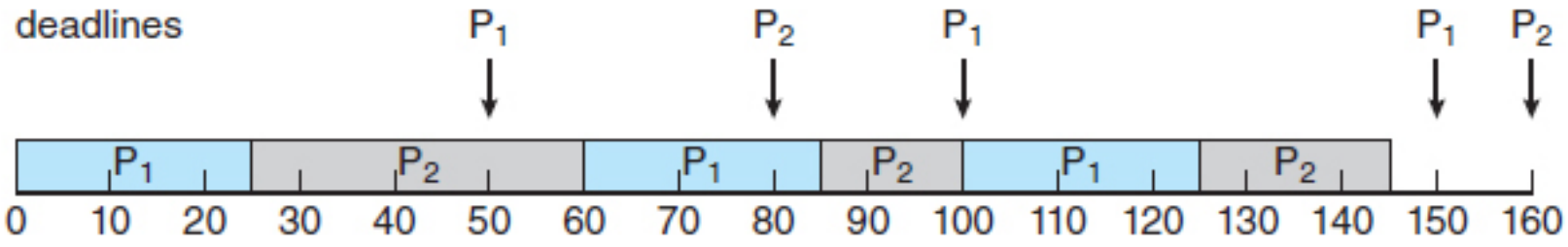


Figure 5.24 Earliest-deadline-first scheduling.

Operating System Examples – Linux

- Linux scheduling
  - ✓ In release 2.6.23 of the kernel, **Completely Fair Scheduler** (**CFS**) became the default Linux scheduling algorithm.
  - ✓ To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class.
  - ✓ Standard Linux kernels implement two scheduling classes: (1) a default scheduling class using the CFS scheduling algorithm and (2) a real-time scheduling class.
  - ✓ Rather than using strict rules that associate a relative priority value with the length of a time quantum, the CFS scheduler assigns a proportion of CPU processing time to each task.
  - ✓ This proportion is calculated based on the **nice value** assigned to each task. Nice values range from -20 to +19, where a numerically lower nice value indicates a higher relative priority.
  - ✓ CFS doesn't use discrete values of time slices and instead identifies a **targeted latency**, which is an interval of time during which every runnable task should run at least once.

Operating System Examples – Linux

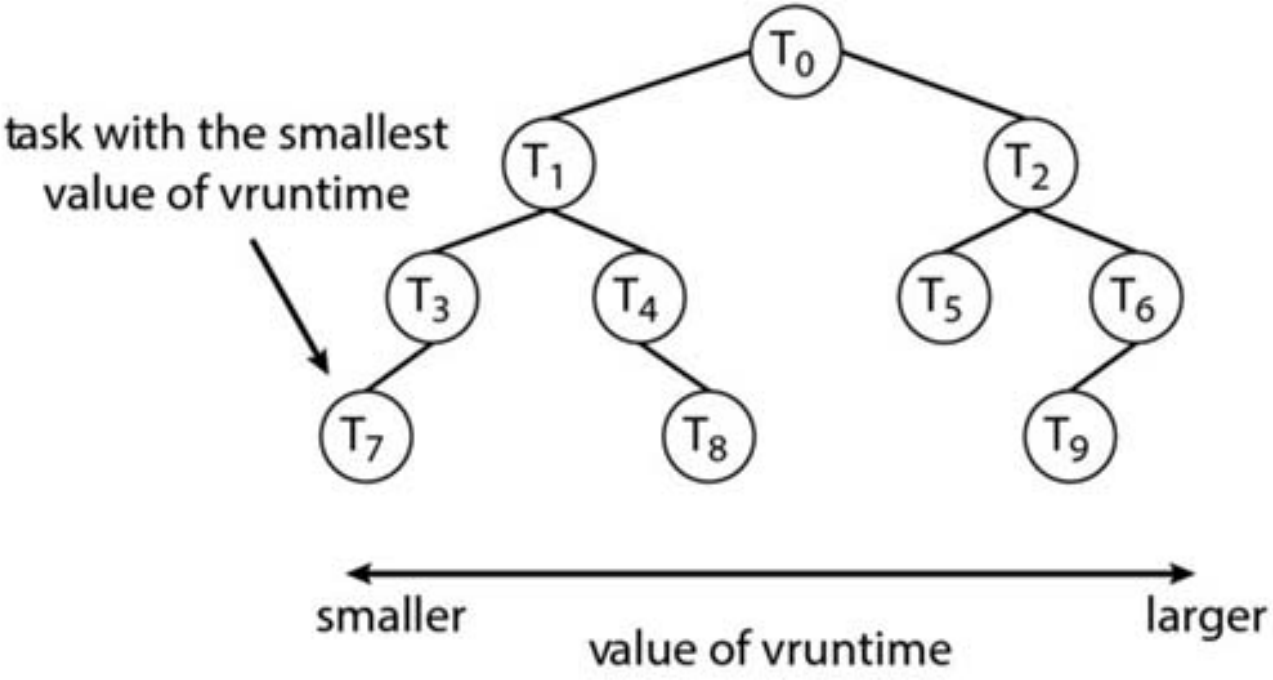
- Linux scheduling
  - ✓ The CFS scheduler doesn't directly assign priorities. Rather, it records how long each task has run by maintaining the **virtual run time** of each task using the per-task variable `vruntime`.
  - ✓ The virtual run time is associated with a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks.
  - ✓ For tasks at normal priority (nice values of 0), virtual run time is identical to actual physical run time.
    - ❖ If a task with default priority runs for 200ms, its `vruntime` will be 200ms. If a lower-priority task runs for 200ms, its `vruntime` will be higher than 200ms. Similarly, if a higher-priority task runs for 200ms, its `vruntime` will be less than 200ms.
  - ✓ To decide which task to run next, the scheduler simply selects the task that has the smallest `vruntime` value. In addition, a higher-priority task that becomes available to run can preempt a lower-priority task.

