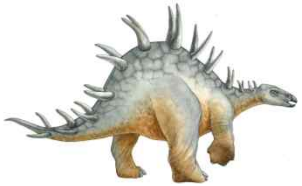
Chapter 5: CPU Scheduling

Chapter 5: CPU Scheduling

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

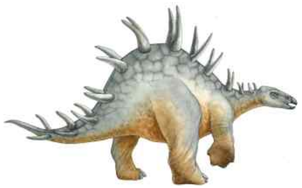


Basic Concepts

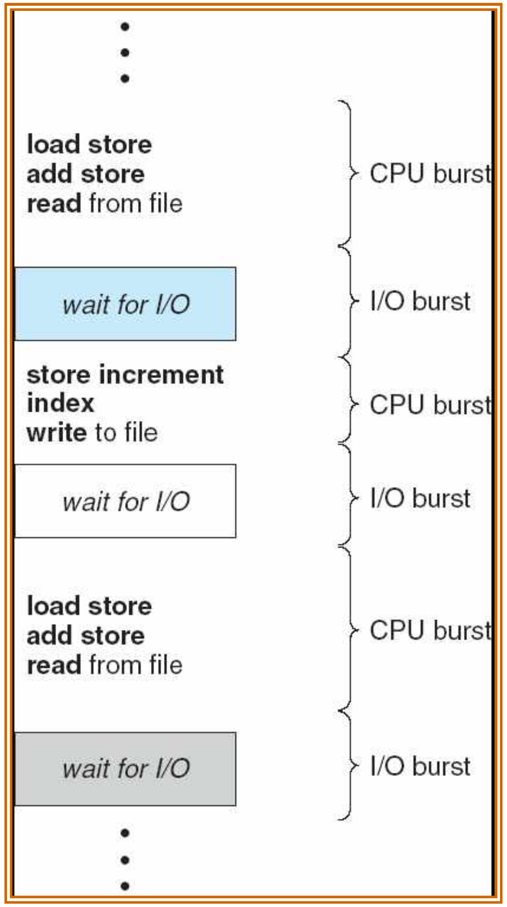
장기 job scheduling
단기 CPU scheduling <=Focus
중기 swapping : Swap In, Swap Out

- 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

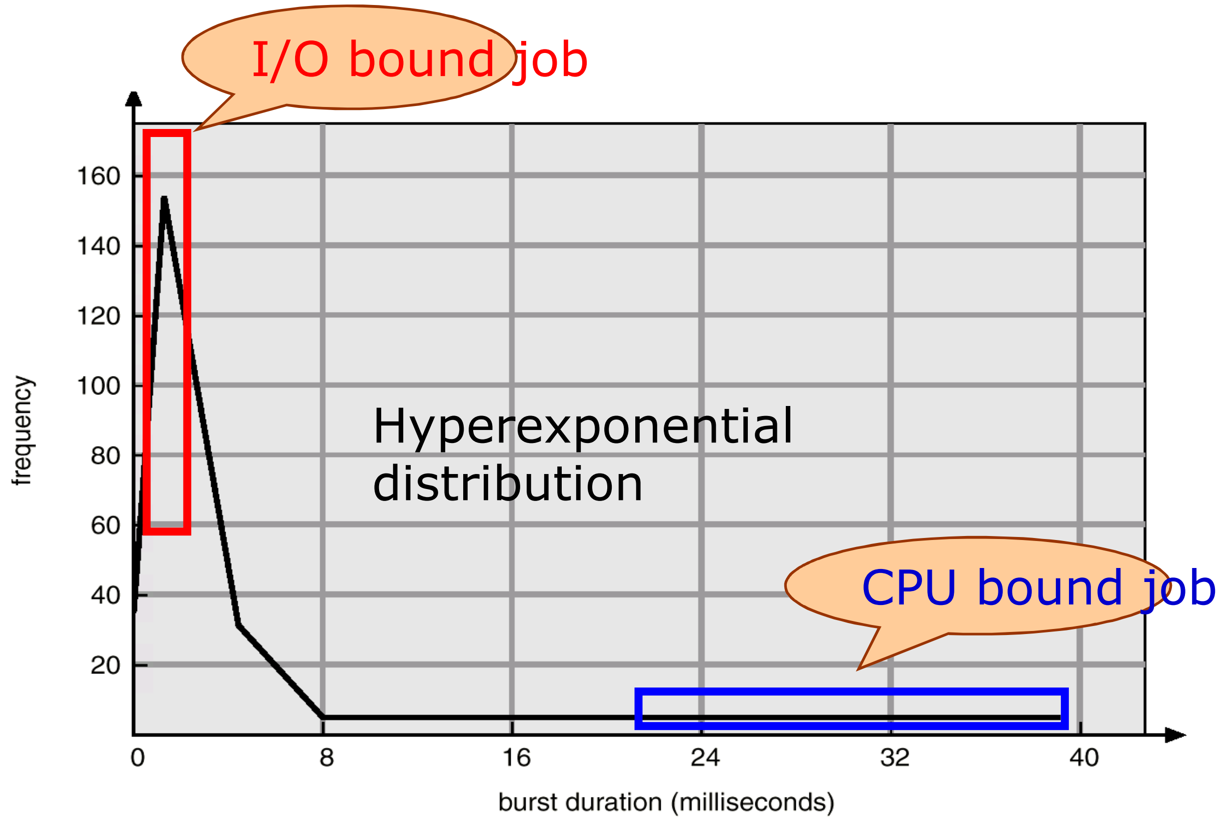
- CPU 스케줄러
  - 단기 스케줄러(short-term scheduler) : ready queue에서 선택
    FIFO(First-In First-Out)큐
    우선순위 큐
    트리
    연결리스트



Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times

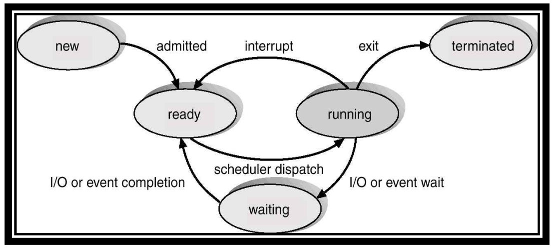


일반적인 시스템에서,
다수의 짧은 CPU burst와 적은 수의 긴 CPU burst로 구성
=> 어떻게 스케쥴링할 것인가?

