5. CPU Scheduling

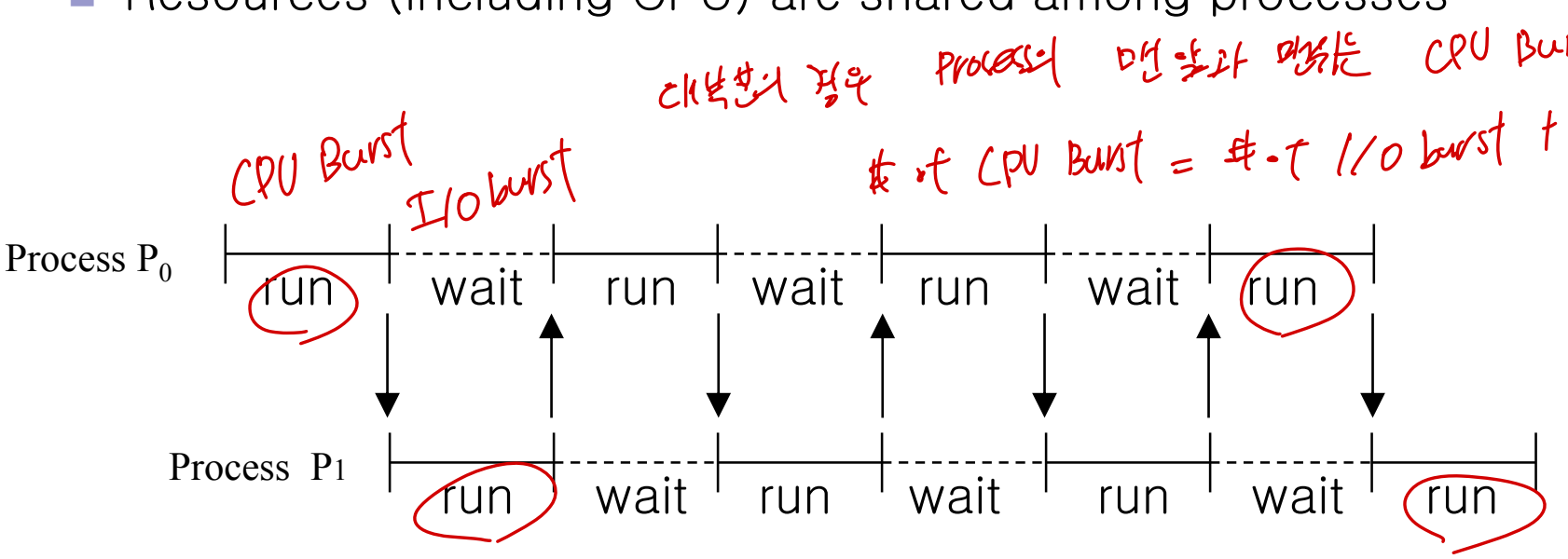
ECE30021/ITP30002 Operating Systems

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Real-time CPU scheduling
- Operation system examples
- Algorithm evaluation

Basic Concepts

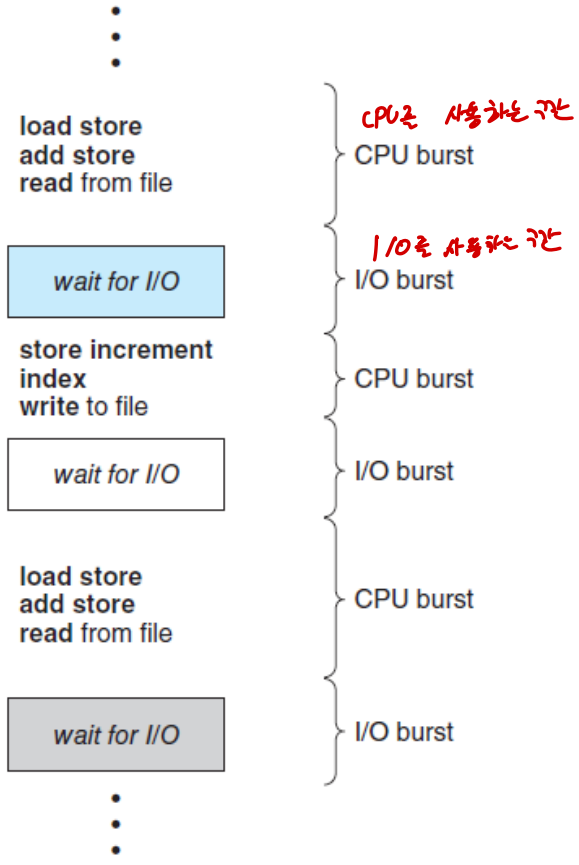
- Motivation: maximum CPU utilization obtained with multiprogramming and multitasking
  - Resources (including CPU) are shared among processes



Multiprogramming

CPU–I/O Burst Cycle

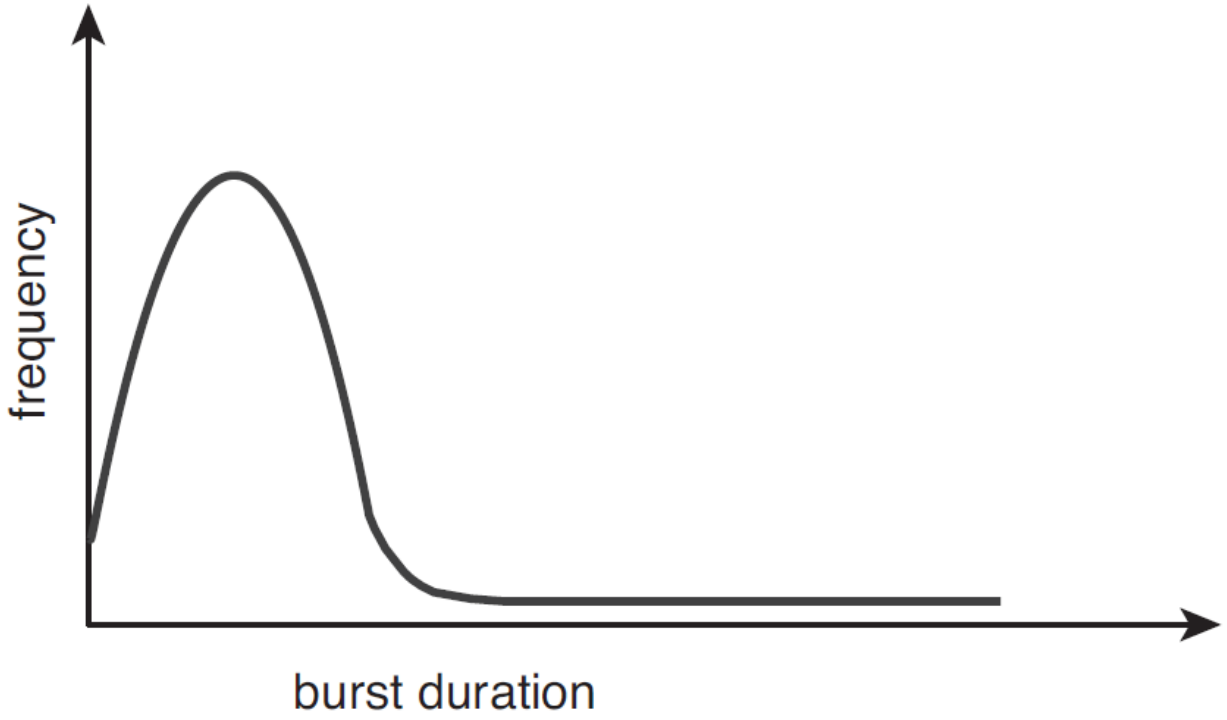
- Process execution consists of cycle of CPU execution and I/O wait
  - First and last bursts are CPU bursts
- Types of processes
  - I/O-bound process
    - Consists of many short CPU bursts
  - CPU-bound process
    - Consists of a few long CPU bursts



Histogram of CPU-burst Durations

- Exponential or hyper exponential

대부분의 CPU-burst의 길이는 짧다 => I/O bound process가 많다



CPU Scheduler

- CPU scheduler (= short term scheduler)
  - Selects a process from ready queue and allocate CPU to it

- Implementation of ready queue
  - FIFO, priority queue, tree, unordered linked list
    - Each process is represented by PCB

Preemptive Scheduling

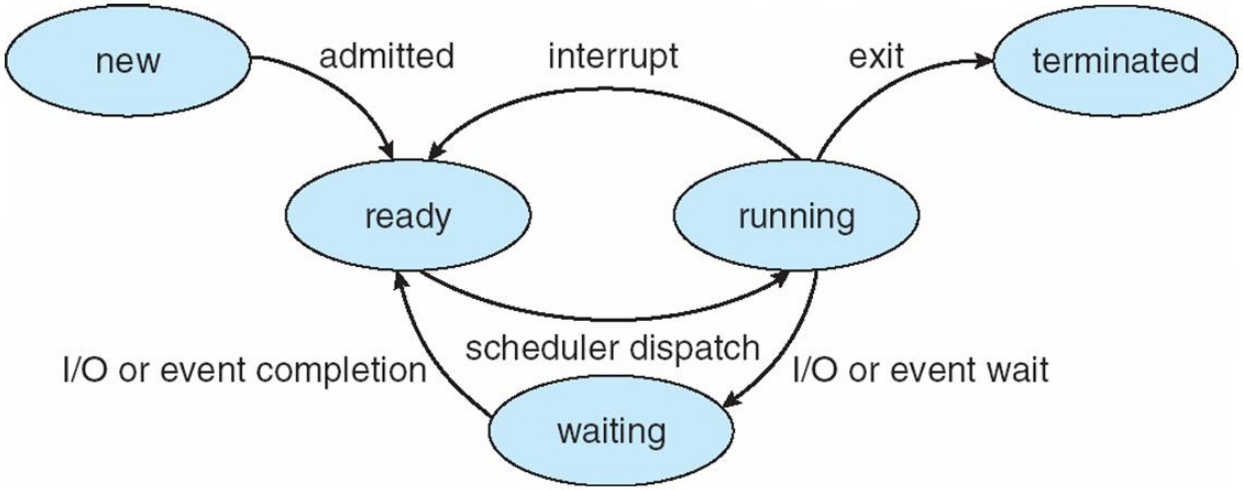
- Preemptive scheduling: scheduling may occur while a process is running.