Operating System Examples – Linux

- CFS Performance
  - ✓ The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Rather than using a standard queue data structure, each runnable task is placed in a **red-black tree** – a balanced binary search tree whose key is based on the value of `vruntime`.
  - ✓ When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable ([Ex] if it is blocked while waiting for I/O), it is removed.
  - ✓ 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(log N) operations.

Operating System Examples – Linux

- CFS Performance
  - ✓ 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.



Operating System Examples – Linux

- Linux scheduling
  - ✓ Linux uses two separate priority ranges: Real-time tasks are assigned static priorities within the range of 0 to 99, and normal tasks are assigned priorities from 100 to 139.
    - ❖ Lower values indicate higher priorities.

Figure 5.26 Scheduling priorities on a Linux system.

Operating System Examples – Windows

- Windows scheduling
  - ✓ A priority-based, preemptive scheduling algorithm
    - ❖ the following six priority classes to which a process can belong
    - ❖ The values for relative priorities
  - ✓ By default, the base priority is the value of the NORMAL relative priority for that class.

	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

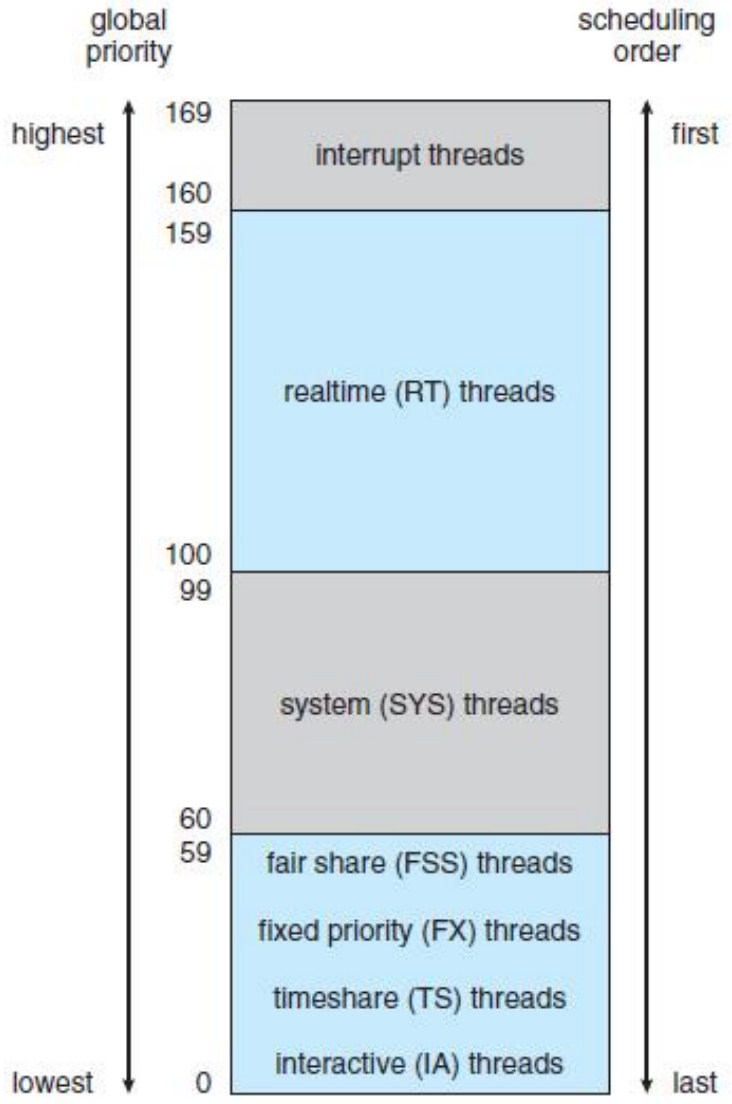
Operating System Examples – Solaris

- The default scheduling class for a process is time sharing
- The inverse relationship between priorities and time quanta:
  - ✓ The lowest priority (0) has the highest time quantum (200ms)
  - ✓ The highest priority (59) has the lowest time quantum (20ms)
- Interactive processes have a higher priority
