Chapter 5: CPU Scheduling

2020.5
Howon Kim

- 정보보호 및 지능형 IoT연구실 – http://infosec.pusan.ac.kr
- 부산대 지능형융합보안대학원 – http://aisec.pusan.ac.kr

Chapter 5: CPU Scheduling

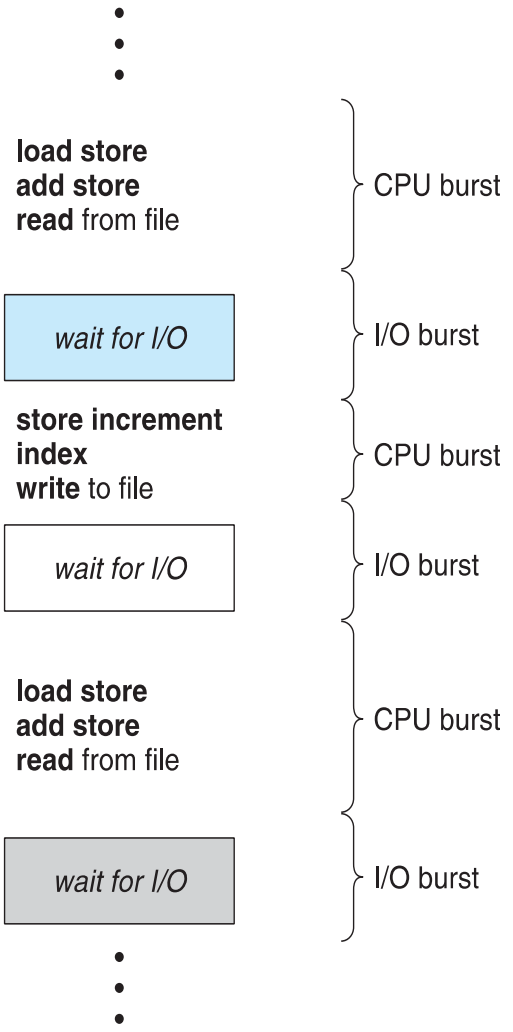
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

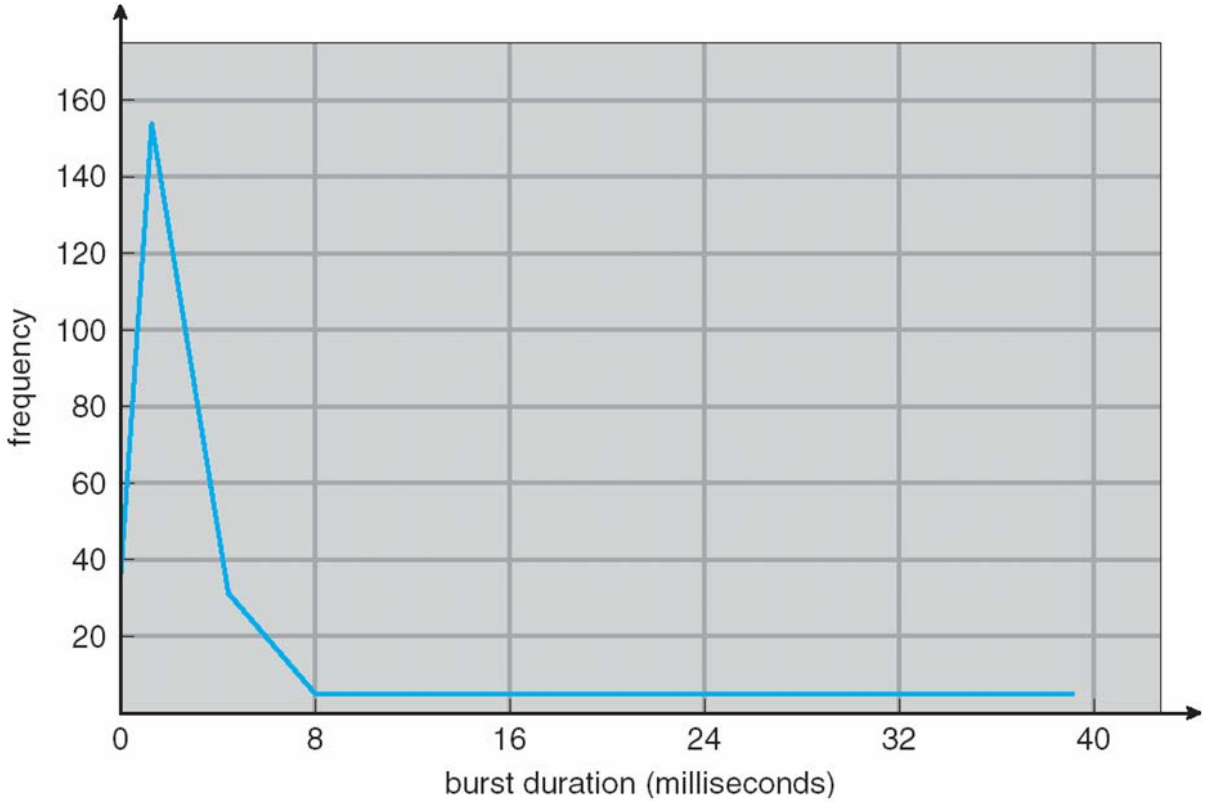
Basic Concepts

- **Maximum CPU utilization** obtained with multiprogramming
- **CPU–I/O Burst Cycle** – Process execution consists of a cycle of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- **CPU burst distribution is of main concern**



< CPU와 입출력 Burst 교차 >

Histogram of CPU-burst Times



CPU burst 지속 시간 측정 사례 → 짧은 CPU burst가 많고 긴 CPU burst는 매우 적음

CPU Scheduler

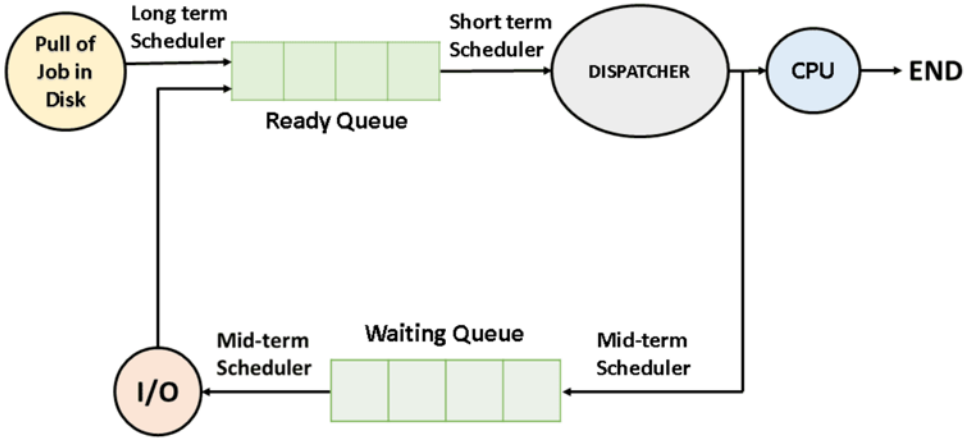
- **Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them**
  - Queue may be ordered in various ways
- **CPU scheduling decisions may take place when a process…**
  1. switches from running to waiting state (I/O 요청으로 인해, CPU 실행에서 waiting state으로 전환시..혹은 child process fork 후, child process가 종료되기를 기다리는 상황)
  2. switches from running to ready state (interrupt 발생. 실행중인 프로세스가 다음 실행 기다리는 형태로 전환되는 경우)
  3. switches from waiting to ready state (I/O 종료 후, 다시 CPU 실행 기다림)
  4. terminates
- **1과 4 상황에서만 scheduling이 발생할 경우, scheduling은 선택 여지 없음. 즉,실행을 위해선 new process가 반드시 선택되어야 함. 하지만, 2,3 상황에서는 선택의 여지가 있음.**
  - 1과 4의 경우에서는 “기존 process가 종료되거나 I/O를 위해 wait 상태로 간 이후,” 새로운 process를 선택함 → scheduling 관점에서 보면 “기존 process 실행이 종료된 후, 새 process 선택”. 즉, non-preemptive scheduling (혹은 cooperative scheduling)
  - 2와 3의 상황은 interrupt 발생 혹은 I/O 종료에 의해, 기존 process 실행이 아닌 ready state에 있는 process 중에서 선택 실행을 의미함 → scheduling 관점에서 보면 preemptive scheduling임.

CPU Scheduler

- **Preemptive Scheduling 에서는 다음을 고려해야함**
  - Consider access to shared data → preemptive scheduling에선 다수의 프로세스가 data를 공유할 때, 경쟁조건 초래 가능하므로 주의 필요
  - Consider preemption while in kernel mode → 어떤 process 의 system call 처리위해 kernel mode 동작이 이뤄지는 데, 이 경우 preemption이 발생하면, kernel 자료 변경 발생. 이때문에 kernel mode 동작시 preemption은 특히 주의 필요
  - Consider interrupts occurring during crucial OS activities → 여러 process 가 interrupt를 겹쳐 호출하지 못하도록 정리 필요. 예를 들어, 한 process에 의한 interrupt 실행 시, 다른 proces의 interrupt가 병행처리 못되도록 하기 위해, 진입점에서 interrupt disable 시키고 출구에서 다시 interrupt enable 시킴

Dispatcher

- **Dispatcher : the module of OS that sets up the execution of the process selected by the scheduler**
- **Dispatcher 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**



< CPU scheduling 그림>

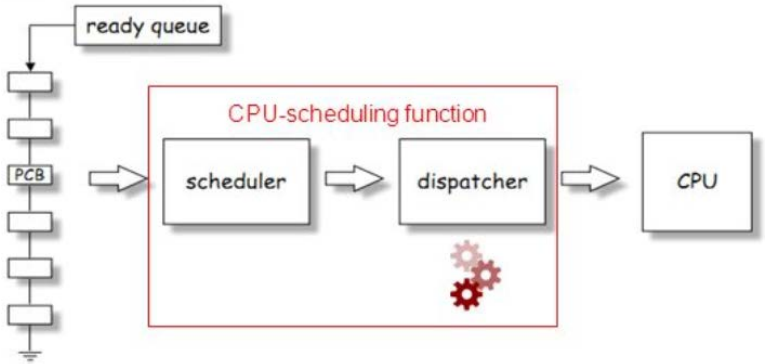


그림: Excerpt from Notesjam

Scheduling Criteria

- CPU scheduler는 다음과 같은 특성을 고려하여 process를 scheduling 함
- CPU utilization (이용율) – keep the CPU as busy as possible
- Throughput (처리량) – # of processes that complete their execution per time unit
- Turnaround time (총 처리시간) – amount of time to execute a particular process. That is, the amount of time taken to complete a request. It is defined as the total time taken between the finishing time and start time. --- The sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- Waiting time (대기 시간) – amount of time a process has been waiting in the ready queue
- Response time (응답 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 Algorithm Optimization Criteria

- Maximize CPU utilization
- Maximize throughput

- Minimize turnaround time
- Minimize waiting time
- Minimize response time

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

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

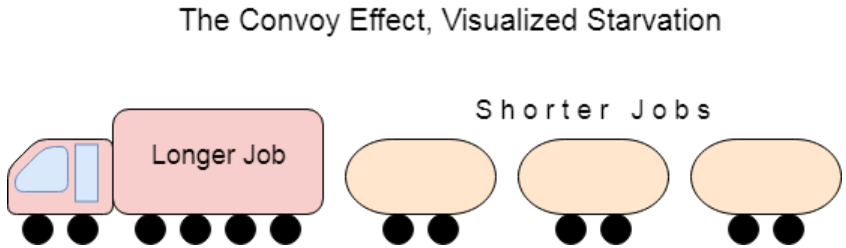
FCFS Scheduling (Cont.)

