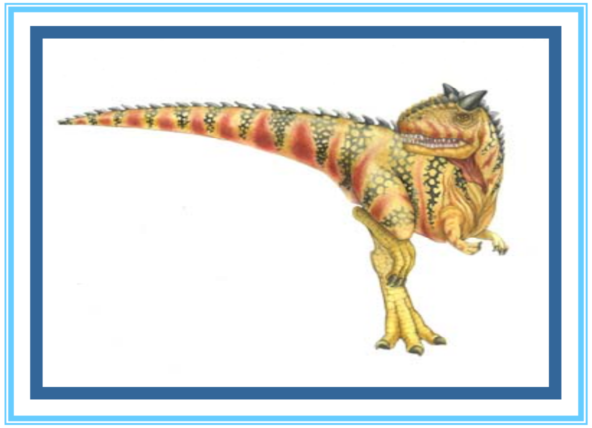
Chapter 5: CPU Scheduling



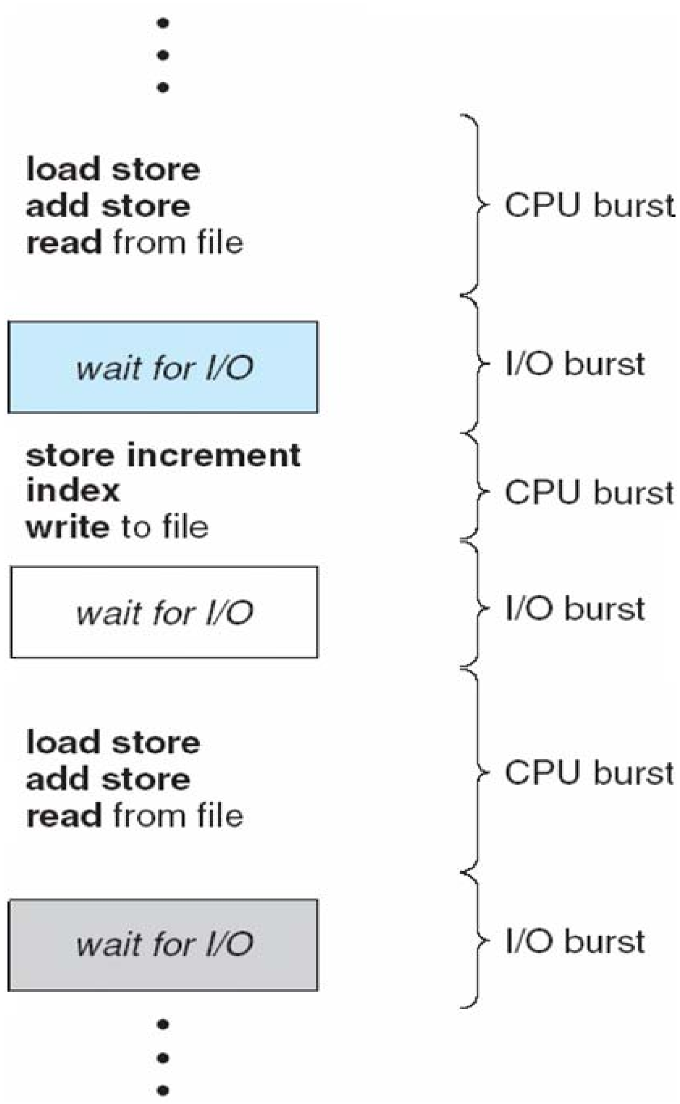
Basic Concepts

장기 job scheduling
단기 CPU scheduling < =Focus
중기 swapping : Swap In, Swap Out

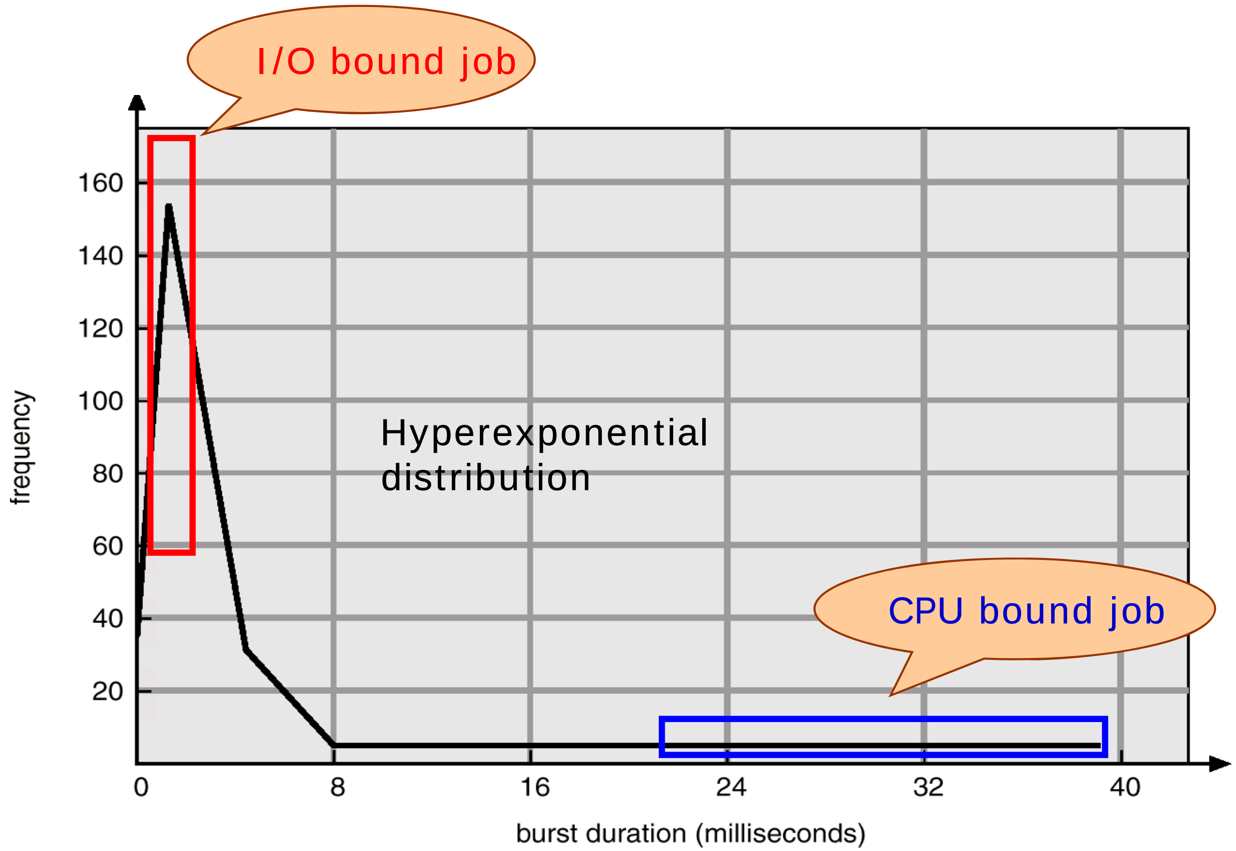
- **CPU-I/O** 버스트 주기**(burst cycle)**
  - **cycle : CPU** 실행**(CPU burst) <--> I/O** 대기**(I/O burst)**
  - **CPU burst** 유형
    - **I/O bound program :** 많은 짧은 **CPU burst** 가짐
    - **CPU bound program :** 적은 아주 긴 **CPU burst** 가짐
- **CPU** 스케줄러
  - 단기 스케줄러**(short-term scheduler) : <u>ready queue</u>**에서 선택

FIFO(First-In First-Out)큐
우선순위 큐
트리
연결리스트

Alternating Sequence of CPU and I/O Bursts



Histogram of CPU-burst Times

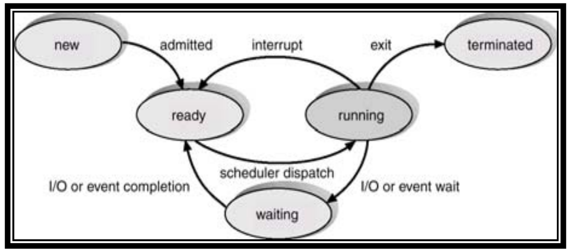


일반적인 시스템에서,
다수의 짧은 CPU burst와 적은 수의 긴 CPU burst로 구성
=> 어떻게 스케쥴링할 것인가?

CPU Scheduler

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

- **CPU scheduling decision time**
  - **running -> waiting (예:I/O request interrupt)**
  - **running -> ready (예: time run out)**
  - **waiting -> ready (예 : I/O 완료 interrupt)**
  - **halt : non preemptive**



- **1**과 **4**에서만 **Scheduling**이 발생할 경우**:** ***nonpreemptive***로 충분
- 모든 경우에서 **Scheduling**이 가능할 경우 **:** ***preemptive***

CPU Scheduler

- 선점(**preemptive**) 스케줄링
  - 특수하드웨어(**timer**)필요
  - 공유 데이타에 대한 프로세스 동기화 필요
- 비선점(**non preemptive**) 스케줄링
  - 특수 하드웨어(**timer**) 없음
  - 종료 또는 **I/O**까지 계속 **CPU**점유