  Ex) interrupt, process with higher priority

When Scheduling Occurs?

1. When a process switches from running state to waiting state 필수
2. When a process switches from running state to ready state optional
   ex) an interrupt occurs
3. When a process switches from waiting state to ready state optional
   ex) completion of I/O I/O를 마친 process가 running state의 process보다 우선순위가 높은경우
4. When a process terminates

-> 1 and 4 are inevitable, but 2 and 3 are optional



Preemptive Scheduling

- Non-preemptive (or cooperative) scheduling
  - Scheduling can occur at 1, 4 only
  - Running process is not interrupted.

- Preemptive scheduling
  - Scheduling can occurs at 1, 2, 3, 4
  - Scheduling may occur while a process is running
  - Requires H/W support and shared data handling

  - Preemptive scheduling can result in race conditions when data are shared among several processes

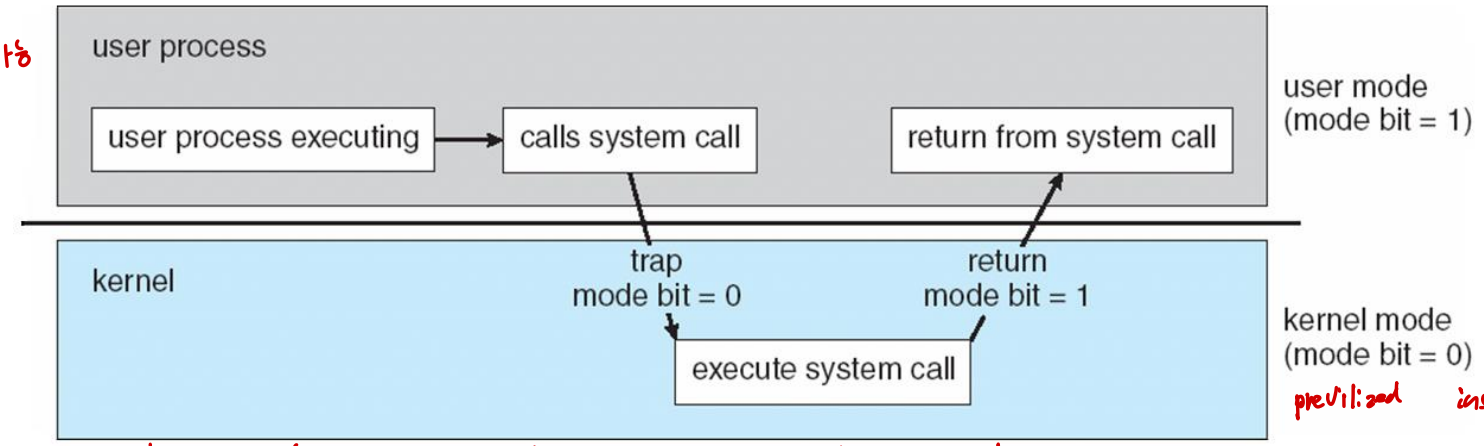
2개 이상의 thread나 process가 돌 때, 순서에 따라 최종적으로 에러나 날 때

ㄴ 극악한 버그, 잡기 어려움

synchronization으로 막음

Preemption of OS Kernel

- **Preemptive kernel**: kernel allows preemption in system call.
  - Desirable for real-time support.

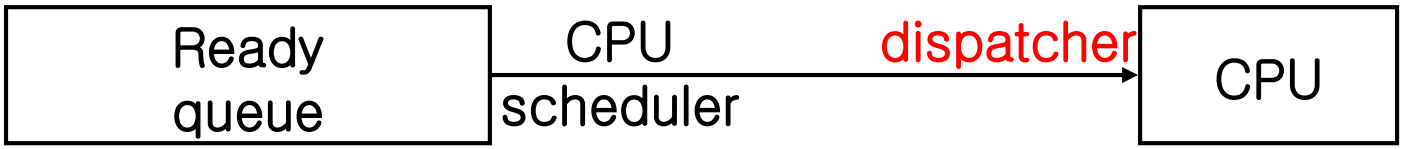


cf. System calls are non-preemptive in most OS.
  - Keeps kernel structure simple

Dispatcher

- Dispatcher: a module that gives control of CPU to the process selected by short-term scheduler
  - Switching context
  - Switching to user mode
  - Jumping to proper location in user program

- Dispatch latency: time from stopping one process to starting another process



Scheduling Criteria

- CPU utilization: keep the CPU as busy as possible
- Throughput: # of processes completed per time unit
- Turnaroundtime: interval from submission of a process to its completion
- Waiting time: sum of periods spent waiting in ready queue
- Response time: time from submission of a request to first response

-> Importance of each criterion vary with systems

- Measure to optimize
  - Average / minimum / maximum value
  - Variance

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Real-time CPU scheduling
- Operation system examples
- Algorithm evaluation

Scheduling Algorithms

- First-come, first-served (FCFS) scheduling
- Shortest-job-first (SJF) scheduling
- Priority scheduling
- Round-robin scheduling
- Multilevel queue scheduling
- Multiple feedback-queue scheduling

First-Come, First-Served Scheduling

- Process that requests CPU first, is allocated CPU first
  - Non-preemptive scheduling
  - Simplest scheduling method

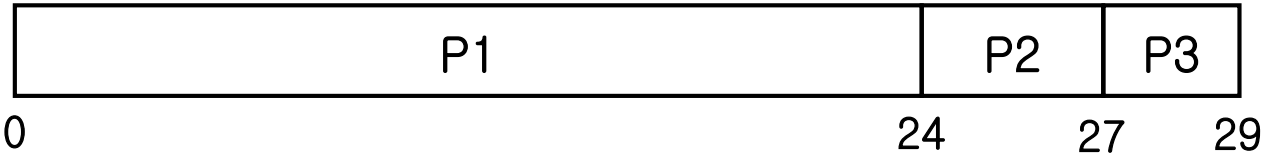
- Sometimes average waiting time is quite long
  - CPU, I/O utilities are inefficient

Ex) Three processes arrived at time 0

Process	Burst Time	Waiting time
P1	24	0
P2	3	24
P3	2	27

*time unit: msec

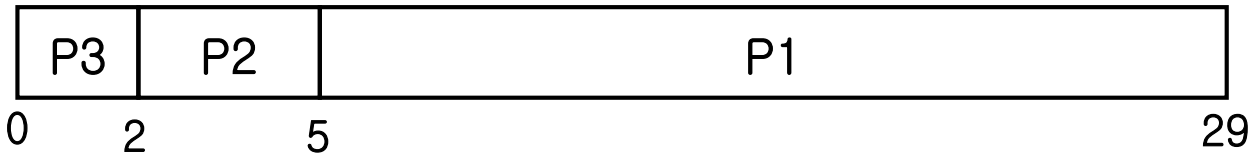
Average waiting time:
(0 + 24 + 27)/3 = 17ms



Shortest-Job-First Scheduling

- Assign to the process with the smallest next CPU burst

Process	Burst Time	Waiting time
P1	24	5
P2	3	2
P3	2	0



Average waiting time:

(5 + 2 + 0)/3 = 2.3 ms

waiting time optimal

↳ burst time을 알아야하므로 정확하게 알수가 없다.

↳ history 분석을 예측은 가능

- SJF algorithm is optimal in minimum waiting time
- Problem: difficult to know length of next CPU burst

Shortest-Job-First Scheduling

- Predicting next CPU burst from history
  - Exponential averaging

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

: weighted average

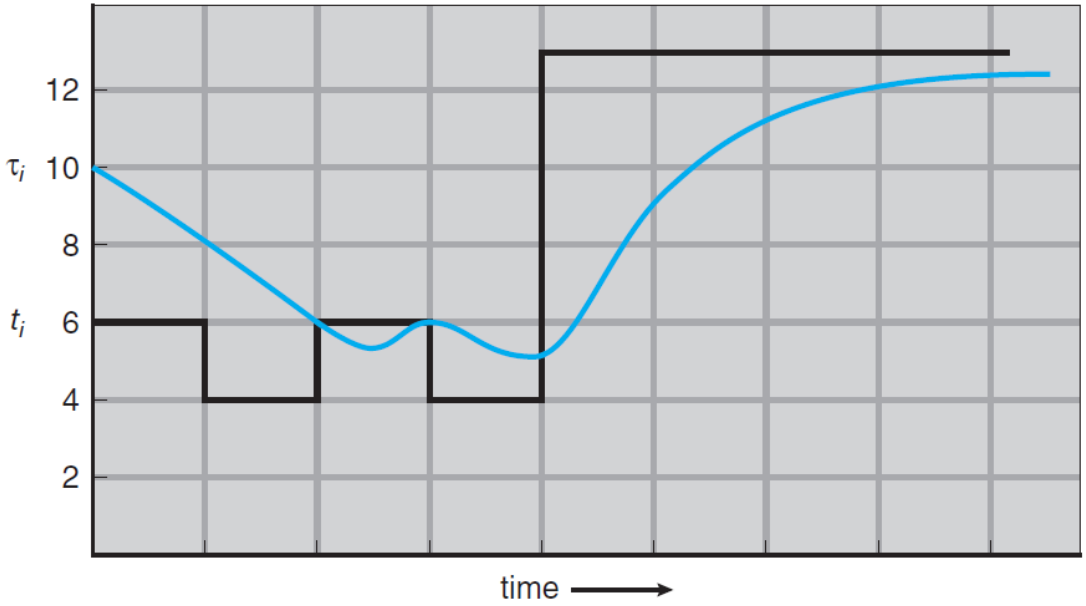
    - t_n : actual length of n-th CPU burst
    - τ_n : prediction for n-th CPU burst
    - α : a coefficient between 0 and 1
      - α = 0: recent history has no effect
      - α = 1: only recent history matters
      - Usually, α = 0.5