Suppose that the processes arrive in this new order:

$$P_2 , P_3 , P_1$$

- **The Gantt chart for the schedule is:**

- **Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$**
- **Average waiting time: (6 + 0 + 3)/3 = 3**
- **Much better than previous case**



- **Convoy effect - short process behind long process**
  - FCFS may suffer from the convoy effect if the burst time of the first job is the highest among all. As in the real life, if a convoy is passing through the road then the other persons may get blocked until it passes completely.
  - If the CPU gets the processes of the higher burst time at the front end of the ready queue then the processes of lower burst time may get blocked which means they may never get the CPU if the job in the execution has a very high burst time. This is called convoy effect or starvation.

https://www.javatpoint.com

Shortest-Job-First (SJF) Scheduling

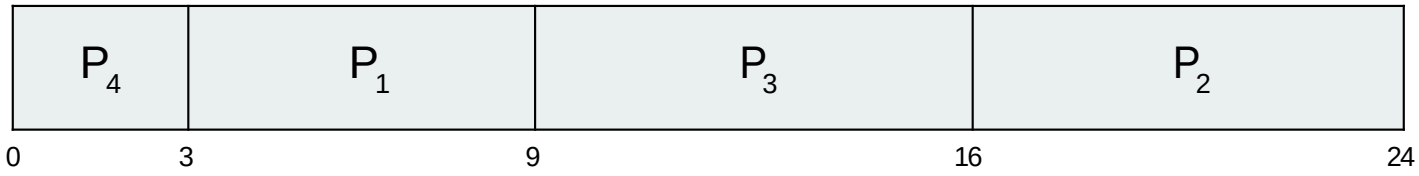
- **Associate with each process the length of its next CPU burst**
  - Use these lengths to schedule the process with the shortest time
  - 즉, CPU burst 시간(사용시간)이 짧은 process를 먼저 scheduling 함
    - Process 전체 길이가 아니라 CPU burst 길이를 고려함

- **SJF is optimal – gives minimum average waiting time for a given set of processes**
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user
  - short term scheduler는 process의 CPU burst time 정보를 알 수 없음 → 이에, 어떤 process의 이전 CPU burst time 고려하여 다음 burst time을 예측하여, 가장 짧은 CPU burst time을 갖는 process를 scheduling 함

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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

Determining Length of Next CPU Burst

- **Can only estimate the length – should be similar to the previous one**
  - Then pick process with shortest predicted next CPU burst
- **Can be done by using the length of previous CPU bursts, using exponential averaging(smoothing)**

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

if $\alpha = 0$, $\tau_{n+1} = \tau_n$, 즉, 다음번 CPU burst 예측값은 실제 CPU burst 길이에 영향없음

if $\alpha = 1$, $\tau_{n+1} = t_n$, 즉, 실제 CPU burst 길이에만 의존

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

예 Exponential Smoothing Forecaset with a = .3

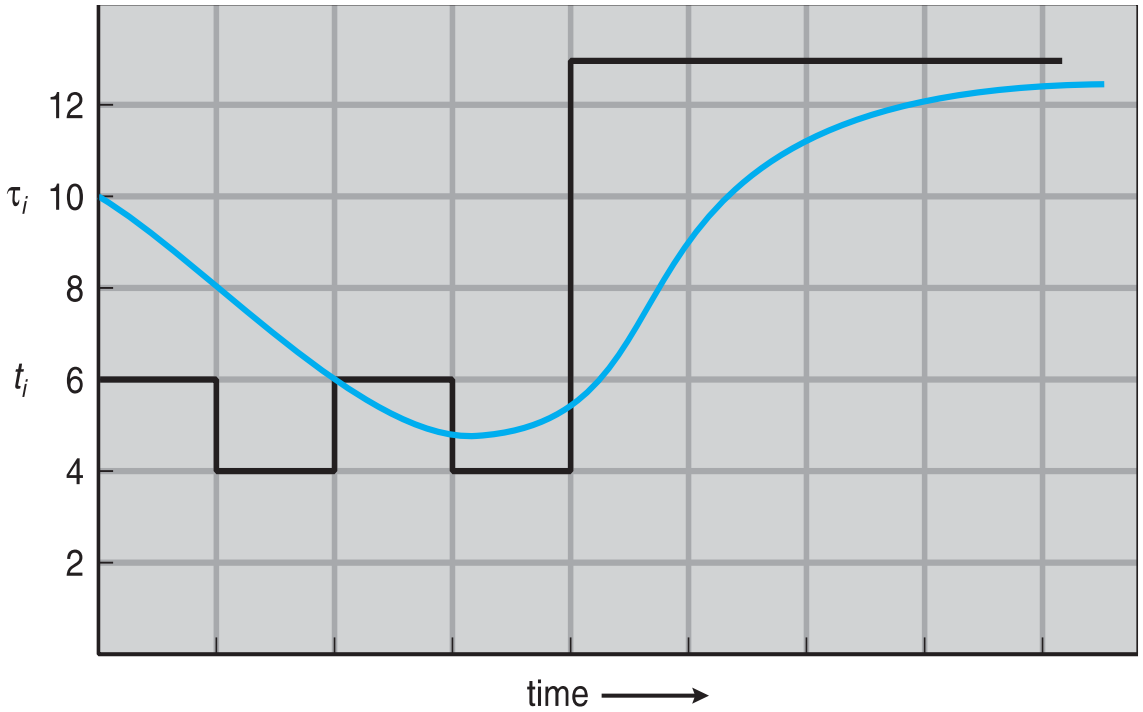
실제값 예측값

Period	Demand	Exponential Smoothing Forecast
1	12	10.00
2	15	10.60
3	11	11.92
4	9	11.64
5	10	10.85
6	8	10.60
7	14	9.82
8	12	11.07
9		11.35

F2 = .3*12 + .7*10
= 3.6 + 7
= 10.6

F3 = .3*15 + .7*10.6
= 11.92

Prediction of the Length of the Next CPU Burst



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

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- **If we expand the formula, we get:**

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

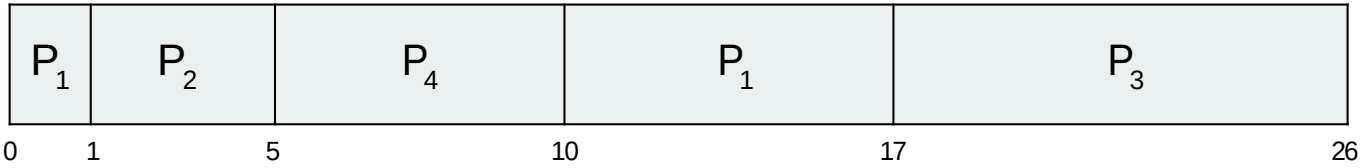
- **Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor**

Example of Shortest-remaining-time-first

- 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

Priority Scheduling

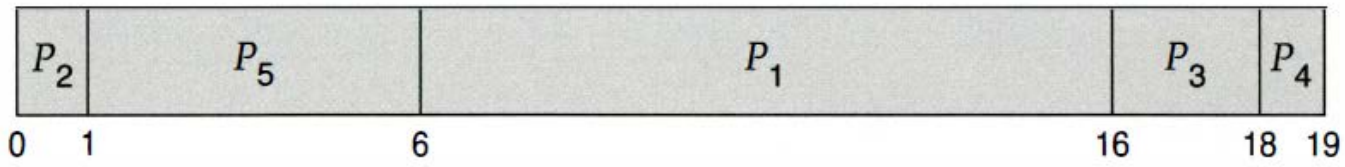
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡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

Example of Priority Scheduling

* Example !

Process	Burst Time	Priority
도착순서 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)

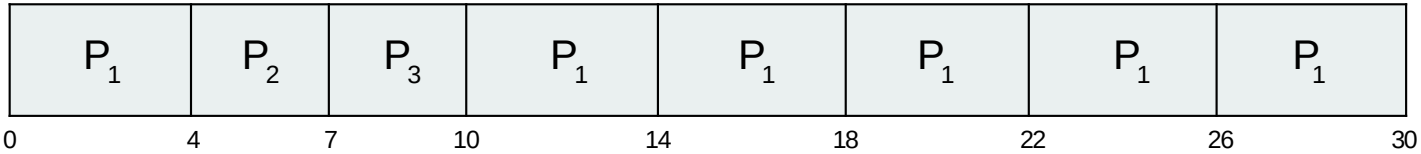
- **Round Robin scheduling 기법은 시분할 시스템을 위해 설계됨**
- **Each process gets a small unit of CPU time (time quantum *q*), usually 10~100 milliseconds.**
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- **If there are *n* processes in the ready queue and the 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**
  - *q* large ⇒ FIFO : 즉, process가 CPU상에서 실행되는 시간 q가 클 경우
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high : CPU burst 시간이 할당된 시간 q보다 크면, context switching 발생하여 실행중이던 process는 ready queue에 저장됨(향후에 다시 scheduling 됨) → 오버헤드 많음

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

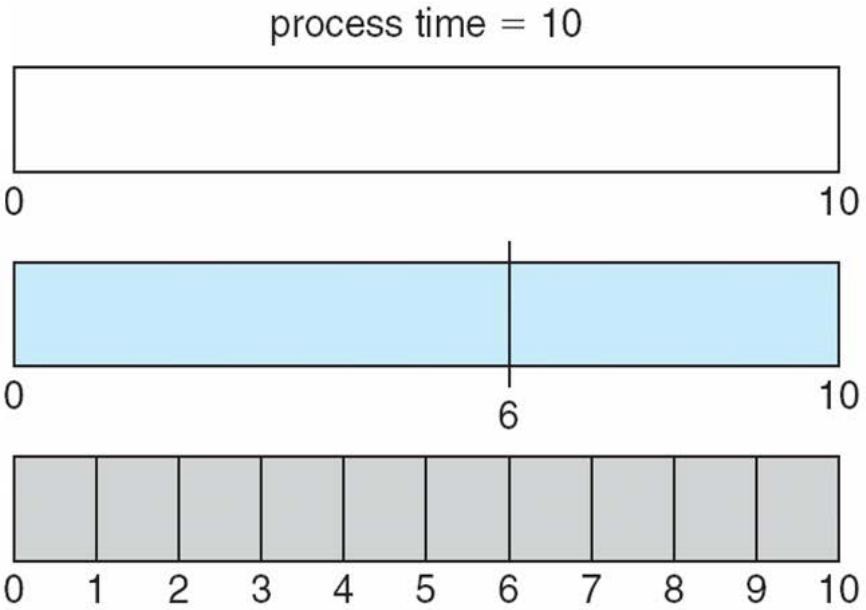
q(Time Quantum) = 4



- Typically, higher average turnaround than SJF, but better *response*
- q should be large compared to context switch time
  - q가 매우 크면, FIFO와 동일하게 됨
  - q가 매우 작으면, overhead 많아짐
- q usually 10ms to 100ms, context switch < 10 usec

• Turnaround time (총 처리시간)
• Response time (첫번째 응답시간)

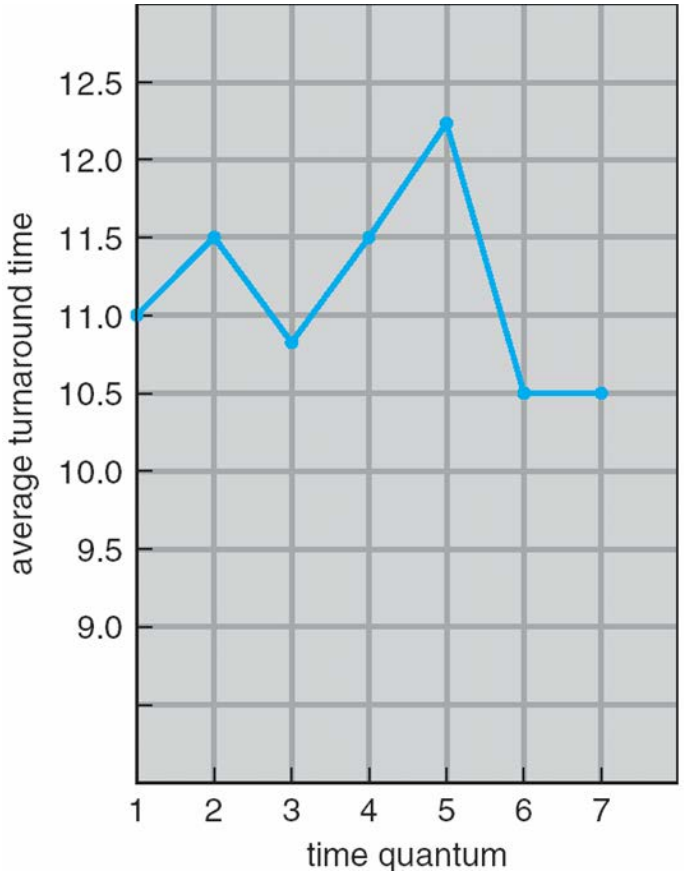
Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