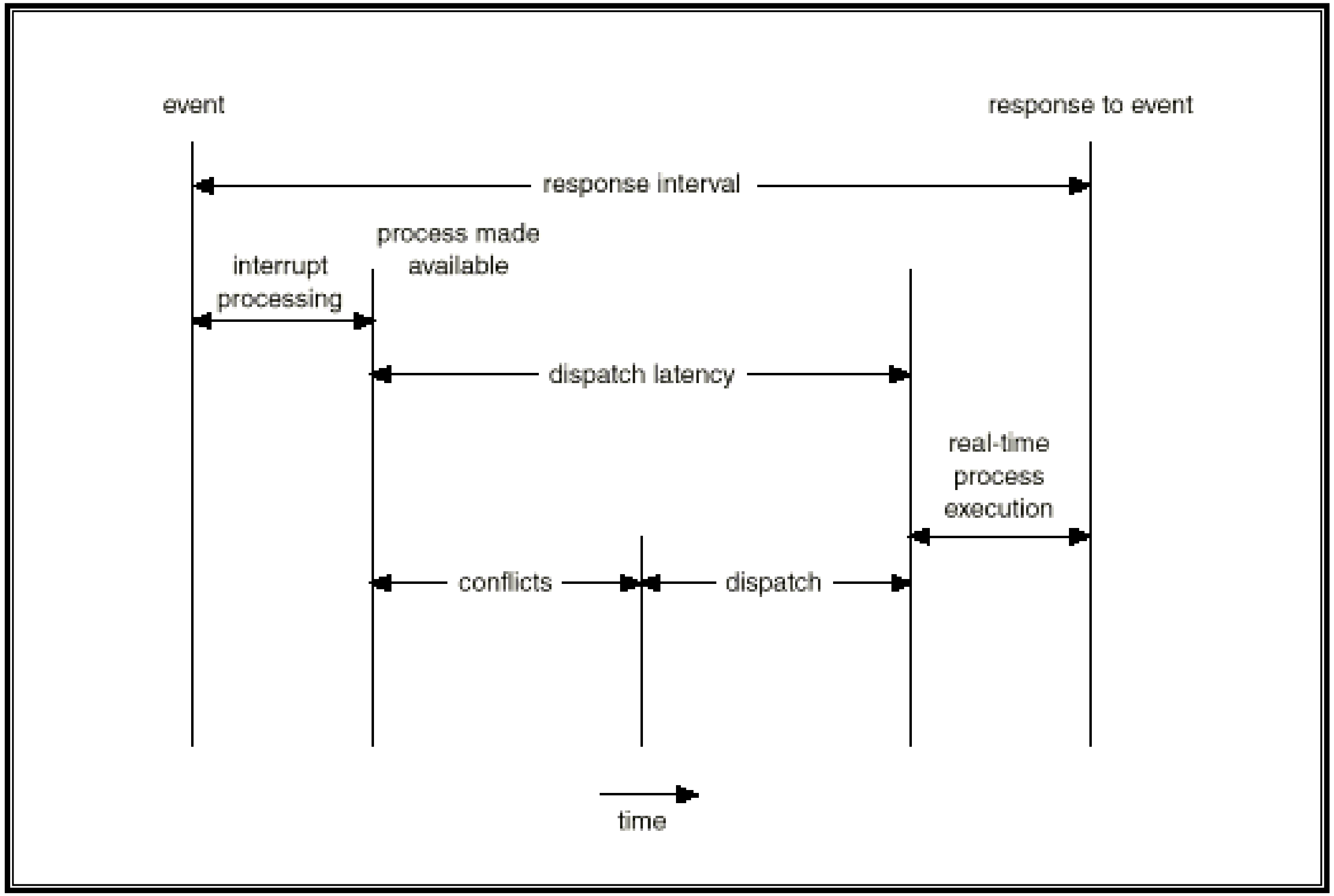
Dispatcher

- **Dispatcher**의 정의
  - **CPU** 스케쥴러에 의해 선택된 프로세스에게 **CPU**에 대한 제어권한을 주는 모듈

- **Dispatcher**의 역할
  - ***switching context***
  - ***switching to user mode***
  - ***jumping to the proper location in the user program***

- ***Dispatch latency***
  - **Dispatcher**가 하나의 프로세스를 정지하고 다른 프로세스의 수행을 시작하는 데까지 소요되는 시간

Dispatch Latency



CPU Scheduling의 성능 기준

- 이용률(**CPU utilization) : 40% ~ 90%**
  - **keep the CPU as busy as possible**
- 처리율(**throughput) :** 단위 시간당 완료된 프로세스 갯수
  - **# of processes that complete their execution per time unit**
- 반환시간(**turnaround time) : system in -> system out** 걸린 시간
  - **amount of time to execute a particular process**
- 대기시간(**waiting time) : ready queue**에서 기다린 시간
  - **amount of time a process has been waiting in the ready queue**
- 응답시간(**response time) :** 대화형 시스템에서 첫 응답까지의 시간
  - **amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)**

Scheduling Algorithms

- FCFS (First-Come First-Served)
- SJF (Shortest-Job-First)
  - SRT (Shortest-Remaining-Time)
- Priority Scheduling
  - HRN(Highest-Response-ratio Next
- RR (Round Robin)
- Multilevel Queue
- Multilevel Feedback Queue

- **CPU Scheduler**

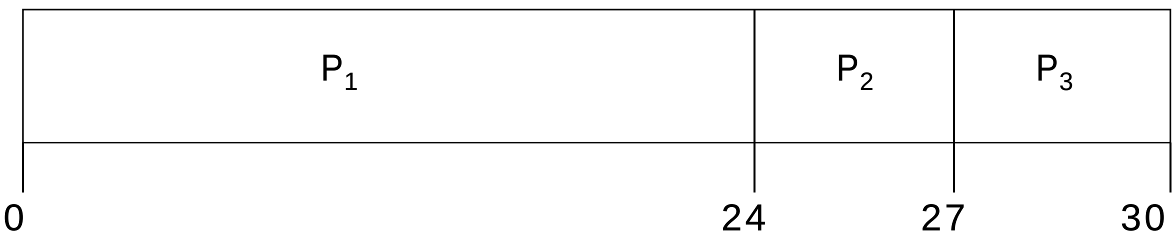
  - http://jimweller.com/jim-weller/jim/java_proc_sched/

First-Come, First-Served (FCFS) Scheduling

선입 선처리(**First-Come, First-Served**) 스케줄링

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:



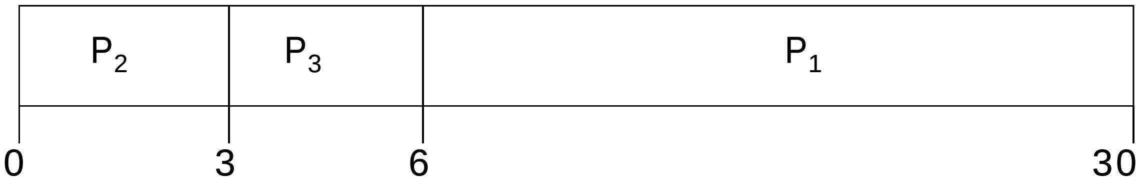
- **Waiting time for P_1 = 0; P_2 = 24; P_3 = 27**
- **Average waiting time: (0 + 24 + 27)/3 = 17**

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 :***
  - **FCFS** 스케쥴링 알고리즘**(I/O Queue**와 **Read Queue**를 가진**)**에 있어서 **CPU-bound** 프로세스**(CPU**를 많이 차지하는**)**와 **I/O bound** 프로세스**(**상대적으로 **CPU**를 적게 사용하는**)**가 있을 때 **CPU-bound** 프로세스로 인해 **I/O bound** 프로세스가 짧은 **CPU**의 할당만으로 **JOB**을 완료할 수 있음에도 불구하고**,** 순서를 기다림으로써 전반적인 시스템 성능이 떨어지는 효과

Shortest-Job-First (SJF) Scheduling

최소 작업 우선(**Shortest-Job-First**) 스케줄링

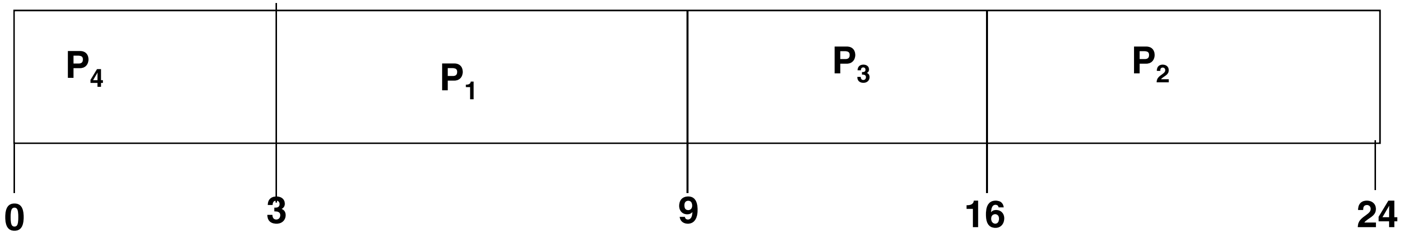
- **SJF Scheduling**의 정의
  - **Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.**

- **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 know as the**
    Shortest-Remaining-Time-First (SRTF).

Example of SJF(Non-preemptive)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- **SJF scheduling chart**

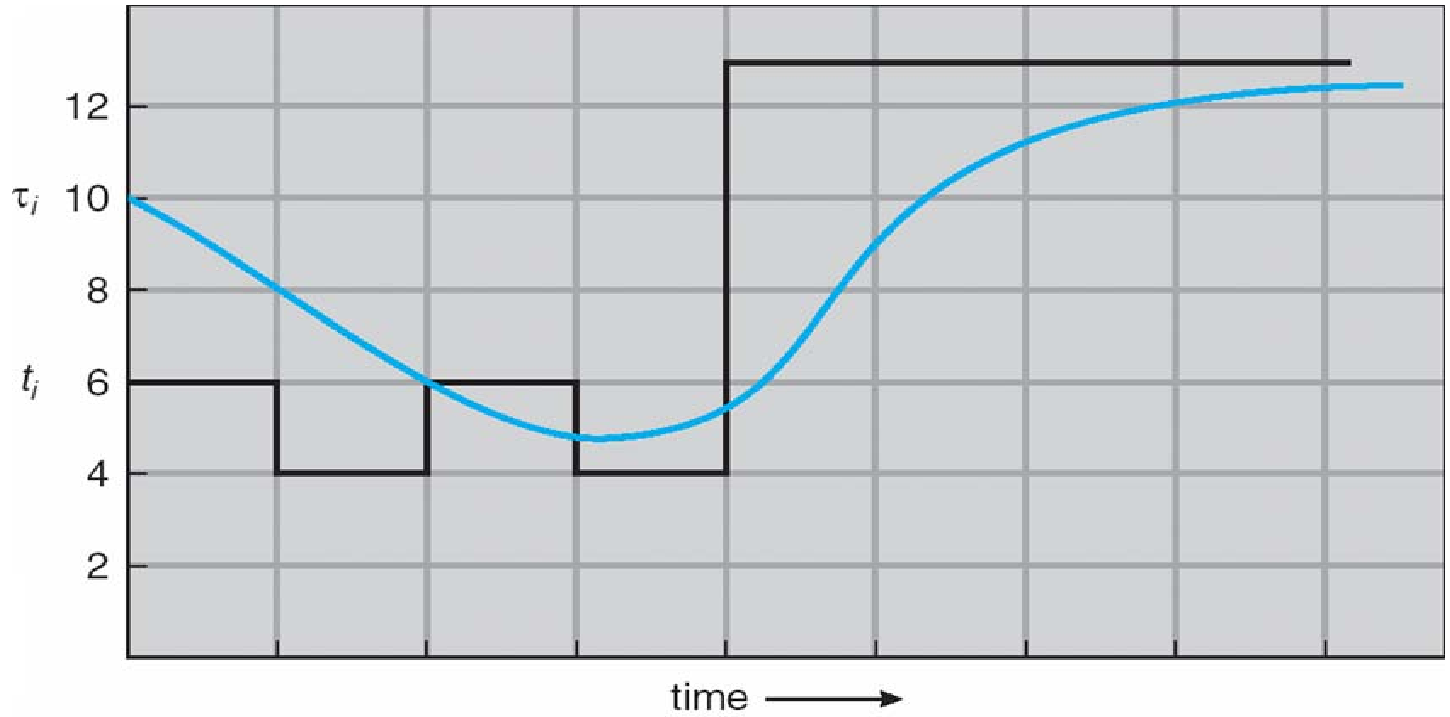


- **Average waiting time = (3 + 16 + 9 + 0) / 4 = 7**

SJF

- **SJF is optimal – gives minimum average waiting time for a given set of processes**
  - **long-term scheduling**에 좋음**(**프로세스 시간의 사용자 예측 치 이용**)**
  - **short-term scheduling** 에는 나쁨 **:** 차기 **CPU burst** 시간 파악이 어려워서
  - 차기 **CPU** 버스트 시간 예측 모델 필요

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	. . .