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



- 1과 4에서만 Scheduling이 발생할 경우: *nonpreemptive*로 충분
- 모든 경우에서 Scheduling이 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/O까지 계속 CPU점유

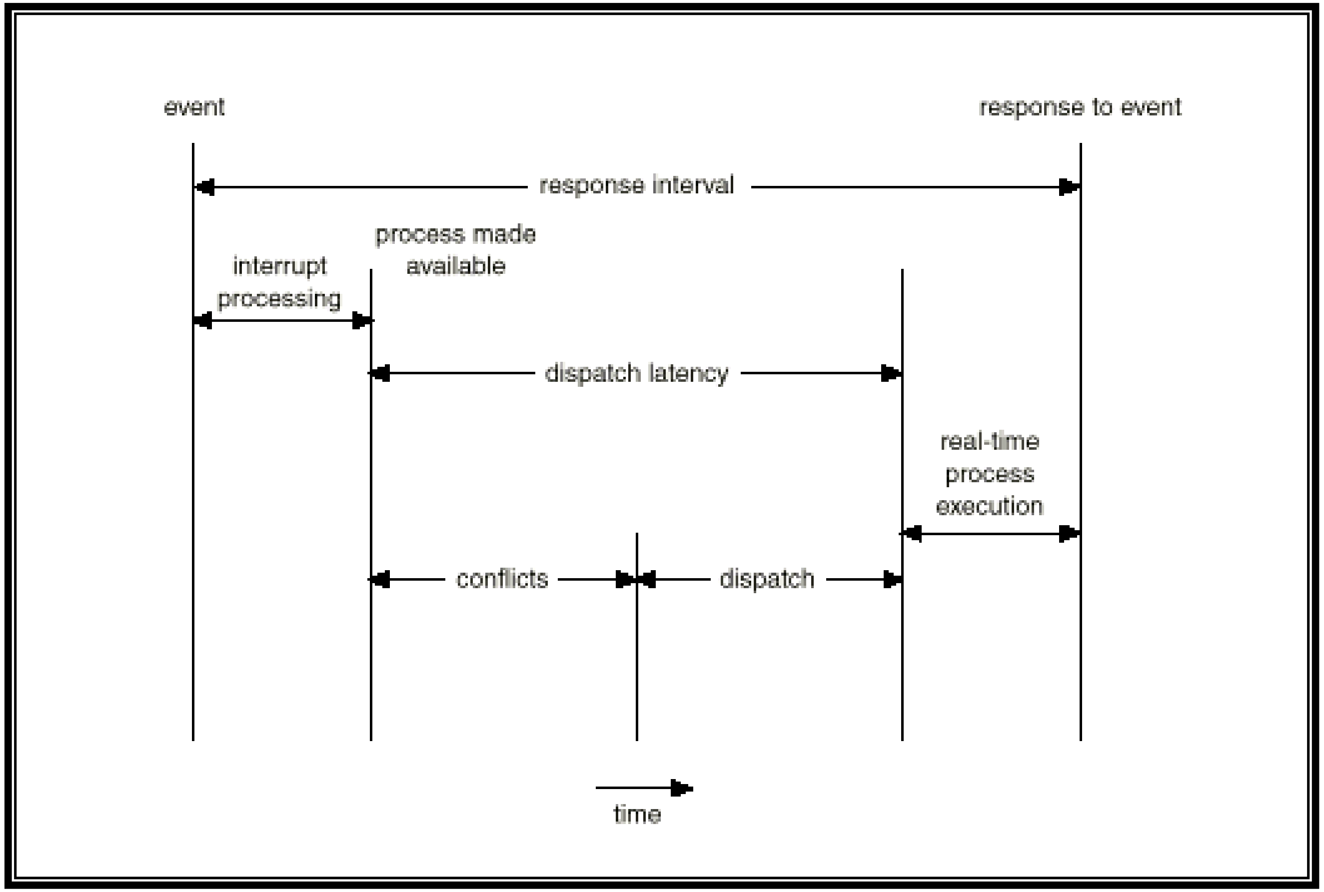
Dispatcher

- Dispatcher의 정의
  - CPU 스케쥴러에 의해 선택된 프로세스에게 CPU에 대한 제어권한을 주는 모듈

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

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

Dispatch Latency

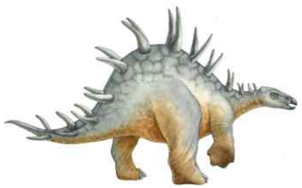


CPU Scheduling의 성능 기준

- 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 queue에서 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***



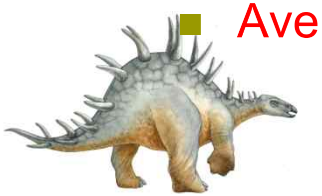
First-Come, First-Served (FCFS) Scheduling

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

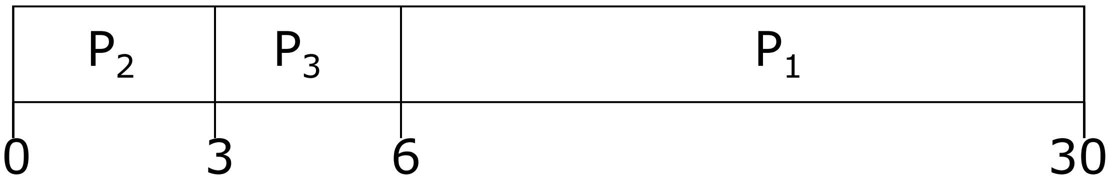


FCFS Scheduling (Cont.)

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 Queue와 Read Queue를 가진)에 있어서 CPU-bound 프로세스(CPU를 많이 차지하는)와 I/O bound 프로세스(상대적으로 CPU를 적게 사용하는)가 있을 때 CPU-bound 프로세스로 인해 I/O bound 프로세스가 짧은 CPU의 할당만으로 JOB을 완료할 수 있음에도 불구하고, 순서를 기다림으로써 전반적인 시스템 성능이 떨어지는 효과

Shortest-Job-First (SJF) Scheduling

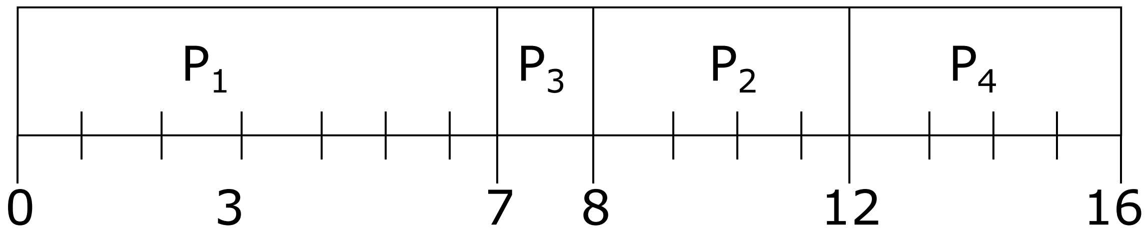
최소 작업 우선(Shortest-Job-First) 스케줄링

- SJF Scheduling의 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).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

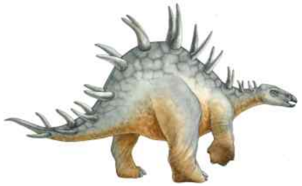
Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