quantum q 시간 줄일 수록
context switching이 많아짐

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

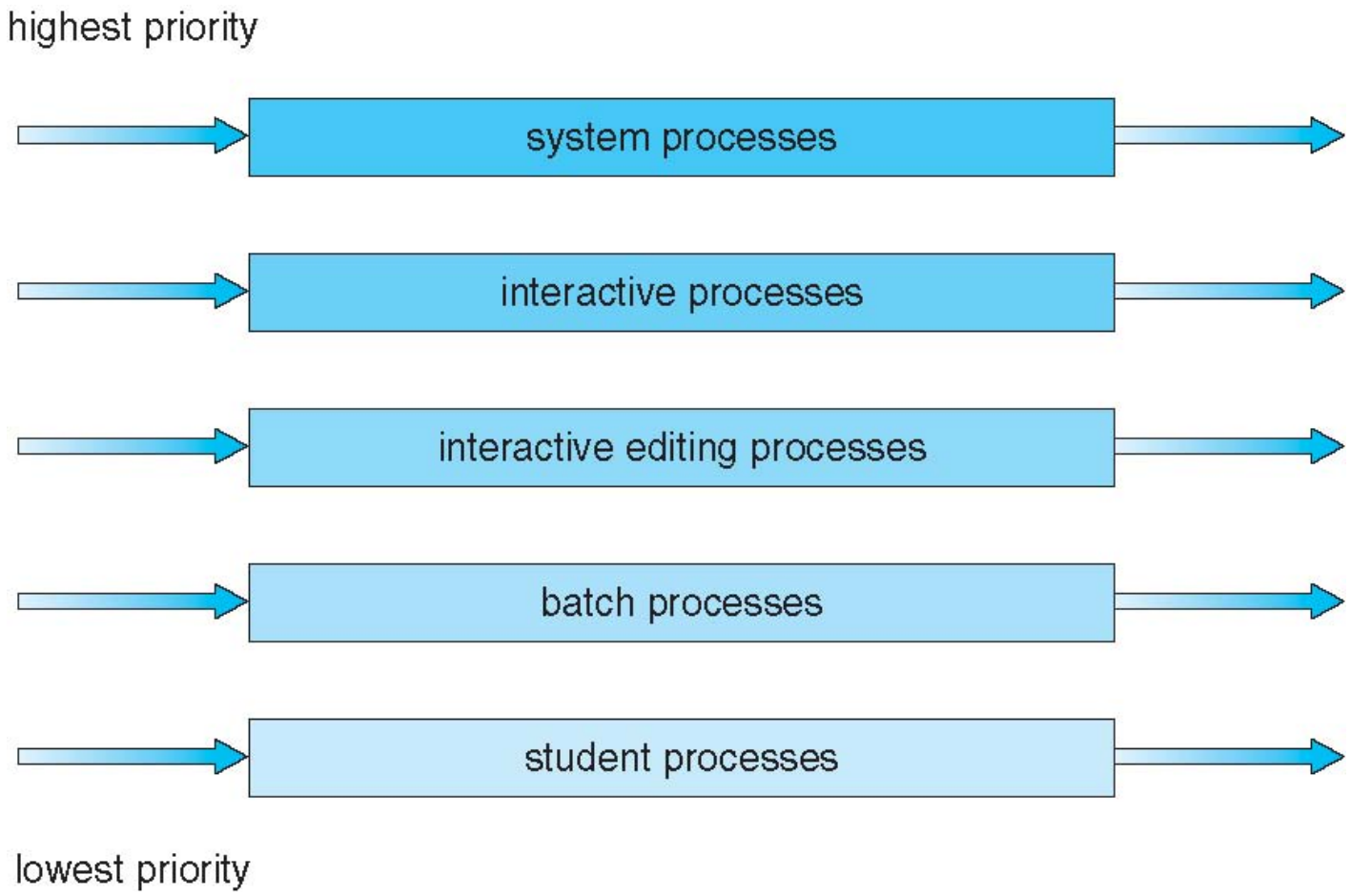
80% of CPU bursts should be shorter than q

한 process 집합의 평균 총 처리 시간(turnaround time)이 q에따라 변하는 것을 보여줌
- q를 키운다고해서 반드시 turn around time이 개선되는(줄어드는) 것은 아님. 너무 키우면 FIFO
- 프로세스들의 CPU burst time을 만족하는 q를 설정하면 되는데... ☺

Multilevel Queue

- **Ready queue is partitioned into separate queues, eg:**
  - **foreground** (interactive)
  - **background** (batch)
- **Processes are assigned to one queue based on some property of the process – 일반적으로 프로세스는 메모리 크기, 프로세스 우선순위 등, 특성에 따라 특정 queue에 할당됨**
- **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% (of CPU time) to foreground in RR
    - 20% of CPU time to background in FCFS

Multilevel Queue Scheduling



예1) batch process queue의 process는 그 상위 queue가 비워있지 않으면 실행 안됨.
하위 우선순위 queue상의 process 수행 중, 더 높은 우선순위 갖는 queue에 process 오면, 그 상위 우선순위가 선점함

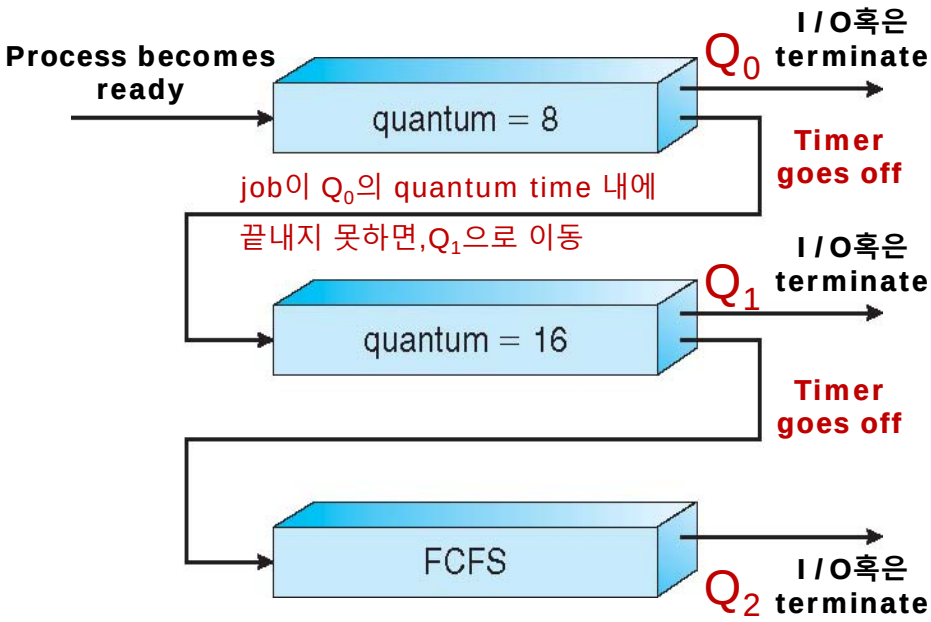
예2) queue간 CPU time 할당해서 사용: 상위 foreground queue는 80% CPU time, background queue는 20% CPU time 등

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



Strict priority among queues, Q1 > Q2 >Q3

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
    - 즉, 동일한 process에 속하는 thread 간 경쟁을 하므로 **"프로세스 경쟁 범위"**라고 함
  - Typically done via priority set by programmer
  - user level thread가 LWP상에서 scheduling 된다는 것은 thread 가 실제 CPU 상에서 **실행중**이라는 것을 의미하지 않음
- **Kernel thread scheduled onto available CPU is system-contention scope (SCS), "시스템 경쟁 범위" – competition among all threads in system.**

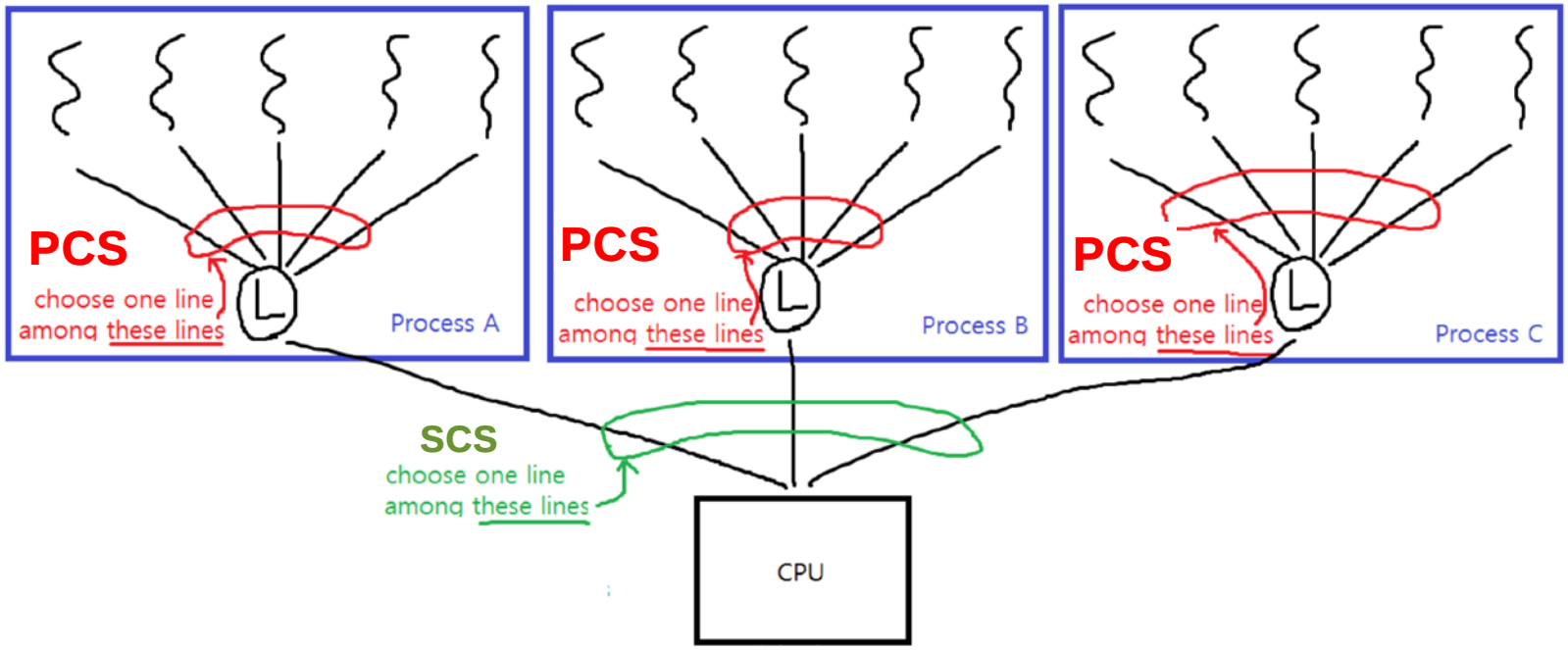
PCS and SCS

- Thread-library schedules user-level-threads on LWP(Lightweight Process).

This is Process-Contention Scope(PCS) because contention occurs between threads belonging to the same process.

OS schedules kernel-threads on CPU.

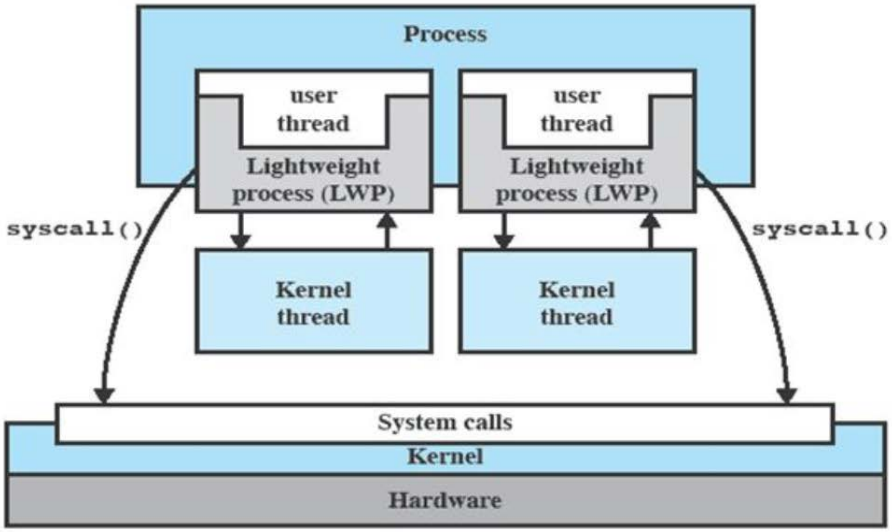
This is System-Contention Scope(SCS) because contention occurs between all threads belonging to the system.



https://softwareengineering.stackexchange.com/questions/301119/pthread-scheduling-system-and-process

참고) LWP(Light weight process)

- **LWP: Lightweight process**
  - A lightweight process (LWP) can be viewed as a mapping between ULTs(User Level Threads) and kernel threads.
  - Each LWP supports ULT and maps to one kernel thread.
  - LWPs are scheduled by the kernel independently, and may execute in parallel on multiprocessors.