Determining Length of Next CPU Burst

- **Can only estimate the length – should be similar to the previous one**
  - **Then pick process with shortest predicted next CPU burst**

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

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

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

- **Commonly,** α **set to ½**
- **Preemptive version called shortest-remaining-time-first**

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:

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

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

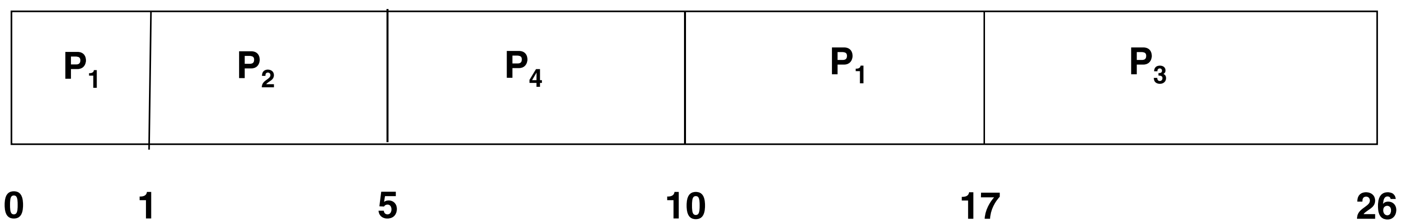
Example of Shortest-remaining-time-first

Preemptive SJF

- 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 msec

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	1.0	4
P_3	2.0	9
P_4	3.0	5

- SJF (preemptive)

  - Average waiting time = ?

SJF(Shortest-Job-First) 스케줄링

- **Job** 의 실행시간이 가장 짧은 작업을 선택
- 장점 **:** 평균 대기시간이 짧다
- 단점 **:**
  - 시분할 구현이 불가능
  - **Starvation** 의 가능성
  - **Job** 의 실행시간 예측이 거의 불가능

Priority Scheduling

NonPreemptive

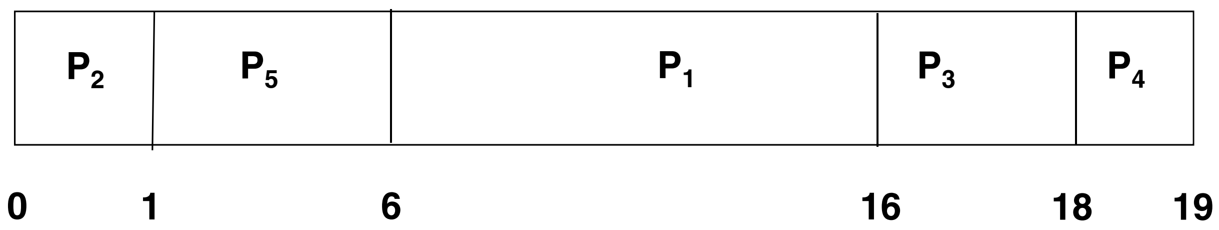
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
  - Problem ≡ Starvation – low priority processes may never execute.
  - Solution ≡ Aging – as time progresses increase the priority of the process.

소문 : 1973년 MIT의 IBM 7094를 폐쇄할때,
1967년의 프로세스가 아직도 수행되지 못한 것을 발견!

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

Round Robin (RR)

- **Time Quantum :**
  - **Each process gets a small unit of CPU time (time quantum q), 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.**
- **Timer interrupts every quantum to schedule next process**
- **Performance**
  - 할당되는 시간이 클 경우 **FIFO** 기법과 같아짐
  - 할당되는 시간이 작은 경우 문맥 교환 및 오버헤드가 자주 발생

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

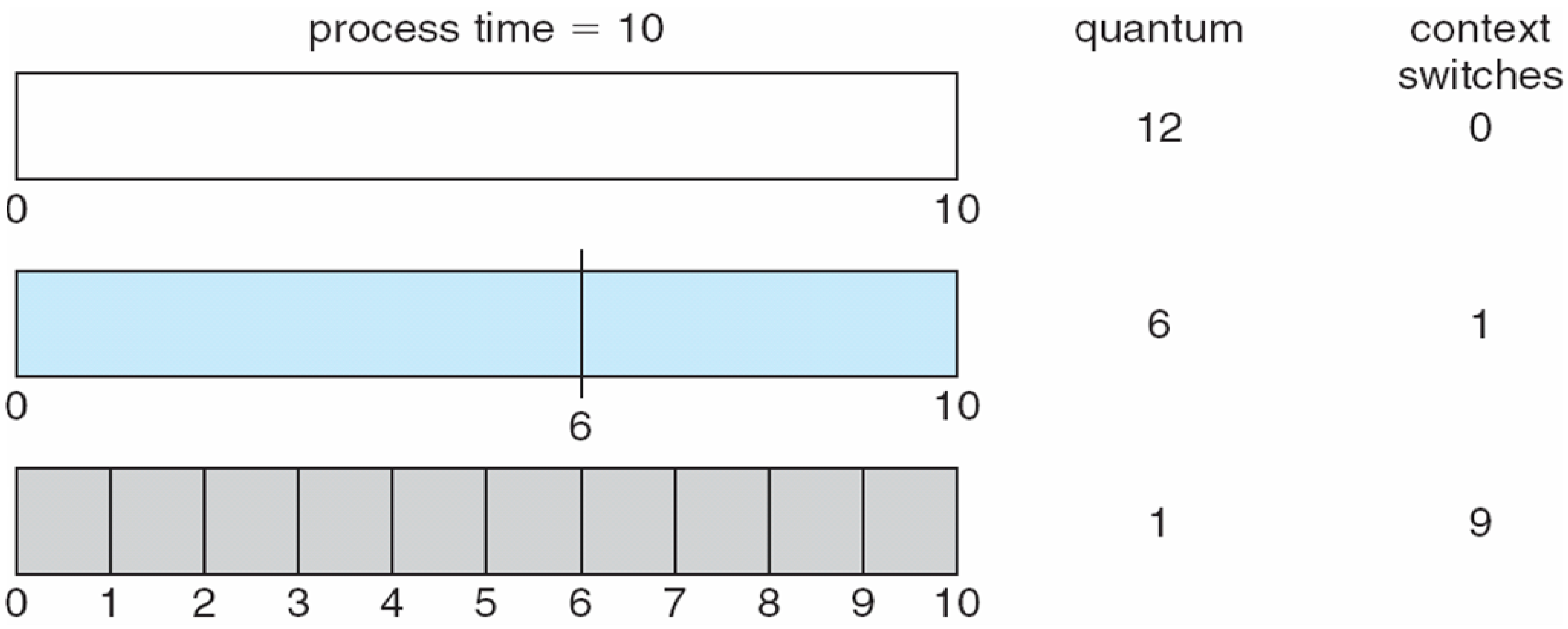
P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

- Typically, higher average turnaround than SJF, but better *response*
  - q should be large compared to context switch time
  - q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time

Context Switch Overhead가 1이라고 한다면,

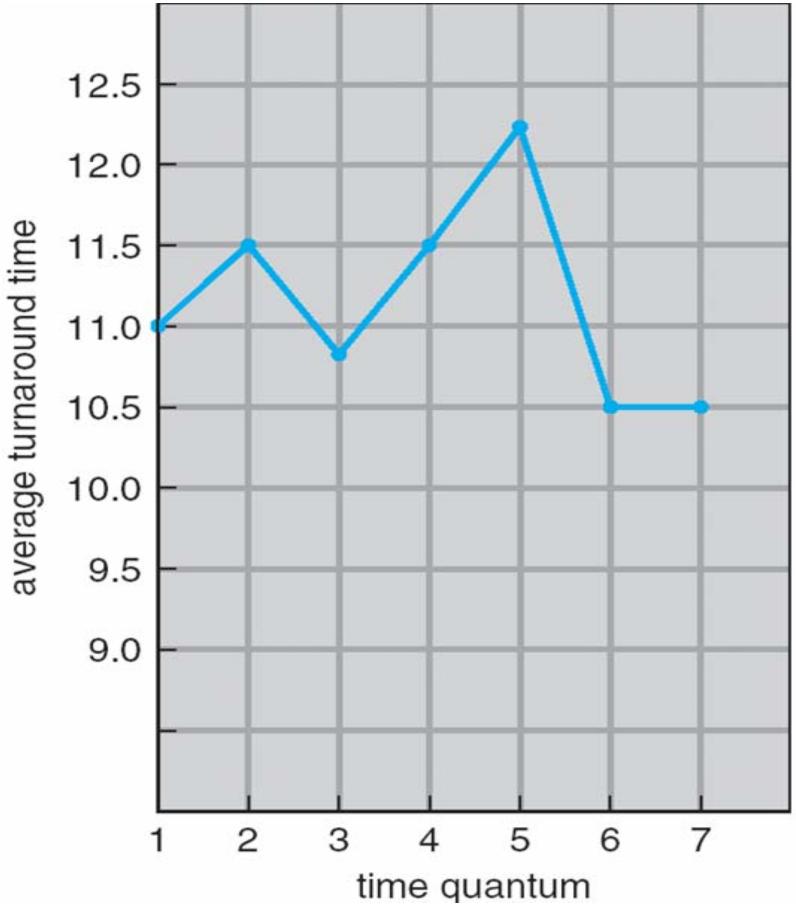


Quantum 의 크기

- 길이
- 고정 대 가변
- 대단히 클 경우 **FIFO** 와 동일
- 작아질수록 문맥교환이 빈번
- 최적치**:** 대부분의 대화형 사용자의 요구가 **quantum** 보다 짧은 시간에 처리될 경우

경험적으로, CPU 버스트의 80%는 Quantum 보다 짧아야 한다!

