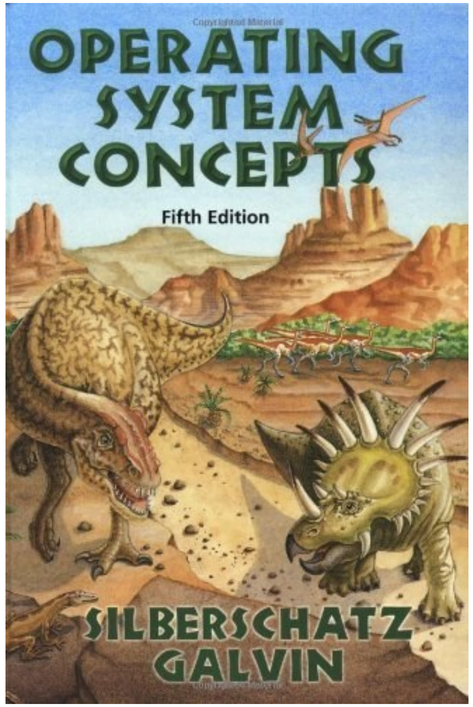
Chapter 5: CPU Scheduling

School of Computing, Gachon Univ.
Joon Yoo



Gachon University
School of Computing

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

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Advanced Scheduling
  - Multiple-Processor Scheduling
  - Real-Time CPU Scheduling

Introduction

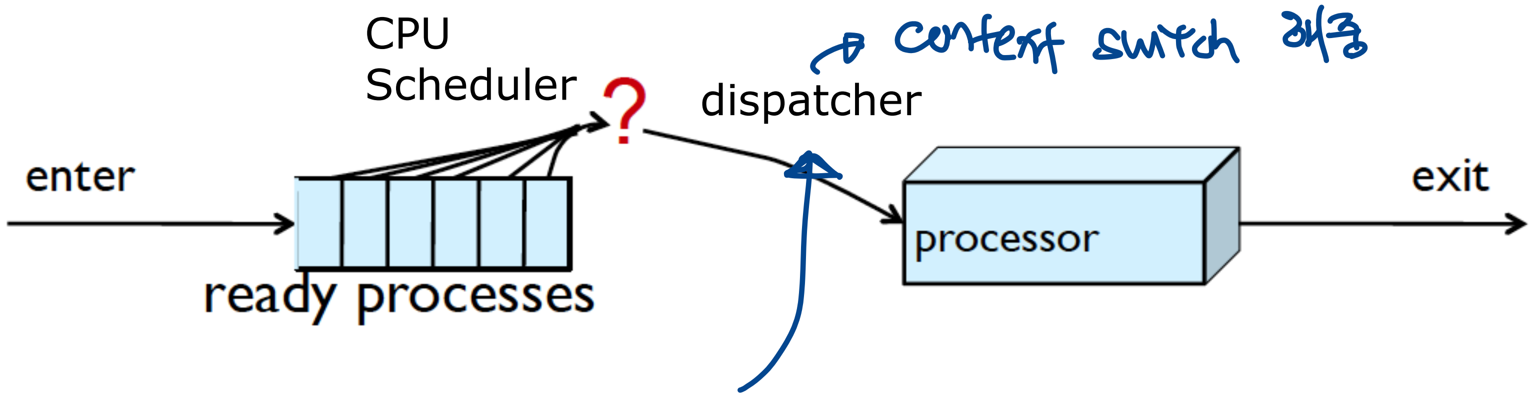
- Multitasking: multiple processes (threads) in the system with one (or more) processor cores
- The role of OS
  - Increases CPU utilization: by organizing processes (threads) so that the CPU always has one to execute
  - Timesharing: by frequent context switching
  - CPU scheduling
    - Allocate CPU time slots to processes (threads) in Ready queue
  - **Note:** Terms "**Process/Thread/CPU/Job/Task** scheduling" will be used inter-changeably in this chapter.

최신 용어 : Process, Thread...