Process	• includes the user's address space, stack, and process control block
User-level Threads	• a user-created unit of execution within a process
Lightweight Processes (LWP)	• a mapping between ULTs and kernel threads
Kernel Threads	• fundamental entities that can be scheduled and dispatched to run on one of the system processors

< Process & Thread in Solaris OS >

ref) OS internals & Design by William Stallings

Pthread Scheduling

- **API allows specifying either PCS or SCS during thread creation**
  - PTHREAD_SCOPE_**PROCESS** schedules threads using **PCS scheduling**
  - PTHREAD_SCOPE_**SYSTEM** schedules threads using **SCS scheduling**
- **Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM**

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

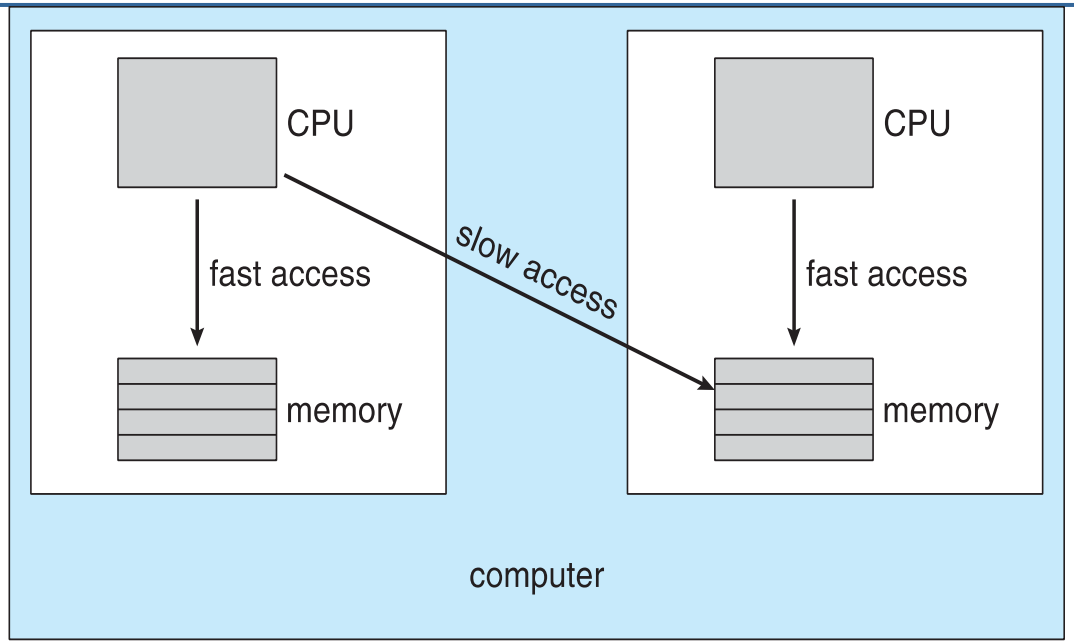
Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Multiple-Processor Scheduling

- **CPU scheduling more complex when multiple CPUs are available**
- **Homogeneous processors within a multiprocessor**
  - 기능이 동일한 경우
- **Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing**
- **Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes**
  - Currently, most common
- **Processor affinity(프로세서 친화성)– process has affinity for processor on which it is currently running (특정 프로세서에서 동작하던 process를 다른 프로세서로 이동시, 관련 데이터 캐쉬를 모두 이동해야 함 → 높은 cost → 이에, 어떤 process는 현재 실행중인 프로세서에 대한 친화성 affinity 가짐)**
  - **soft affinity – OS가 동일한 처리기에서 process 실행시키려고 노력은 하지만, 보장하지는 않음**
  - **hard affinity – 시스템 호출 단계에서 이미 프로세스는 자신이 실행될 처리기 집합을 명시함**
  - Variations including **processor sets**

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

위와 같은 Non-uniform Memory Access(NUMA)인 경우, 자신의 main memory access 빠름 → 이에, 해당 process상의 데이터 memory placement algorithm은 이를 고려하여, 해당 process가 동작되는 것과 가까운 memory에 배치됨

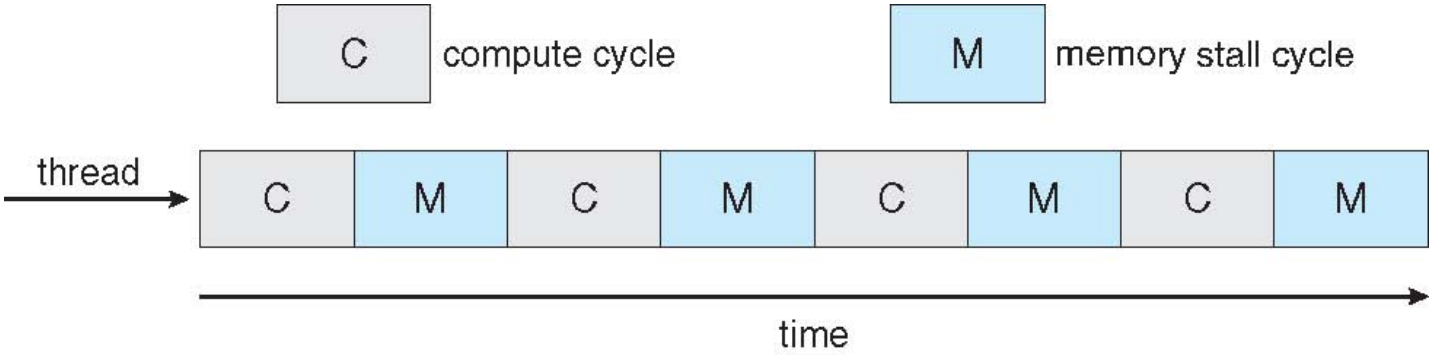
Multiple-Processor Scheduling – Load Balancing

- **If SMP, need to keep all CPUs loaded for efficiency**
- **Load balancing attempts to keep workload evenly distributed**
  - Load balancing(부하 균등화)를 위해 사용되는 두 가지 방식
  - Push migration – periodic task checks load on each processor, and if found pushes task from overloaded processor to other processors. (A special task가 모니터링 함)
  - Pull migration – idle processors pulls waiting task from busy processor (processor가 함)

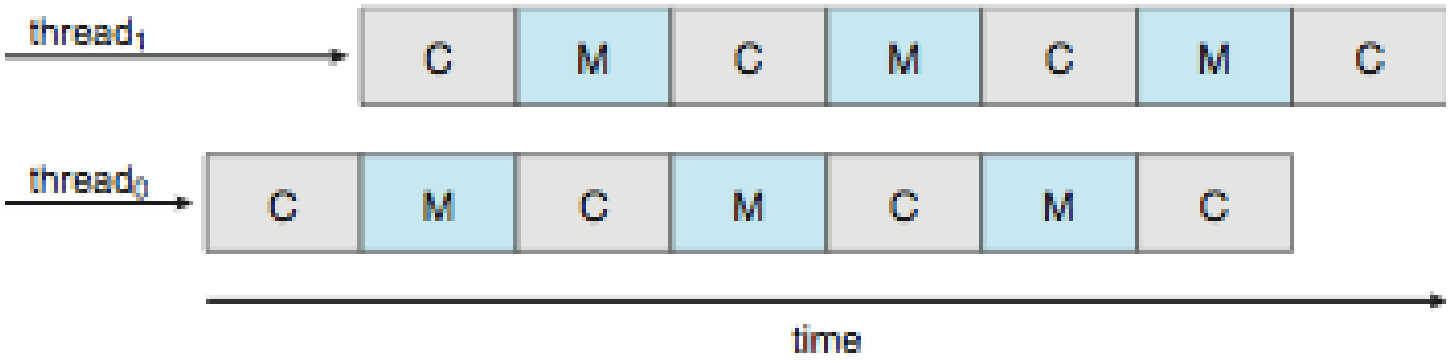
Multicore Processors

- **Recent trend to place multiple processor cores on same physical chip**
- **Faster and consumes less power**
- **Multiple threads per core may complicate scheduling issues:**
  - 다중 코어 프로세서 상에서, 특정 프로세스가 memory 접근시, available한 데이터를 얻을 때까지 많은 시간 소요됨 → memory stall 발생
    - Memory stall은 cache miss에 의해 발생하는 경우가 많음
  - Takes advantage of memory stall to make progress on another thread while memory retrieve(stall) happens
    - 즉, 여러 thread를 core에 할당한 후,
    - 어떤 thread가 memory stall을 경험하는 경우(즉, memory 접근을 위해 기다림), core는 다른 thread를 실행함(즉, 다른 thread가 core에 scheduling 됨)

Multithreaded Multicore System



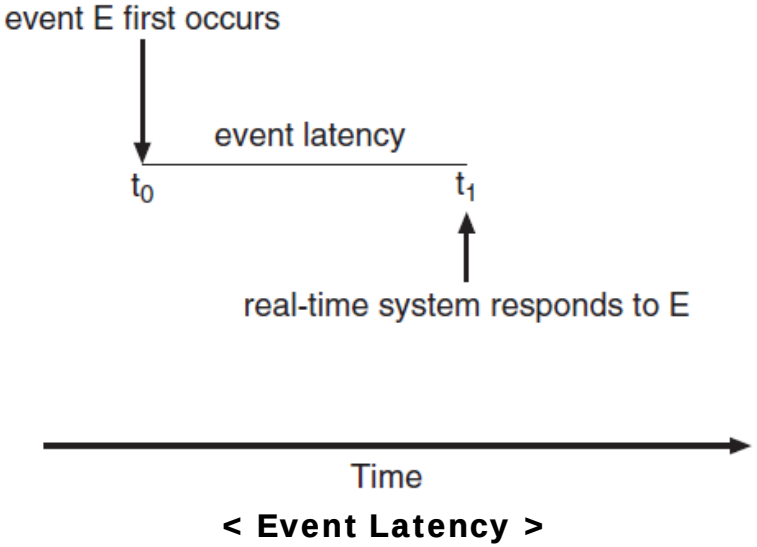
이 그림은 memory stall이 발생하고 있는 모습을 보여줌. 즉, 프로세서가 메모리상의 데이터가 가용해지기를 기다리고 있음. 예에서는 50% 정도의 시간이 낭비되고 있음



두개의 thread(thread1, thread0)를 번갈아가면서 수행하여, memory stall을 역으로 활용하여 두개의 thread를 동시에 수행함

Real-Time CPU Scheduling

- 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
- Event Latency – event가 발생해서 그 event에 대한 서비스가 수행될때 까지의 시간

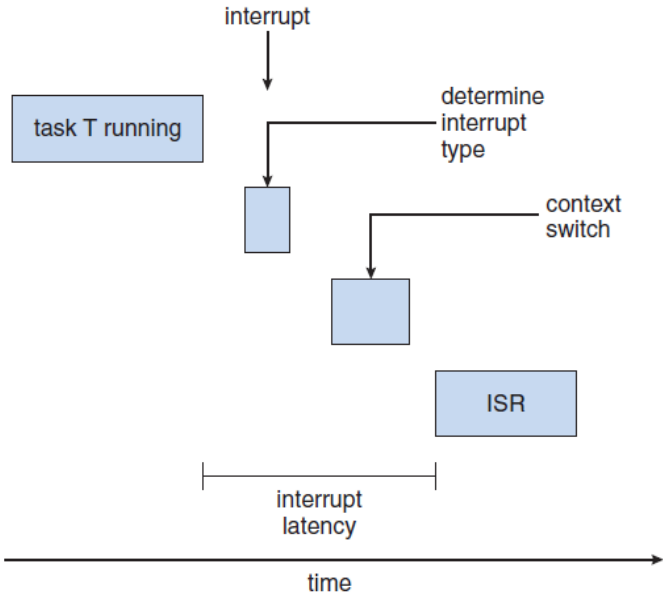


< Event Latency >

- 실시간 시스템의 성능에 영향을 주는 두 가지 주요 요소
  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

Interrupt Latency

- **Interrupt Latency** – the period of time from the arrival of an interrupt at the CPU to the start of the routine(ISR) that services the interrupt
  - When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred.
  - It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR).