Note: older history affects less

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

Prediction of CPU Burst

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

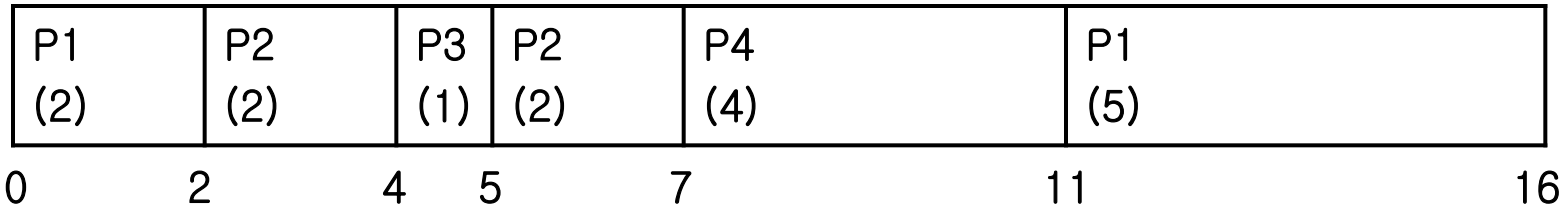


CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Shortest-Job-First Scheduling

- Preemptive version of SJF scheduling
  - Shortest-remaining-time-first scheduling (끝나는시간 - 시작시간) - 실행시간

Process	Arrival Time	Burst Time	Waiting Time
P1	0	~~7~~ 8 0	9
P2	2	~~4~~ 2 0	1
P3	4	~~1~~ 0	0
P4	5	~~4~~ 0	2



Average waiting time: 3

Priority Scheduling

- CPU is allocated the process with the highest priority
  - Equal-priority processes: FCFS

Note: each process has its priority

- In this text, lower number means higher priority

Ex)

Process	Burst Time	Priority	Waiting Time
P1	10	3	6
P2	1	1	0
P3	2	4	16
P4	1	5	18
P5	5	2	1

Average waiting time: 8.2

Priority Scheduling

I/O bound process가 CPU bound process보다 대체로 우선순위가 높다.

- Priority can be assigned internally and externally
  - Internally: determined by measurable quantity or qualities
    - Time limit, memory requirement, # of open files, ratio of I/O burst and CPU burst, …
  - Externally: importance, political factors
- Priority scheduling can be preemptive or non-preemptive.
- Major problems
  - Indefinite blocking (= starvation) of processes with lower priorities 우선순위가 낮은 프로세스가 계속 CPU time을 못받는 상황

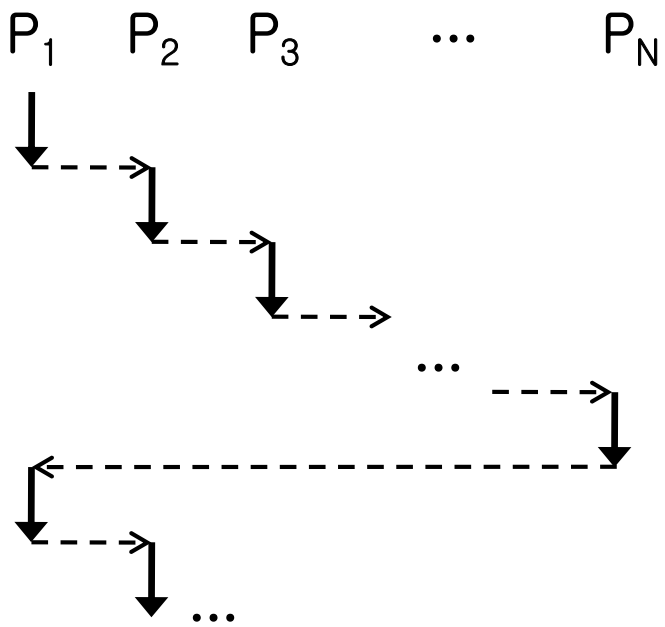
  -> Solution: aging (gradually increase priority of processes waiting for long time)

Round-Robin Scheduling

- Similar to FCFS, but it's preemptive
  - Designed for time-sharing systems
  - CPU time is divided into time quantum (or time slice)
    - A time quantum is 10~100 msec.

    Cf. switching latency: 10 μsec msec = 1000 μsec
  - Ready queue is treaded as circular queue
  - CPU scheduler goes around the ready queue and allocate CPU time up to 1 time quantum

Round-Robin Scheduling

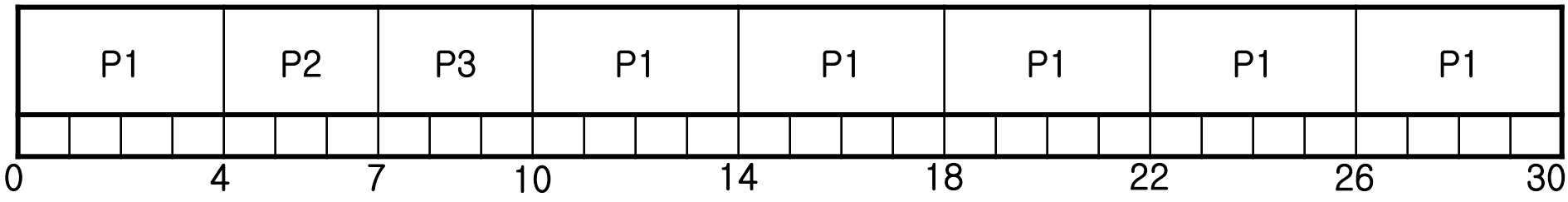


Round-Robin Scheduling

Ex) Time quantum = 4

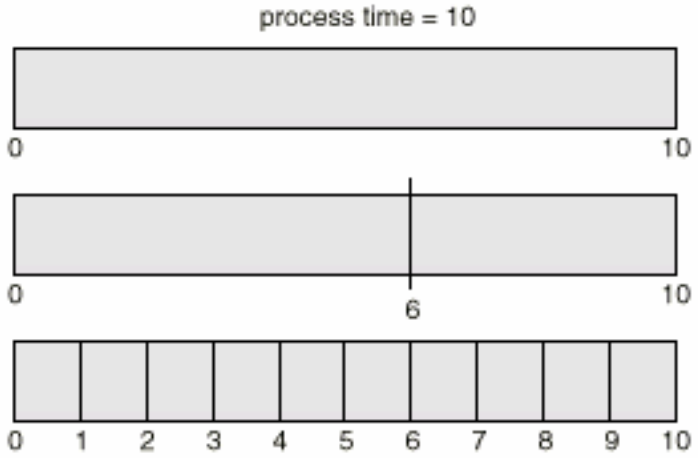
Process	Burst Time	Waiting Time
P1	24	6
P2	3	4
P3	3	7

Average waiting time: 5.66



Round-Robin Scheduling

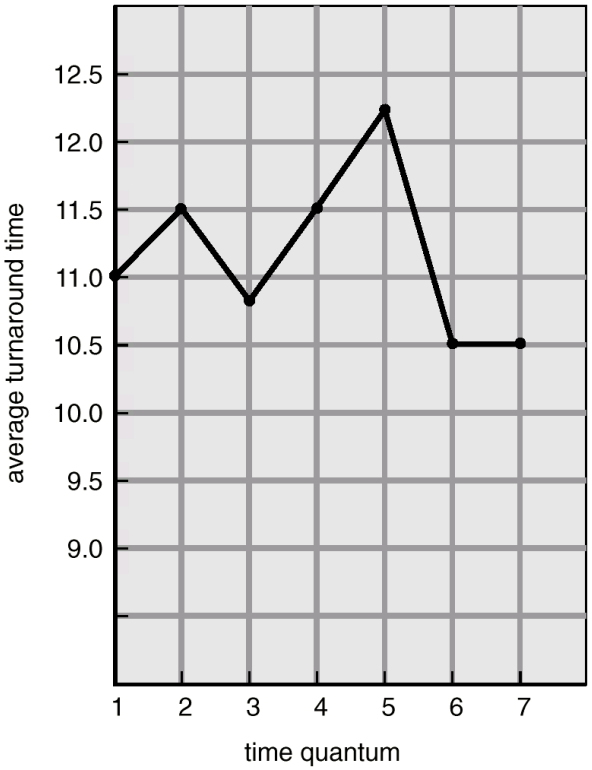
- Performance of RR scheduling heavily depends on size of time quantum.
  - Time quantum is small: processor sharing
  - Time quantum is large: FCFS
- Context switching overhead depends on size of time quantum.



quantum	context switches
12	0
6	1
1	9

Round-Robin Scheduling

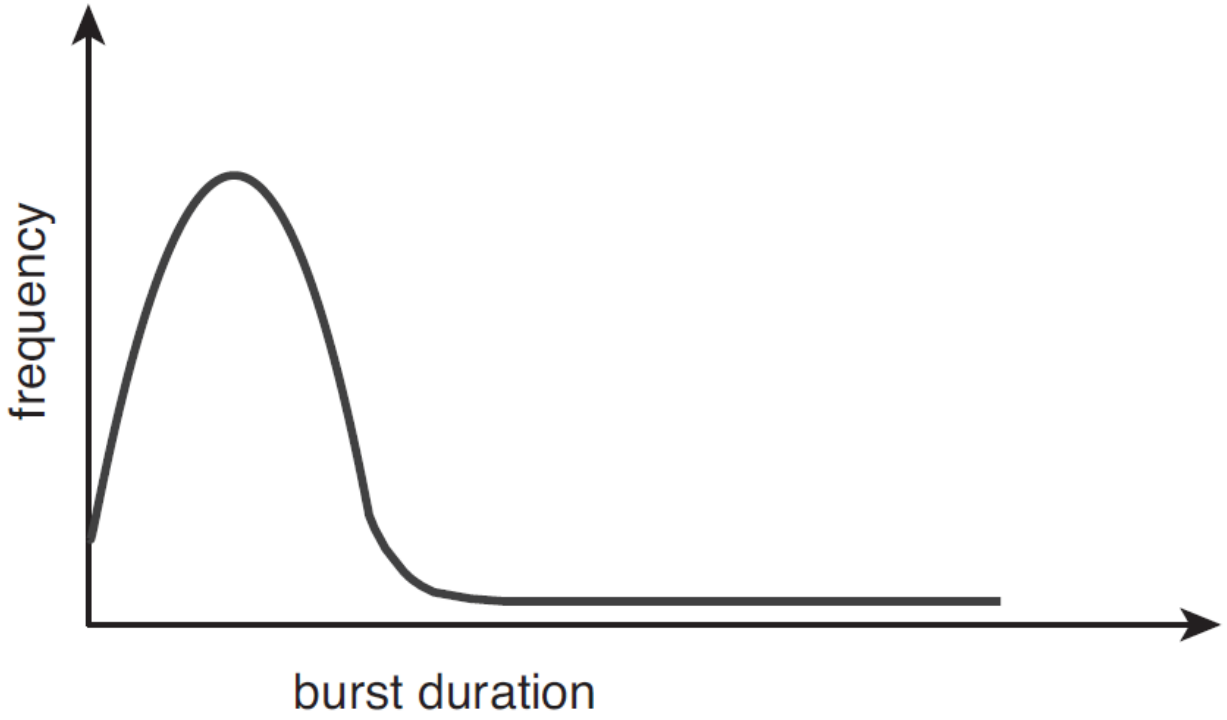
- Turnaroundtime also depends on size of time quantum
  - Average turnaround time is not proportional nor inverse-proportional to size of time quantum
  - Average turnaroundtime is improved if most processes finish their next CPU burst in a single time quantum
  - However, too long time quantum is not desirable
  - A rule of thumb: about 80% of CPU burst should be shorter than time quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