왜 이렇게 혼용하지? → OS 과거의 용어 : Job, Task

다 같은 의미이다

CPU Scheduler



- **CPU Scheduler** — 프로세스 선택
  - **<u>An OS component</u>** that performs the **CPU Scheduling**
    - Which process (or thread) should I run next?
  - Allocates CPU time slots to processes (or threads)

Dispatcher

context switch

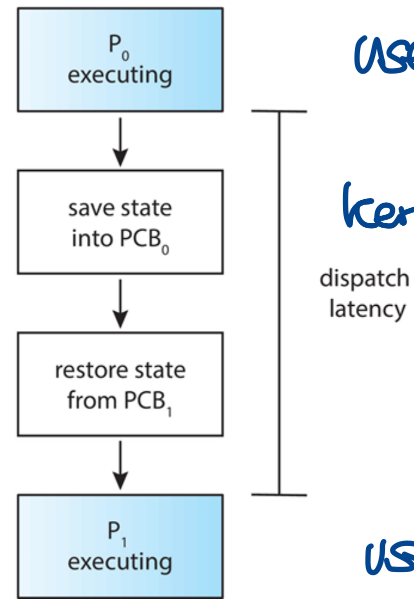
- **Dispatcher**
  - **<u>CPU Scheduler module</u>** that gives control of the CPU to the process selected by the <u>scheduler</u>
    - **context switching**
    - switching to **user mode**
    - jumping to the proper location (**PC**) in the user program to restart that program