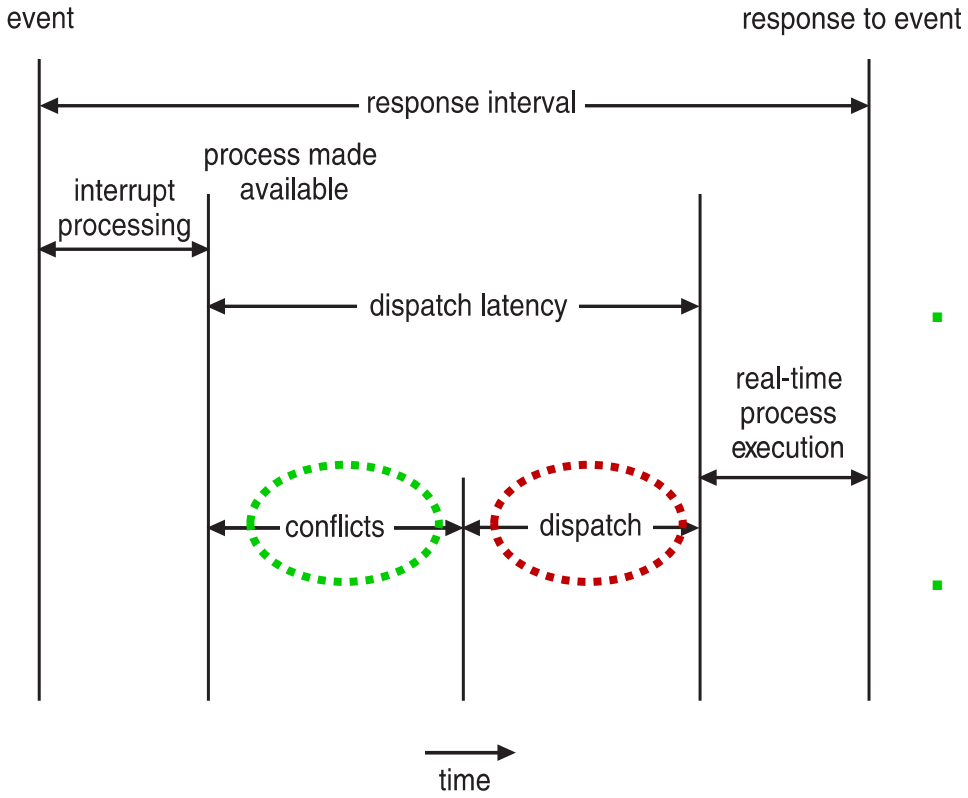


< Interrupt Latency >

- Interrupt Latency에 영향을 주는 요인 중 하나는 kernel data 구조가 update 되는 동안 interrupt를 disable 시키는 것
  - Real time OS에서는 interrupt가 disable 되는 시간이 매우 짧아야 함

Dispatch Latency

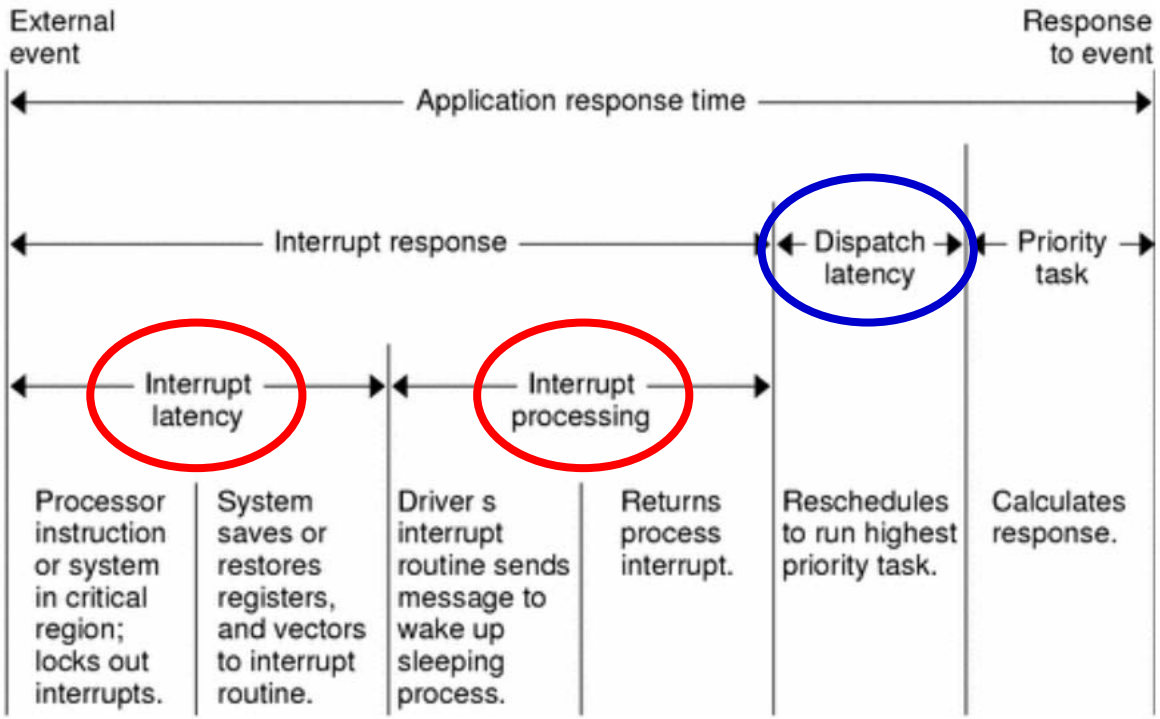
- **Dispatch Latency** – Scheduling dispatcher가 한 프로세스를 stop(block)하고 다른 프로세스를 시작하는데 걸리는 시간 → 실시간 OS에서는 dispatch latency가 최소화되어야 함 → Dispatch latency를 최소화하는 가장 효과적 방법: Preemptive kernel !
  - Dispatcher : the module of OS that sets up the execution of the process selected by the scheduler
- **Diagram of the dispatch latency is shown below:**



- **Conflict phase** in dispatch latency has two components:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes
- **Following the conflict phase,** the **dispatch phase** schedules the high-priority process onto an available CPU.

참고: Process Dispatch Latency 설명

Figure 10–2 Application Response Time



- The overall application response time consists of the interrupt response time, the dispatch latency, and remaning application response time.
- The interrupt response time for an application includes both the interrupt latency of the system and the device driver's own interrupt processing time.
  - The interrupt latency is determined by the longest interval that the system must run with interrupts disabled. This time is minimized in SunOS using synchronization primitives that do not commonly require a raised processor interrupt level.

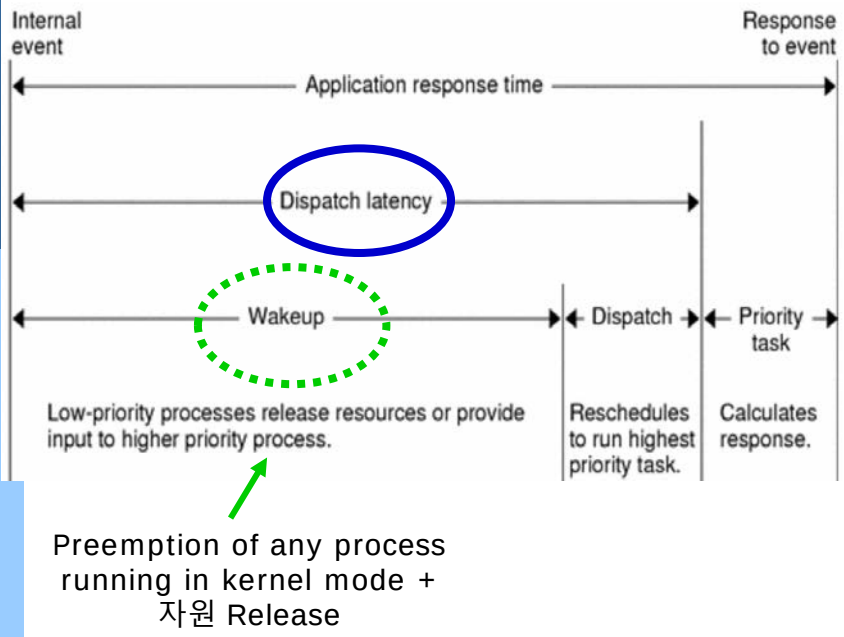
- During interrupt processing, the driver's interrupt routine wakes the high-priority process and returns when finished. The system detects that a process with higher priority than the interrupted process is now ready to dispatch and dispatches the process.

- The time to switch context from a lower-priority process to a higher-priority process is included in the dispatch latency time.

https://docs.oracle.com/cd/E19683-01/816-5042/chap7rt-21297/index.html

참고: Process Dispatch Latency 설명

Figure 10–3 Internal Dispatch Latency



- It illustrates the internal dispatch latency and application response time of a system.
- The response time is defined in terms of the amount of time a system takes to respond to an internal event.

- **The dispatch latency of an internal event** represents the amount of time that a process needs to wake up a higher priority process.

- **The dipatch latency** also includes the time that the system takes to dispatch the higher priority process.

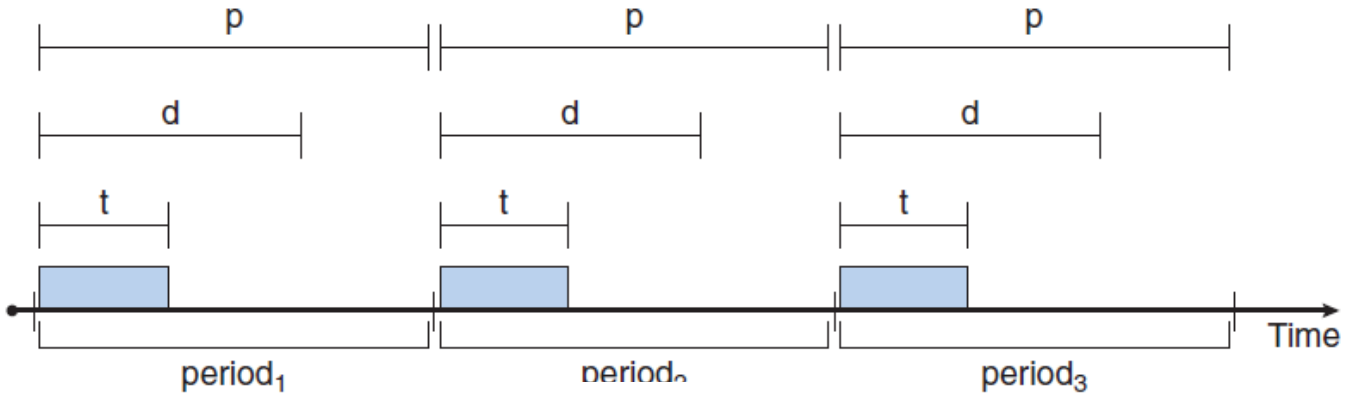
- **The application response time** is the amount of time that a driver takes to: wake up a higher-priority process, release resources from a low-priority process, reschedule the higher-priority task, calculate the response, and dispatch the task.

- Interrupts can arrive and be processed during the dispatch latency interval. This processing increases the application response time, but is not attributed to the dispatch latency measurement. Therefore, this processing is not bounded by the dispatch latency guarantee.

Priority-based Scheduling

- **Real time OS는 실시간 프로세스가 CPU를 필요로 할 때, 바로 응답해줘야 함**
- **So, the scheduler must support preemptive, priority-based scheduling**
  - However, providing a preemptive, priority-based scheduling means **that it only guarantees soft real-time functionality**
  - **Hard real-time systems** must further guarantee that real-time tasks **will be serviced in accordance with their deadline requirements** and making such guarantees requires additional scheduling features.
- Before we proceed with the details of the individual schedulers, however, we must define certain characteristics of the processes that are to be scheduled:
- **Processes have new characteristics: periodic ones require CPU at constant intervals**
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is 1/*p*

scheduler는 마감시간 이내에 완수할 수 있는 프로세스의 실행을 허락하고 그렇지 않는 프로세스는 실행을 허락하지 않을 수 있음 → **Scheduler의 admission control mechanism**



< Periodic task >

Rate Monotonic Scheduling

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

- For example:
- Period of Process P1, P2: $p_1 = 50$ and $p_2 = 100$
- Processing time of P1 & P2: $t_1 = 20$, $t_2 = 35$

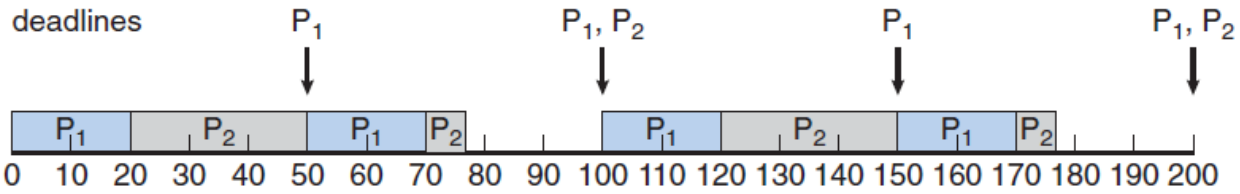
- The deadline for each process requires that it complete its CPU burst by the start of its next period.
- We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines.

- 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 percent.
  - 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 Montonic Scheduling

- Now 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.
- The execution of these processes in this situation is shown in Figure 17.
- P_1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline.
- P_2 starts running at this point and runs until time 50.