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

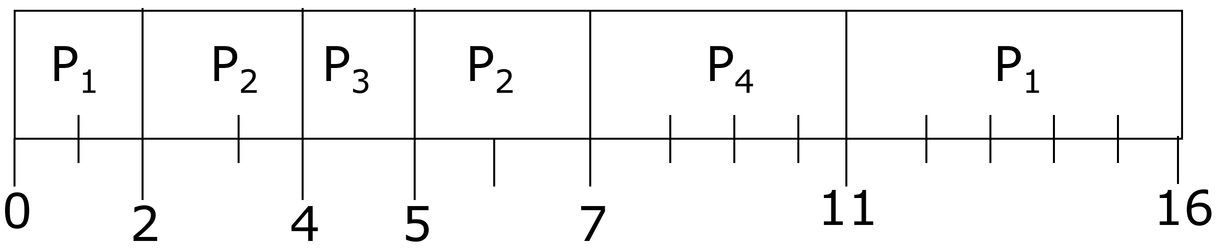


Example of Preemptive SJF

Preemptive

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

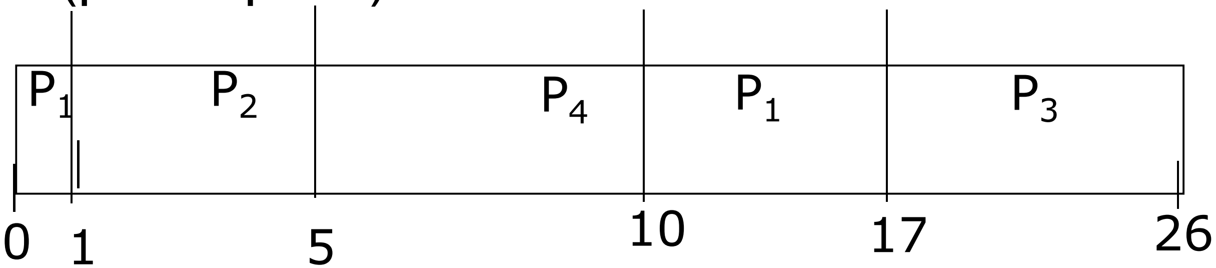


  - Average waiting time = (9 + 1 + 0 +2)/4 = 3

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

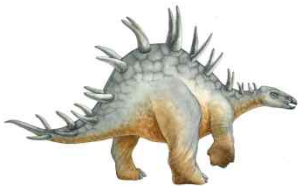
- SJF (preemptive)



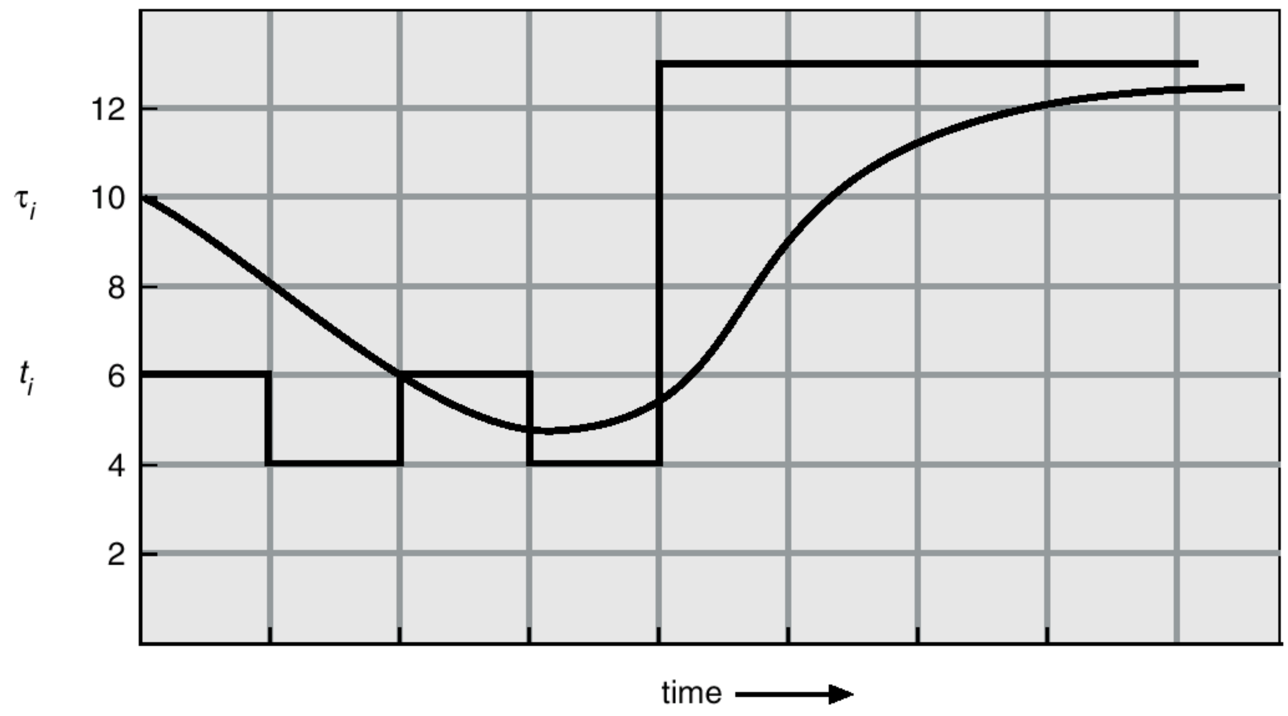
  - Average waiting time = ?

SJF

- SJF is optimal – gives minimum average waiting time for a given set of processes
  - long-term scheduling에 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



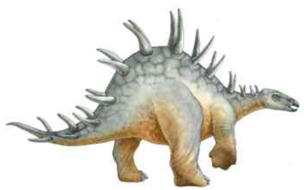
CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

alpha = 1/2

Determining Length of Next CPU Burst

- Can only estimate the length
- 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. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

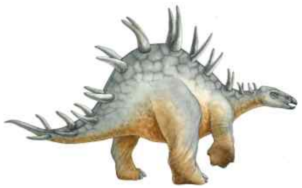


Examples of Exponential Averaging

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

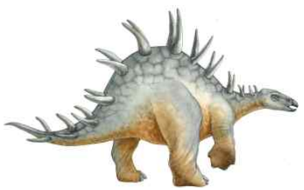
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_n - 1 + \ldots$$
$$+ (1 - \alpha)^j\, \alpha\, t_n - 1 + \ldots$$
$$+ (1 - \alpha)^{n=1}\, t_n\, \tau_0$$

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



SJF(Shortest-Job-First) 스케줄링 (nonpreemptive 기법)

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

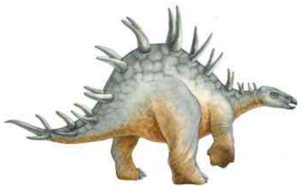


Priority Scheduling

NonPreemptive

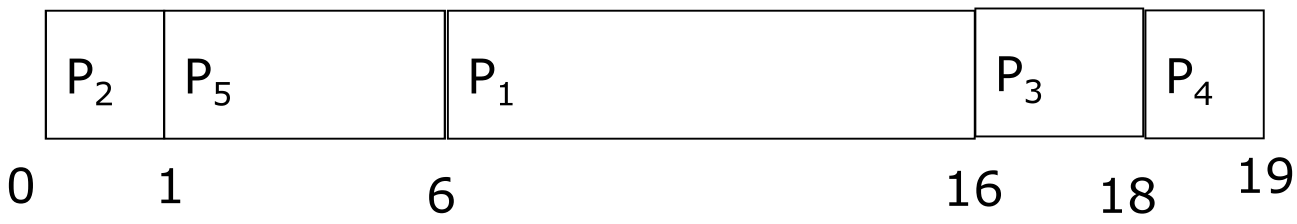
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
  - Problem ≡ Starvation – 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!