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

HRN(Highest-Response-ratio Next) NonPreemptive

- **HRN(Highest-Response-ratio Next) 스케쥴링**
  - **SJF** 는 짧은 **job** 을 지나치게 선호
    - 실행시간이 긴 프로세스에 불리한 **SJF** 기법을 보완하기 위한 것으로 대기시간과 서비스 시간을 이용하는 기법
  - 우선순위를 계산하여 그 숫자가 가장 높은 것부터 낮은 순으로 우선순위가 부여
  - 우선순위 = $\dfrac{\text{대기시간 + 서비스시간}}{\text{서비스시간}}$

작업	대기시간	서비스시간
A	5	5
B	10	6
C	15	7
D	20	8

- A : (5 + 5) / 5 = 2
- B : (10 + 6) / 6 = 2.67
- C : (15 + 7) / 7 = 3.14
- D : (20 + 8) / 8 = 3.5

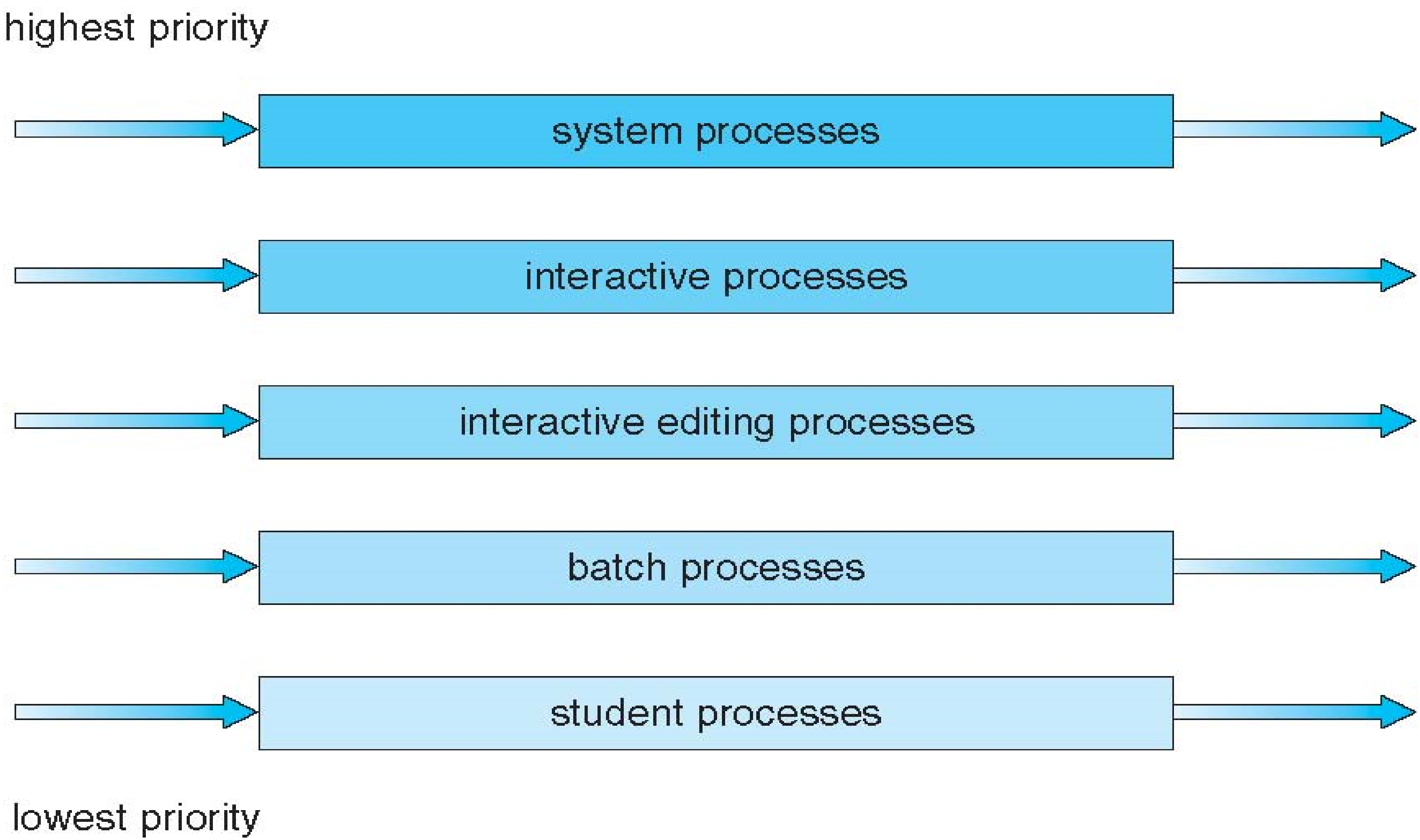
※ 우선순위가 가장 높은 것은 D

Multilevel Queue

Preemptive

- 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

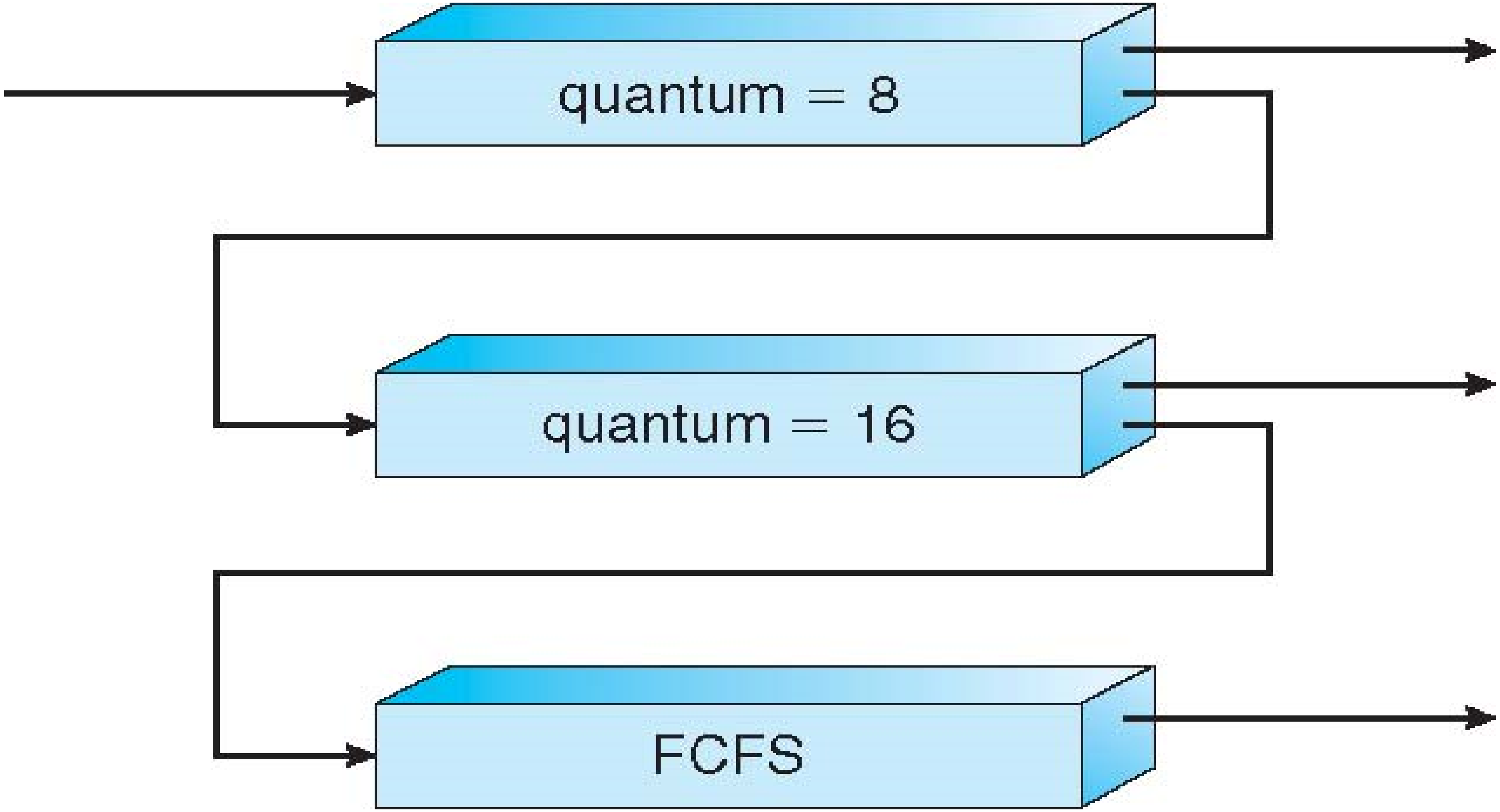
- **A process can move between the various queues; aging can be implemented this way**