user

PCB에 기존정보 저장

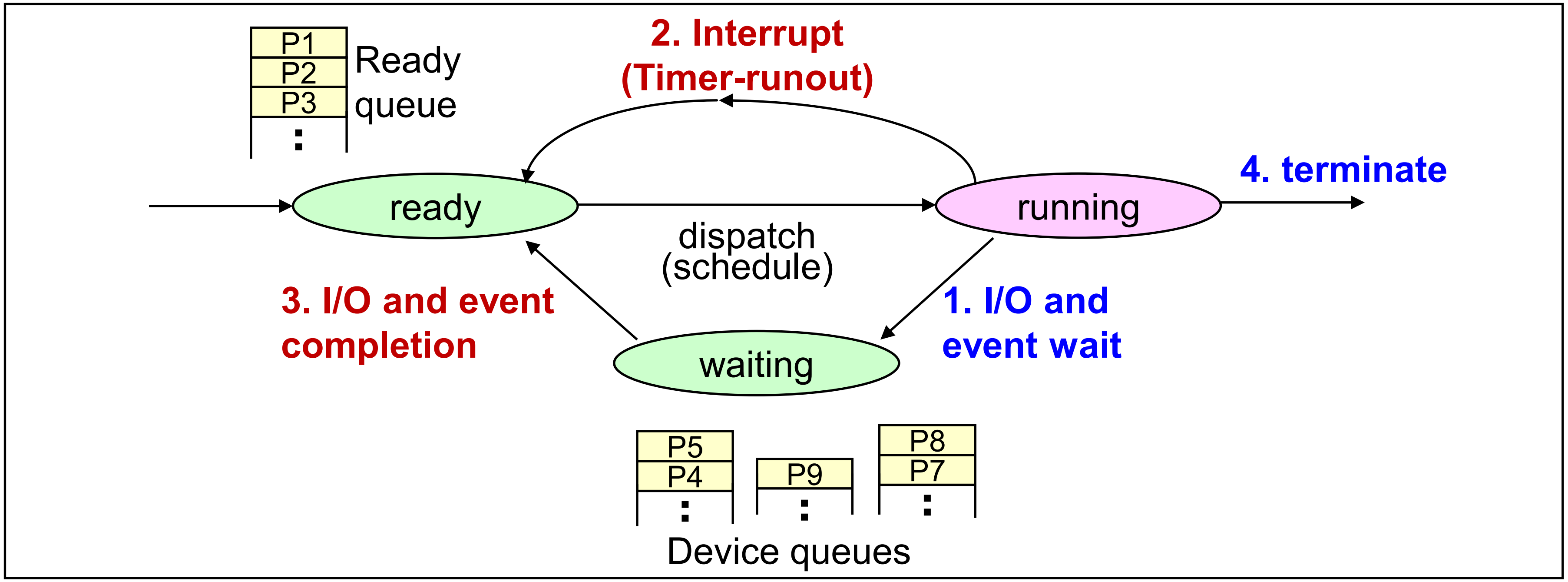
kernel

CPU에 PCB_1의 정보를 가져와서 실행한다



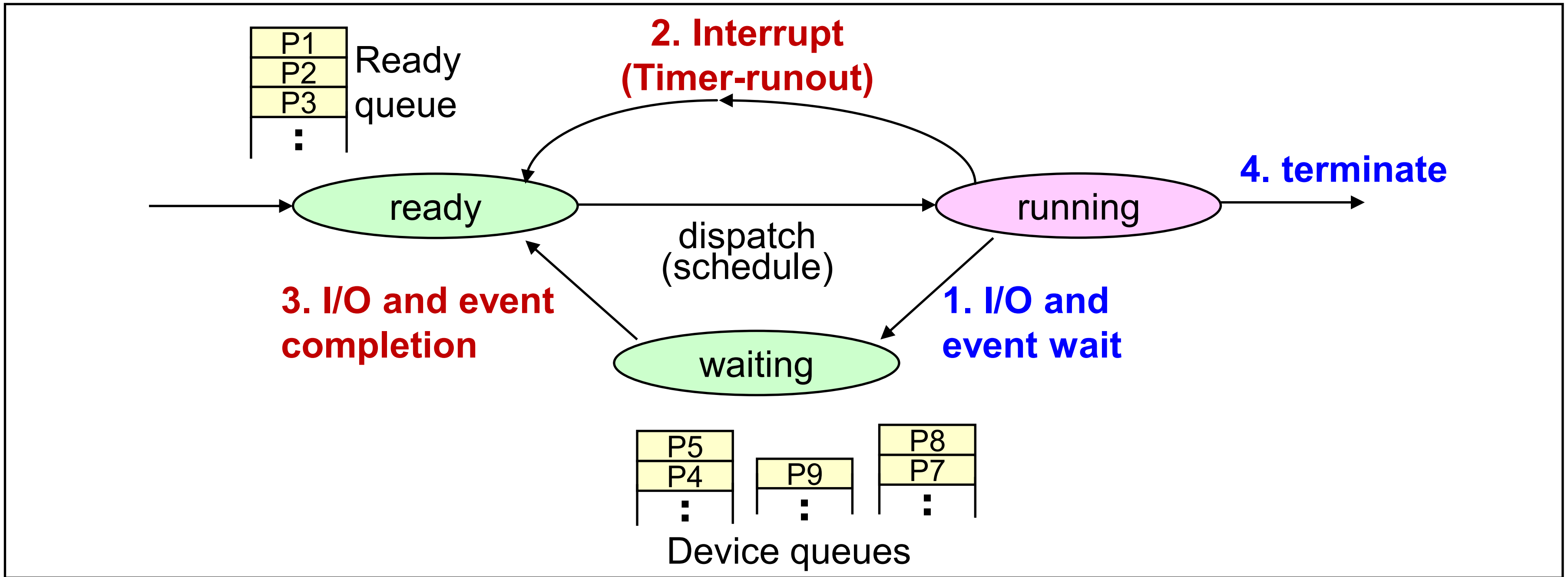
user

Re: Process and State Transition



- CPU scheduling decisions my take place under above four circumstances

Re: Process and State Transition



- Three states : ready, running, and waiting
- CPU scheduling decisions may take place when a process:

스스로 종료

강제로 종료

→ Interrupt

Must dispatch a new process

1. **Switches from running to waiting state**
2. Switches from running to ready state
3. Switches from waiting to ready
4. **Terminates**

timer

Can make a decision!
- should scheduler dispatch a new process or continue with old one?

Preemptive (vs. Non-preemptive

- **Non-preemptive** → 프로세스가 스스로 나감!
  - Once CPU is allocated to a process, it holds it till it
    - It gives up by itself (스스로): Terminates (case 4) or blocks itself to wait for I/O – (case 1)
    - It does not give up for case 2 & 3
  - Process uses the CPU until it <u>voluntarily</u> releases it - <u>**No preemption**</u>

- **Preemptive** → 프로세스를 강제로 종료
  - Currently running process may be <u>**interrupted**</u> and moved to the ready state by the OS
  - CPU may be <u>**preempted**</u> to another process independent of the intention of the running process
    - Most modern OSes: Windows, Mac OS, Linux, ...

└ 최근 OS는 거의 다 preemptive !

Preemptive vs. Non-preemptive

- **Non-preemptive scheduling**
  - Pros: Low context switch overhead
  - Cons : May result in longer mean response time (not good for for time-sharing systems and real-time systems)

요청을 하고 응답이 오기까지 걸린 시간

└ time-sharing가 잘 안될수도 있음

- **Preemptive scheduling**
  - Cons: race conditions (need synchronization – Ch. 6-7)

↳ interrupt 된 위치에 따라 값이 바뀔수도 있다!