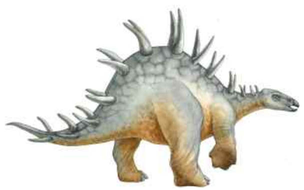


Priority Scheduling

Process	Burst Time	Priority
P1	10	3
P2	*1*	1
P3	*2*	4
P4	*1*	5
P5	*5*	2



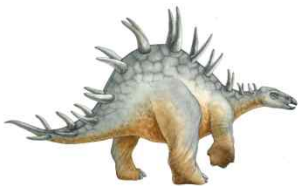
평균 대기 시간 : 8.2초



Round Robin (RR)

Preemptive

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.
    - 할당되는 시간이 클 경우 FIFO 기법과 같아짐 9908
    - 할당되는 시간이 작은 경우 문맥 교환 및 오버헤드가 자주 발생



Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

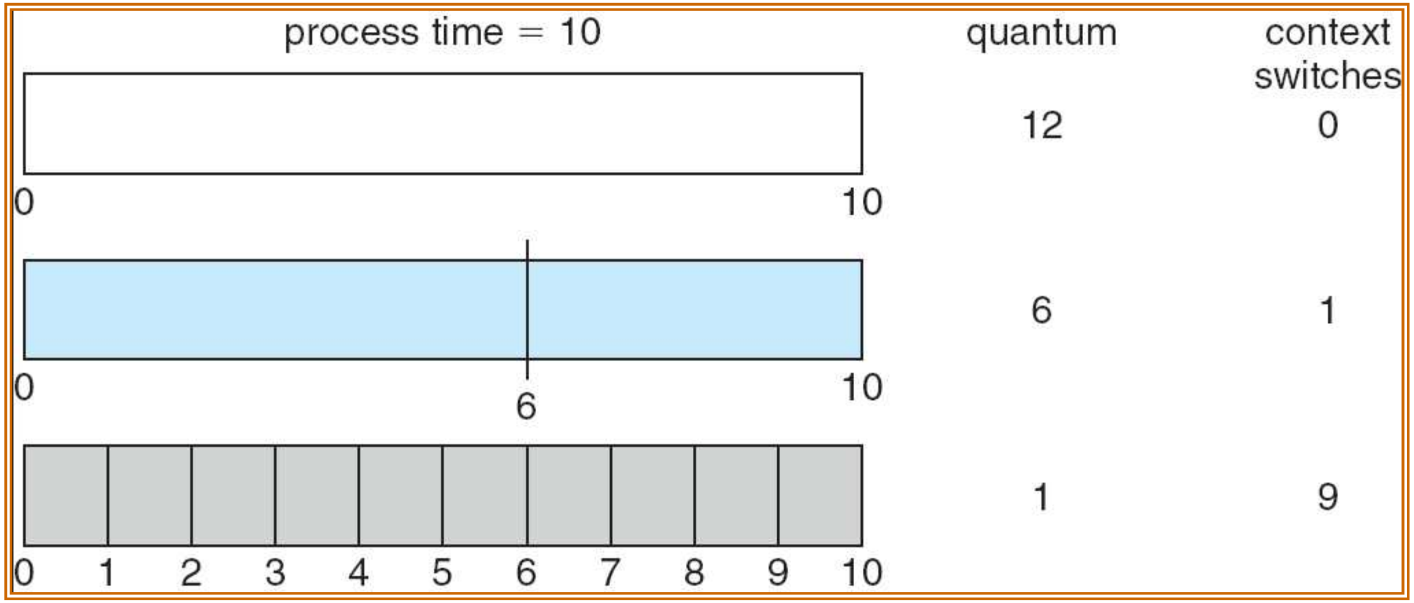
- The Gantt chart is:

	P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3
0	20	37	57	77	97	117	121	134	154	162

- Typically, higher average turnaround than SJF, but better *response*.

Time Quantum and Context Switch Time

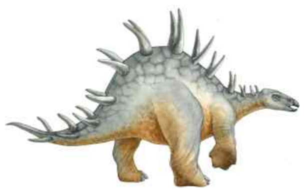
Context Switch Overhead가 1이라고 한다면,



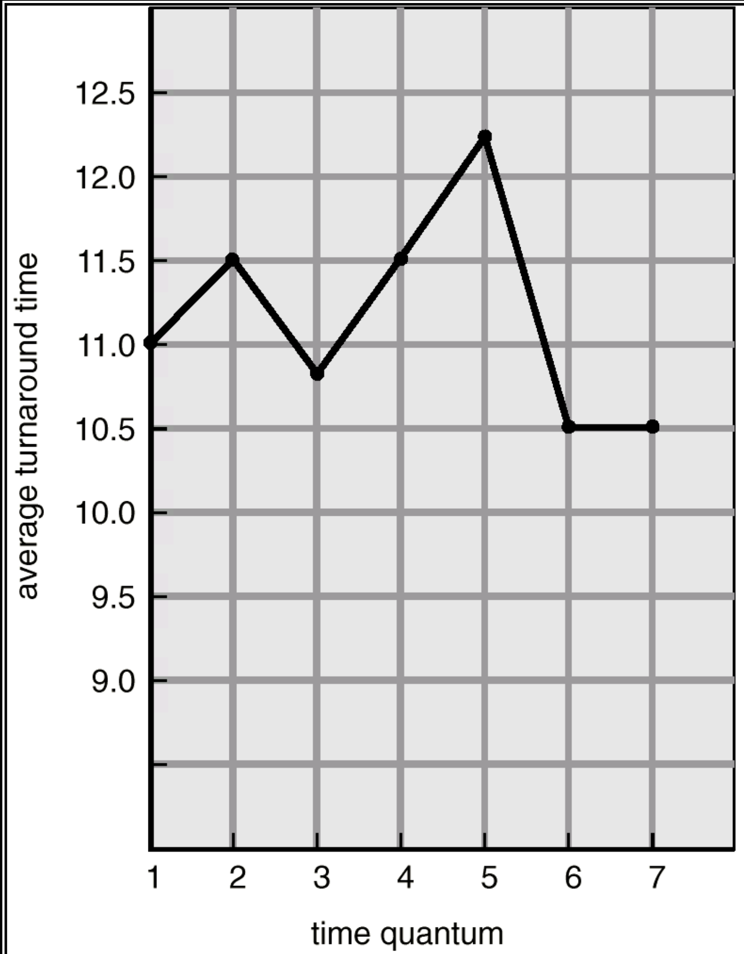
Quantum 의 크기

- 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

SRT(Shortest-Remaining Time) Preemptive

- SRT(Shortest-Remaining-Times First) 스케쥴링 : preemptive
  - SJF 를 Preemptive 기법으로 변형
  - 대기 list 상의 job 중 남아있는 실행시간 추정치가 가장 작은 작업 선택

