V. CPU 스케줄링

Topics

1. 기본 개념
2. 스케줄링 기준
3. **스케줄링 알고리즘**
4. 다중처리기 스케줄링
5. 실시간 CPU 스케줄링
6. 운영체제 사례

CPU Scheduling Algorithms

도착순서 기반:

- First-Come-First-Served (FCFS)
- Round-Robin

우선순위 기반:

- Priority
- Shortest-Job-First (SJF)
  = SPN(Shortest Process Next)
  ≒SRTN(Shortest Remaining Time Next, 최소 잔여 시간)
  ≒HRRN(High Response Ratio Next)

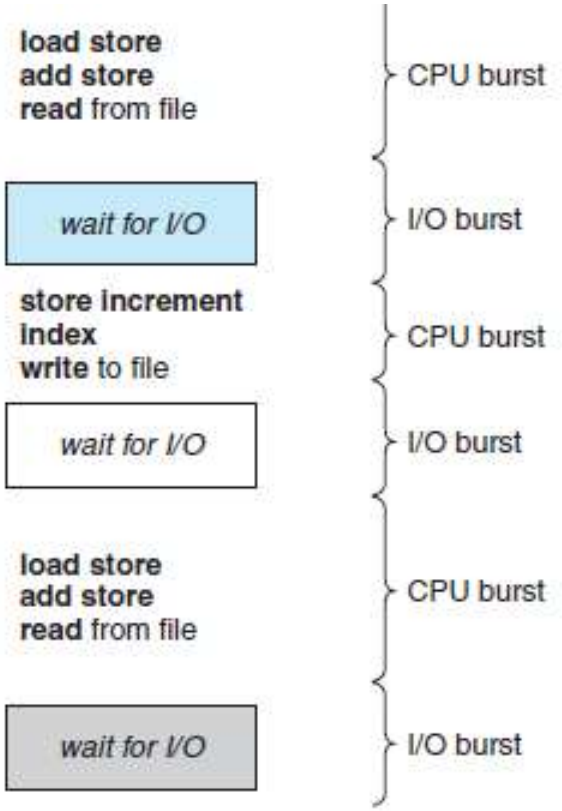
다단계 큐:

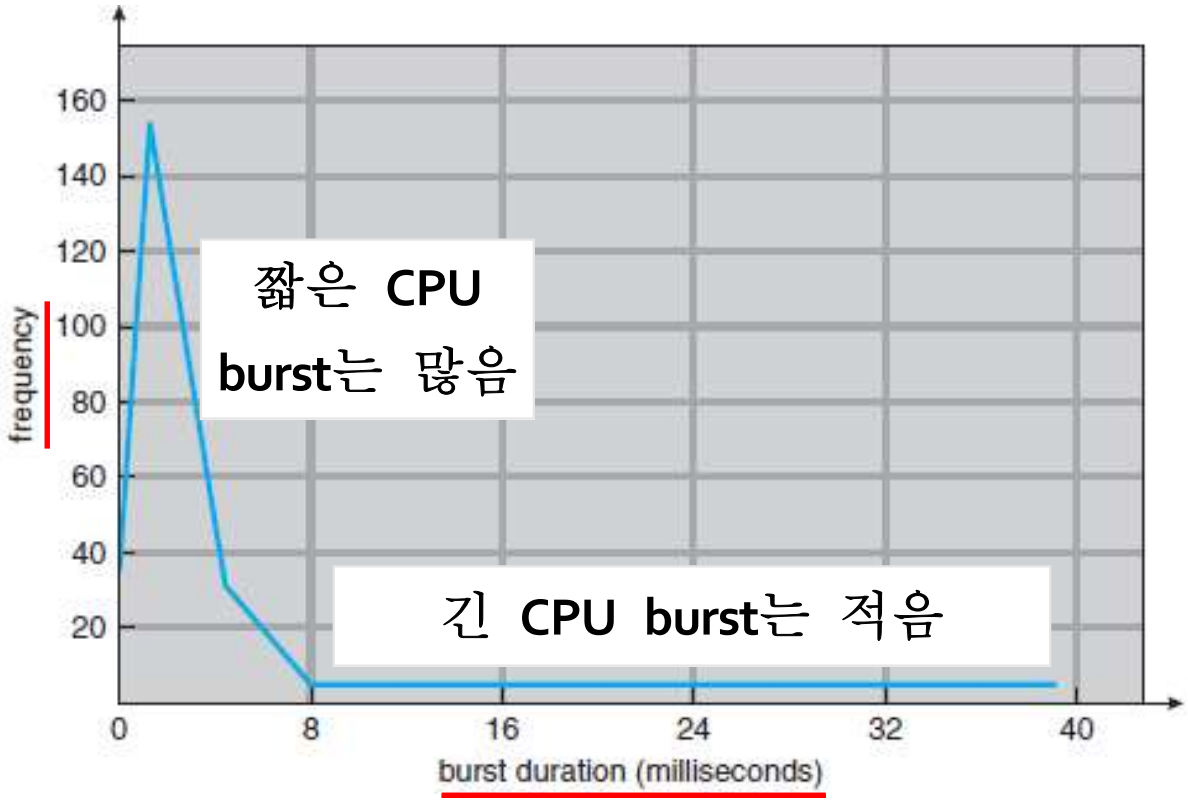
- Multi-level Queue
- Multi-level Feedback Queue

1. 기본 개념

❑ 프로세스 실행의 특성

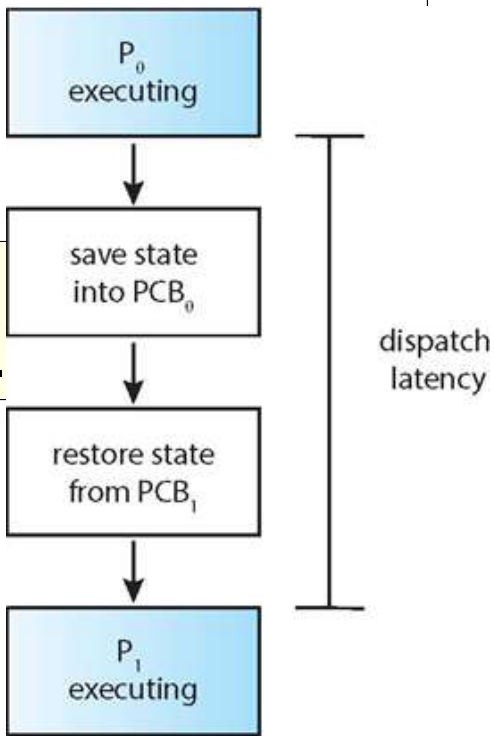
- 프로세스 실행: A cycle of CPU bursts and I/O bursts
- 통계적으로 짧은 CPU burst는 많고, 긴 CPU burst는 적음
- **CPU-bound, I/O-bound program** ⇨ 스케줄링 알고리즘 선택에 영향을 줌





❑ CPU scheduling and Dispatching

CPU Scheduling	Dispatching
ready queue에서 다음에 실행시킬 프로세스를 선택하는 것.	선택된 프로세스에게 CPU control을 넘겨줌: - context switching - user mode 전환 - PC 설정
Dispatch Latency 현행 프로세스 중단부터 다른 프로세스 시작까지 걸리는 시간.	