- **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

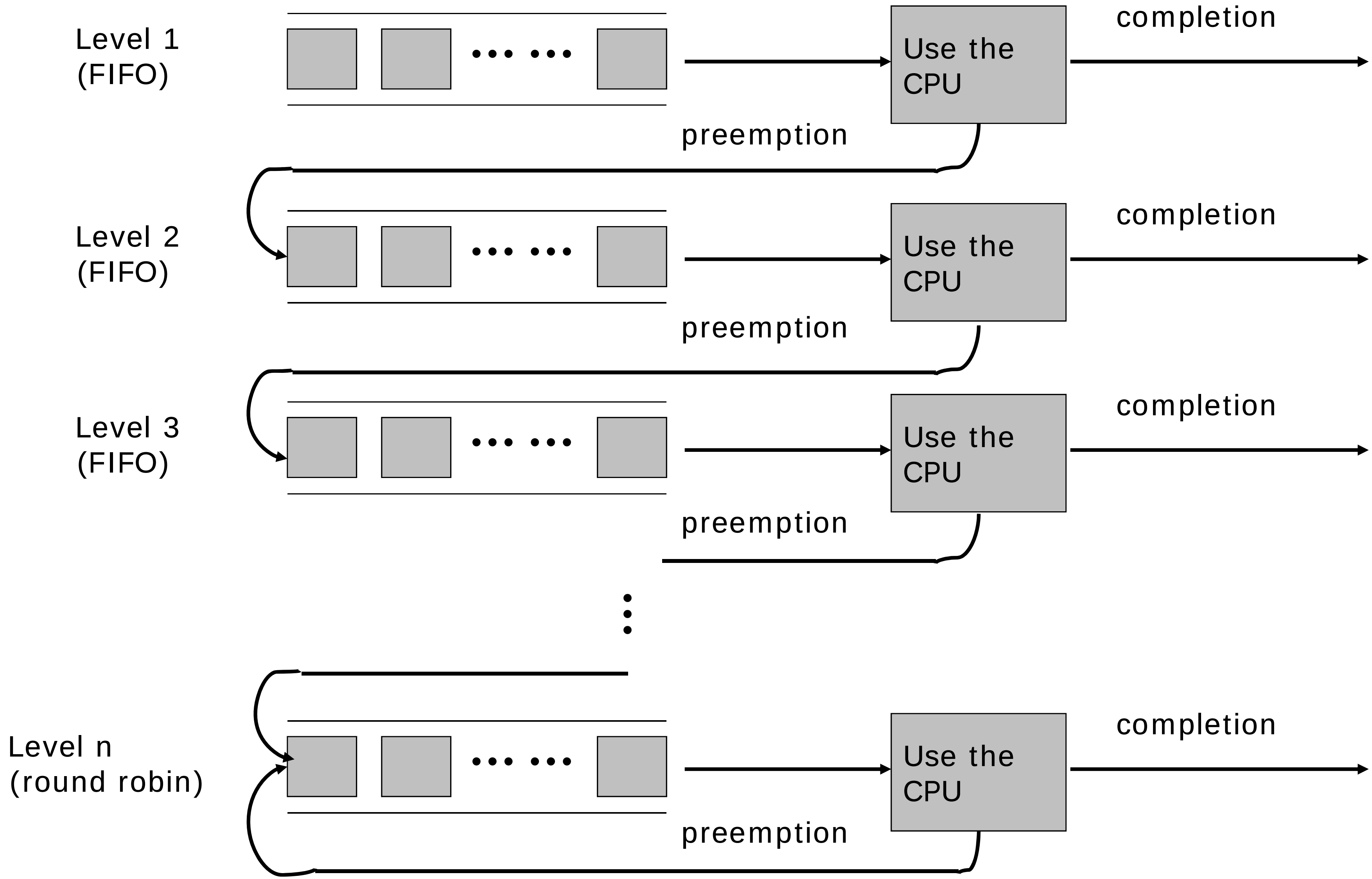
- **Three queues:**
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- **Scheduling**
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved 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



Multilevel Feedback Queue: Preemptive

- 프로세스의 특성에 따라 처리
- 짧은 작업에 우선권
- **IO** 위주의 작업에 우선권 **(IO** 장치를 충분히 사용**)**
- **CPU-bound / IO-bound** 를 빨리 파악
- **CPU bound-job :** 계산위주의 작업 **(**점차 아래로 이동**)**
- **IO bound-job : (**상위 **level** 에서 처리**)**


Level 1
(FIFO)
Use the CPU
completion
preemption
Level 2
(FIFO)
Use the CPU
completion
preemption
Level 3
(FIFO)
Use the CPU
completion
preemption
Level n
(round robin)
Use the CPU
completion
preemption

Thread Scheduling

- Distinction between user-level and kernel-level threads
  - When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as Process-Contention Scope (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is System-Contention Scope (SCS) – competition among all threads in system

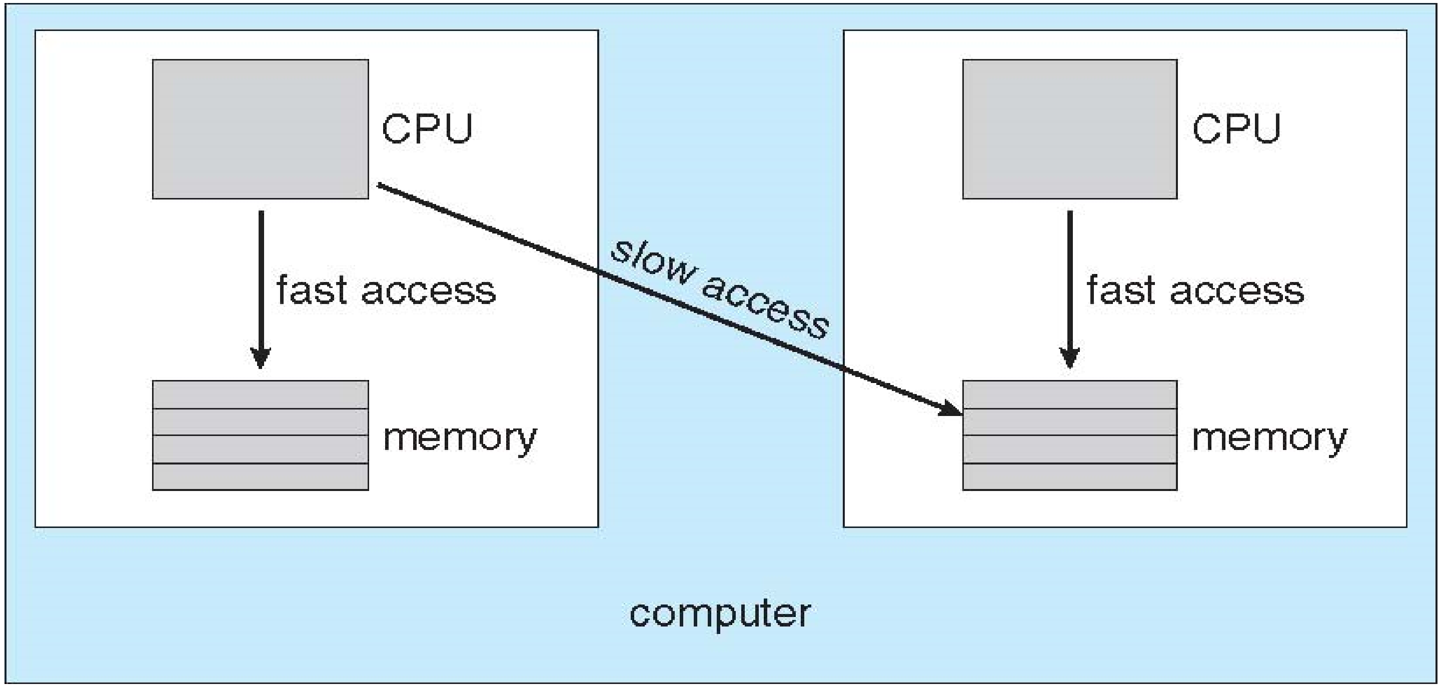
Multiple-Processor Scheduling

- ***Asymmetric multiprocessing***
  - *하나의 **processor**가 **scheduling** 하므로 자료 공유가 없음*
- ***Symmetric multiprocessing(SMP)***
  - *각 **processor**가 독자적으로 **scheduling***
  - ***Load sharing*** **:** 공동의 **Ready Queue** 사용 가능

- 처리기 친화성**(Processor Affinity)**
  - **CPU core**의 **cache** 활용성을 높이기 위해 같은 **core**를 선호하는 것
  - **Hard Affinity, Soft Affinity**

- **Load Balancing**
  - **Push :** 특정 태스크가 주기적으로 부하 검사
  - **Pull :** 쉬고 있는 프로세서에서 다른 프로세서의 **load**를 가져옴

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

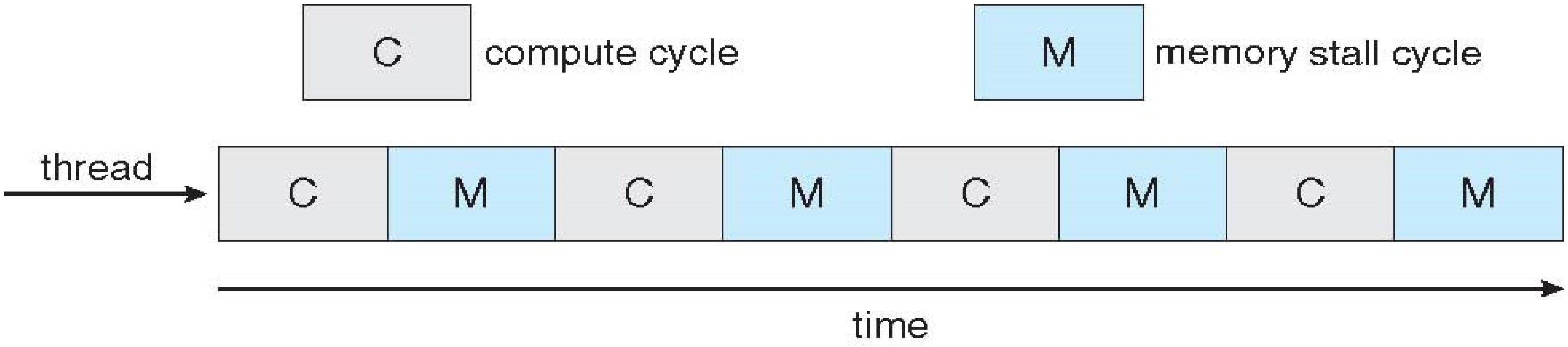
Multicore Processors

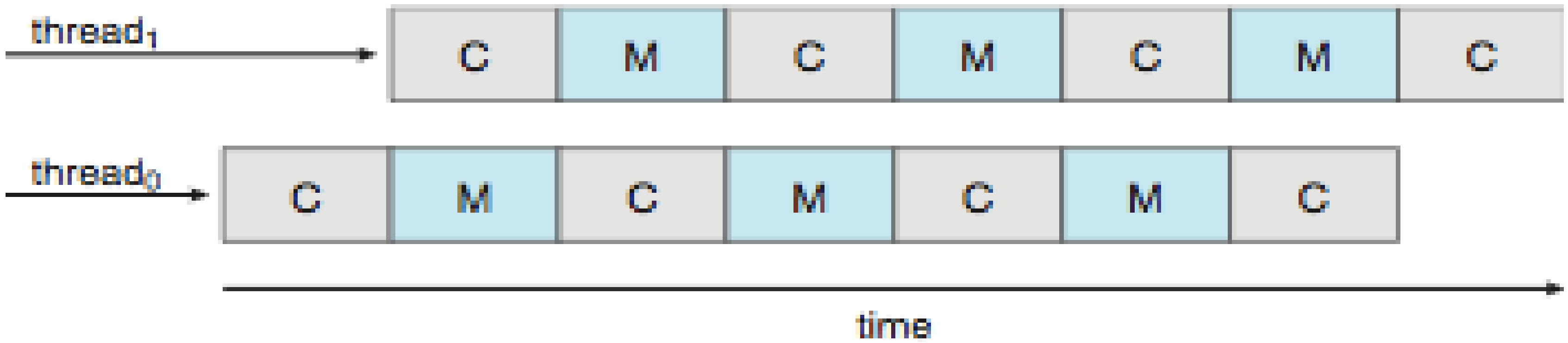
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multicore Processors