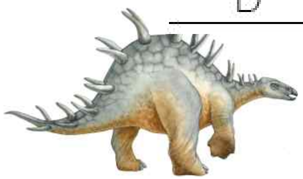
HRN(Highest-Response-ratio Next) NonPreemptive

- 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 = $\frac{\text{대기시간} + \text{서비스시간}}{\text{서비스시간}}$

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

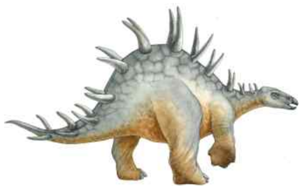
※ 우선순위가 가장 높은 것은 D



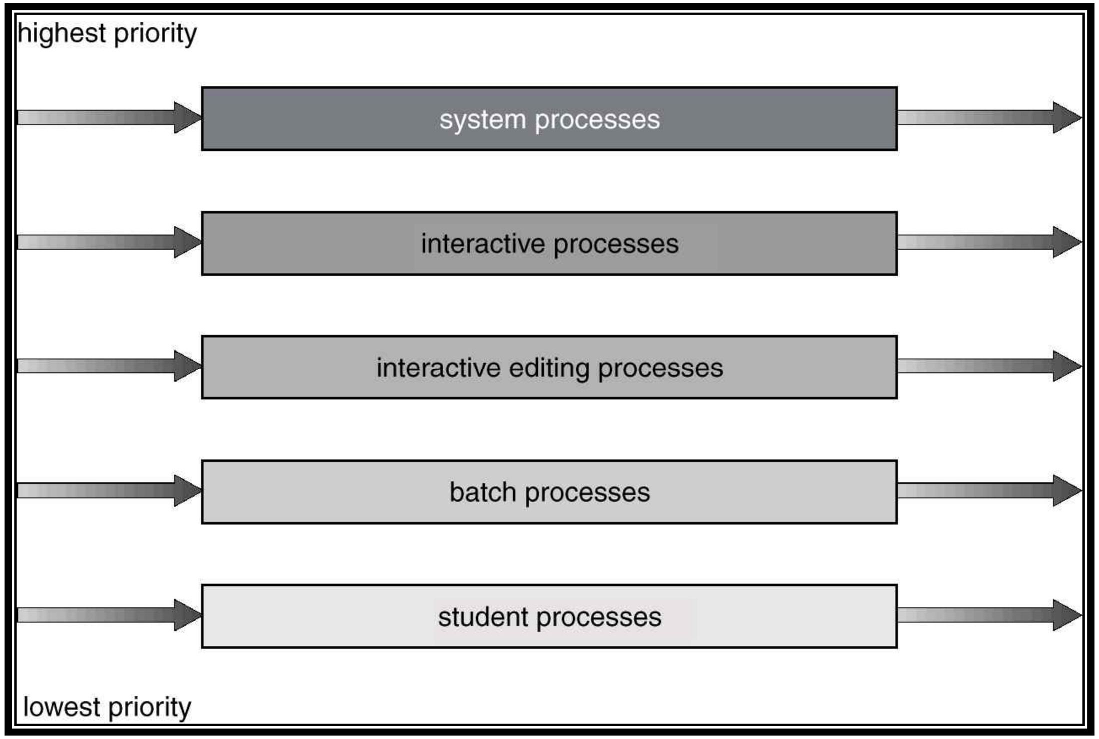
Multilevel Queue

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



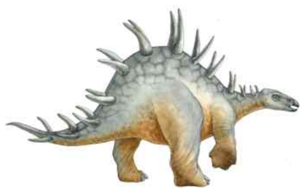
Multilevel Feedback Queue

Preemptive

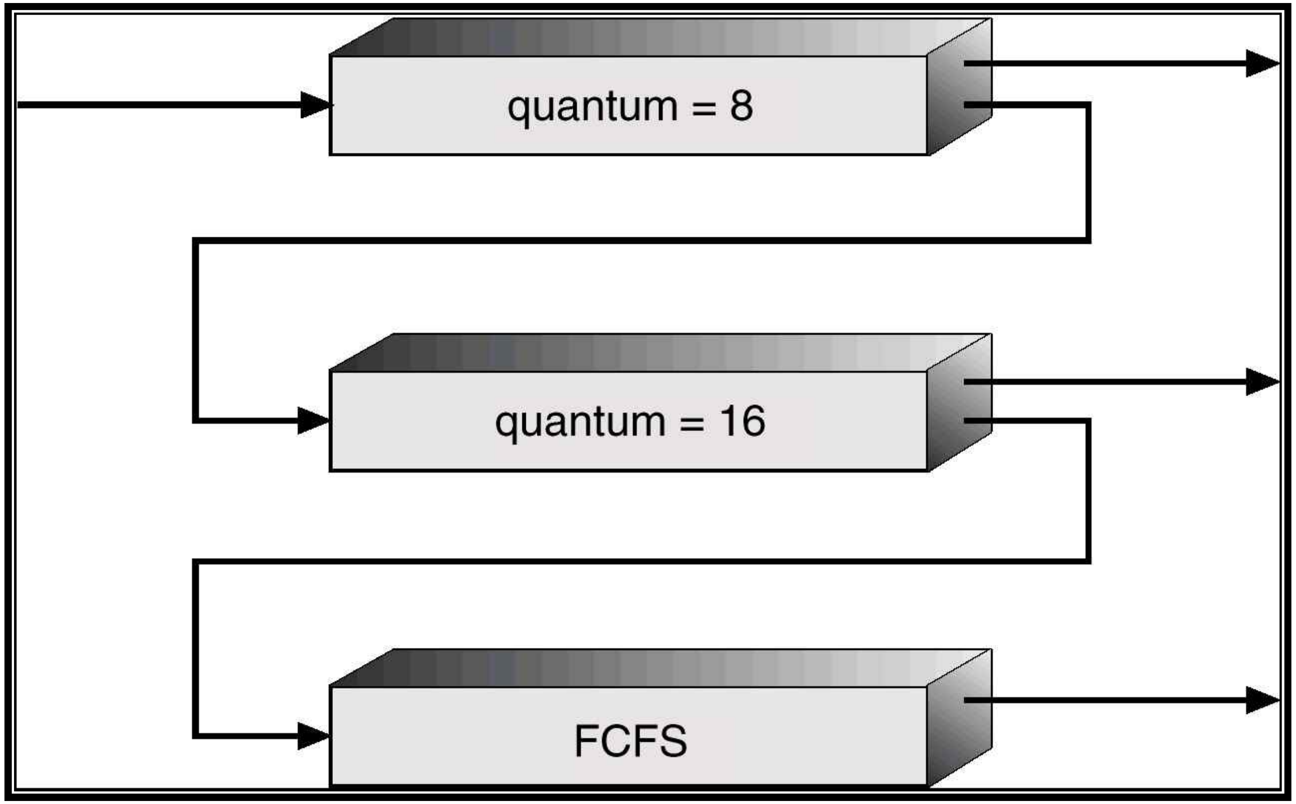
- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

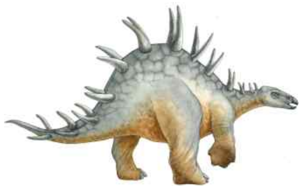
Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – 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.