- Scheduling 유형
  - **선점형, 비선점형**
  - *Preemption* 선점

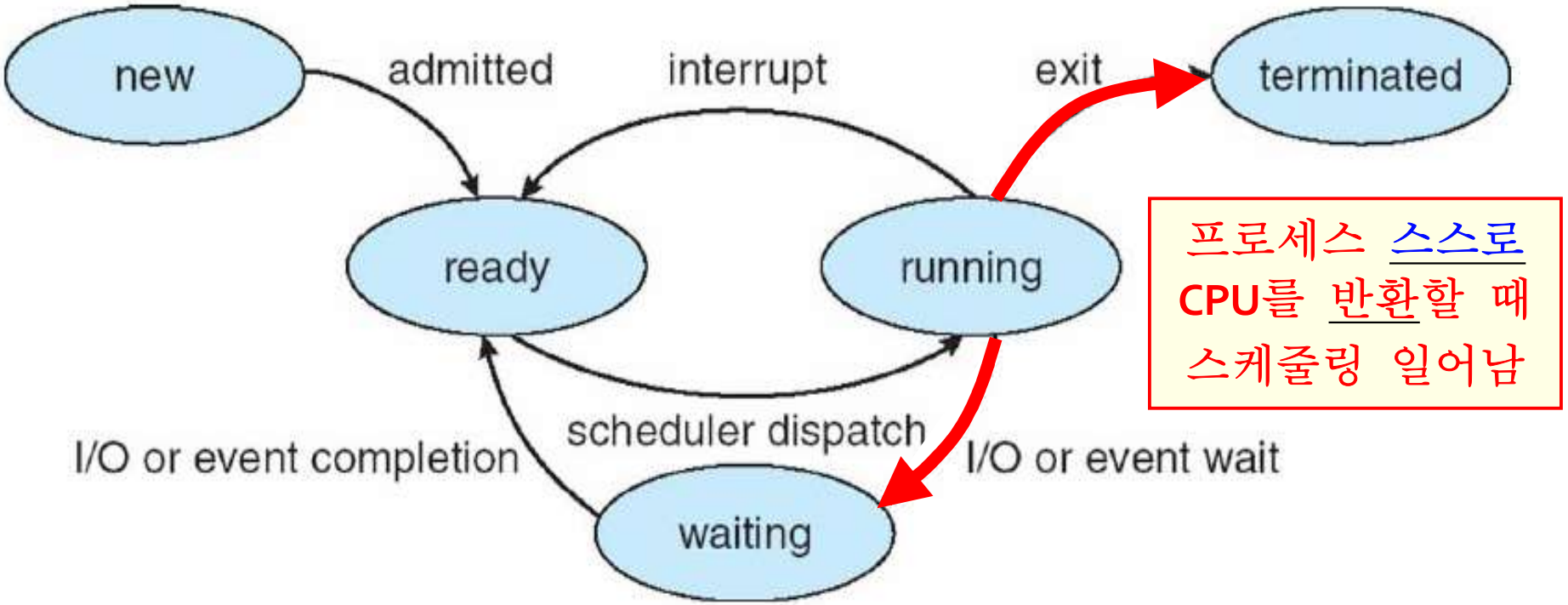
    Temporarily interrupting a process being carried out

    with the intention of resuming the task at a later time.(인터럽트란 말을 사용한 이유)

- **비선점 스케줄링 Non-preemptive scheduling**

  Running process continues to execute

  until ① it terminates or ② it blocks itself.



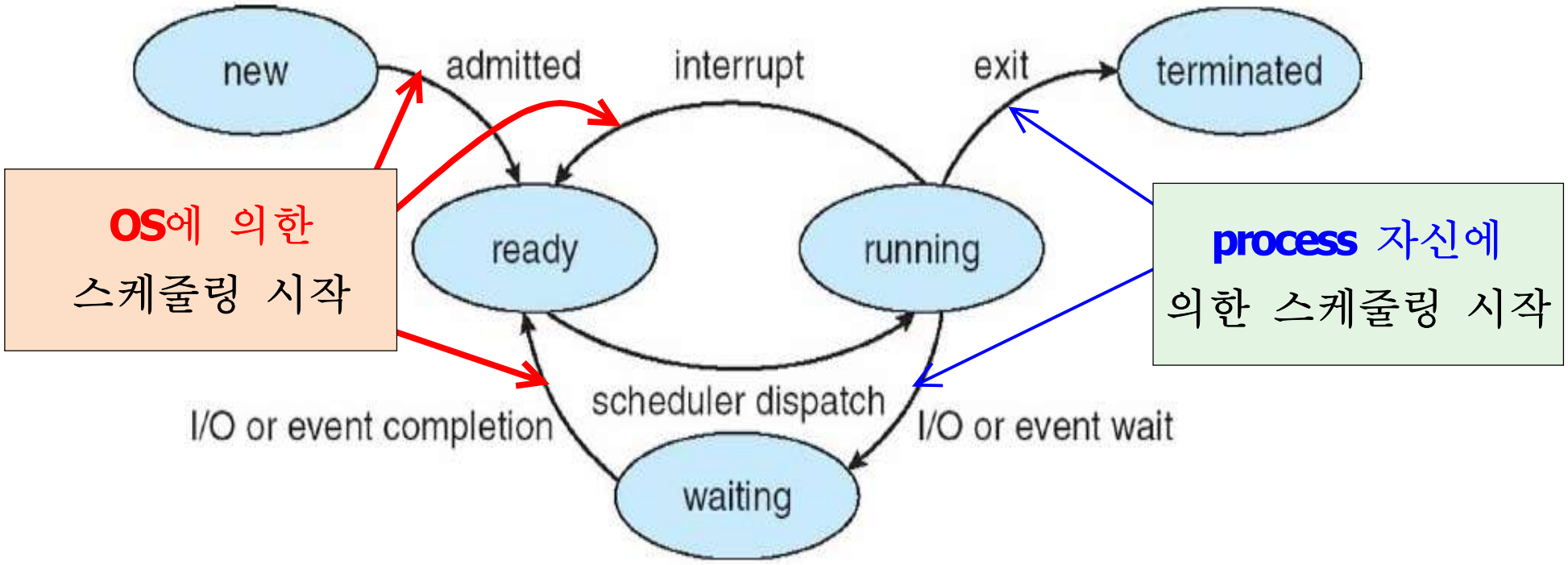
- **선점 스케줄링 Preemptive scheduling**

  Currently running process may be interrupted and moved to READY state by OS:

  ③ By timer interrupt

  ④ By arrival of a high-priority in READY state.

(즉 프로세스 자신의 CPU burst 완료 前 OS에 의해 preemption 가능)



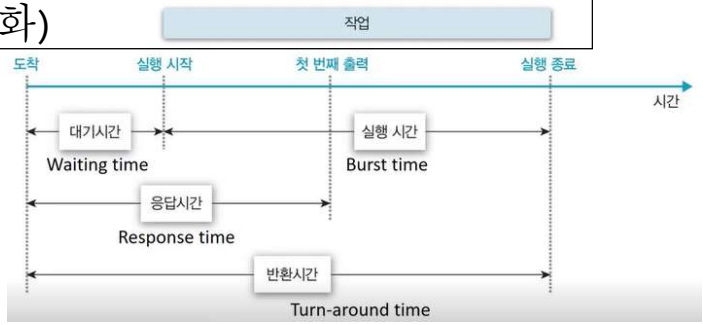
(note) 선점 스케줄링의 Overhead

공유 데이터 접근 도중 **preempted**	프로세스 동기화(Synchronization) 필요
kernel mode에서 **preempted**	시스템호출 완료 후 **preemption**
중요한 **OS** 작업 실행 도중 **interrupted**	**Interrupt_disable;** **{ crucial OS activity }** **Interrupt_enable;**

2. 스케줄링 기준

❑ 스케줄링 時 프로세스 선택에 영향을 주는 요소들

	Scheduling Criteria	설명
전체 시스템 관점	CPU 이용률 ↑ *CPU utilization*	실제로 **40-90%** 사이에 있어야 함
	처리량 ↑ *Throughput*	단위 시간당 완료된 프로세스 개수
개별 프로세스 관점	총처리 시간 ↓ *Turnaround time*	프로세스 제출에서 완료까지 걸리는 시간
	대기 시간 ↓ *Waiting time*	**Ready queue** 내에서 대기한 총시간
	응답시간 ↓ *Response time*	• 하나의 요청 제출~응답 생성(**not** 출력) 시간 • **Interactive system**에서 중요 응답시간 변동폭 최소화 중요 (not 평균 응답시간의 최소화)



3. 스케줄링 알고리즘

알고리즘	프로세스 선택 기준	스케줄링 유형
선입 선처리	도착순서	Non-preemptive only
Round Robin	도착순서	Preemptive only
우선순위	우선순위	Both
최단 작업 우선	다음 CPU burst time 길이	Shortest Job First (SJF) Shortest Remaining Time First (SRTF)
다단계 큐	고정 우선순위	Preemptive only *and* 큐 間 Preemptive
다단계 피드백 큐	동적 우선순위	

(1) 선입 선처리 스케줄링 First Come First Served Scheduling (FCFS)

프로세스가 **Ready queue**에 도착한 순서로 **CPU**를 할당함.