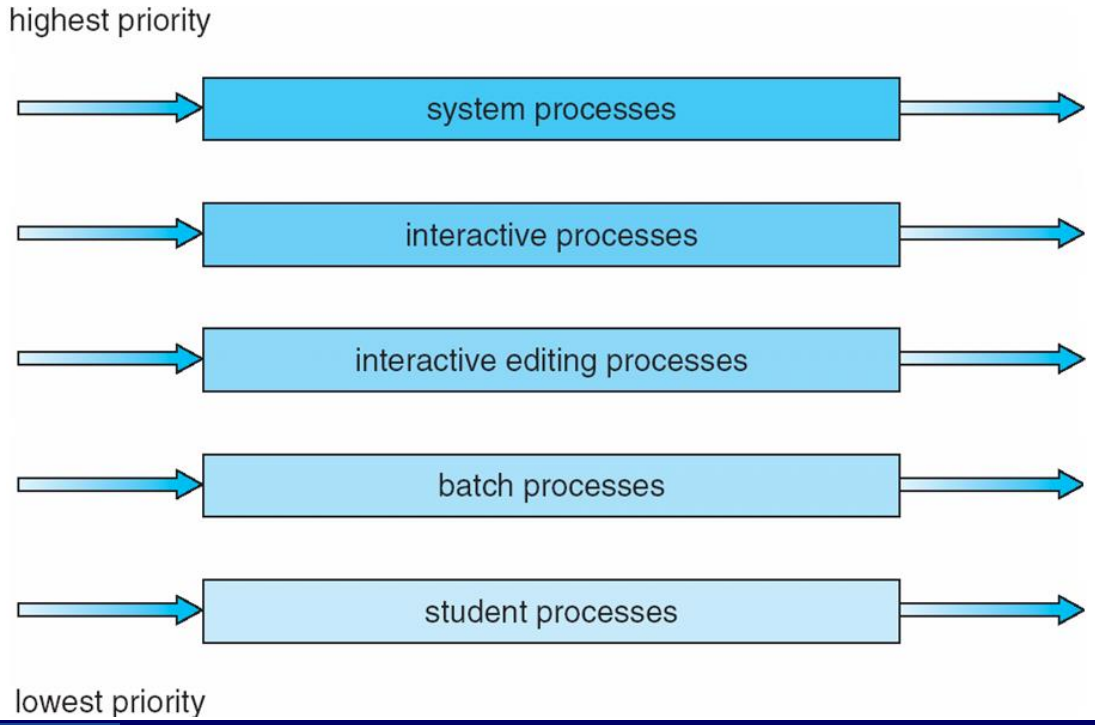
Histogram of CPU-burst Durations

- Exponential or hyper exponential



Multilevel Queue Scheduling

- Classify processes into different groups and apply different scheduling
  - Memory requirement, priority, process type, …
- Partition ready queue into several separate queues



Multilevel Queue Scheduling

- Each queue has its own scheduling algorithm
- Scheduling among queues
  - Fixed-priority preemptive scheduling
    - A process in lower priority queue can run only when all of higher priority queues all empty
  - Time-slicing among queues

    Ex) foreground queue (interactive processes): 80%

    background queue (batch processes): 20%
- Assignment of a queue to a process is permanent.

Multilevel Feedback-Queue Scheduling

- Similar to multilevel queue scheduling, but a process can move between queues.

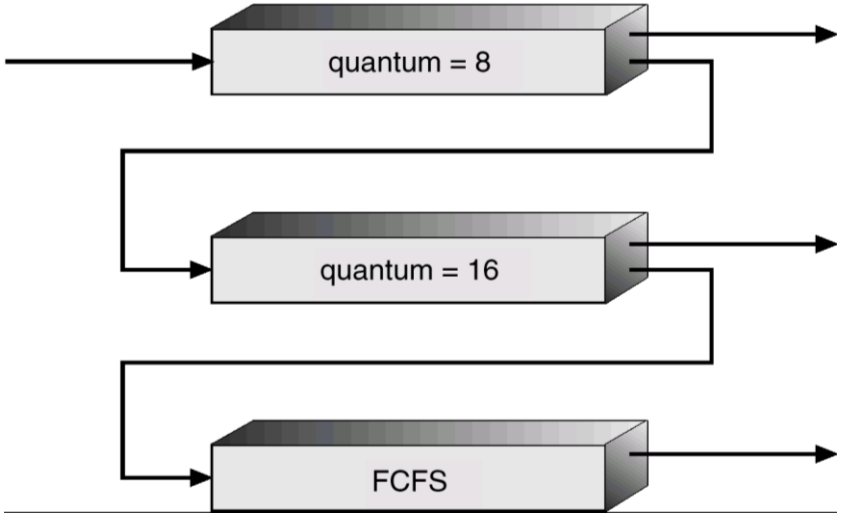
- Idea: separate processes according to characteristics of their CPU bursts.
  - If a process uses too much CPU time, move it to lower priority queue.
  - I/O-bound, interactive processes are in higher priority queues.

Multilevel Feedback-Queue Scheduling

Ex) Ready queue consists of three queues (0~2)

- Q_0 (time limit = 8 milliseconds)
- Q_1 (time limit = 16 milliseconds)
- Q_2 (FCFS)

➔ A new process is put in Q_0. if it exceeds time limit, it moves to lower priority queue



Multilevel Feedback-Queue Scheduling

- Parameters to define a multilevel feedback-queue scheduler
  - # of queues
  - Scheduling algorithm for each queue
  - Method to determine when to upgrade a process to higher priority queue
  - Method to determine when to demote a process to lower priority queue
  - Method to determine which queue a process will enter when it needs service

➔ The most complex algorithm

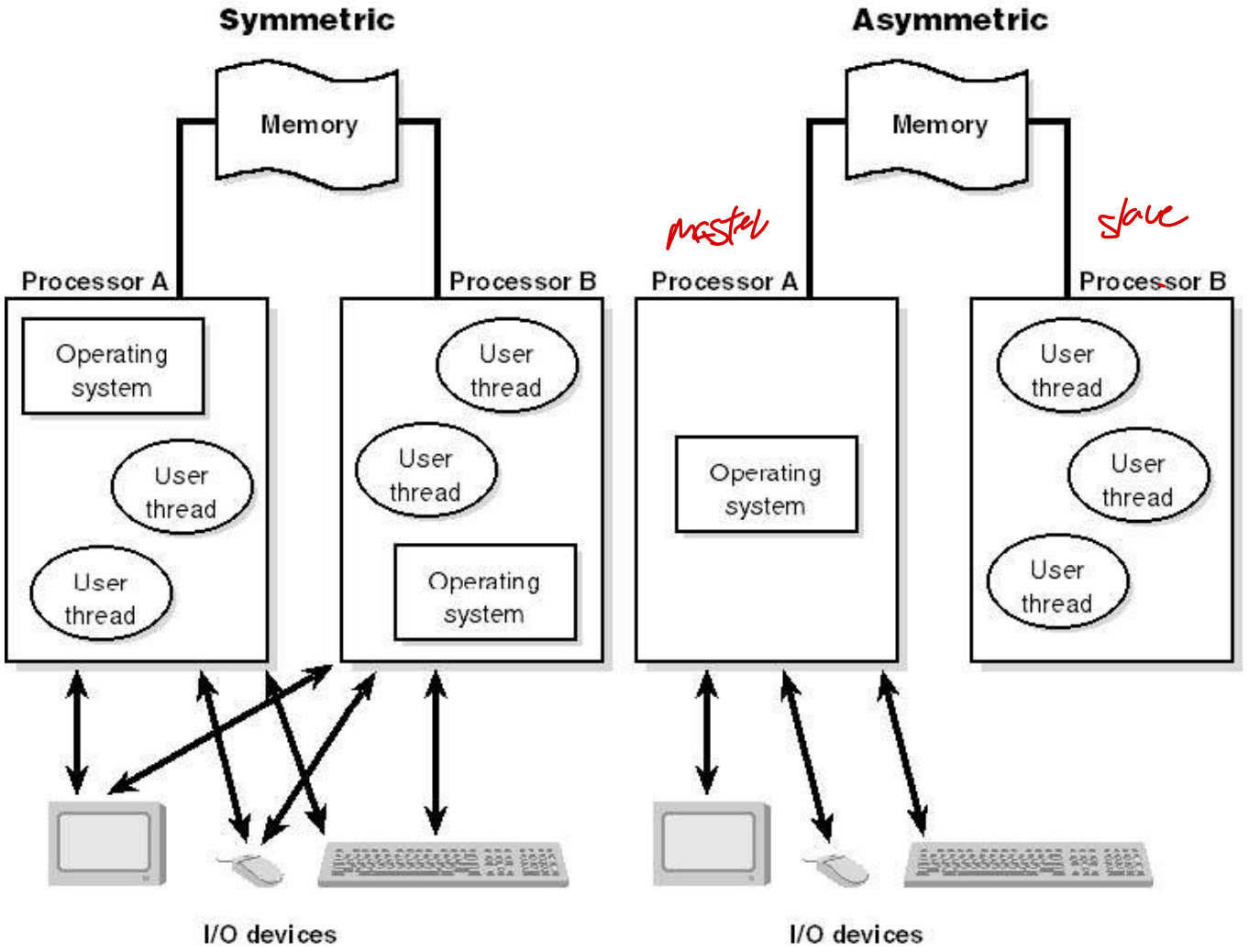
Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Real-time CPU scheduling
- Operation system examples
- Algorithm evaluation