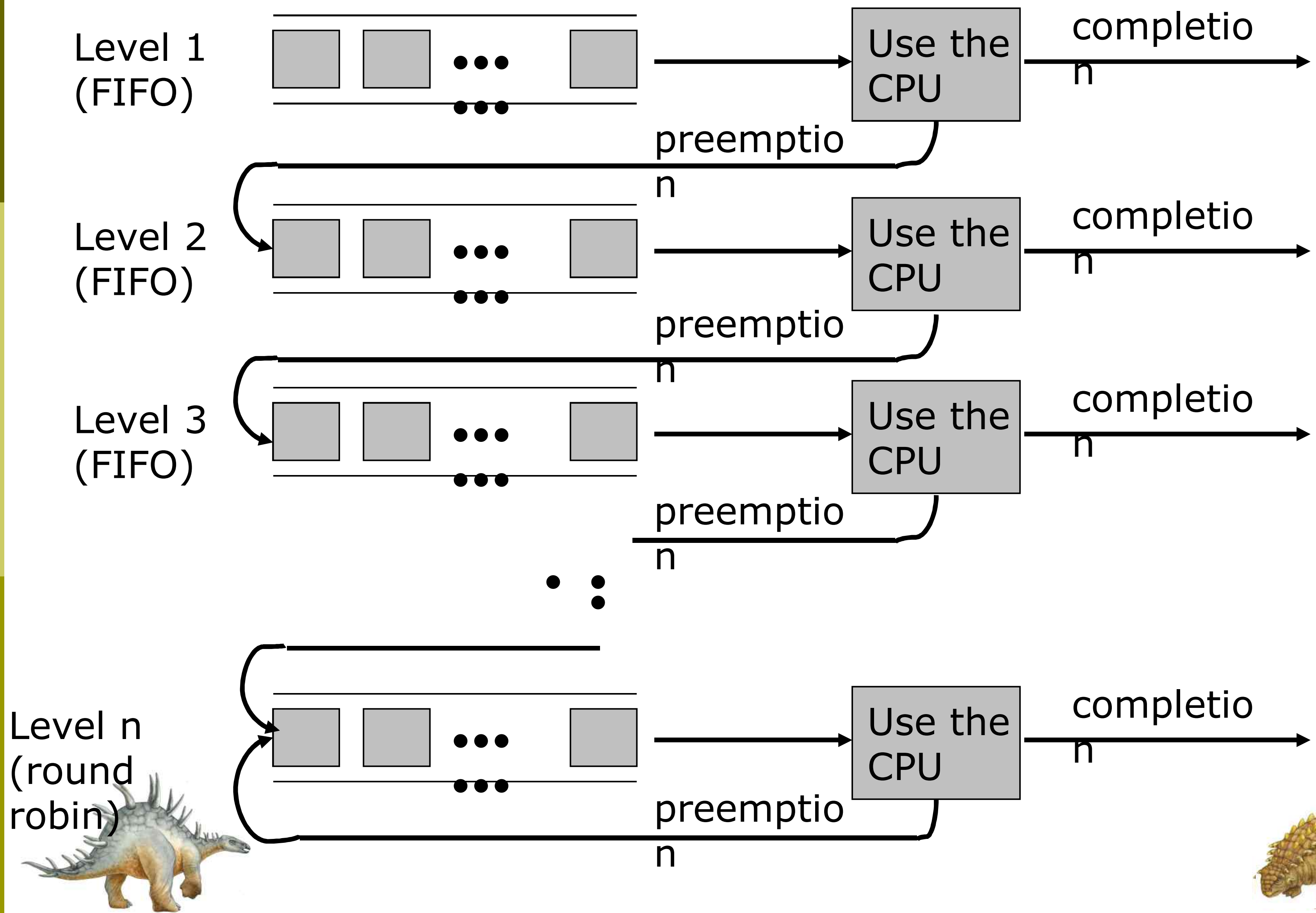
Multilevel Feedback Queues





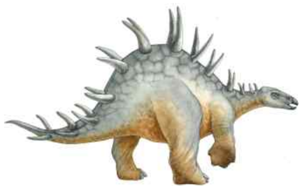
Multilevel Feedback Queue: Preemptive

- 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)

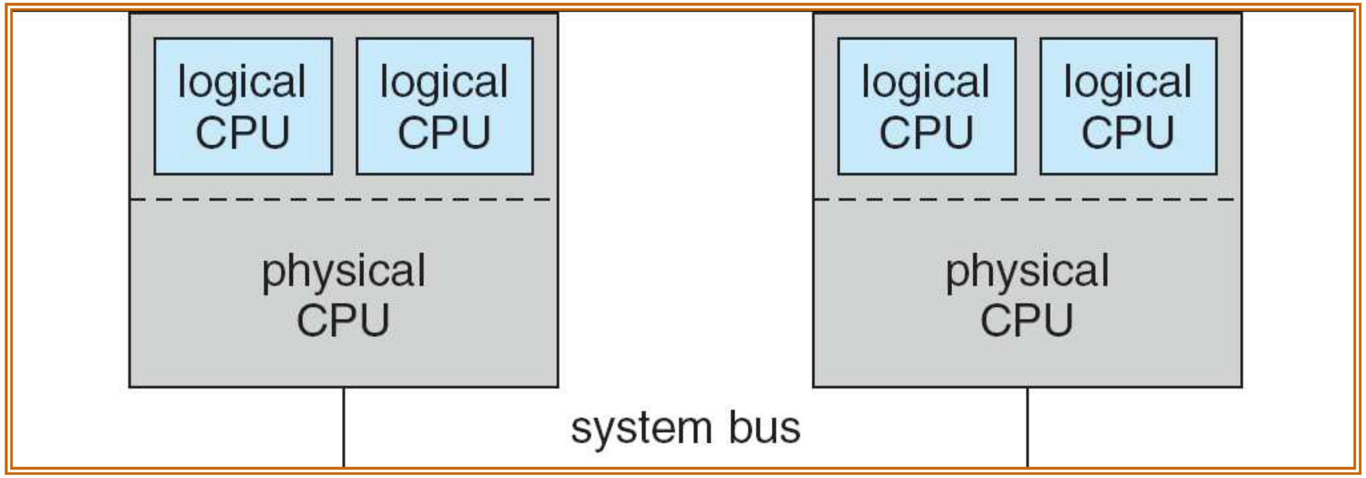

Level 1 (FIFO)
Level 2 (FIFO)
Level 3 (FIFO)
Level n (round robin)
Use the CPU
Use the CPU
Use the CPU
Use the CPU
completion
completion
completion
completion
preemption
preemption
preemption
preemption