(FIFO Ready queue에서 가장 오랫동안 대기한 프로세스 선택)

비선점형	프로세스 자신이 ***CPU burst***를 완료할 때 스케줄링 시작.

프로세스	Burst time
P1	6
P2	3
P3	100
도착순서: P1→P2→P3	

Gantt chart for schedule

- 평균 대기시간 = (0 + 6 + 9)/3 = 5.00
- 만약 도착순서가 P3 → P1 → P2 이면 평균 대기시간은
  (0 + 100 + 106)/3 = 68.67 (Convoy Effect)

(장점) 가장 간단한 알고리즘 이며,
스케줄러 구현이 용이함 (FIFO Ready queue를 이용).

(단점) 평균 대기시간이 매우 길어질 수 있음.

Convoy Effect

long process가 앞에 있으면 많은 short process가 오랫동안 대기함.

(예) 많은 I/O-bound 프로세스 앞에 CPU-bound 프로세스가 실행 중.

(2) Round-Robin Scheduling (RR)

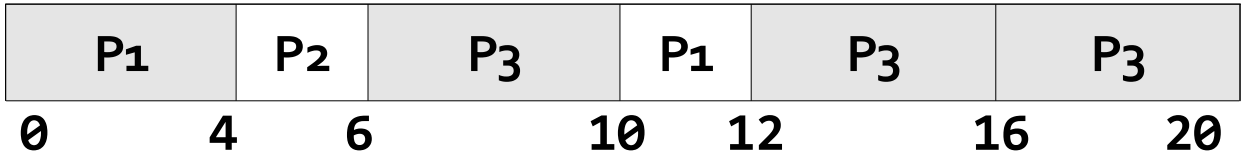
프로세스가 Ready queue에 도착한 순서로 CPU를 할당함 (FCFS와 동일)

선점형	timer interrupt에 의해 스케줄링이 시작됨. (현재 실행중인 프로세스는 ready queue의 tail에 추가됨)
*time-sharing system*의 전형적인 CPU sharing 기법: RR = FCFS + Preemption by timer interrupt	

프로세스	Burst time
P1 P2 P3	6 2 12
도착순서: P1→P2→P3.	

Assume: Time Quantum(Time Slice) = 4.00

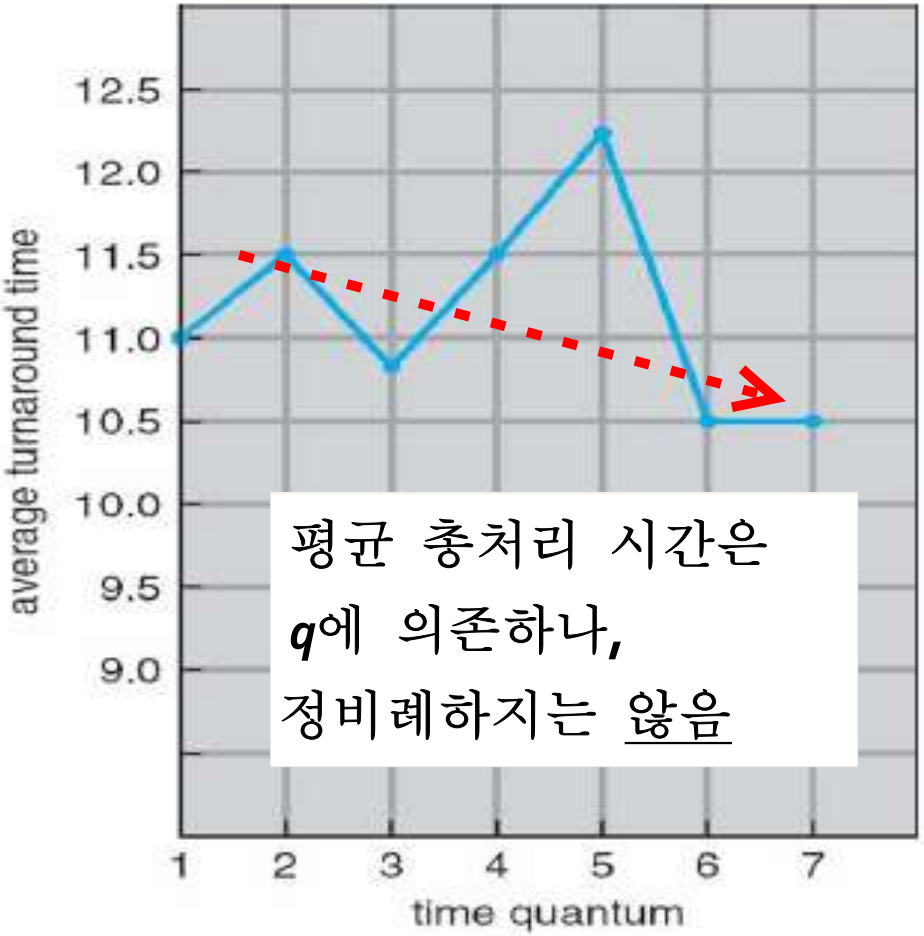
Gantt chart for schedule



- 평균 대기시간 = ((6) + (4) + (6+2))/3 = 6.00
- 평균 총처리 시간 = (12 + 6 + 20)/3 = 12.67
- Response time이 좋음.

❑ Time quantum, *q*

- **context switching time** (보통 10μs 미만) 보다 커야함: 보통 q = 10-100ms.
- *q*가 커질수록 **FCFS**에 가까워짐 (퇴보함)
- **Turnaround time**과의 관계



Burst time:

(P1, 6) (P2, 3) (P3, 1) (P4, 7)

q=3) (13+6+7+17)/4 = 43/4 = 10.75

q=5) (15+8+9+17)/4 = 49/4 = 12.25

경험적으로
프로세스의 **CPU burst** 中 **80%** 이상이
q 보다 짧아야 좋아짐.

(3) 우선순위 스케줄링 Priority Scheduling

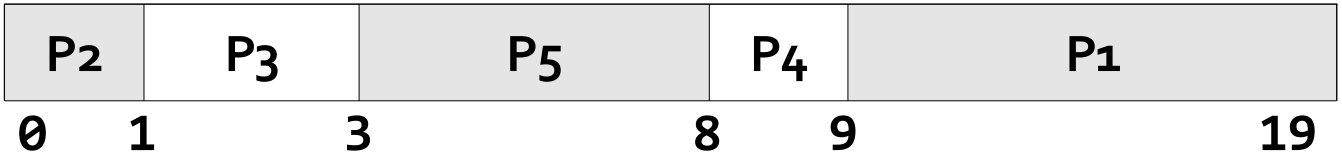
우선순위가 가장 높은 프로세스 선택

비선점형	running process가 자신의 *CPU burst*를 완료할 때 스케줄링 시작.
선점형	우선순위가 높은 프로세스가 Ready queue 도착 즉시 스케줄링 시작.

프로세스	Burst time	우선순위
P1	10	5
P2	1	1
P3	2	2
P4	1	4
P5	5	3
동시 도착.		

(예) 비선점형

Gantt chart for schedule



- 평균 대기시간 = (9 + 0 + 1 + 8 + 3)/5 = 4.20

❑ 우선순위

- Internal priority) 측정치를 사용하여 프로세스의 우선순위를 계산함.

  (예) time limit, 메모리 요구, open file 개수, CPU/IO burst 비율

- External priority) 운영체제 외적 기준에 의해 설정됨.