Multiple-Processor Scheduling

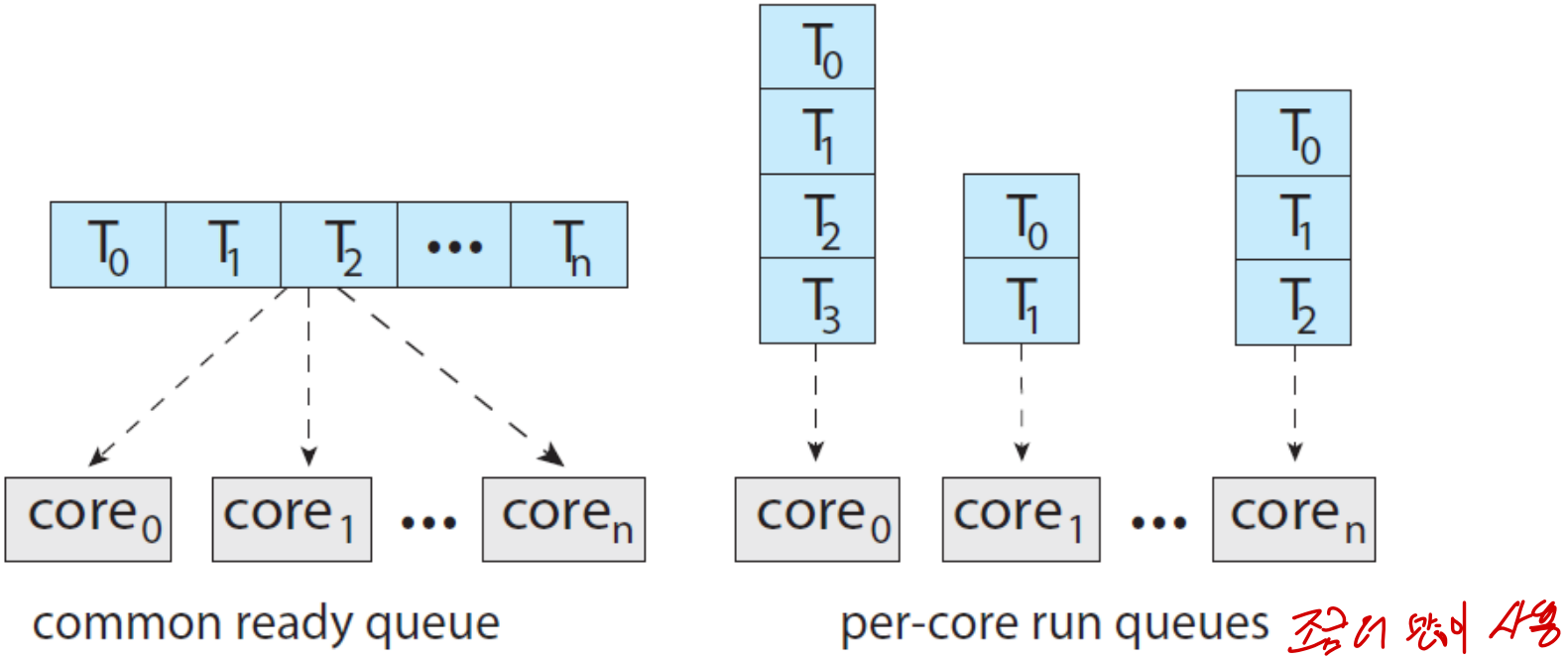
- Multiple-processor system
  - Load sharing is possible.
  - Scheduling problem is more complex.
- There are many trials in multiple-processor scheduling, but no generally best solution
- In this text, all processors are assumed identical.
  - Any process can run on any processor

Symmetric vs. Asymmetric Multiprocessing



Multi-processor Scheduling

- Two possible strategies
  1. All threads may be in a common ready queue.
     - Needs to ensure two processors do not choose the same thread nor lost from the queue
  2. Each processor may have its own private queue of threads.



친화성, 친화력

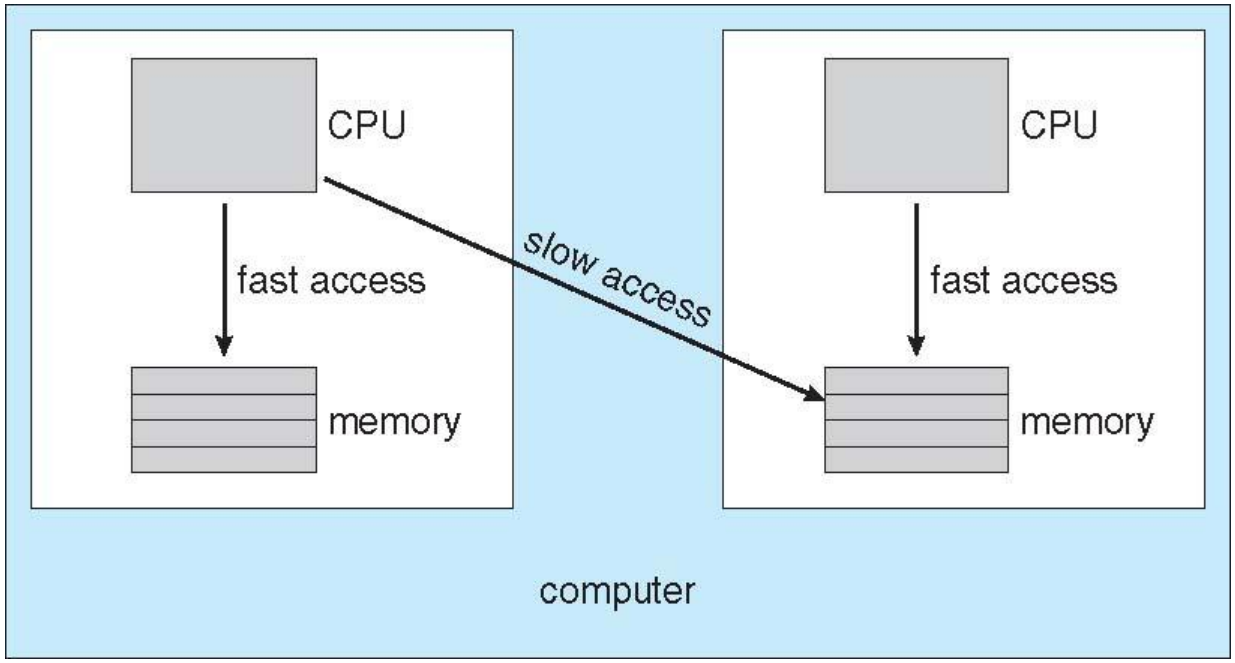
Processor Affinity

- Overhead of migration of processes from one processor to another processor
  - All contents of cache should be invalidated and re-populated

- Processor affinity: keeping a process running on the same processor to avoid migration overhead
  - **Soft affinity**: Although OS attempts to keep a process running on the same processor, a process may migrate between processors.
  - **Hard affinity**: It is possible to prevent a process from being migrated to other processor

Processor Affinity

- NUMA(Non-Uniform Memory Access) and CPU scheduling
  - A CPU has faster access to some parts of main memory than to other parts



Load Balancing

- Load balancing: attempt to keep the workload evenly distributed across all processors.
  - Necessary for system where each processor has its own ready queue (It's true for most modern OS's)

- Two general approaches
  - Push migration
    - A specific task periodically checks the load on each processor
    - If unbalance is found, move processes to idle or less-busy processors
  - Pull migration
    - An idle processor pulls a waiting task

  Note! Push and pull migration can be implemented in parallel.