Multiple-Processor Scheduling

- *Asymmetric multiprocessing*
  - *하나의 processor가 scheduling 하므로 자료 공유가 없음*
- *Symmetric multiprocessing(SMP)*
  - *각 processor가 독자적으로 scheduling*
  - *Load sharing* : 공동의 Ready Queue 사용 가능
- 처리기 친화성(Processor Affinity)
- Load Balancing
  - Push : 특정 태스크가 주기적으로 부하 검사
  - Pull : 쉬고 있는 프로세서에서 다른 프로세서의 load를 가져옴

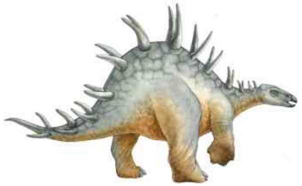


Typical SMT architecture

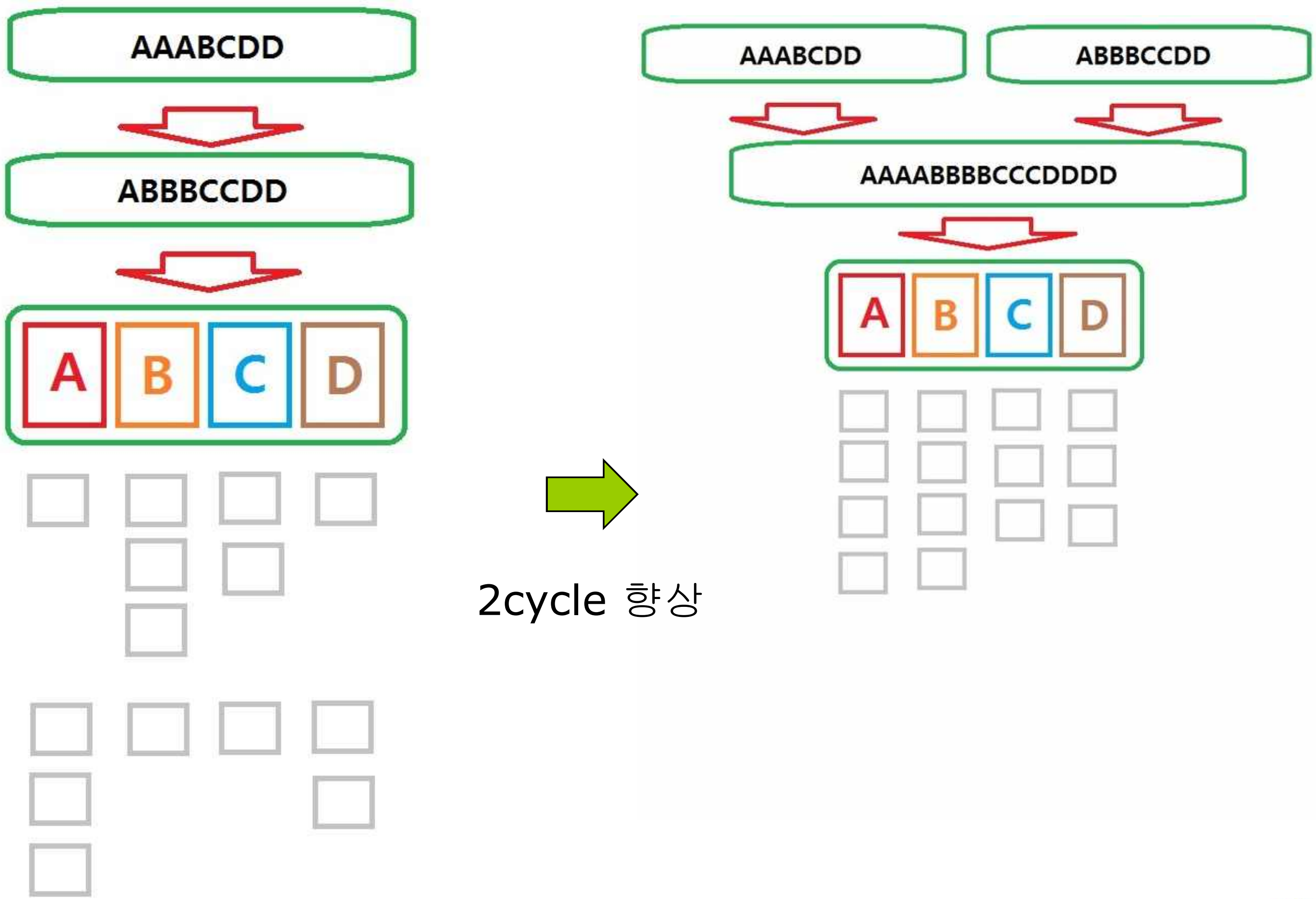


SMT : Symmetric multithreading
- provide multiple logical- rather than physical- processors