  (예) 프로세스의 중요도, payment, 수행 부서(주체), 정치적 요인

❑ Starvation (또는 Indefinite blocking)

- 자신 보다 우선순위가 높은 프로세스들 때문에
  Ready process가 오랫동안 CPU를 할당받지 못하는 것.

- Aging) 오랫동안 ready 상태인 프로세스의 우선순위를 주기적으로 높여줌.

(4) 최단 작업 우선 스케줄링 Shortest Job First Scheduling (SJF)

다음 CPU burst time(T)이 가장 짧은 프로세스 선택
(단, T가 같으면 도착 순서대로)

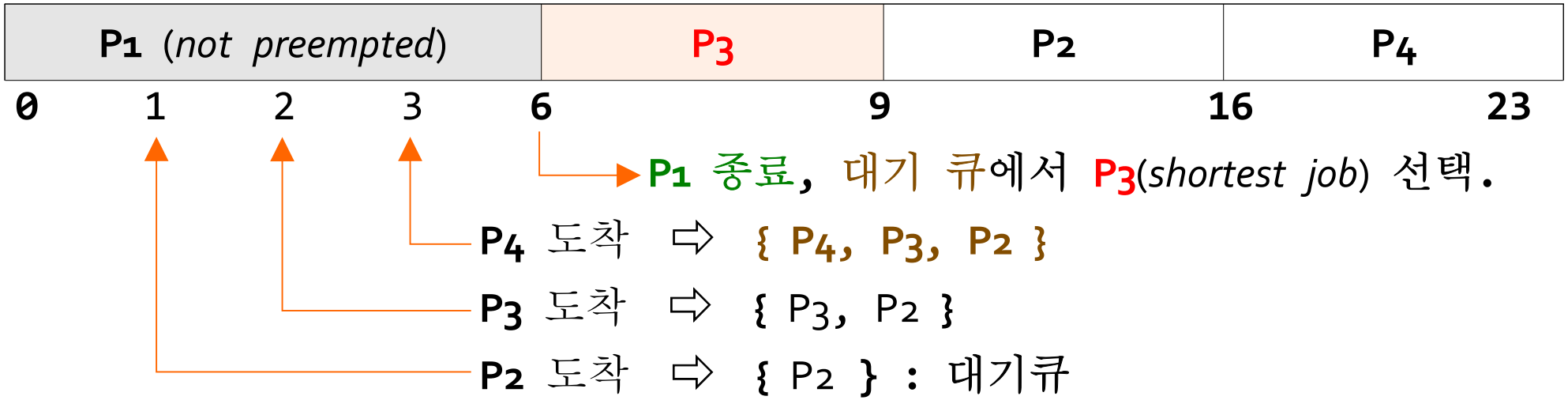
비선점형	현행 프로세스가 자신의 *CPU burst*를 완료할 때 스케줄링 시작.
선점형	remaining T가 더 짧은 프로세스가 ready queue에 도착 즉시 시작. called Shortest Remaining Time First (SRTF)

- 주어진 프로세스 집합에 대해 최소 평균대기시간 보장 (최적 알고리즘)
  (∵) short process를 앞에 두면 대기시간이 단축되는 것은 명확함.

- 프로세스의 next CPU burst 길이는 사전에 알 수는 없으나, 예측 가능.
  (예) 지수평균법 (Exponential averaging)

(예) 비선점형: Shortest Job First (SJF)

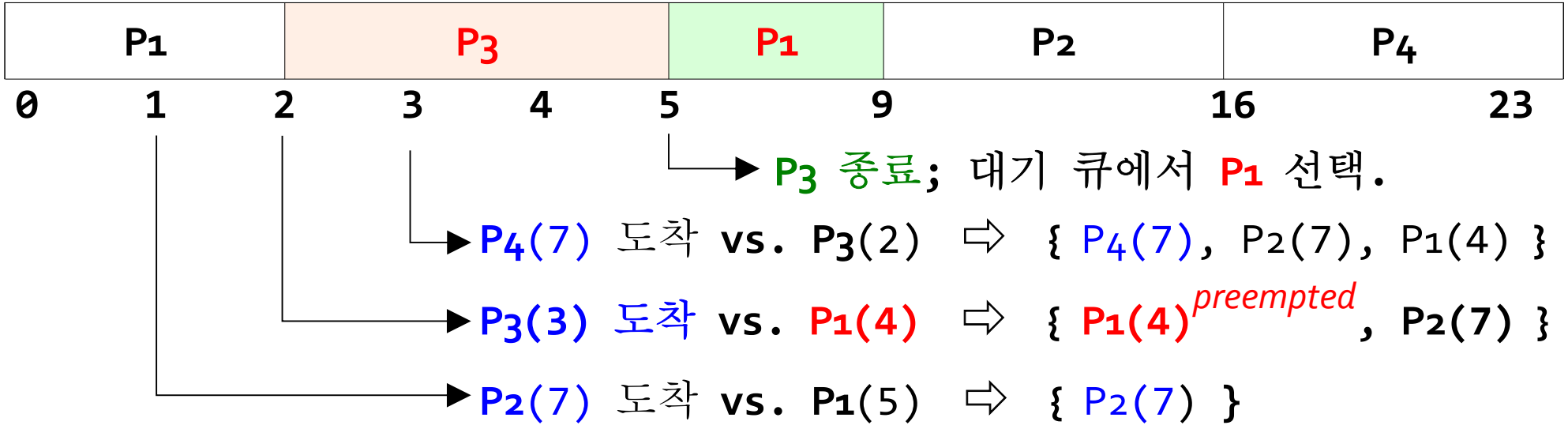
프로세스	Burst time	도착 시각
P1	6	0
P2	7	1
P3	3	2
P4	7	3



평균 대기시간 = (0 + (9-1) + (6-2) + (16-3))/4 = 6.25

(예) 선점형: Shortest Remaining Time First (SRTF)

프로세스	(Remaining) Burst time	도착 시각
P1	6	0
P2	7	1
P3	3	2
P4	7	3



평균 대기시간 = ((5-2) + (9-1) + 0 + (16-3))/6 = 6.00

Next CPU burst time의 예측

(참고) 산술 평균 방법

이전 CPU burst time들의
산술평균을 예측치로 사용

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i$$

$$= \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

where:

- S_{n+1} : $(n+1)$번째 예측치
- T_i : i번째 실제 길이
- S_1 : 첫 번째 예측치

(note)

T_n과 $\sum_{i=1}^{n-1} T_i$ 間 가중치 없음

지수 평균 방법

최근과 과거 CPU burst time들 간에
가중치를 둠.

$$S_{n+1} = \alpha T_n + (1-\alpha) S_n$$

$$where \ (0 \le \alpha \le 1)$$

$S_2 = \alpha T_1 + (1-\alpha) S_1$

$S_3 = \alpha T_2 + (1-\alpha) S_2 = \alpha T_2 + (1-\alpha)\alpha T_1 + (1-\alpha)^2 S_1$