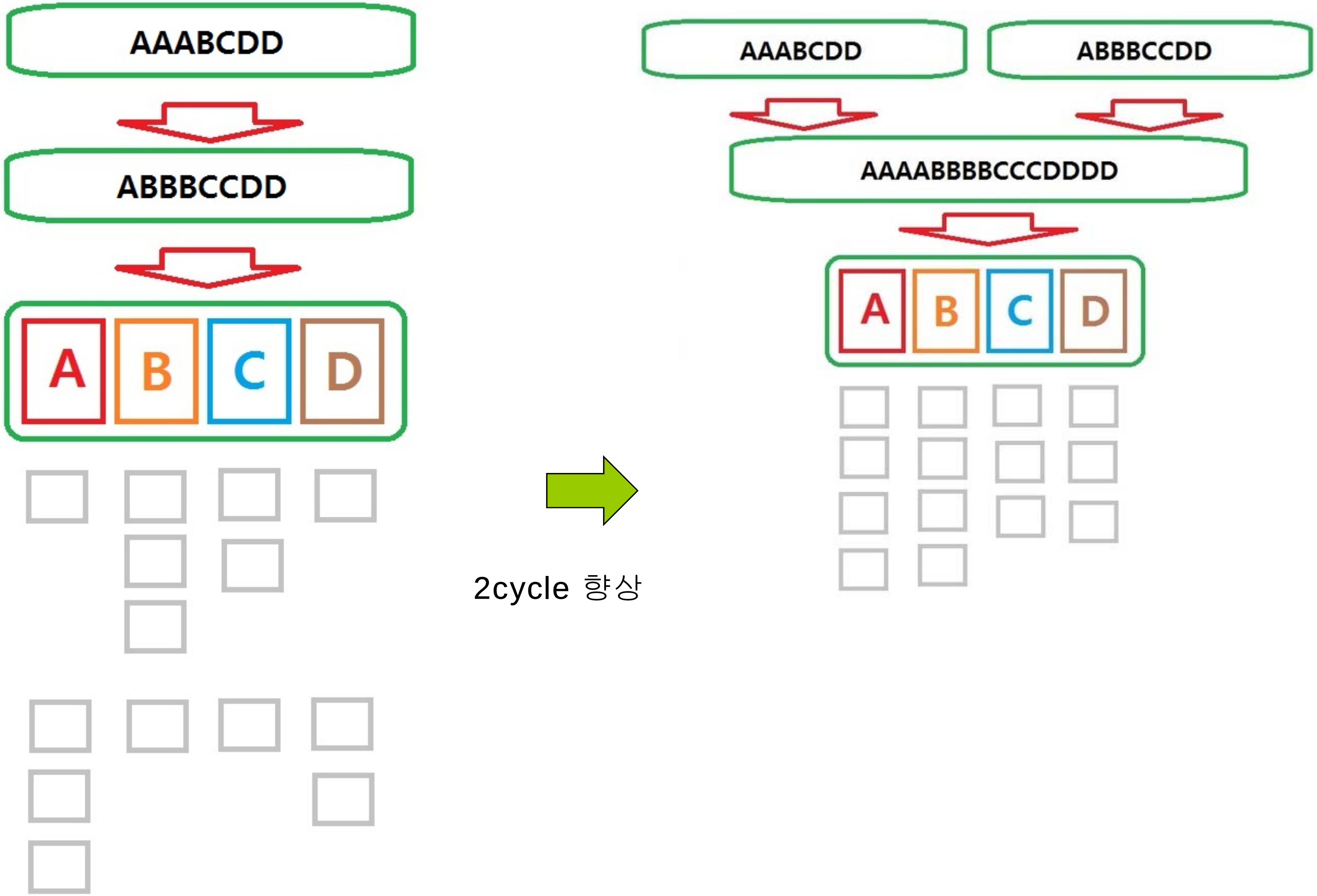
- There are two ways to multi-thread a processor:
  - *Coarse-grained* multithreading switches between threads only when one thread blocks, say on a memory read. Context switching is similar to process switching, with considerable overhead.
  - *Fine-grained* multithreading occurs on smaller regular intervals, say on the boundary of instruction cycles. However the architecture is designed to support thread switching, so the overhead is relatively minor.

Multithreaded Multicore System





Hyperthreading : Best Case



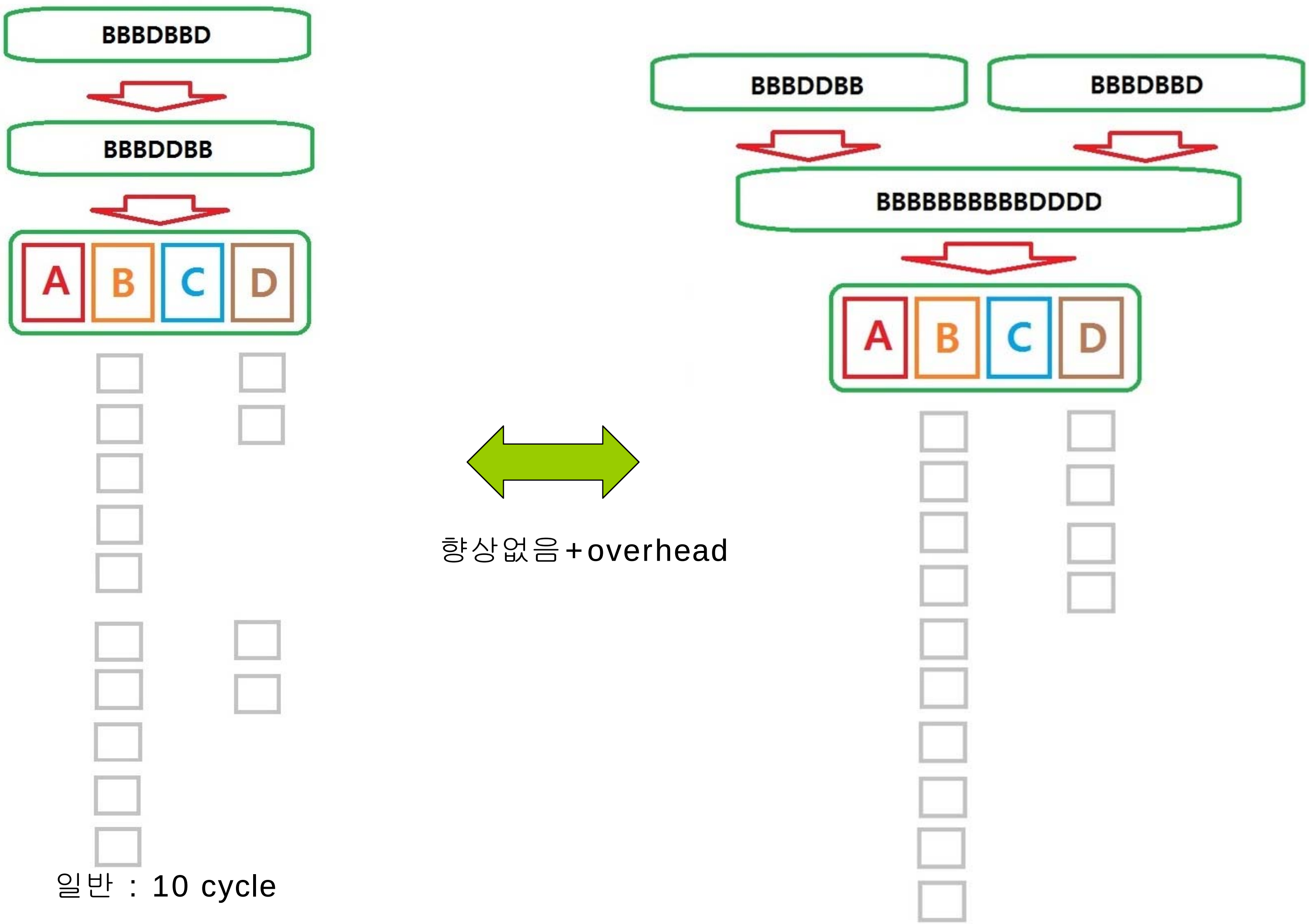
2cycle 향상

일반 : 6 cycle

Hyperthreading : 4 cycle

출처 : http://blog.naver.com/jky

Hyperthreading : Worst Case



출처 : http://blog.naver.com/jky

Hyperthreading : 10 cycle

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn't own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