- CPU-bound processes have a lower priority

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Operating System Examples – Solaris

- Each thread belongs to one of six classes.
- Threads in the real-time class are given the highest priority.
- the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run.
- The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread.



Algorithm Evaluation

- Defining the criteria to be used in selecting an algorithm
  - ✓ Maximizing CPU utilization
  - ✓ Maximizing throughput
- Deterministic modeling
  - ✓ One type of analytic evaluation
  - ✓ takes a particular predetermined workload and defines the performance of each algorithm for that workload
  - ✓ Example

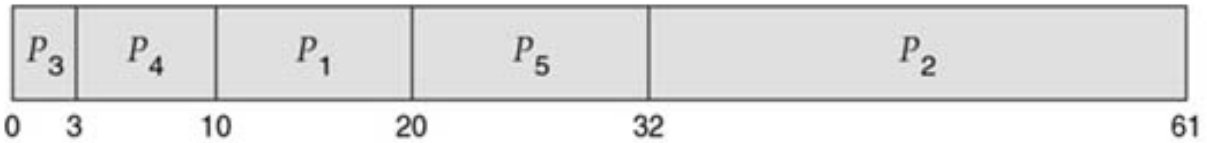
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Algorithm Evaluation

- FCFS algorithm

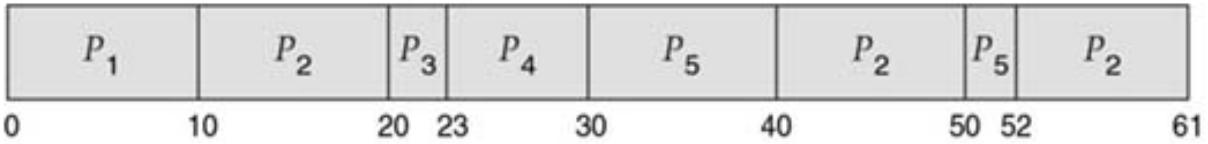
✓ Average waiting time = (0 + 10 + 39 + 42 + 49) / 5 = 28

- SJF algorithm



✓ Average waiting time = (10 + 32+ 0 + 3 + 20) / 5 = 13

- RR algorithm



✓ Average waiting time = (0 + 32 + 20 + 23 + 40) / 5 = 23

Algorithm Evaluation

- Queueing models
  - ✓ There is no static set of processes (or times) to use for deterministic modeling
    - ❖ What can be determined is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or estimated. ⇒ mathematical formula
  - ✓ Queueing-network analysis
    - ❖ Let n be the average queue length, let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue
      - » **$n = \lambda \times W$**
- Simulation
  - ✓ Involves programming a model of the computer system
- Implementation
  - ✓ The only completely accurate way

Evaluation of CPU schedulers by simulation