- Period(deadline): $p_1 = 50$ and $p_2 = 100$
- Processing time: $t_1 = 20$, $t_2 = 35$



< Fig 17. Rate-monotonic scheduling >

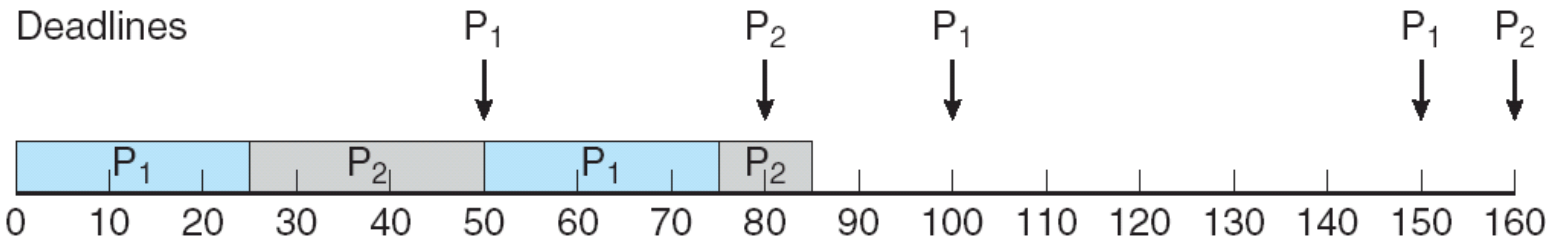
- At this time, P_2 is preempted by P_1, although it still has 5 milliseconds remaining in its CPU burst.
- P_1 completes its CPU burst at time 70, at which point the scheduler resumes P_2.
- P_2 completes its CPU burst at time 75, also meeting its first deadline.

- The system is idle until time 100, when P_1 is scheduled again.

Rate Montonic Scheduling

- **Rate-monotonic scheduling으로 스케줄링할 수 없는 경우에는 다른 static priority 알고리즘으로도 스케줄링할 수 없는 것으로 알려져 있음**

- **Let's next examine a set of processes that cannot be scheduled using the rate-monotonic algorithm.**



< Fig18. **Missing deadlines with rate-monotonic sch**eduling >

- **Process P1**, *period* $p_1 = 50$, $t_1 = 25$
- **Process P2**, *period* $p_2 = 80$, $t_2 = 35$
- P_1의 시간이 짧기때문에 rate-monotonic scheduling에서는 P_1에 우선 순위 부여함 → P_1이 시작되어 25 수행 후, P_2가 수행됨.
- 그런데 t=50에서 P_2는 P_1에 의해 선점됨. 결국 P_2는 자신의 deadline=80을 만족하지 못함

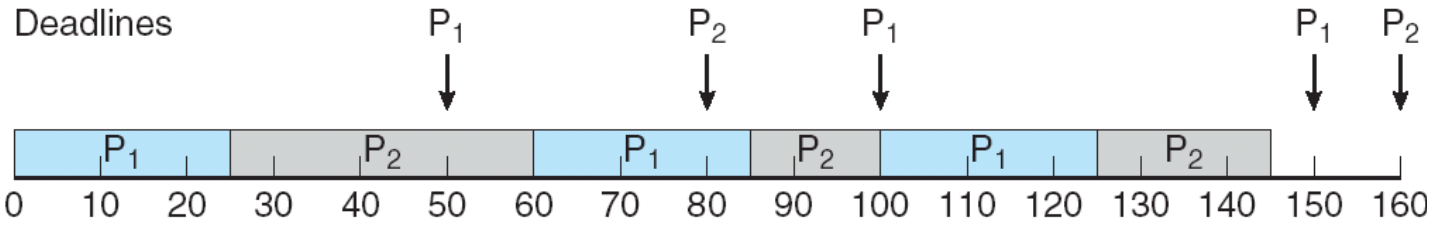
https://en.wikipedia.org/wiki/Rate-monotonic_scheduling

Earliest Deadline First Scheduling (EDF)

- **Priorities are assigned dynamically according to deadlines:**
  the earlier the deadline, the higher the priority;

  the later the deadline, the lower the priority

- **Process P1**, *period* $p_1 = 50$, $t_1 = 25$
- **Process P2**, *period* $p_2 = 80$, $t_2 = 35$



< Fig19. Earliest deadline first scheduling >

- 처음에는 프로세스 P1의 마감시간이 더 빠르기 때문에 **P1이 P2보다 우선 순위 높음**
- rate-monotonic 스케줄링에서처럼 t=50에서 P1이 P2를 선점했 던 것과는 달리 EDF 스케줄링에서는 **P2가 계속 수행. →** 왜냐하면 EDF에서는 deadline이 더 빠른게 우선 순위 높으므로, **P2가 P1보다 우선순위 높아짐(P2's next deadline (time 80) is earlier than that of P1 (time 100)).**
- Thus, both P1 and P2 meet their first deadlines.
- Process P1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. P2 begins running at this point, only to be preempted by P1 at the start of its next period at time 100. P2 is preempted because P1 has an earlier deadline (time 150) than P2 (time 160). At time 125, P1 completes its CPU burst and P2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P1 is scheduled to run once again.

Proportional Share Scheduling

- ***T* shares are allocated among all processes in the system**
- **An application receives *N* shares where *N* < *T***
- **This ensures each application will receive *N* / *T* of the total processor time**
- **Example**
  - T=100인 time share가 있으며, 3개의 process, A,B,C가 있다고 가정하자.
  - A← 50, B← 15, C← 20을 할당할때, A는 모든 처리기 시간의 50%, B:15%, C:20%를 할당받는 것임
  - Proportional share scheduling(일정 몫 할당 스케줄링)은 process가 특정 시간 share를 할당 받는 것을 보장하는 “승인 제어 정책”과 함께 동작해야 함

POSIX Real-Time Scheduling

- The POSIX.1b standard : Real-time extensions
- 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

- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t_tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Operating System Examples

- **Linux scheduling**
  - History of Linux Scheduler
  - General Scheduler for Linux: O(1), CFS
  - Real-time Scheduler for Linux
- **Windows scheduling**
- **Solaris scheduling**

Linux Scheduler(1) - Brief History

- **Linux 1.0 : Simple linked list of Runnable Processes**
- **Linux 2.0 : SMP support**
- **Linux 2.5 : O(1) scheduler**
  - Priority sorted array of lists
- **Linux 2.6.23 : CFS**
  - Completely Fair Scheduler

Linux Scheduler(2) – O(1)

- **Linux O(1) scheduler**
- **Version 2.5 moved to constant order *O*(1) scheduling time**
  - Constant time required to pick the next process to execute
  - Processes divided into 2 types
    - Scheduling real time processes
      - Priorities from 0 to 99
    - Scheduling normal processes (Interactive, Batch)
      - Priorities from 100 to 139
  - Worked well, but poor response times for interactive processes

Linux Scheduler(2) – CFS

- **Completely Fair Scheduler (CFS)**
  - Elegant handling of I/O and CPU bound processes
- **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
    1. default
    2. real-time
- **Quantum calculated based on nice value from -20 to +19**
  - Lower value is higher priority
  - Calculates **targeted latency(목적 지연시간)** – interval of time during which task should run at least once
  - Target latency can increase if number of active tasks increases
- **CFS scheduler maintains per task virtual run time in variable `vruntime`**
  - The virtual runtime is a concept introduced to achieve fairness.
  - It takes the nice value as well as the physical execution time of that task into account. That is, virtual run time is the weighted time a task has run on the CPU.
  - $vruntime$ **= (**$execution\ time$ **/ Load weight of the task)** $* NICE\_O\_LOAD$, where NICE_O_LOAD is the default nice value.
  - 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**

Linux Scheduler(2) – CFS

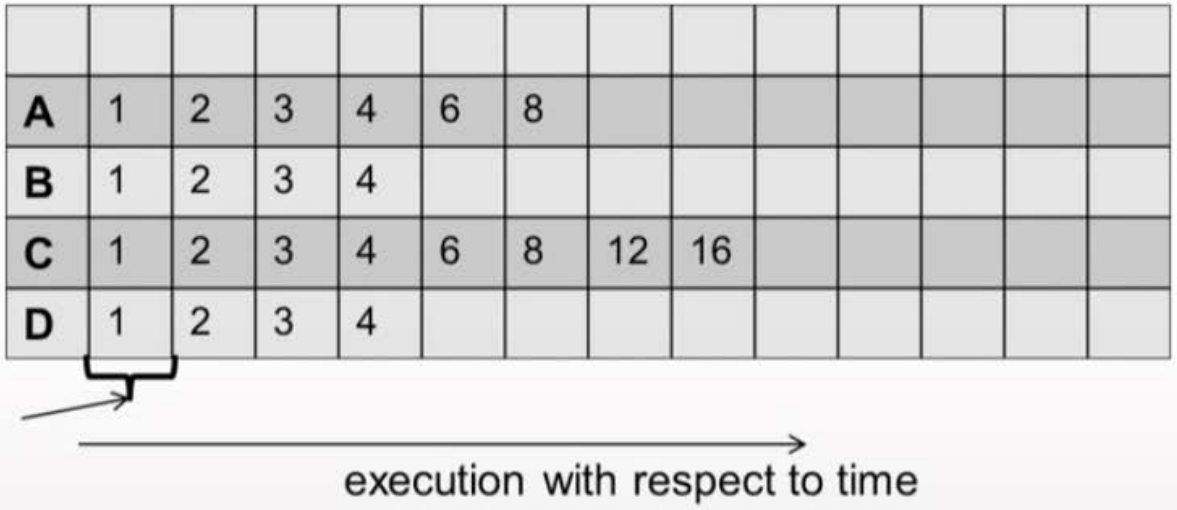
- **Ideal Fair Scheduling**

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness



Ref) IIT Madras, prof. Chester Rebeiro
http://www.cse.iitm.ac.in/~chester/

Linux Scheduler(2) – CFS

Ideal Fair Scheduling

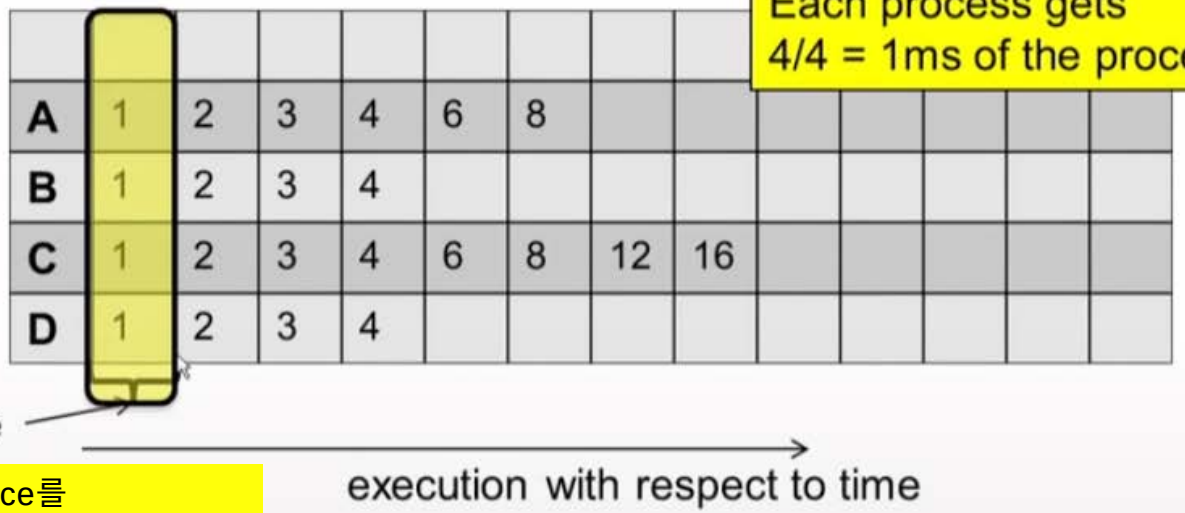
Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

Each process gets
4/4 = 1ms of the processor time



4ms의 time slice를
프로세스(A~D)가 균등하게
나눠서 사용

Ref) IIT Madras, prof. Chester Rebeiro
http://www.cse.iitm.ac.in/~chester/

Linux Scheduler(2) – CFS

Ideal Fair Scheduling

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

4ms의 time slice를 C 프로세스가 사용

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

프로세스 B와 D 실행이 끝나면, 4ms의 time slice를 A, C 프로세스 2개가 균등하게 2ms 씩 사용

Ref) IIT Madras, prof. Chester Rebeiro
http://www.cse.iitm.ac.in/~chester/

참고) CFS – 각 process의 Targeted Latency의 균등분할

CFS now has a weighted proportion of processor time assigned to each process. To determine the actual length of time each process runs, CFS needs to divide the proportions into a fixed period. That period is called the *target latency*, as it represents the scheduling latency of the system. To understand the target latency, let's assume it is set to 20 milliseconds and that there are two runnable processes of the same priority. Thus, each process has the same weight and is assigned the same proportion of the processor, 10 milliseconds. Thus, CFS will run one process for 10 milliseconds, then the other for 10 milliseconds, and then repeat. If there are 5 runnable processes on the system, CFS will run each for 4 milliseconds.