  -> Linux, ULE scheduler for FreeBSD

- Load balancing can counteract the benefit of processor affinity

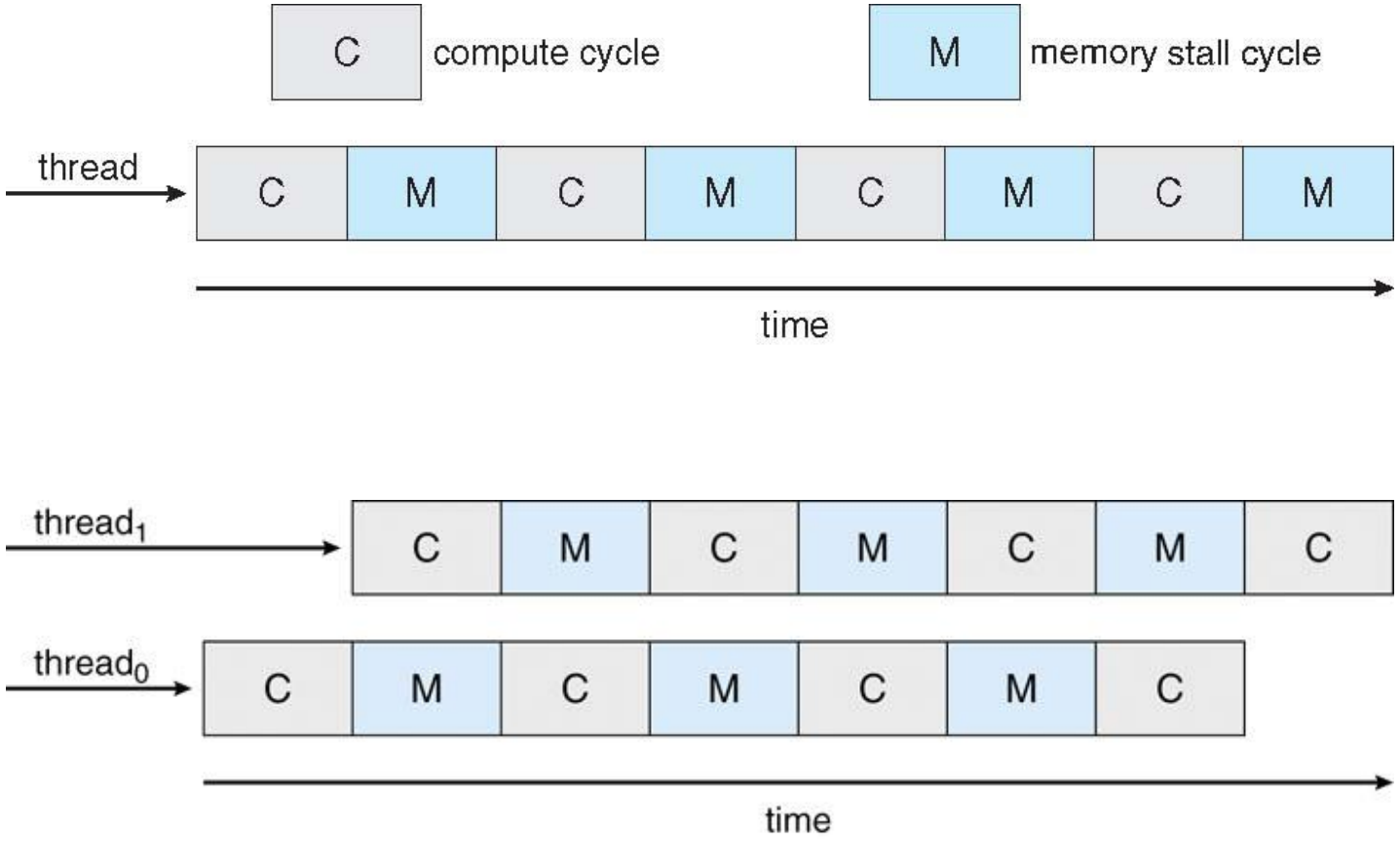
Multi-core Processors

- Multi-core processor: multiple processor cores on the same physical chip
  - Recent trend
  - Faster and consume less power than systems in which each processors has its own physical chip

- Scheduling issues on multi-core processor
  - Memory stall: for various reasons, memory access spends significant amount time waiting for the data to become available.

    Ex) accessing data not in cache
  - Remedy: multithreaded processor cores
    - Two or more hardware threads are assigned to each core

Multi-Threaded Processor Cores



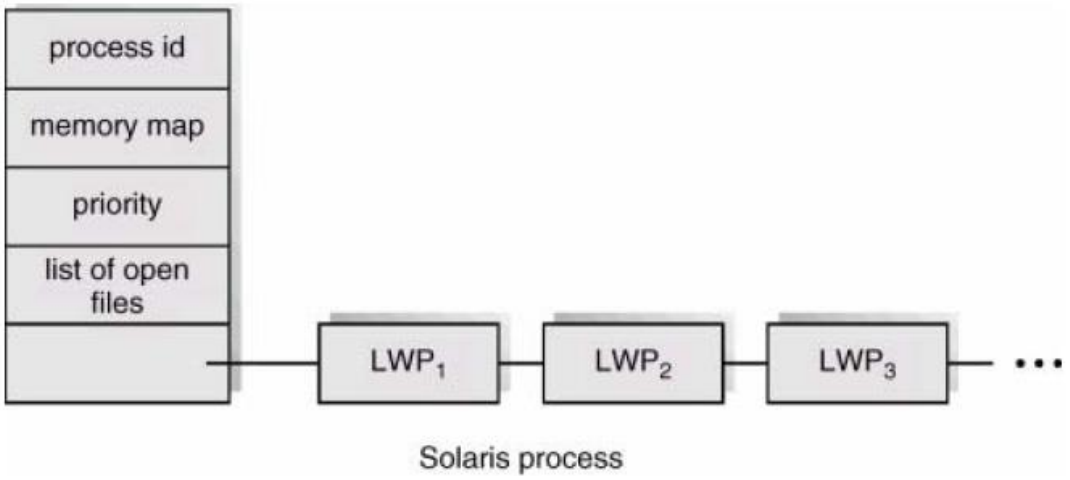
Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Real-time CPU scheduling
- Operation system examples
- Algorithm evaluation

Thread Scheduling

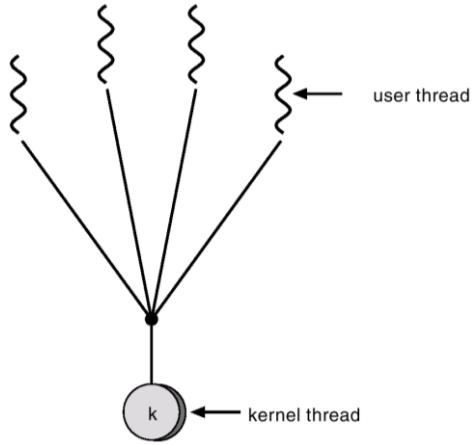
- Threads
  - User thread -> supported by thread library
    - Scheduled indirectly through LWP
  - Kernel thread -> supported by OS kernel

- Actually, it is not processes but kernel threads that are being scheduled by OS

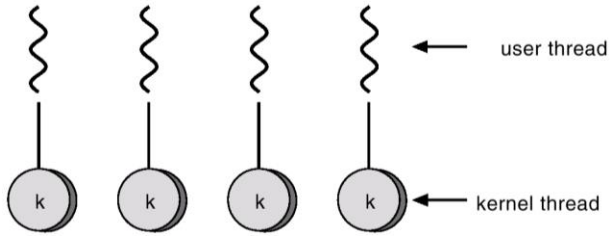


Contention Scope

- Process-contention scope (PCS)
  - Competition for LWP among user threads
    - Many-to-one model or many-to-many model
  - Priority based
- System-contention scope (SCS)
  - Competition for CPU among kernel threads



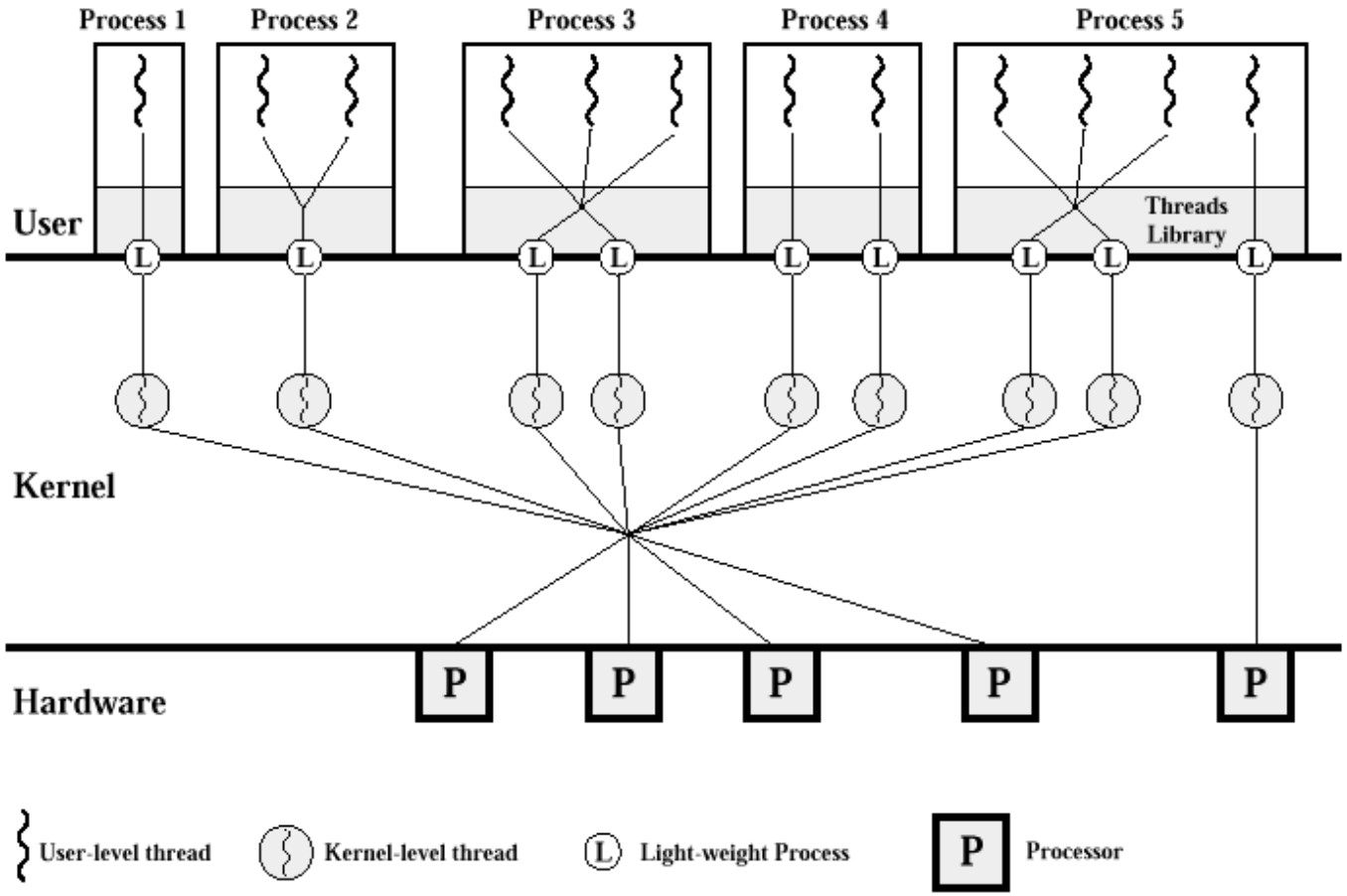
< many-to-one >



< one-to-one >

Scheduler Activation and LWP

- Connection between user / kernel threads through LWP



Pthread Scheduling

- In thread creation with Pthreads, we can specify PCS or SCS
  - PCS(PTHREAD_SCOPE_PROCESS):
    - In many-to-one, only PCS is possible
  - SCS(PTHREAD_SCOPE_SYSTEM):
    - In one-to-one model, only SCS is possible