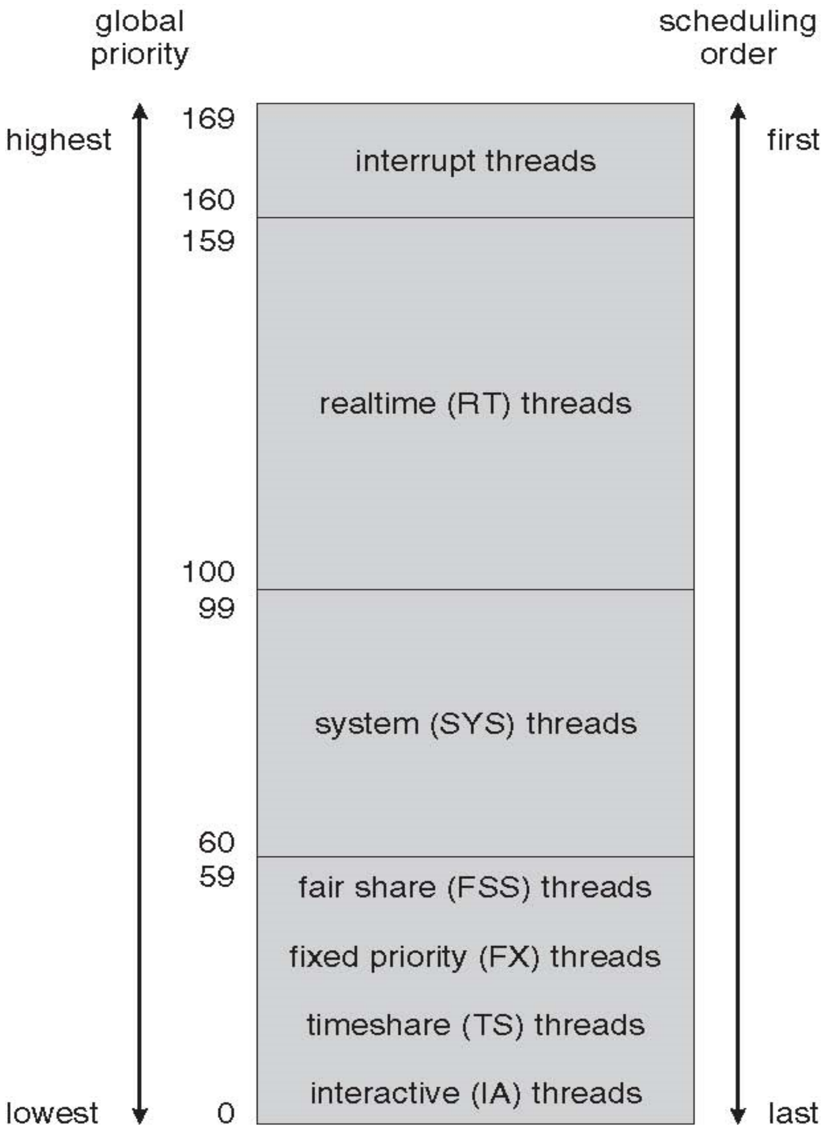
Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default)
  - Interactive
  - Real time
  - System
  - Fair Share
  - Fixed priority
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris Dispatch Table

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

Solaris Scheduling



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

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 higher-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
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost

Windows XP 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

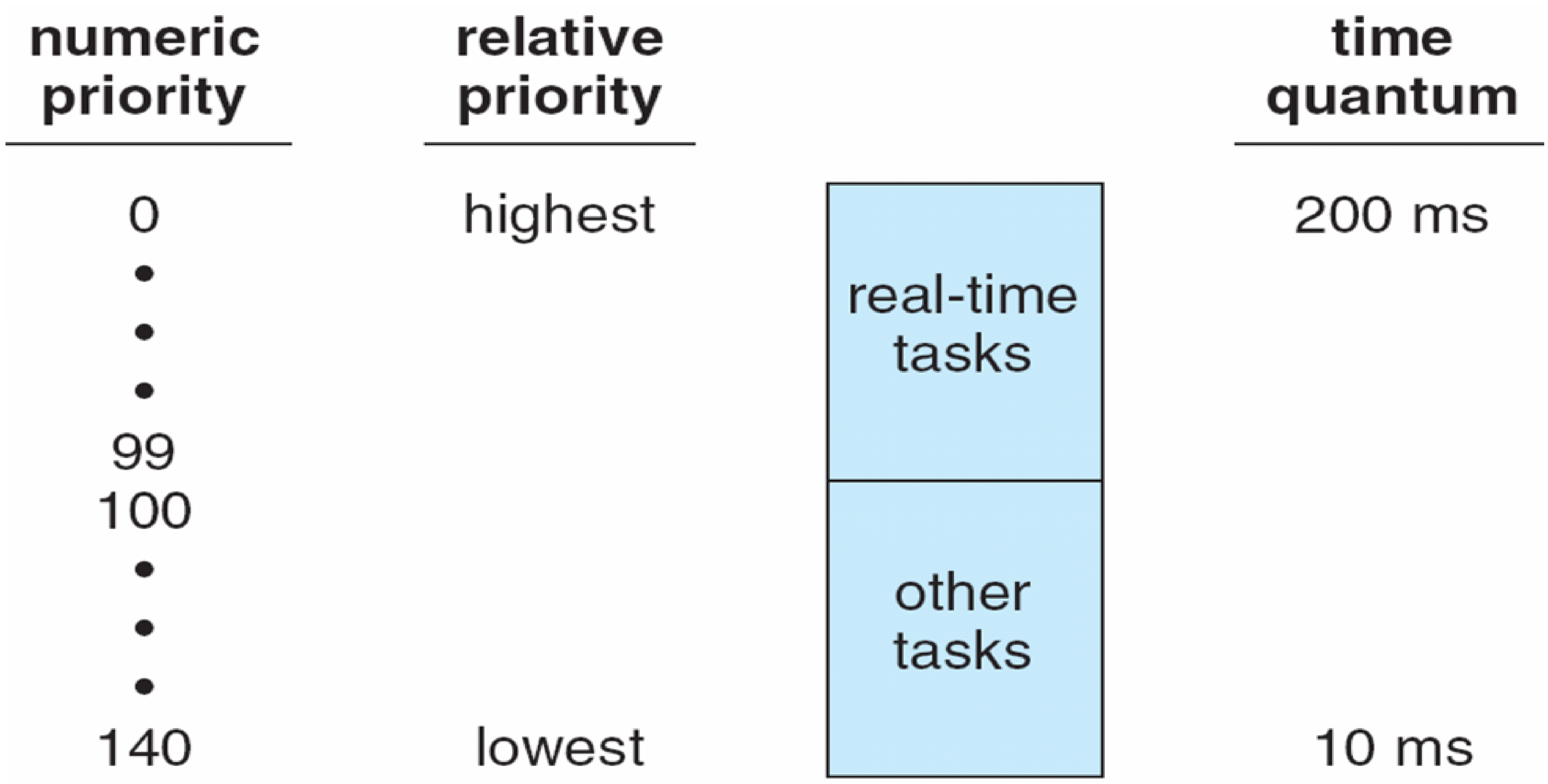
Linux Scheduling

- 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 priority
- 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
  - Two priority arrays (active, expired)
  - Tasks indexed by priority
  - When no more active, arrays are exchanged

Linux Scheduling (Cont.)

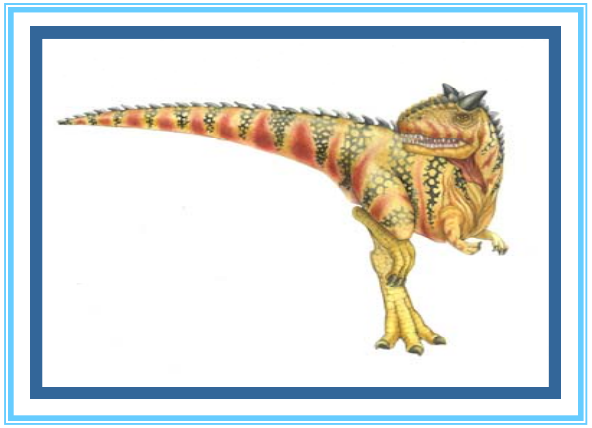
- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- All other tasks dynamic based on *nice* value plus or minus 5
  - Interactivity of task determines plus or minus
    - More interactive -> more minus
  - Priority recalculated when task expired
  - This exchanging arrays implements adjusted priorities

Priorities and Time-slice length



List of Tasks Indexed According to Priorities

End of Chapter 5



Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- Deterministic modeling
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

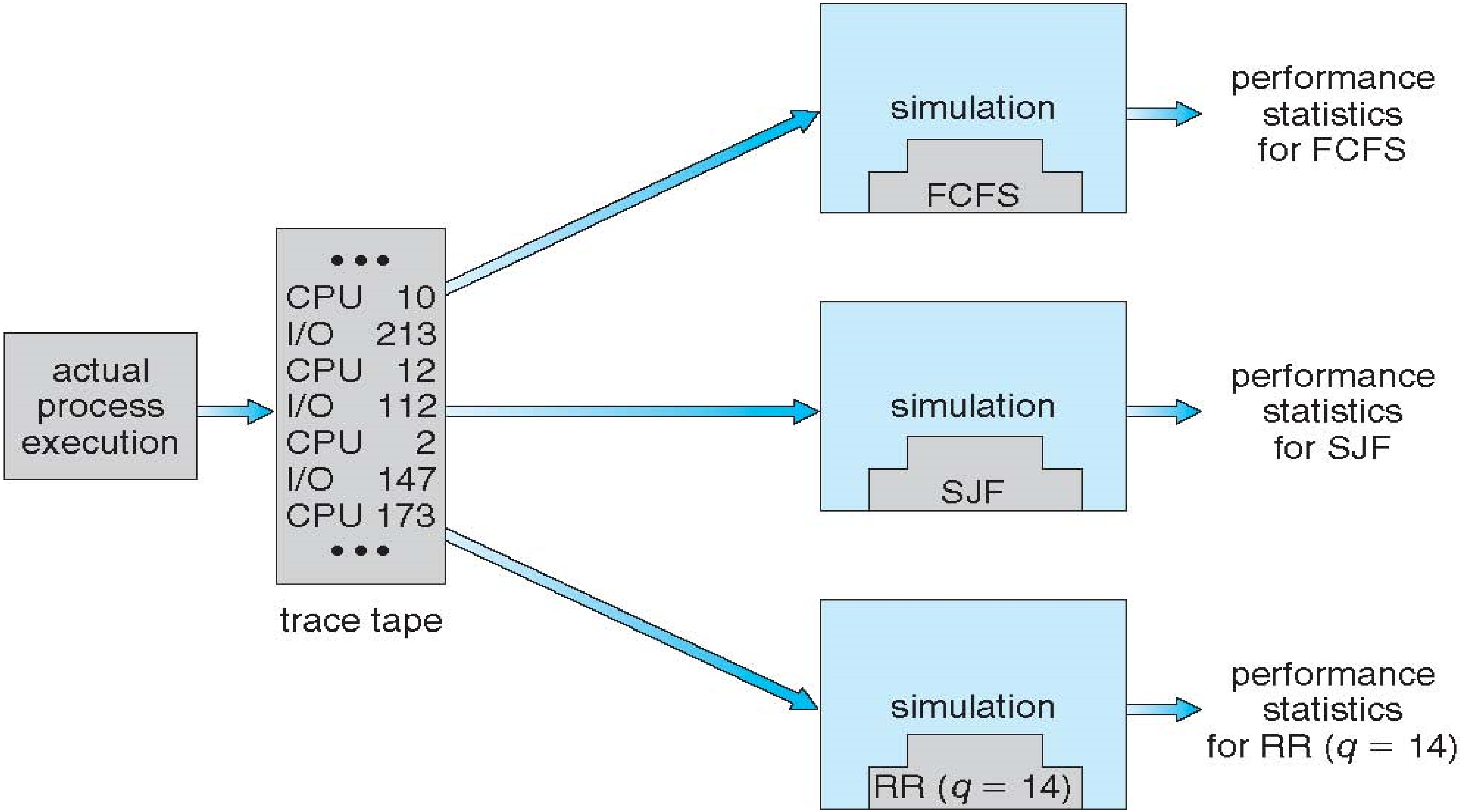
Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus
  $n = \lambda \times W$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