Target latency가 20ms로 setting된 경우. 2개의 process에대해 10ms 할당됨

100개의 process 있는 경우,
각 proces는 100us 할당됨
(context switching 소요 시간을 고려하면?)

Target latency / Process	(1) 20ms	(2) 20ms	(3) 20ms
P1	10ms	5ms	100 us
P2	10ms	5ms	100 us
P3	-	5ms	100 us
P4	-	5ms	100 us
…	-	-	**…**
P200	-	-	100 us

Linux System Programming: Talking Directly to the Kernel and C Library by Robert Love, CH6.

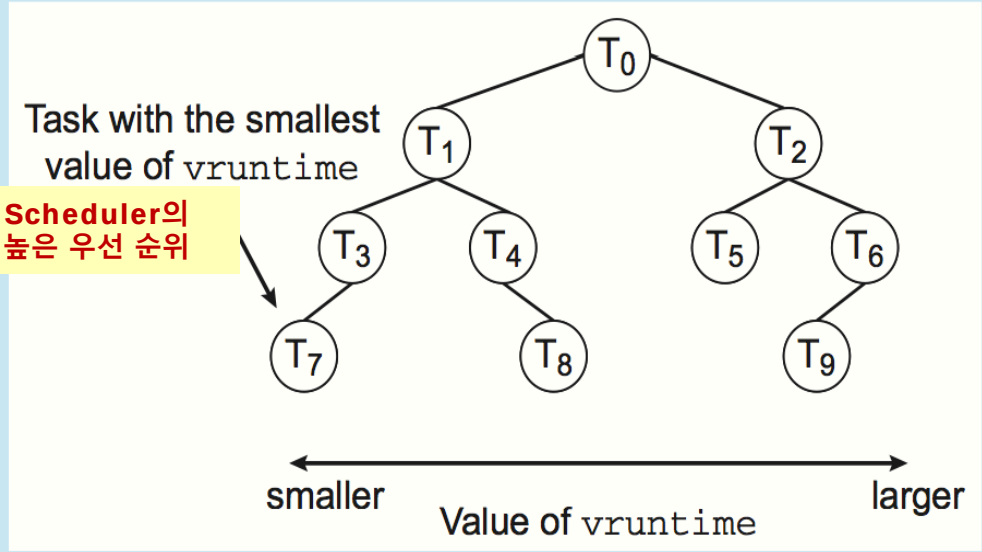
참고) Mininum Granularity 개념 도입 필요성

So far, so good. But what if we have, say, 200 processes? With a target latency of 20 milliseconds, CFS would run each of those processes for only 100 microseconds. Due to the cost of context switching from one process to another, known as *switching costs*, and the reduced temporal locality, the system's overall throughput would suffer. To deal with this situation, CFS introduces a second key variable, the minimum granularity. The *minimum granularity* is a floor on the length of time any process is run. All processes, regardless of their allotted proportion of the processor, will run for at least the minimum granularity (or until they block). This helps ensure that switching costs do not consume an unacceptably large amount of the system's total time at the expense of honoring the target latency. That is, when the minimum granularity kicks in, fairness is violated. With typical values for the target latency and minimum granularity, in the common case of a reasonable number of runnable processes, the minimum granularity is not applied, the target latency is met, and fairness is maintained.

Linux System Programming: Talking Directly to the Kernel and C Library by Robert Love, CH6.

참고) vruntime – 각 process의 수행 CPU 시간

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

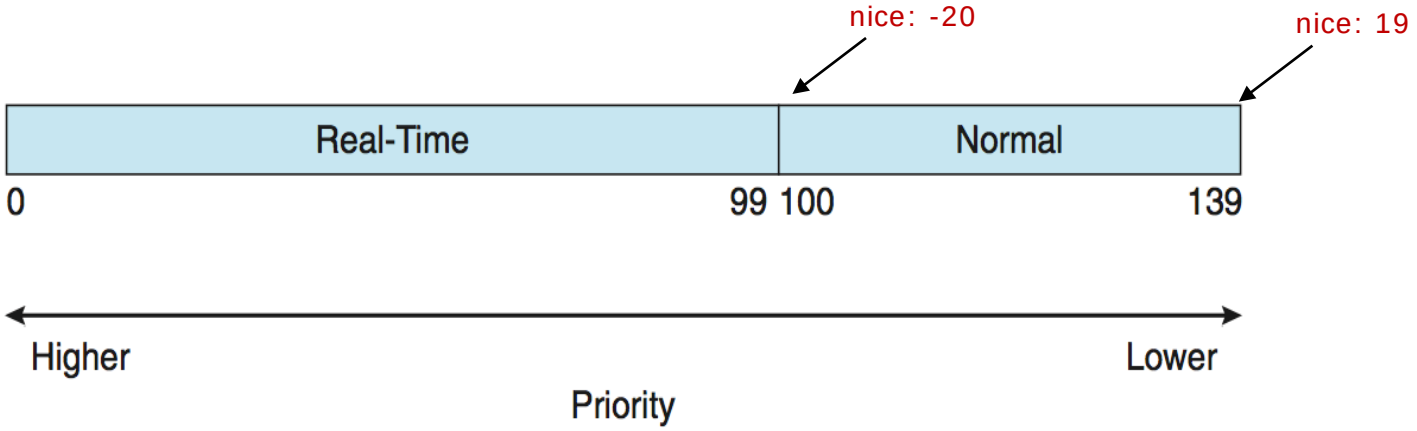


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and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

- vruntime: the amount of time a process got CPU time (process가 지금까지 얼마나 많은 CPU time을 얻었는지를 알려줌)
  - low vruntime value indicates that the process is "processor deprived" .(해당 proces의 작은 vruntime 값은 그 동안 CPU 할당이 적었다는 것을 의미함)
  - high vruntime value indicates that the process acquired "considerable processor time" (높은 vruntime 값은 그동안 많은 CPU 시간을 할당받았다는 것을 의미)
  - 이에, proces간 fair한 scheduling을 위해 그동안 실행이 적게된(즉, vruntime이 적은) process 에 높은 scheduling 우선순위를 줌

Linux Scheduler(3) – Real-time scheduler

- **Linux의 Real-time Scheduler - Real-time scheduling according to POSIX.1b**
  - Real-time tasks have static priorities
- **Real-time plus normal map into global priority scheme**
  - Nice value of -20 maps to global priority 100
  - Nice value of +19 maps to priority 139



Linux Scheduler - 요약

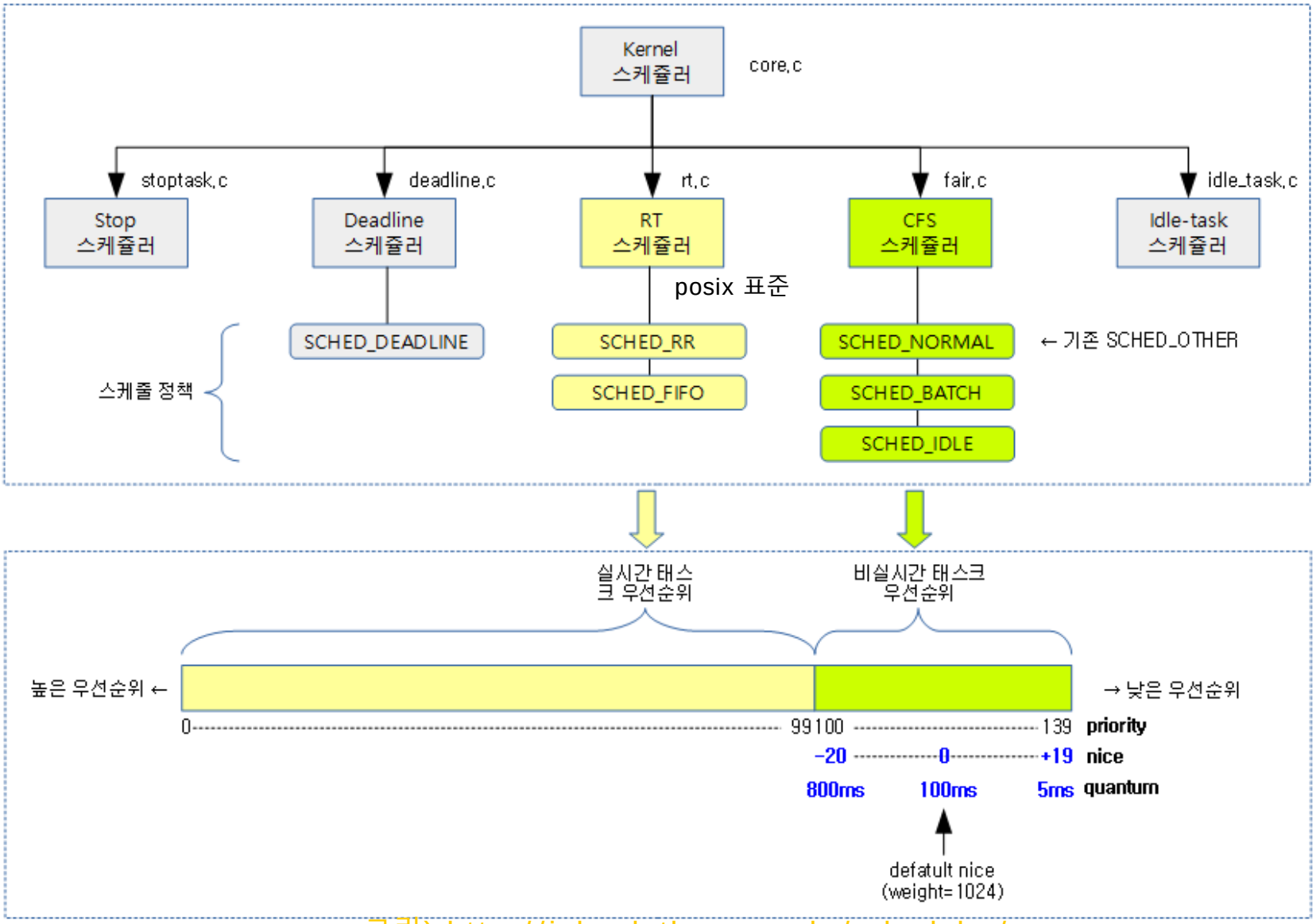


그림) http://jake.dothome.co.kr/scheduler/

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- Dispatcher is scheduler
- Thread runs until (1) it is blocked, (2) allocated time slice is over, (3) it is 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

Real time class

Variable 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

상대적
우선순위

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
- **프로세스는 대개 NORMAL_PRIORITY_CLASS에 속함 (단, 생성시 parent process가 IDLE_PRIORITY_CLASS가 아니며, 특정 class로 명시되지 않은 경우)**
- **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
If quantum expires, priority lowered, but never below base

상대적 우선순위

Real time class

Variable 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

Windows Priority Classes

- 스레드가 생성될 때, 해당 스레드가 속한 class에서 기본 우선순위를 배정받음
- 예: 각 class의 기본 우선 순위는 normal.