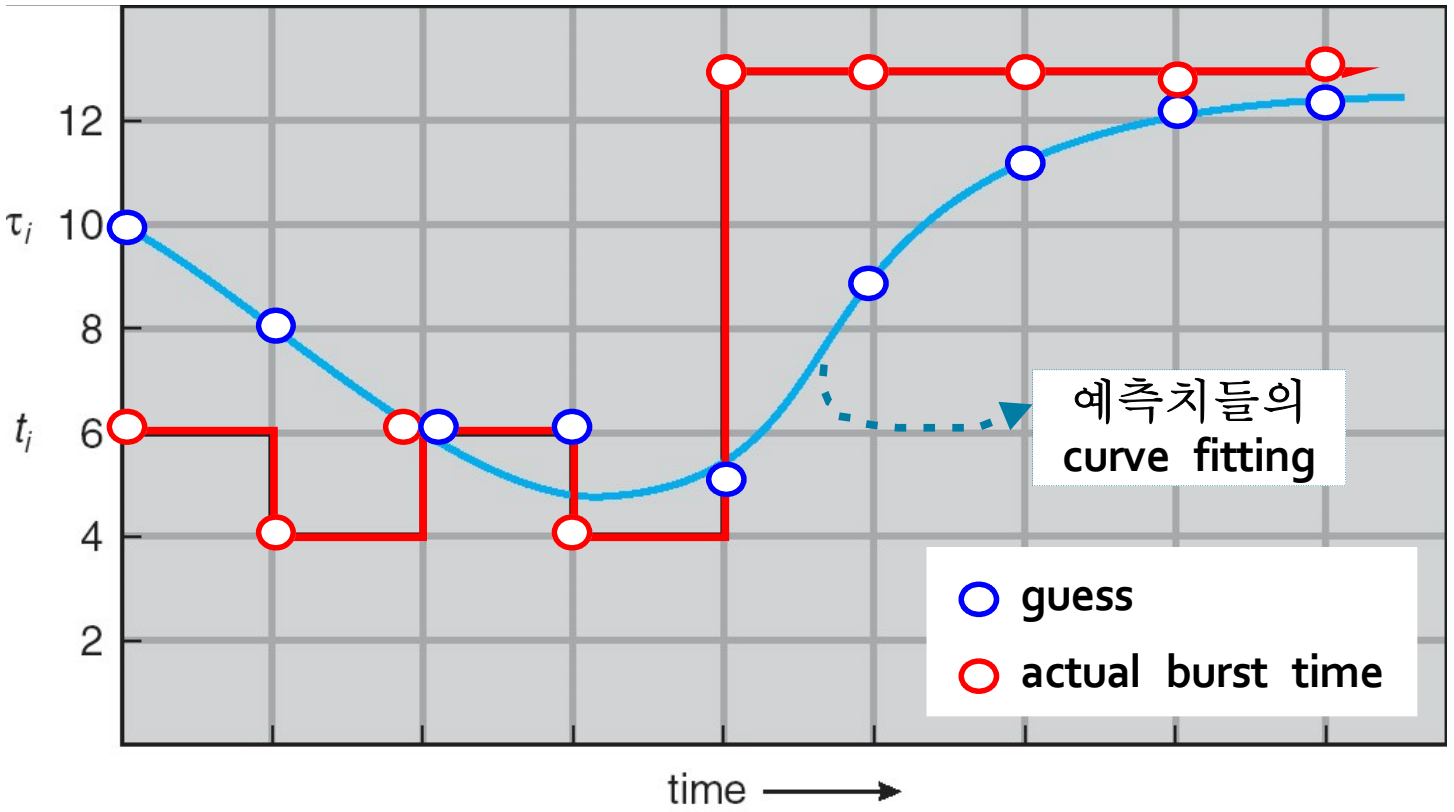
$S_4 = \alpha T_3 + (1-\alpha) S_3 = \alpha T_3 + (1-\alpha)\alpha T_2 + (1-\alpha)^2 \alpha T_1 + (1-\alpha)^3 S_1$

...

$$S_{n+1} = \alpha T_n + \ldots + (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_1$$

⇨ 과거 CPU burst time 일수록 가중치가 낮음.

(예) α = ½일 때 next CPU burst 길이 예측: $\tau_{n+1} = \frac{1}{2}t_n + \frac{1}{2}\tau_n$



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

(5) 다단계 큐 스케줄링 Multi-level Queue Scheduling

- 다단계 대기 큐; 각 큐는 서로 다른 **Priority**와 **Scheduling Algorithm**을 가짐.
- **admit** 時, 프로세스 속성(유형)에 따라 특정 큐에 영구 배정함; **Fixed priority**.

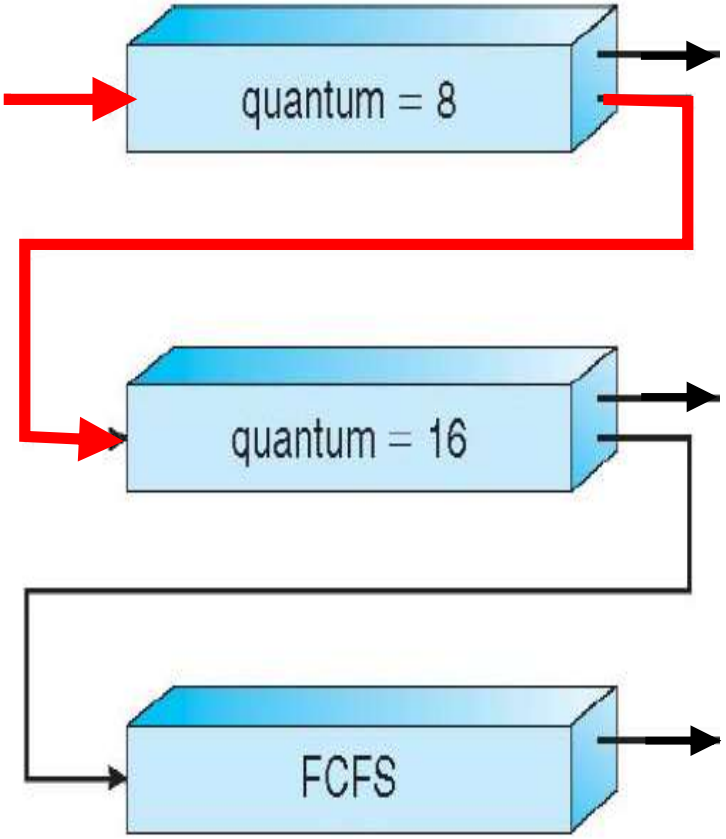
- 큐 間 스케줄링 **(Preemptive scheduling)**
  - 한 레벨의 큐는 자신의 모든 상위 큐가 **empty**일 때만 스케줄링 됨.
  - 실행중인 하위 큐의 프로세스는 상위 큐에 프로세스가 도착하면 **preempted.**

(6) 다단계 피드백 큐 스케줄링 Multi-level Feedback Queue Scheduling (MFQ)

- 다단계 대기 큐, 큐 間 스케줄링은
  다단계 큐 스케줄링과 同一함.

- **admit** 時 **process**는 최우선 큐에 배정되고,
  이 후 자신의 **CPU burst** 특성에 따라
  하위 큐로 이동함; **Dynamic Priority**.

- 설계 時 고려 사항:
  - 큐의 개수, 각 큐의 스케줄링 방법
  - **process**의 최초 큐 배정 방법
  - 프로세스의 큐 間 이동 방법
    - 하위 큐로 이동
    - 상위 큐로 이동 **(Aging-priority** 변경)



4. 다중 처리기 스케줄링

❑ 두 가지 스케줄링 방법

(1) 하나의 처리기(*Master Server*)가 스케줄링, 입출력, 시스템 행위를 수행하고, 나머지 처리기들은 **User Code**만 실행함. Asymmetric **Multiprocessing**.

(2) 각 처리기는 독자적으로 스케줄링을 수행함. **Symmetric Multiprocessing**.

- **Global Ready Queue) A single system-wide ready queue**
  - 문제점) 한 프로세스를 여러 번 선택; 프로세스 유실 **(Lost process)**.
- **Private Ready Queue) Per-processor ready queue**
  - 문제점) 처리기 間 부하 불균형. **(note) Process Migration**.
- 거의 모든 운영체제에서 사용: **Windows, Solaris, Linux, Mac.**

❑ SMP(Symmetry Multi Processor)에서의 이슈(Issues)들

(1) Processor Affinity 처리기 친화성

Affinity - 동족 관계, 유사성, 맞는 성질, 좋아함, 친화력, 유연(類緣)