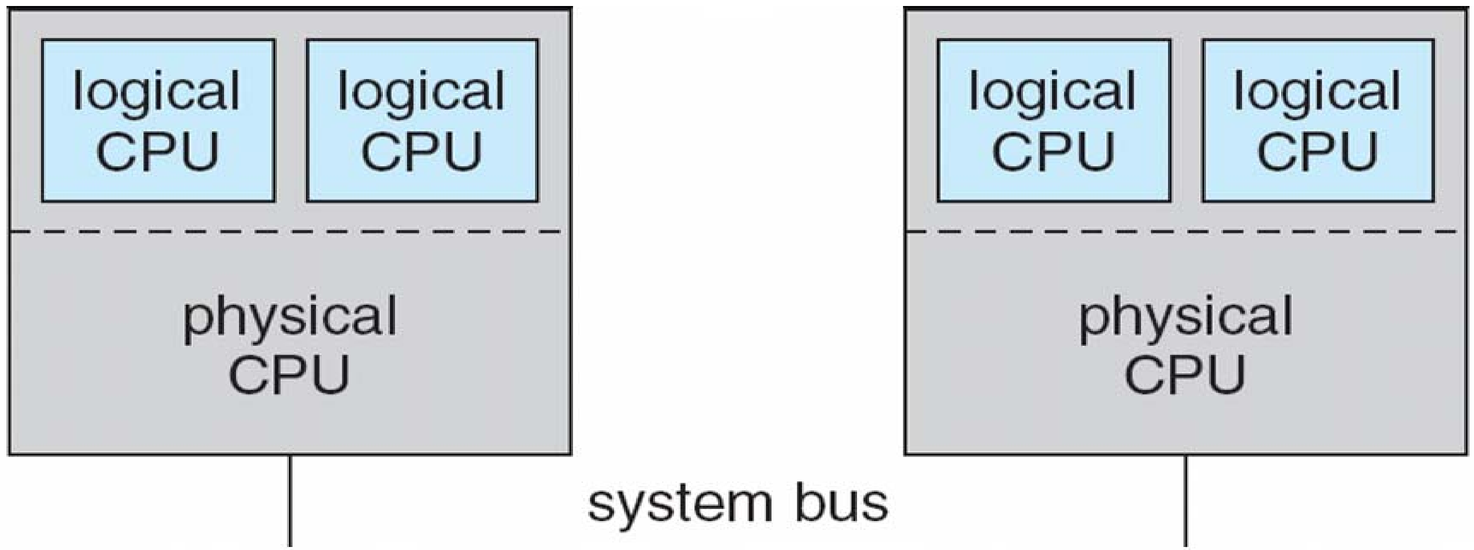
Evaluation of CPU Schedulers by Simulation



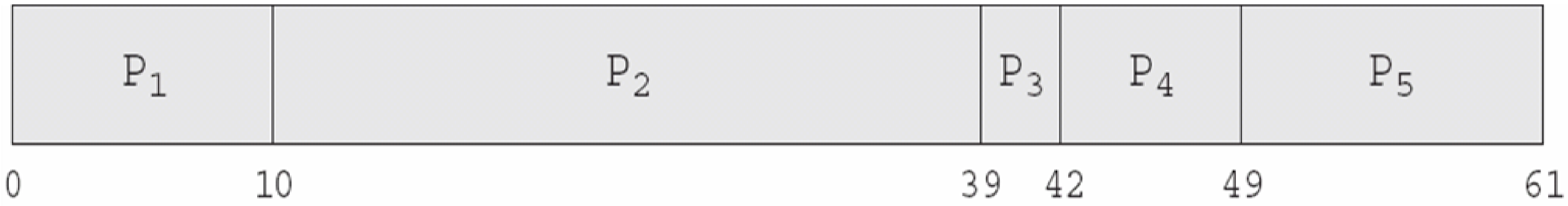
Implementation

- **Even simulations have limited accuracy**
- **Just implement new scheduler and test in real systems**
  - **High cost, high risk**
  - **Environments vary**
- **Most flexible schedulers can be modified per-site or per-system**
- **Or APIs to modify priorities**
- **But again environments vary**

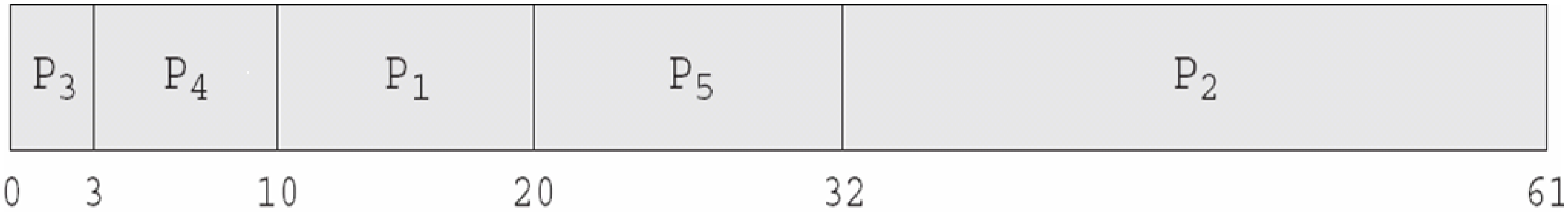
5.08


logical CPU
logical CPU
physical CPU
logical CPU
logical CPU
physical CPU
system bus

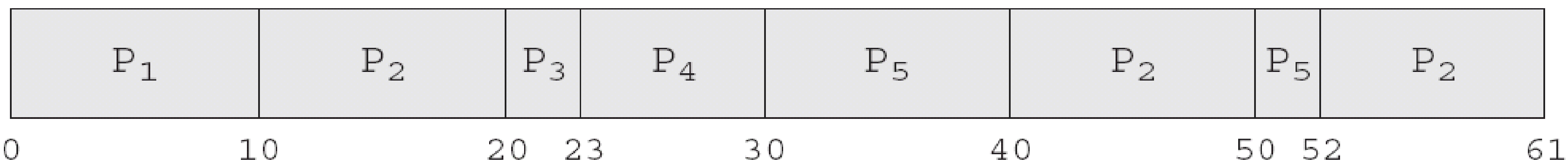
In-5.7



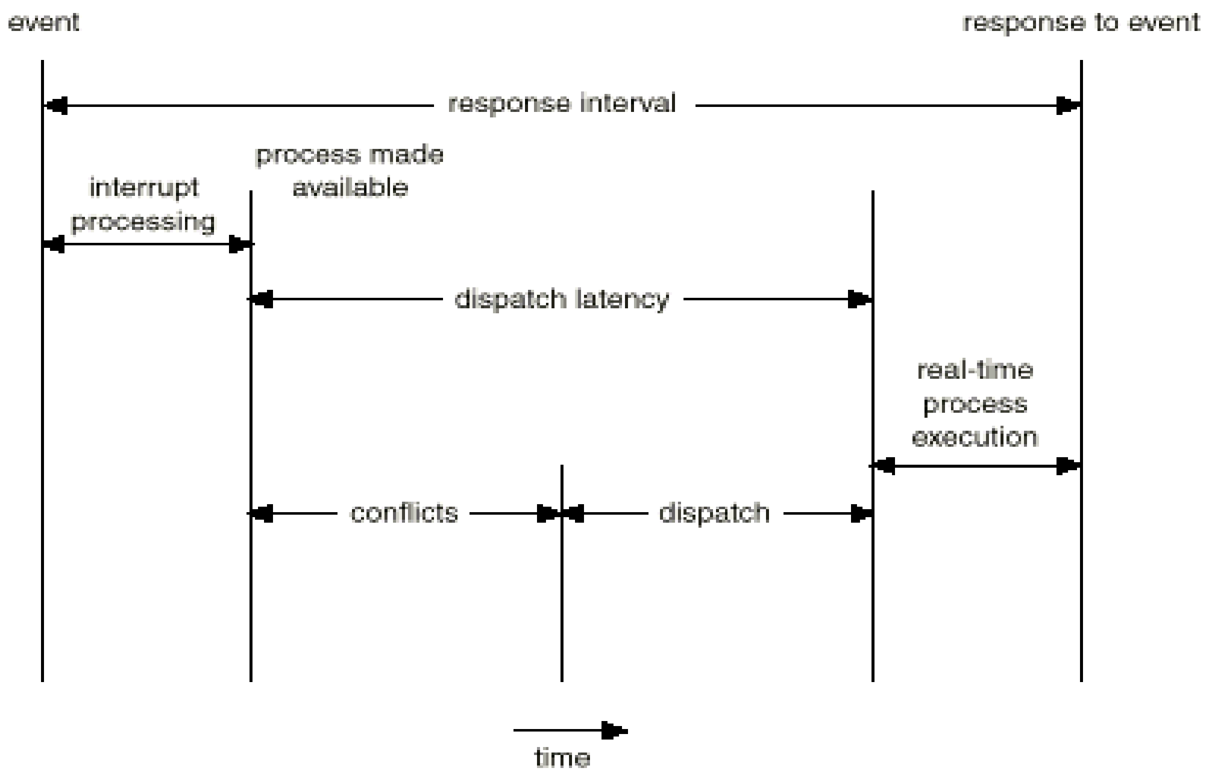
In-5.8



In-5.9



Dispatch Latency



Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm

- FIFO Queue is Used if There Are Multiple Threads With the Same Priority

Java Thread Scheduling (Cont.)

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

Time-Slicing

Since the JVM Doesn't Ensure Time-Slicing, the yield() Method May Be Used:

```
while (true) {
    // perform CPU-intensive task
    . . .
    Thread.yield();
}
```

This Yields Control to Another Thread of Equal Priority

Thread Priorities

Priority	Comment
Thread.MIN_PRIORITY	Minimum Thread Priority
Thread.MAX_PRIORITY	Maximum Thread Priority
Thread.NORM_PRIORITY	Default Thread Priority

Priorities May Be Set Using setPriority() method:

```
setPriority(Thread.NORM_PRIORITY + 2);
```

Solaris 2 Scheduling