이를 위해 synchronization이 필요!

Preemptive vs. Non-preemptive

- Which scheduling policy
  - can immediately take care of Interrupts? P
  - has less context-switching overhead? N
  - has less response time? P
  - can use time-sharing? P
  - do you think is used more generally? P

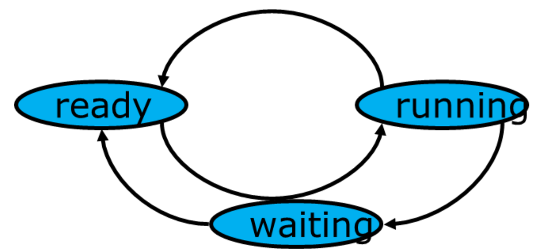
Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Advanced Scheduling
  - Multiple-Processor Scheduling
  - Real-Time CPU Scheduling

Scheduling Criteria

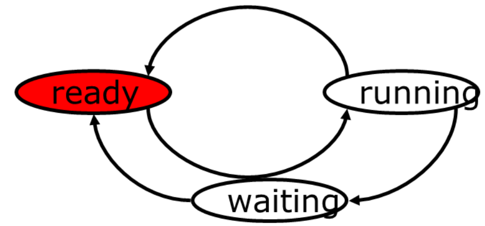
- **CPU utilization** (%) CPU 활용도 → OS 관점에서는 CPU 사용량↑ 이면 좋은것임
  - Higher is better (0~100%)
  - In a real system, 40% (lighted loaded) to 90% (heavily loaded)
- **Throughput** (# of processes completed / time) (Throughput) ↑
  - Higher is better 시간당 처리할 수 있는 프로세스 수 single core < multicore
  - Long process (1 process/several seconds), short process (tens of process/second)
- **Turnaround time** (s): time for each process to complete
  - Lower is better
  - Time spent in the ready queue, executing CPU, and waiting (for I/O, ...)

ready running waiting



Scheduling Criteria

- **Waiting time** (s): sum of time spent waiting in the **ready queue** → ready queue 에서 기다린 시간
  - Lower is better
  - CPU-scheduling algorithm affects *only* the time spent on the ready queue (compare with turnaround time)



- **Response time** (s): time from request to first response
  - Lower is better
  - Used for Interactive and real-time systems
  - e.g., HTTP request to Web page load response

Context switching 에 따라 시간 달라짐

CPU를 쓰려고 들어와서, CPU 응답까지 걸린 시간

왜?

running 시간도 scheduling 와는 관련없음. waiting도 마찬가지. 오직 ready에 있는 시간이 scheduling으로 관리할 수 있다.

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Advanced Scheduling
  - Multiple-Processor Scheduling
  - Real-Time CPU Scheduling

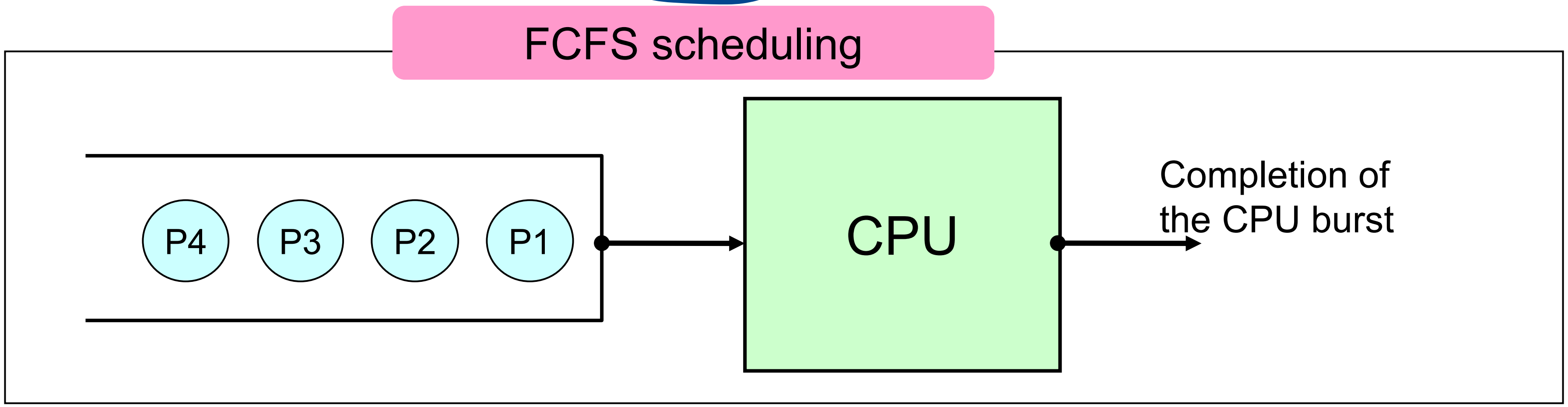
Scheduling Algorithms

- CPU Scheduling Algorithm
  - Algorithm that decides which of the processes in the ready queue is to be allocated the CPU