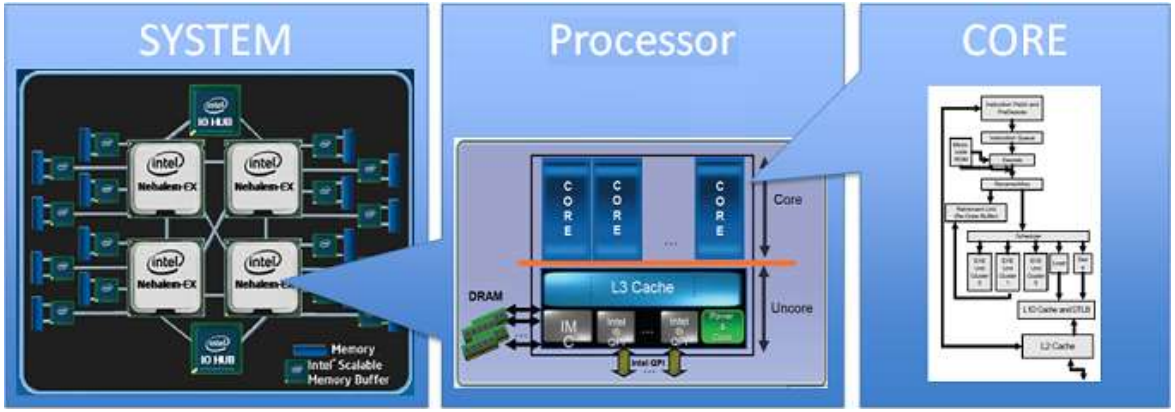
- 하나의 프로세스가 하나의 처리기에서 실행되도록 시도하는 것:
  - 프로세스 이주를 불허함 (Hard affinity 강한 친화성)
  - 노력하지만 보장 못함 (Soft affinity 약한 친화성)

(2) Load Balancing 부화 균등화

Private ready queue 방식에서 처리기 間 부하 균등을 시도하는 것:

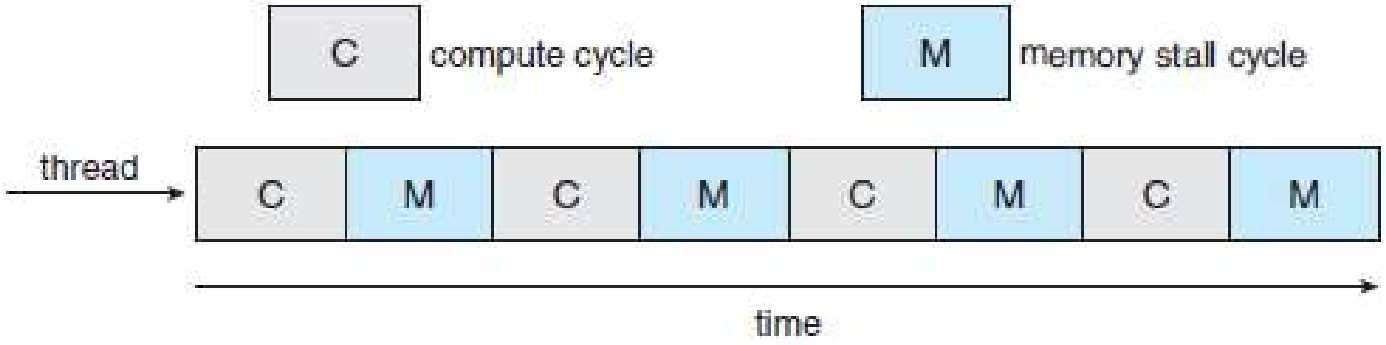
- Push migration) 특정 process가 주기적으로 모든 처리기의 부하를 검사하고, 과부하 처리기의 프로세스들을 부하가 적은 처리기로 보냄.
- Pull migration) idle 처리기가 busy 처리기의 프로세스를 자신에게 가져옴.
  (예) Linux: push 이주(200㎳ 마다) + pull 이주 (자신의 큐가 empty일 때)
- 처리기 친화성과 상충됨
  - 항상 pull (balancing 우선) vs. (불균형>임계치) 경우 이주 (affinity 우선)

(3) Multi-core Multiprocessor에서의 스케줄링



(note) **Multicore Processor**에서의 **Memory Stall** Stall - 마구간, 외양간, 지저분한 방, 엔진 정지, 실속(失速)

여러 **core**가 하나의 **memory**를 사용(접근) 하므로, **core**가 메모리 접근 時 오랫동안 기다리는 상황(*memory stall*, 메모리 멈춤) 발생.
core(CPU) time이 낭비됨.

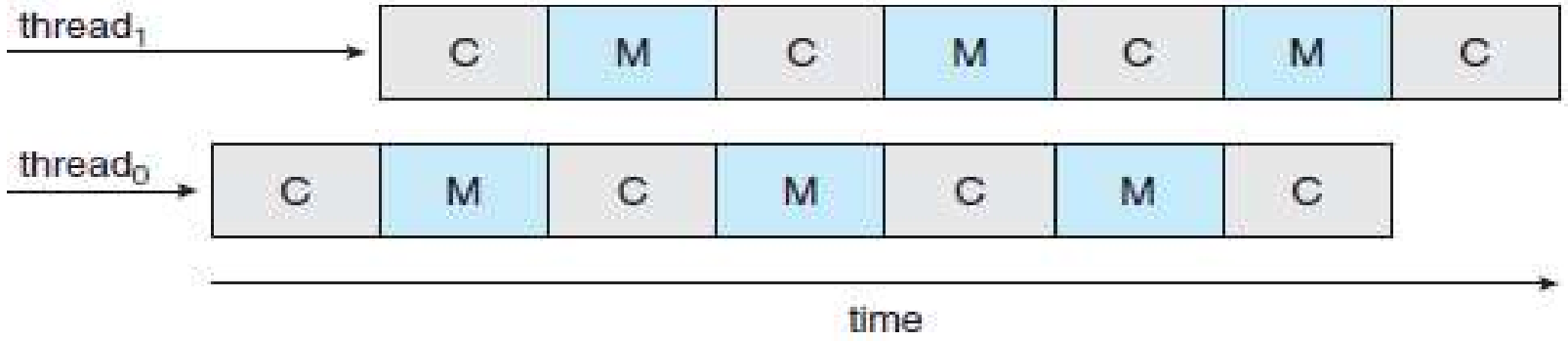


(note) **Multi-threaded Multi-core system**

- 하나의 **core**가 여러 **H/W thread**를 가지며,
  하나의 **H/W thread**는 하나의 **S/W thread**를 실행함. (다음 page 참고)

- **Interleaved multi-threading** or File-grained multi-threading)
  하나의 **core**가 여러 **S/W thread**를 번갈아 실행함-**time sharing**:

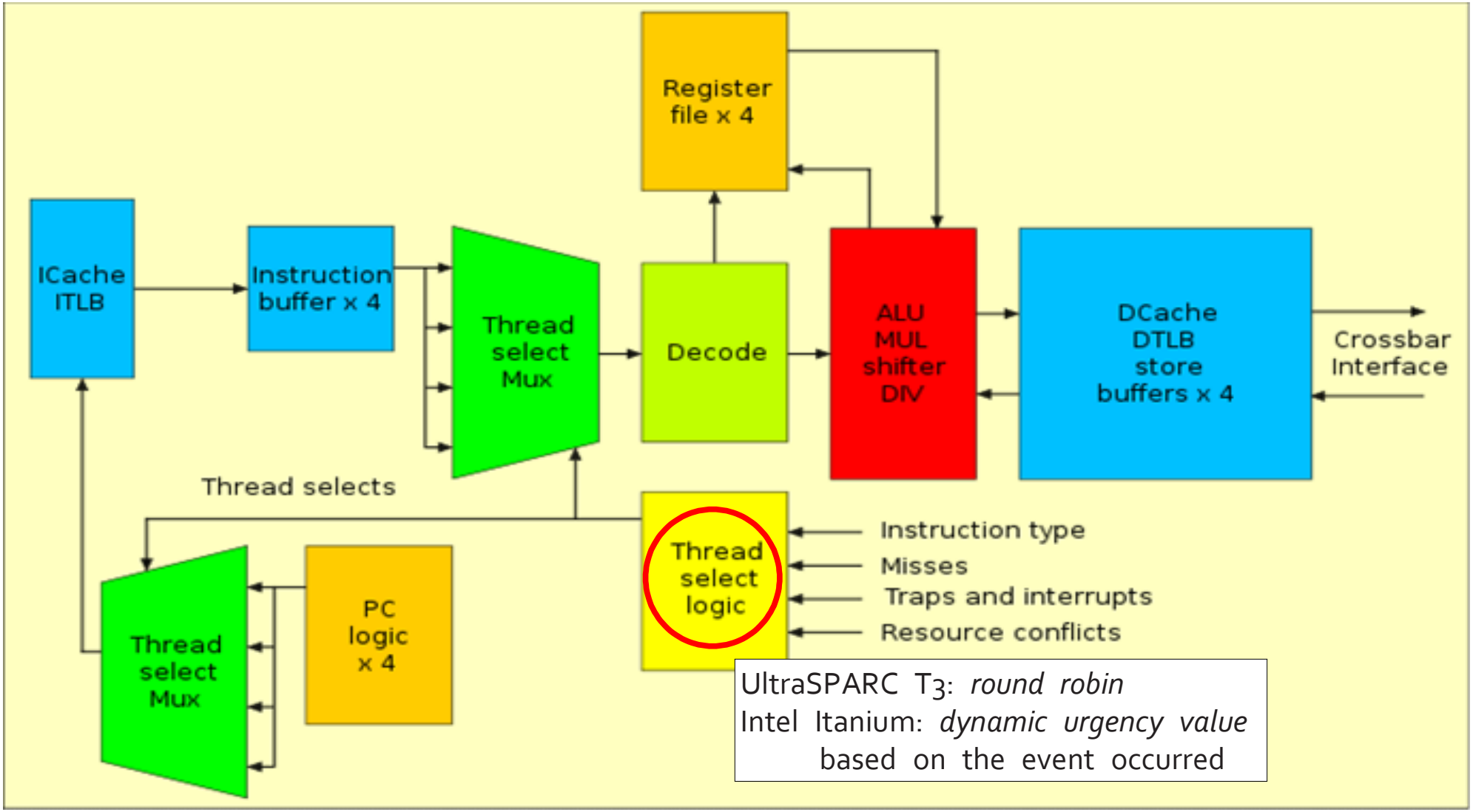
  (예) **thread$_0$**이 메모리 접근時, **thread$_1$**이 실행됨, **vice versa.**



  (note) 다중 코어 처리기에서의 "메모리 멈춤" 문제가 해결됨.

- 다중스레드 다중코어 시스템에서의 스케줄링:
  (단계 1) **Scheduler**가 각 **H/W thread** 상에서 실행될 **S/W thread**를 선정함.
  (단계 2) **Core**(*thread selection logic*)가 다음 실행할 **H/W thread**를 결정함.

UltraSPARC T1 Processor Core (4 H/W threads/core)



(note) What are the difference between Process, Thread and Task?

Process:

A program in execution is known as 'process'. A program can have any number of processes. Every process has its own address space.

Thread:

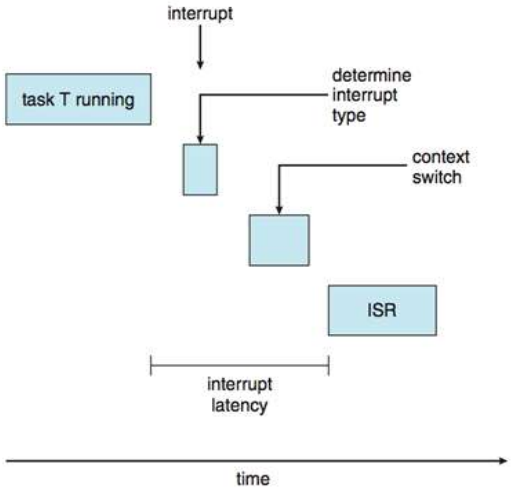
Threads uses address spaces of the process. The difference between a thread and a process is, when the CPU switches from one process to another, the current information needs to be saved in Process Descriptor(Process Control Block) and load the information of a new process. Switching from one thread to another is simple.

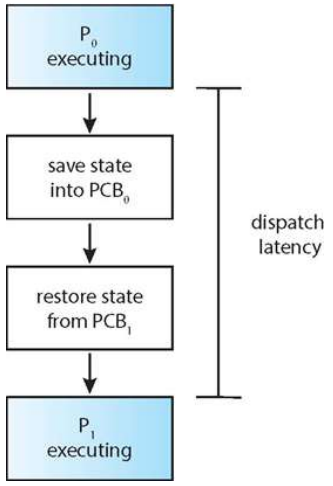
Task:

A task is simply a set of instructions loaded into the memory. Threads can split themselves into two or more simultaneously running tasks.

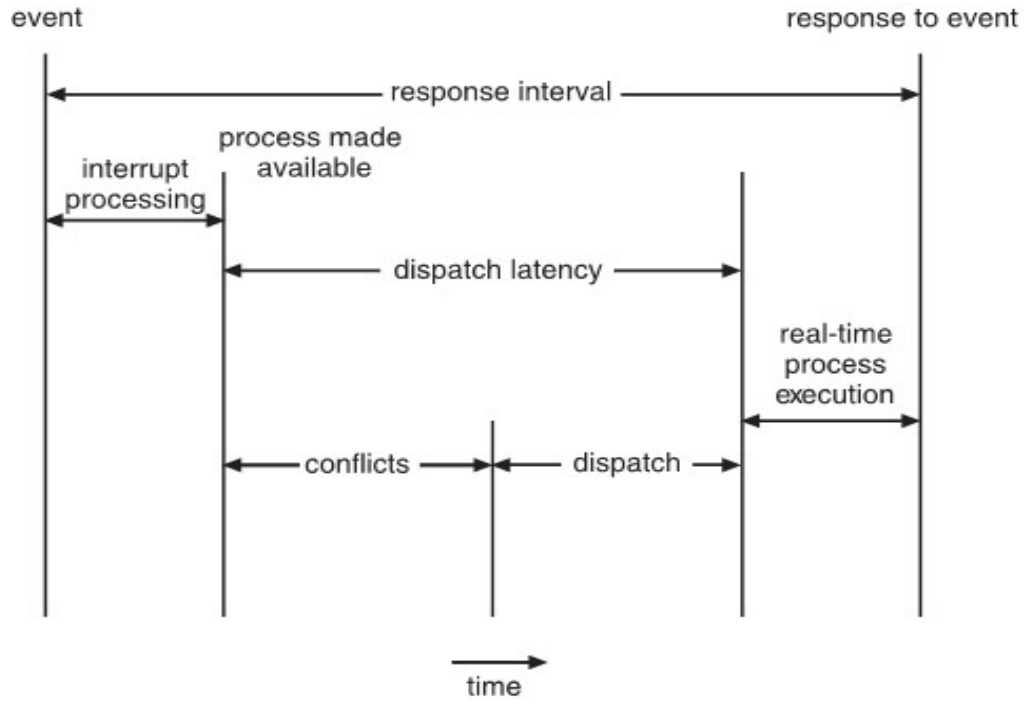
5. 실시간 CPU 스케줄링

- 실시간 시스템 구분
  - Soft real-time systems – no guarantee as to when critical real-time process will be scheduled
  - Hard real-time systems – task must be serviced by its deadline
- 지연시간 최소화
  - 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.





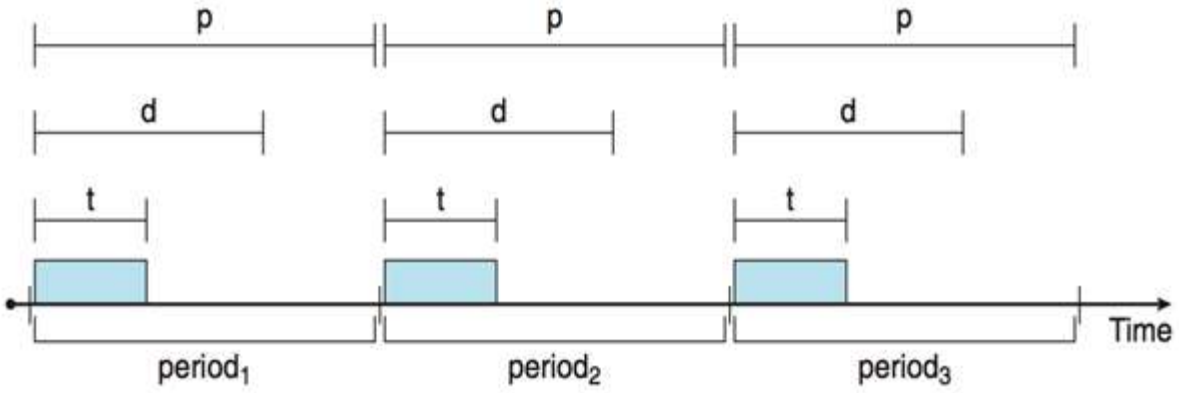
- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



❑ 우선순위 기반의 스케줄링(Priority Based Scheduling)

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling. But only guarantees soft real-time.