=>Hyperthreading



Hyperthreading : Best Case

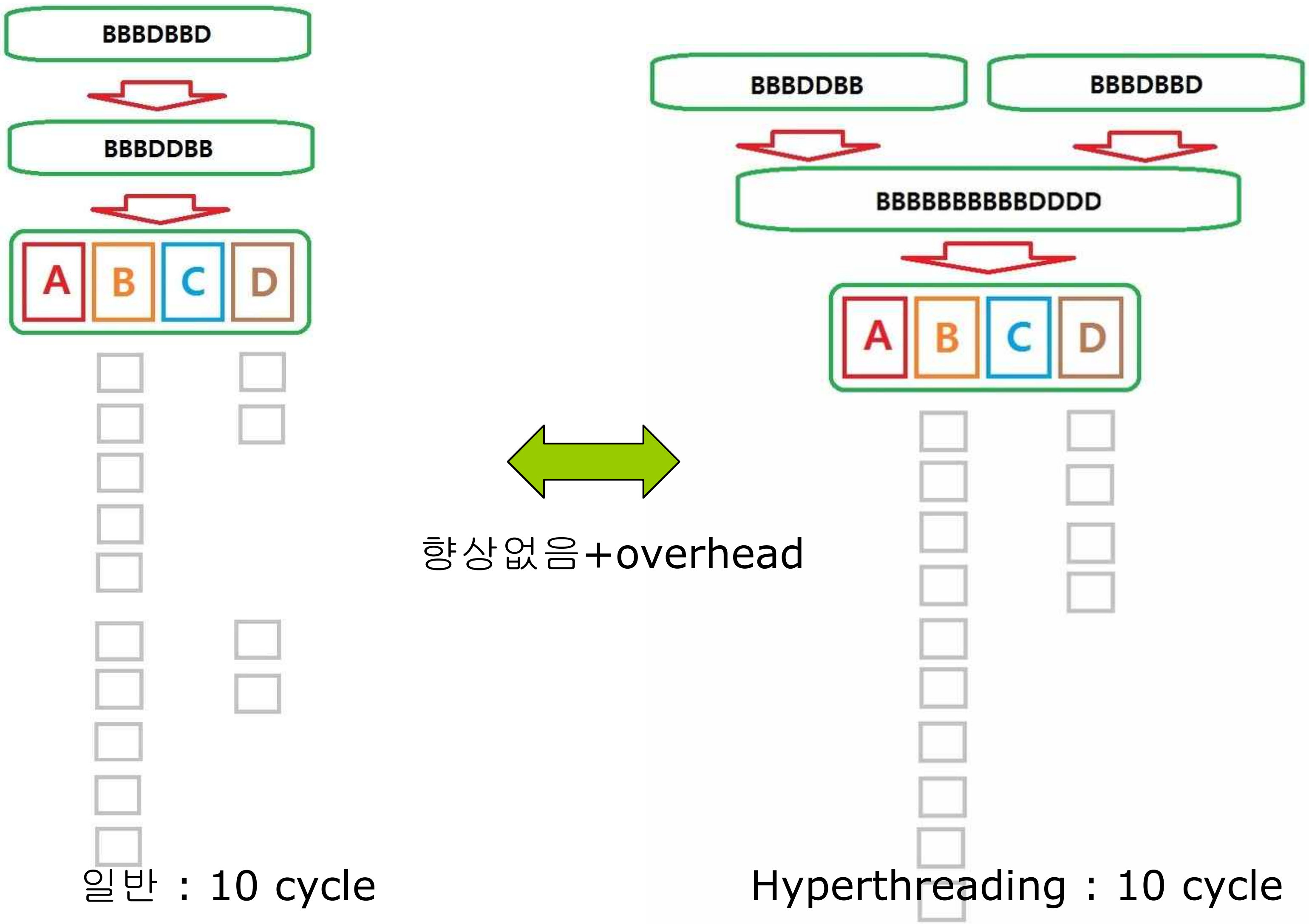


일반 : 6 cycle

Hyperthreading : 4 cycle

출처 : http://blog.naver.com/jky

Hyperthreading : Worst Case



향상없음+overhead

일반 : 10 cycle

Hyperthreading : 10 cycle

출처 : http://blog.naver.com/jky

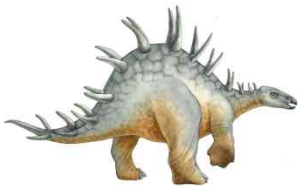
Deadline 스케쥴링 (기한부 스케쥴링)

- 각 job 이 마감시간을 가짐
- 각 job 이 마감시간내에 처리되도록 스케쥴
- 문제점: 구현이 거의 불가능
  - Deadline 을 사용자가 예측 불가능
  - 일부 사용자 희생
  - Overhead 가 큼

Thread Scheduling

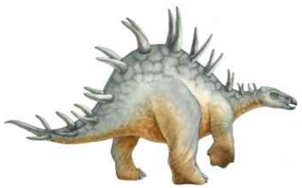
- Local Scheduling – How the threads library decides which thread to put onto an available LWP

- Global Scheduling – How the kernel decides which kernel thread to run next



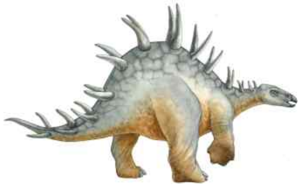
Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
         int i;
        pthread t tid[NUM THREADS];
        pthread attr t attr;
        /* get the default attributes */
        pthread attr init(&attr);
        /* set the scheduling algorithm to PROCESS or SYSTEM */
        pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
        /* set the scheduling policy - FIFO, RT, or OTHER */
        pthread attr setschedpolicy(&attr, SCHED OTHER);
        /* create the threads */
        for (i = 0; i < NUM THREADS; i++)
                pthread create(&tid[i],&attr,runner,NULL);
```



Pthread Scheduling API

```
    /* 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)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```



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

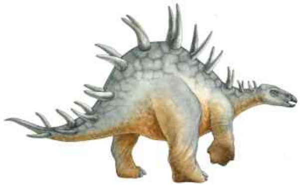
Time-Slicing

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

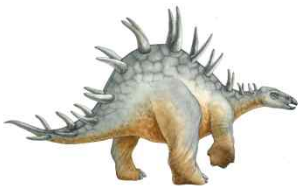


Thread Priorities

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);



Scheduler - TP

```
/**
 * Scheduler.java
*/
public class Scheduler extends Thread
{
  private CircularList queue;
  private int timeSlice;
  private static final int DEFAULT_TIME_SLICE = 1000; // 1초

  public Scheduler() {
    timeSlice = DEFAULT_TIME_SLICE;
    queue = new CircularList();
  }

  public Scheduler(int quantum) {
    timeSlice = quantum;
    queue = new CircularList();
  }
```

Scheduler - TP

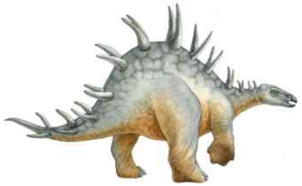
```
// adds a thread to the queue
 public void addThread(Thread t) {
   t.setPriority(2);
   queue.addItem(t);
 }

 // this method puts the scheduler to sleep for a time
   quantum
 private void schedulerSleep() {
   try {
     Thread.sleep(timeSlice);
   } catch (InterruptedException e)
   {};
 }
```

Scheduler - TP

```
public void run() {
   Thread current;
   // set the priority of the scheduler to the highest priority
   this.setPriority(6);

   while (true) {
     current = (Thread)queue.getNext();
     if ( (current != null) &&
          (current.isAlive()) {
       current.setPriority(4);
       schedulerSleep();
       current.setPriority(2);
     }
   }
  }
}
```



```java
/**
 * TestScheduler.java
 * This program demonstrates how the scheduler operates.
 * This creates the scheduler and then the three example threads.
 */

public class TestScheduler
{
  public static void main(String args[]) {
  /**
   * This must run at the highest priority
   * to ensure that it can create the scheduler and the example
   * threads. If it did not run at the highest priority, it is
   * possible that the scheduler could preemt this and not allow
   * it to create the example threads.
   */
```

```java
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);

 scheduler CPUScheduler = new scheduler();
 CPUScheduler.start();

 TestThread t1 = new TestThread("Thread 1");
 t1.start();
 CPUScheduler.addThread(t1);

 TestThread t2 = new TestThread("Thread 2");
 t2.start();
 CPUScheduler.addThread(t2);

 TestThread t3 = new TestThread("Thread 3");
 t3.start();
 CPUScheduler.addThread(t3);
}
}
```