- Various algorithms
  - First-Come, First-Served (FCFS) Scheduling 선착순
  - Shortest-Job-First (SJF) Scheduling 짧은 process부터
  - Round-Robin Scheduling 돌아가면서 차례로
  - Priority Scheduling 우선순위부터
  - Earliest Deadline First Scheduling 마감일 빠른 것부터

First-Come, First-Served (FCFS)

- FCFS(First-Come-First-Served) scheduling
  - Simplest scheme using FIFO queue



  - **<u>Non-preemptive</u>** scheduling

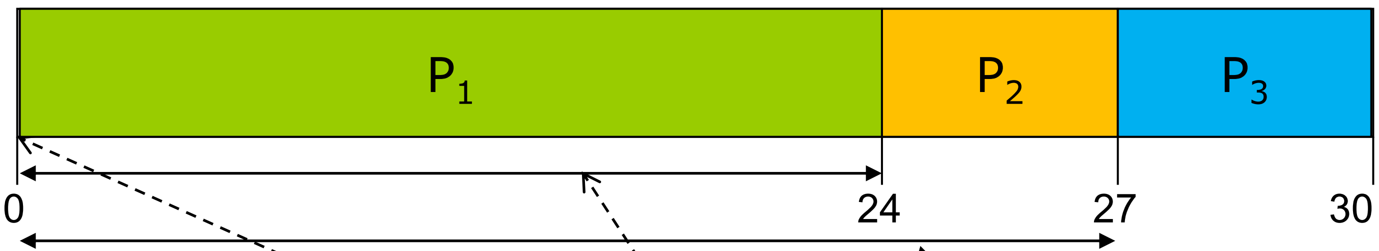
First-Come, First-Served (FCFS) Scheduling

All arrived at time 0 in order P_1, P_2, P_3

Process	CPU 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:

$P_1 \rightarrow P_2 \rightarrow P_3$



- **Waiting time** for P_1 = 0; P_2 = 24; P_3 = 27
  - **Average waiting time** (WT): (0 + 24 + 27)/3 = **17**
  - Turnaround time (TT): (24+27+30)/3 = 27

Can we do better?