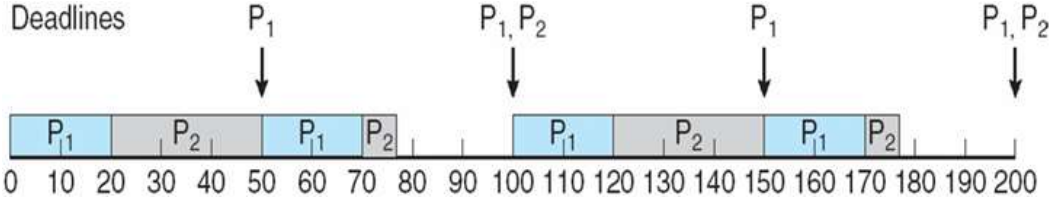
- For hard real-time must also provide ability to meet deadlines.
- Processes have new characteristics: periodic ones require CPU at constant intervals
  - * Has processing time *t*, deadline *d*, period *p*
  - * $0 \leq t \leq d \leq p$
  - * Rate of periodic task is $1/p$



❑ Rate Monotonic 스케줄링

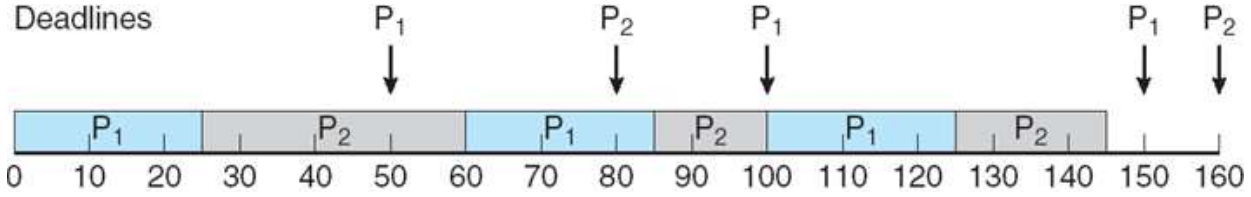
- A priority is assigned based on the inverse of its period. Shorter periods = higher priority, Longer periods = lower priority

P_1 is assigned a higher priority than P_2.



❑ Earliest Deadline First(EDF) 스케줄링

- Priorities are assigned according to deadlines. the earlier the deadline, the higher the priority, the later the deadline, the lower the priority.



❑ 일정 비율의 몫(Proportionate Share) 스케줄링

- T shares are allocated among all processes in the system.
- An application receives N shares where $N < T$.
- This ensures each application will receive N / T of the total processor time

❑ POSIX 실시간 스케줄링

- POSIX.1b standard. API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority.
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority.

6. 운영체제 사례

❑ Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling 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 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
  * 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.
  * Rather than quantum based on fixed time allotments, based on proportion of CPU time.
  * 2 scheduling classes included, others can be added - default, real-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.

❑ Windows Scheduling

※ Priority class 別 Ready Queue

- 최상위 우선순위를 가진 스레드 선택.
- Real time threads can preempt non-real time
- 스레드는 다음이 발생할 때까지 실행됨:
  - Blocked
  - Preempted at time quantum
  - Preempted by a higher-priority thread
- 32 level priority scheme

32 가지 우선순위		Priority Class					
		Real-time	가변 클래스: timeout時 우선순위 ↓, wakeup時 ↑				
			High	Above normal	Normal (default)	Below normal	Idle
상대적 우선순위	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal (default)	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

- Variable class is 1-15, real-time class is 16-31
- Priority 0 is memory-management thread
- If no run-able thread, runs idle thread
- 우선순위 변경
  - timeout 時 강등. 단, 자신의 base priority 보다 낮아지지는 않음.
  - wakeup 時 승격.

❑ Solaris

- **Priority-based, Preemptive thread scheduling**

- 6 가지 스케줄링 유형(Scheduling Class)

Real Time (RT)	최상위 우선순위, 고정 우선순위, 고정 time quantum
System (SYS)	kernel thread, 고정 우선순위, no time quantum
Time Sharing (TS) *(default)*	**Multi-level feedback queue** 동적 우선순위: time quantum 완전 사용⬇, sleep에서 복귀⬆
Interactive (IA)	**Multi-level feedback queue** 동적 우선순위: windowing application 우선순위⬆
Fair Share (FSS)	프로세스 집합에게 CPU 공유량*CPU share*을 배정하고, process들에게 공평하게(동등하게) CPU time을 배정함.
Fixed Priority (FP)	고정 우선순위

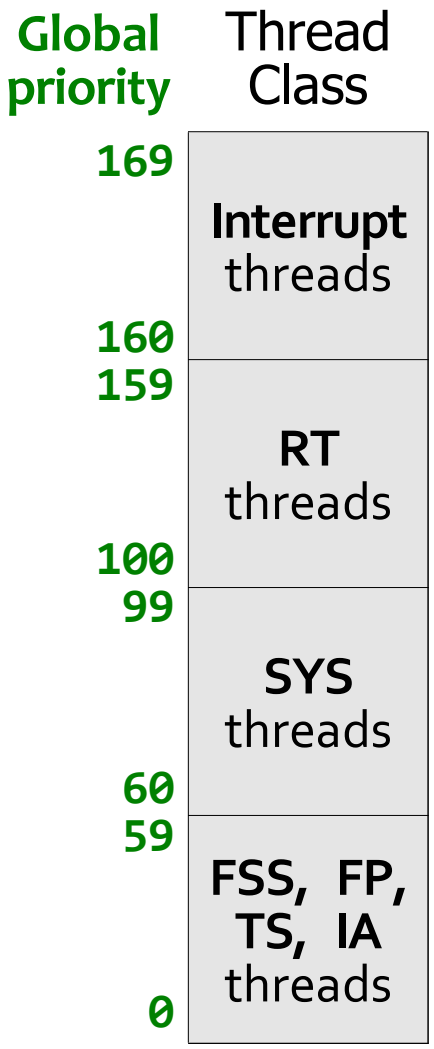
- **Solaris Scheduler**

- class priority를 *global priority*로 변환

- 최상위 우선순위를 가진 스레드 선택

- 스레드는 다음이 발생할 때까지 실행됨:
  - **Blocked**
  - **Preempted at time quantum**
  - **Preempted by a higher-priority thread**

- 우선순위가 동일한 스레드들은
  round-robin queue 사용



❑ Java Thread Priority

- **10 priorities:** Thread.MIN_PRIORITY(1) ~ Thread.MAX_PRIORITY(10)
  - **Priority 0**는 **JVM**이 생성한 스레드에게만 부여됨.

- 스레드의 우선순위 부여:
  - 최초로 **default priority** Thread.NORM_PRIORITY(5)를 부여함.
  - 프로그램에서 명시적으로 변경하지 않는 한 우선순위 변경되지 않는다.
  - 새로 생성되는 스레드 **parent thread**의 우선순위를 물려받음.

  (note) JVM은 **host OS** 상에서 구현되므로
  Java thread의 우선순위는 매핑된 **kernel thread**와 연관됨.

- 명시적 우선순위 변경 **using** Thread.currentThread().setPriority(1≦p≦10)

  Thread.currentThread().setPriority(Thread.NORM_PRIORITY + 3);