- REALTIME_PRIORITY_CLASS—24
- HIGH_PRIORITY_CLASS—13
- ABOVE_NORMAL_PRIORITY_CLASS—10
- NORMAL_PRIORITY_CLASS—8
- BELOW_NORMAL_PRIORITY_CLASS—6
- IDLE_PRIORITY_CLASS—4

Real time class

Variable class

상대적 우선순위

	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

- 스레드의 시간 할당량이 만료되면, 해당 스레드는 interrupt 됨

Solaris

- **Priority-based scheduling**
- **Six classes available**
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- **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

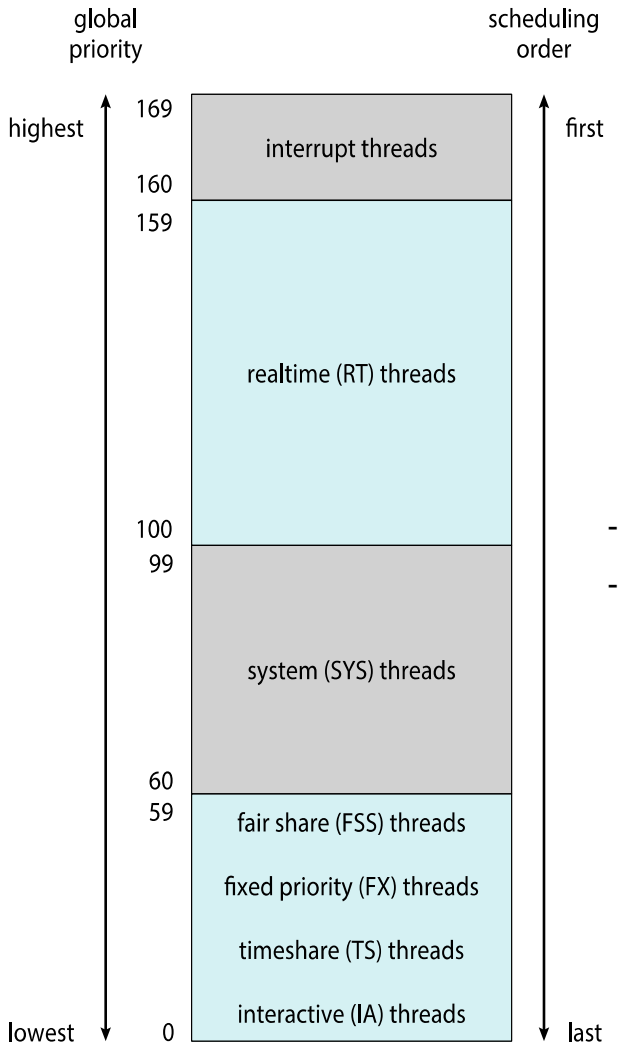
Solaris Dispatch Table

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

높은
우선순위

- Interactive와 Time-sharing 스케줄링 클래스에 대한 dispatch table 사례
- 총 60개의 우선순위 있음
- Time quantum(시간 할당량), 20ms가 가장 우선 순위 높음을 알 수 있음
  - 우선순위와 Time quantum(시간 할당량)은 여기서는 서로 반비례관계임
  - 이는, Interactive 즉, 대화형 프로세스는 높은 우선순위를 가지고 CPU 위주의 프로세스는 낮은 우선순위를 가짐 → 이를 통해 대화형 프로세스에 좋은 response time 특성을 제공하고 CPU 위주의 process 에는 좋은 throughput 특성을가지게 함
- Time quantum expired 된 경우, 즉, 시간할당량을 모두 사용한 스레드의 새로운 우선 순위 정보
- Return from sleep : 입출력 wait 와 같은 수면 대기상태에서 복귀한 스레드의 우선 순위 정보

Solaris Scheduling



- Real time thread : 높은 우선 순위 제공
- System (SYS) thread : scheduler나 page daemon 같은 kernel thread 용

Algorithm Evaluation

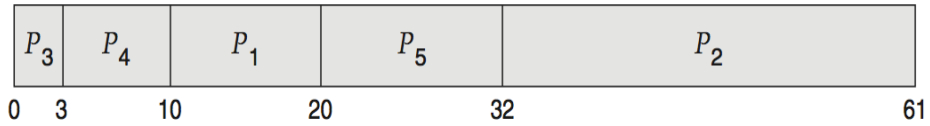
- **How to select CPU-scheduling algorithm for an OS?**
- **Determine criteria, then evaluate algorithms**
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- **Consider 5 processes arriving at time 0:**

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

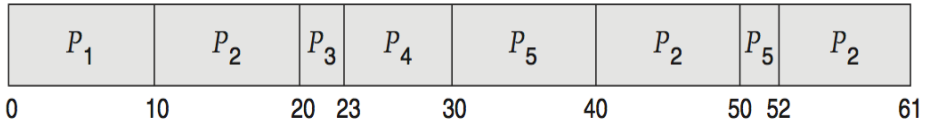
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCFS is 28ms:

  - Non-preemptive SJF is 13ms:



  - RR is 23ms:



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

Queueing Models

- 많은 시스템에서 실행되는 process 들은 실제 날마다 변화함. 이에, deterministic modeling을 사용하기 어려움 → 이에, CPU와 I/O burst 분포를 근사값 계산하거나 단순하게 추정하여 이를 사용
- **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:**

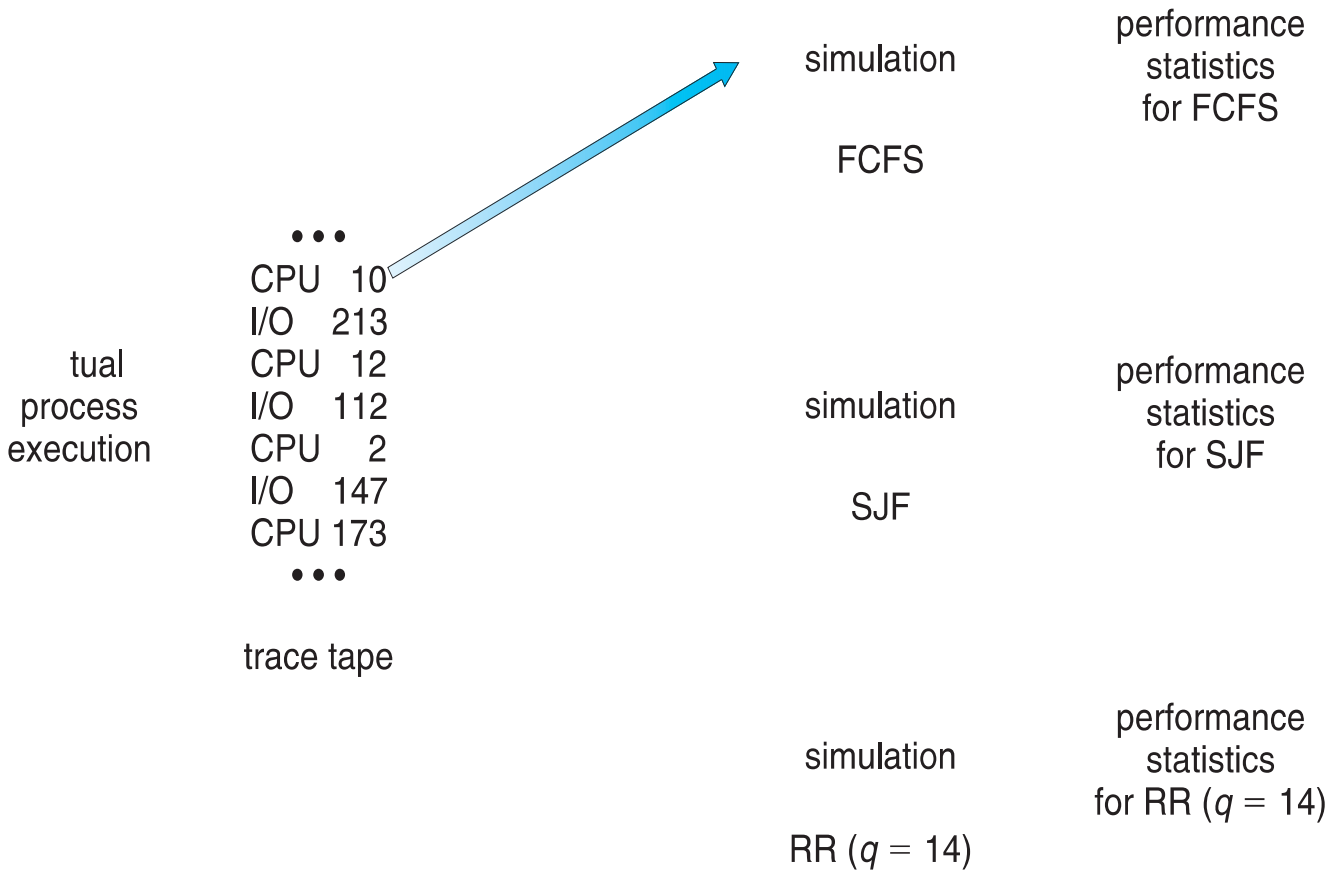
  $$n = \lambda \times W$$

  - 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



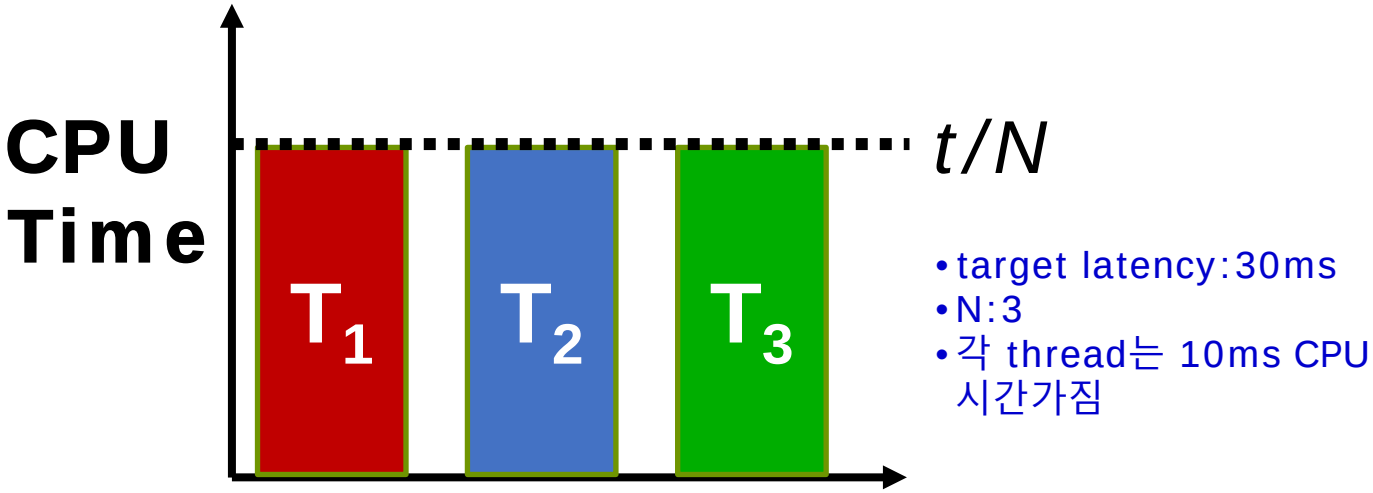
Implementation

- 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

End of Chapter 5

참고) vruntime 개념 추가 설명

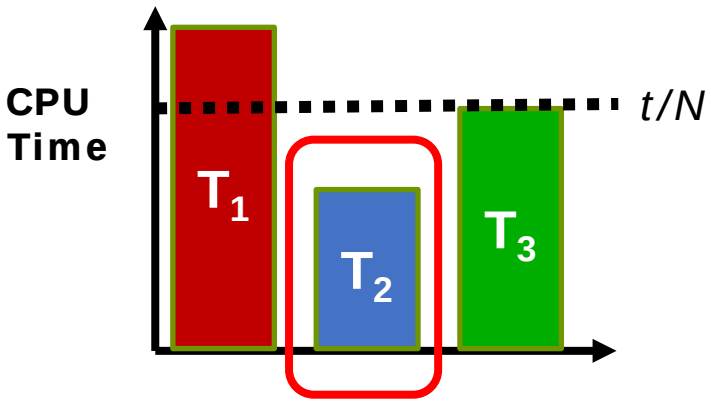
- **Linux CFS**
  - **Goal :** Each process gets an equal share of CPU
  - *N* threads "simultaneously" execute on 1/N^{th} of CPU



이상적으로 위 3개의 thread는 target latency t를 각각 1/3씩 분할한 CPU 시간을 가짐

참고) vruntime 개념 추가 설명

- 하지만, 실제 상황에서는 동일한 CPU time 만큼 실행되지는 않음
  - 각 thread(process)의 CPU time을 tracking 필요



- 수행 시간 합한 값 : vrumtime

  - 각 thread 실제 CPU time tracking을 통해 각 thread의 수행 시간이 fair하게 되도록 계속 노력
    - 위 그림에서 thread T2의 CPU 시간이 가장 적음 → 시간 할당 늘임

참고) vruntime 기반 CFS 사례

Constraint 1: Target Latency

- Period of time over which every process gets service
- Preserves response time
- **Target Latency: 20ms. when 4 Processes,**
  - Each process gets 5ms time slice
- **Target Latency: 20 ms. when 200 Processes,**
  - Each process gets 0.1ms time slice
  - Recall Round-Robin: Huge context switching overhead

Constraint 2: Minimum Granularity

- Minimum length of any time slice
- Protects throughput
- Target Latency 20ms, Minimum Granularity 1ms, 200 processes
  - Each process gets 1ms time slice

참고) vruntime 기반 CFS 사례

- **What if we want to use nice to change priority?**
  - Key Idea: Assign a weight w_i to each process i
  - Originally: time for each process, $Q = \text{Target Latency} * \frac{1}{N}$
  - Now: $Q_i = (w_i / \sum_p w_p) * \text{Target Latency}$
- **example**
  - Target Latency = 20ms, Minimum Granularity = 1ms
  - Two CPU-Bound Threads
    - Thread *A* has weight 1
    - Thread *B* has weight 4
  - Time slice for *A* → 4 ms
  - Time slice for *B* → 16 ms

참고) vruntime 기반 CFS 사례

- Track a thread's virtual runtime rather than its true physical runtime