CPU & I/O Bursts

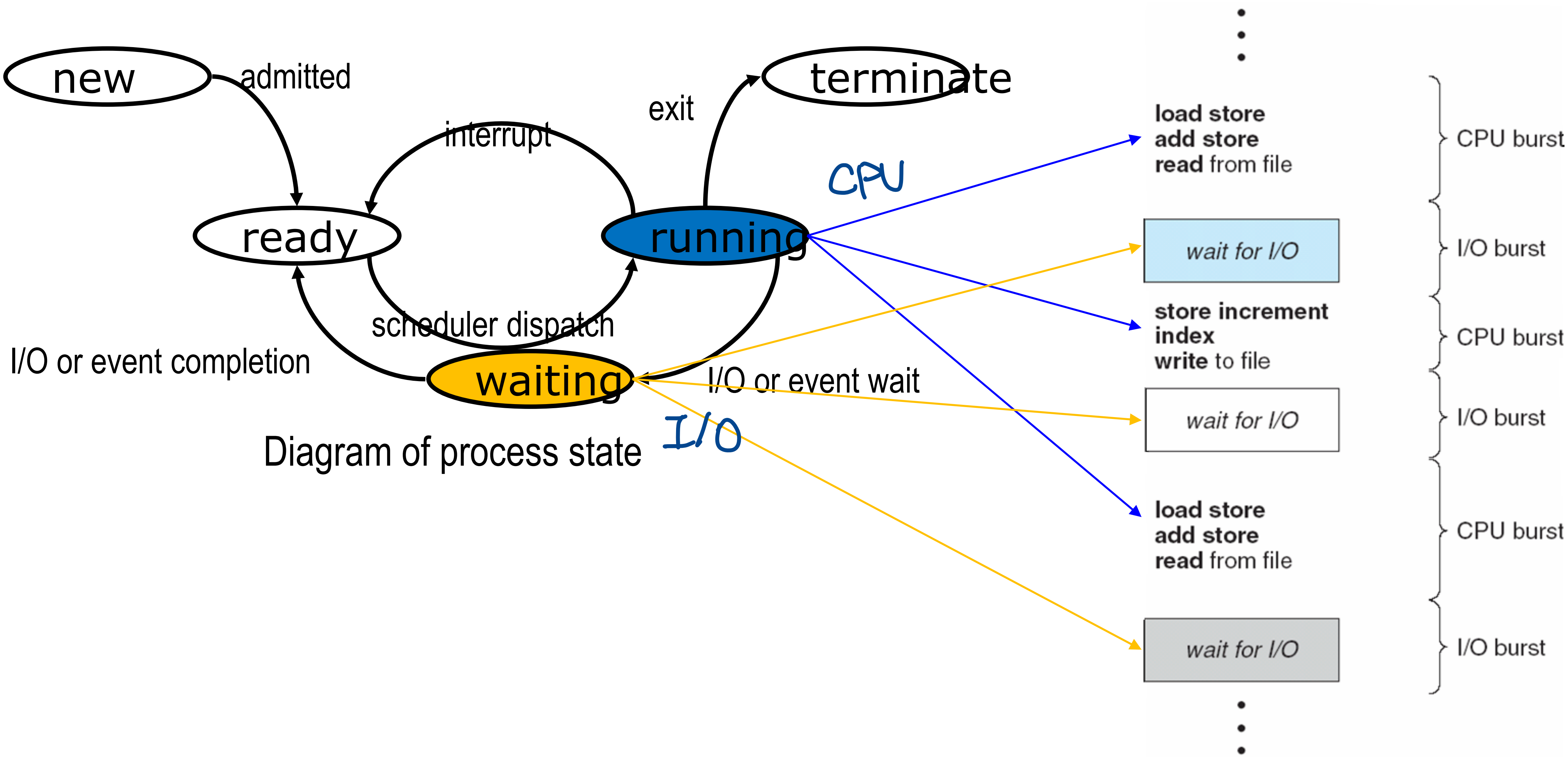
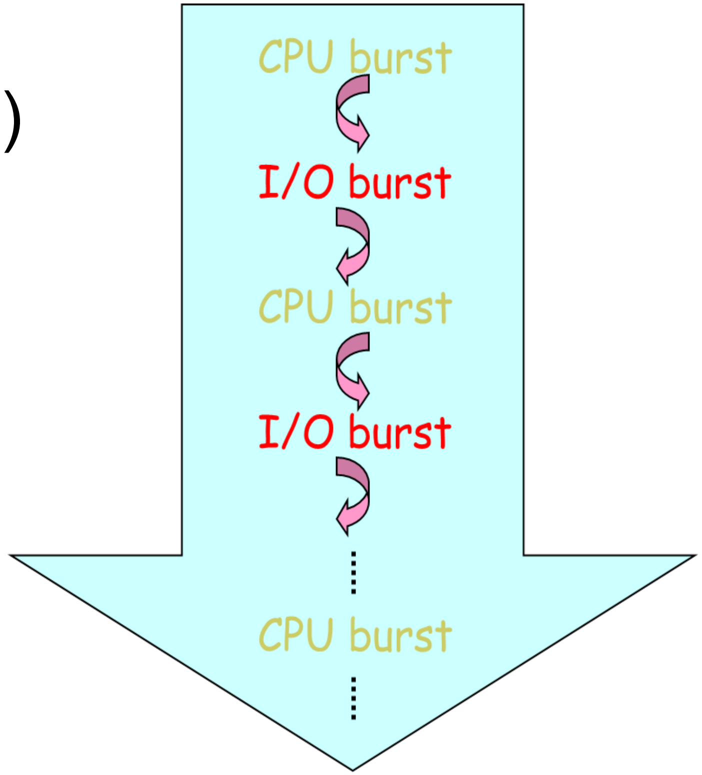