  Note: On certain system, only certain values are allowed
    - Linux, MacOS X

- Related functions
  - pthread_attr_setscope(pthread_attr_t *attr, int scope)
  - pthread_attr_getscope(pthread_attr_t *attr, int *scope)

Pthread Scheduling

Example)

```
pthread_t tid;
pthread_attr_t attr;
int scope;

pthread_attr_init(&attr);

pthread_attr_getscope(&attr, &scope);
// scope is either PTHREAD_SCOPE_PROCESS
// or PTHREAD_SCOPE_SYSTEM

pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM)

pthread_create(&tid, &attr, runner, NULL);
```

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Real-time CPU scheduling
- Operation system examples
- Algorithm evaluation

Real-time CPU Scheduling

- Real-time Operating Systems (RTOS)
  - OS intended to serve real-time systems

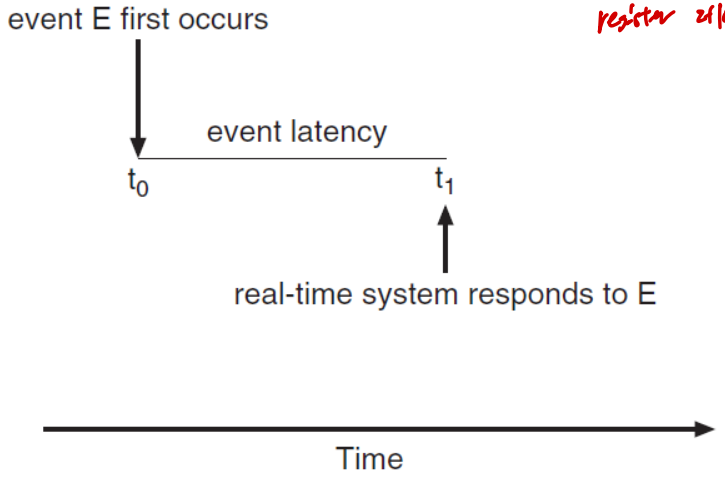
  Ex) Antirock brake system (ABS) requires latency of 3~5 msec

- Soft real-time systems
  - No guarantee as to when a critical real-time process will be scheduled.
  - Guarantee only that the process will be given preference over noncritical processes.

- Hard real-time systems
  - A task must be serviced by its deadline
  - Service after the deadline has expired is the same as no service at all.

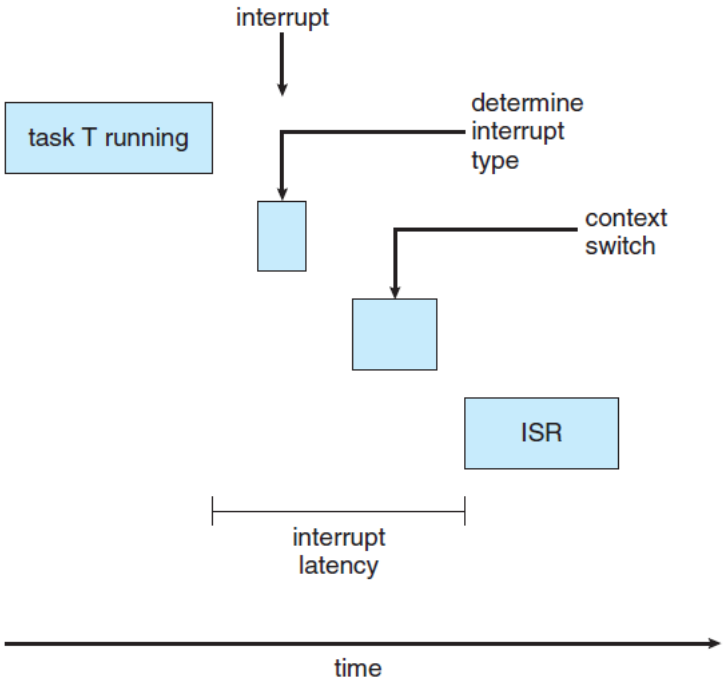
Minimizing Latency

- Most real-time systems are event-driven system
  - When an event occurs, the system must respond to and service it as quickly as possible
- Event latency: the amount of time that elapses from when an event occurs to when it is serviced
  - Interrupt latency, dispatch latency



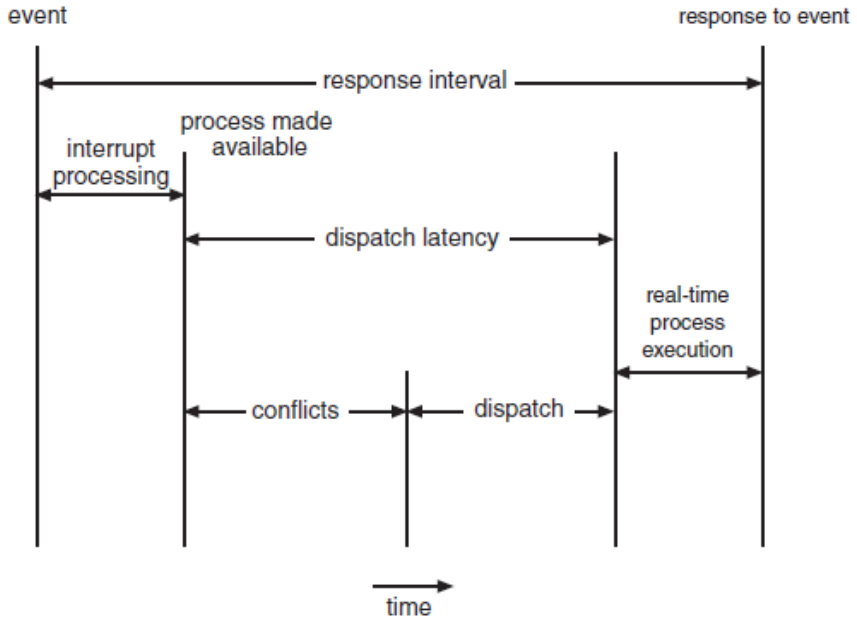
Interrupt Latency

- Interrupt latency: the period of time from the arrival of interrupt at the CPU to the start of the routine that services the interrupt.



Dispatch Latency

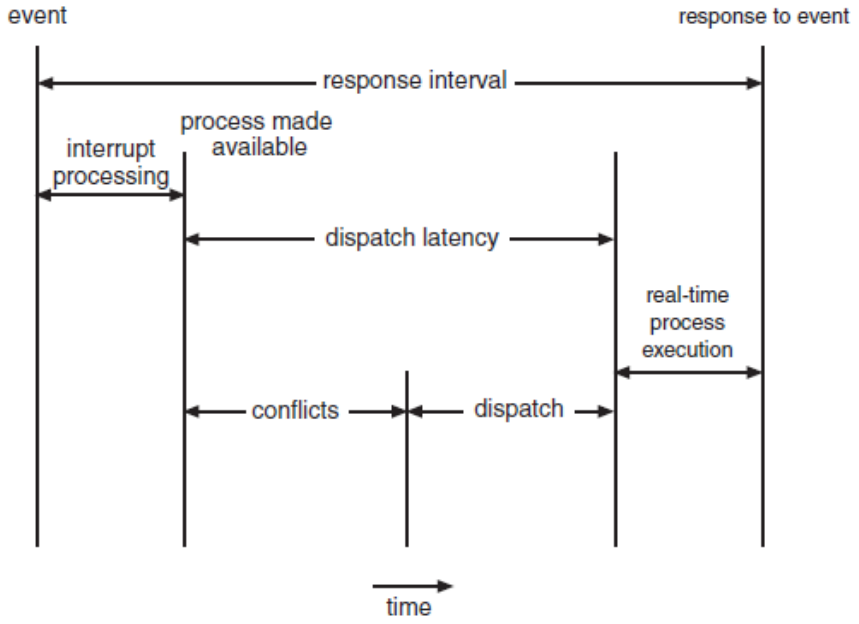
- Dispatch latency: the amount of time required for the scheduling dispatcher to stop one process and start another
  - Preemptive kernels are the most effective technique to keep dispatch latency low.



Dispatch Latency

빨리 종료해야 process가 리소스가 필요한데, 다른 process가 사용하고 있을때

- Conflict phase
  1. Preemption of any process running in the kernel
  2. Release of resources occupied by low-priority processes needed by a high-priority process
- Dispatch phase schedules the high-priority process onto an available CPU.



priority inversion

우선순위 0 1 2

α b C

b가 리소스가 필요한데 α가 리소스를 가지고 있어서 b보다도 우선순위가 높아질때 현상

리얼타임 프로그램이 CPU 독점해서안됨

Priority-based Scheduling

- The most important feature of RTOS is to respond immediately to a real-time process
  - The scheduler should support priority-based preemptive algorithm
  - Most OS assign highest priority to real-time processes
  - Preemptive, priority-based scheduler only guarantees soft real-time functionality.

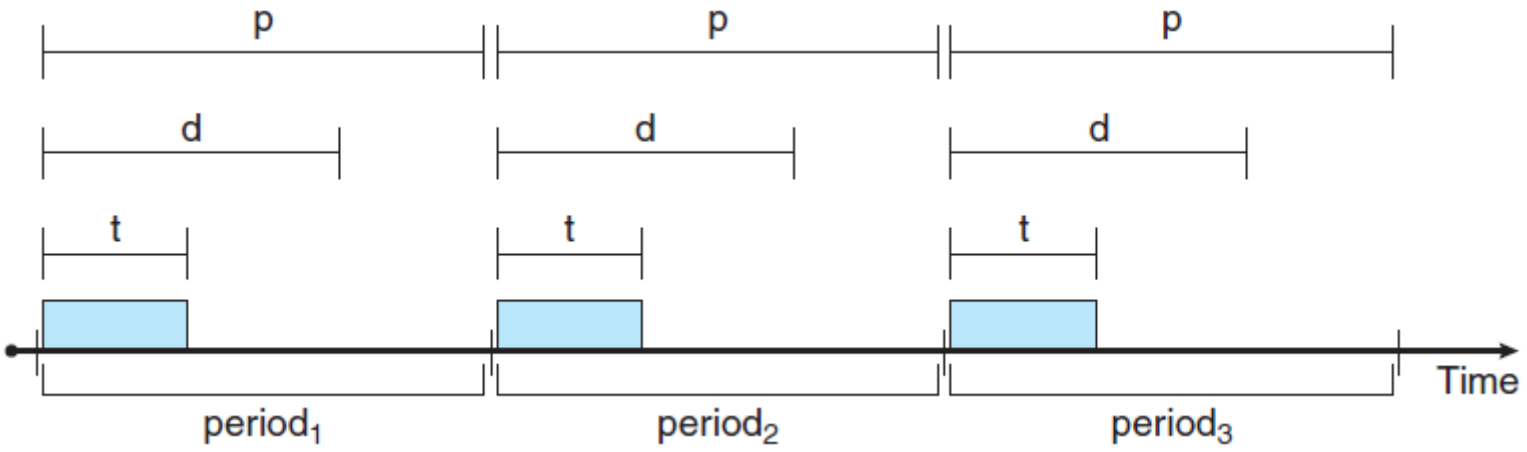
- Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements
  - Requires additional scheduling features.

Periodic Process

- **Periodic processes** require the CPU at constant intervals (periods).
  - A fixed processing time t
  - A deadline d by which it must be serviced by the CPU
  - A period p

$$0 \leq t \leq d \leq p$$

  - The rate of a periodic task: $1/p$



Scheduling with Deadline Requirement

- A process may have to announce its deadline requirements to the scheduler

유저가 요청한 process가 deadline 안에 처리 가능한지 체크

- Using an admission-control algorithm, the scheduler does one of two things:
  - Admits the process, guaranteeing that the process will complete on time, or
  - Rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

Rate-Monotonic Scheduling

- Rate-monotonic scheduling algorithm
  - Schedules periodic tasks using a static priority policy with preemption.
  - Each periodic task is assigned a priority inversely based on its period.
  - The processing time of a periodic process is assumed the same for each CPU burst.

Rate-Monotonic Scheduling

- Rate-monotonic scheduling is considered optimal
  - If a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.

- Limitation
  - CPU utilization is bounded, and it is not always possible to maximize CPU resources fully.
  - Worst-case CPU utilization: $N(2^{1/N} - 1)$
    - N: # of processes

Earliest-Deadline-First Scheduling

periodic 처리 알고리즘

- Earliest-deadline-first (EDF) scheduling
  - The earlier the deadline, the higher the priority
- Requirements
  - When a process becomes runnable, it must announce its deadline requirements to the system.
  - Does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst.
- Theoretically optimal
  - Theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent.

Proportional Share Scheduling

- Proportional share schedulers
  - Allocates T shares among all applications.
  - An application can receive N shares of time, thus ensuring that the application will have N / T of the total processor time
  - Must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time

Unix 참고

POSIX Real-Time Scheduling

- Scheduling classes for real-time threads
  - SCHED_FIFO: FCFS policy
    - No time slicing among threads of equal priority
    - The highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks
  - SCHED_RR: a round-robin policy.
    - Similar to SCHED_FIFO
    - Time slicing among threads of equal priority.
  - SCHED_OTHER: system-specific
    - Implementation is undefined

- POSIX API for getting/setting scheduling policy
  - pthread_attr getschedpolicy(pthread_attr_t *attr, int *policy);
  - pthread_attr setschedpolicy(pthread_attr_t *attr, int policy);

POSIX Real-Time Scheduling

```
int i = 0, policy = 0;
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");
}
```

```
/* 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);
```

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Real-time CPU scheduling
- Operation system examples
- Algorithm evaluation

Operating System Examples

- Linux
- Windows
- (Solaris)

Linux Scheduler

- History
  - Traditional UNIX scheduling algorithm (ver. < 2.5)
    - Not adequate support for SMP system
    - Not scale well as # of tasks increases
  - O(1) scheduling algorithm (ver. ≥ 2.5)
    - O(1) complexity
    - Increased support for SMP
    - Poor response time interactive processes
  - Completely Fair Scheduler (CFS) (ver. ≥ 2.6)

Linux Scheduler

- Based on scheduling classes
  - Each class is assigned a specific priority.
    - Default scheduling class (CFS algorithm)
      - Static priority 100 ~ 139
    - Real-time scheduling class (SCHED_FIFO, SCHED_RR)
      - Static priority 0 ~ 99
    - If necessary, new scheduling classes can be added
  - Allows different scheduling algorithms based on the needs of the system (e.g. server vs. mobile)
  - Scheduler selects highest-priority task belonging to highest-priority class

Linux CFS Scheduler

- CFS scheduler assigns a proportion of CPU processing time to each task
  - The portion is calculated from nice value.
    - Nice value: relative priority in [−20, +19], default is 0
      - Higher value represents lower priority (‘nice’ to other tasks)
  - Proportions of CPU time are allocated from the value of targeted latency.
    - Targeted latency: interval of time during which every runnable task should run at least once
      - Default and minimum values
      - Can increase if the number of active tasks in the system grows beyond a certain threshold

Linux CFS Scheduler

- The scheduler simply selects the task that has the smallest vruntime value
  - Per-task variable vruntime(virtual run time)
    - vruntime = <actual physical run time> * <decay factor>
  - Decay factor
    - Normal priority tasks: 1.0
    - Lower priority task: higher decay factor (> 1.0)
    - Higher priority task: lower decay factor (< 1.0)

    ➔ Higher priority tasks are likely to have smaller vruntime
- Higher-priority task can preempt a lower-priority task.

Linux CFS Scheduler

- I/O-bound tasks: run only for short periods before blocking for additional I/O
  ➔ vruntime will be lower

- CPU-bound tasks: exhaust its time period whenever it has an opportunity to run on a processor
  ➔ vruntime will be higher

➔ Gives I/O-bound tasks higher priority than CPU-bound tasks

Windows Scheduling

- Priority-based, preemptive scheduling
  - Dispatcher handles scheduling
    - Dispatcher ensures the highest priority thread will always run
  - 32-level priority scheme
    - Real-time class (16~32)
    - Variable class (1~15)
    - Memory management (0)
  - If no ready thread is found, dispatcher runs idle threads

Windows Scheduling

- Priority classes
  - A process belongs to one of 6 levels of priority class
  - Each thread belongs to one of 7 levels of relative priority

	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 Scheduling

- Altering priority
  - Priority class of a process: SetPriorityClass()
  - Base priority of a thread: SetThreadPriority()
- Adjusting priority for variable-priority class
  - Lower priority of a thread that expired entire time quantum
    - Never lowered below the base priority
  - Boost priority of a thread that released from waiting
    - Amount of increment vary with what it was waiting
- Scheduling of foreground / background processes
  - Time quantum for the foreground process is increased by some factor (typically, 3)

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Real-time CPU scheduling
- Operation system examples
- Algorithm evaluation

Algorithm Evaluation

- First of all, evaluation criteria should be determined

  Ex)

  1. Maximum response time is 1 second
  2. Satisfying 1, maximize CPU utilization
  3. Maximize throughput such that turnaround time is linearly proportional to total execution time

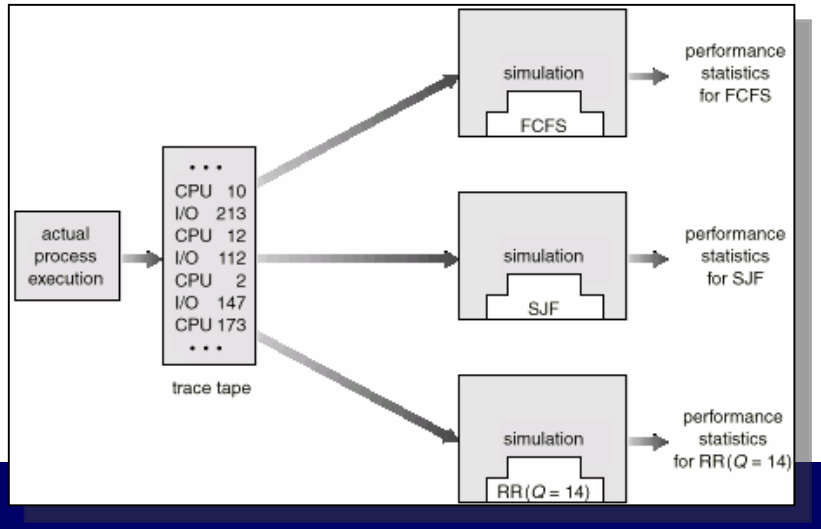
- Evaluation methods
  - Deterministic modeling
  - Queueing models
  - Simulations

Algorithm Evaluation

- Deterministic modeling
  - Evaluation for a particular predetermined workload
- Queueing models
  - Evaluation based on probabilistic distribution of…
    - CPU and I/O burst
    - Arrival time of CPU burst
  - Description of system
    - Network of servers, each of which has a ready queue
    - Given arrival rates and service rates, performance can be computed probabilistically

Algorithm Evaluation

- Simulations, using a programming model of computer system
  - Generator of process, CPU burst time, arrival, departure, …
  - Usually, the generator randomly generates the simulated events based of some distribution
  - Limitation: simulated situation isn't exactly same with the real situation
    - Especially, distribution gives no information about order of events
    - Remedy: using a record of real system. (trace tape)



Algorithm Evaluation

- Implementation: the only way for accurate evaluation
  - Performance depend not only on scheduling algorithm and OS support, but also user's interaction
  - Environment will change

    Ex) If short process is given higher priority, user may break a large process into several smaller processes.
  - Some OS's provides flexibility to alter scheduling scheme
    - Performance turning for specific (set of) applications
    - API's to modify priority of a process or thread