Diagram of process state

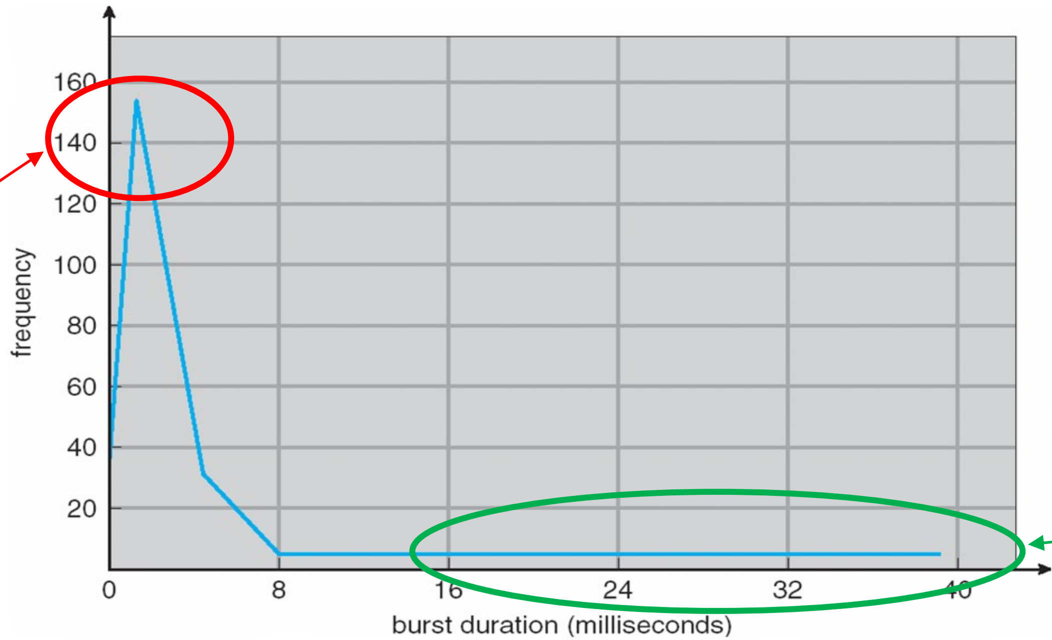
CPU & I/O Bursts

- Process execution consists of cycles of CPU bursts and I/O bursts
  - **CPU burst:** CPU executions (running state)
    - e.g., sorting a million-entry array in RAM
  - **I/O burst:** I/O wait (waiting state)
    - e.g., disk read/write, networking, printing
- **CPU burst time** is an important factor(criteria) for scheduling algorithms

CPU 이용시간이 scheduling에 중요함



Histogram of CPU-burst Durations



CPU bursts distribution

- Large number of **short CPU bursts** and small number of **long CPU bursts**

I/O bound process

↓

I/O 작업 위주 → CPU burst time

짧게, 자주

CPU bound process

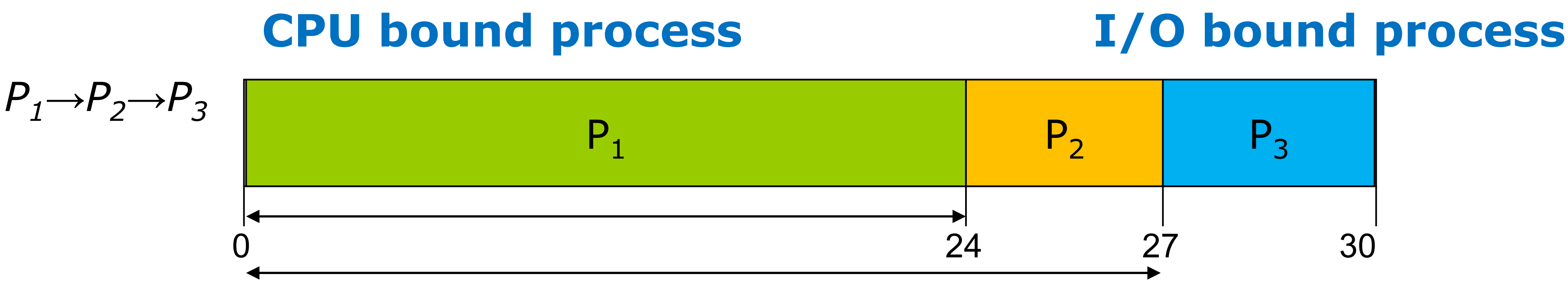
↓

계산 작업 위주 → CPU burst time

길게 ~

FCFS: Convoy Effect

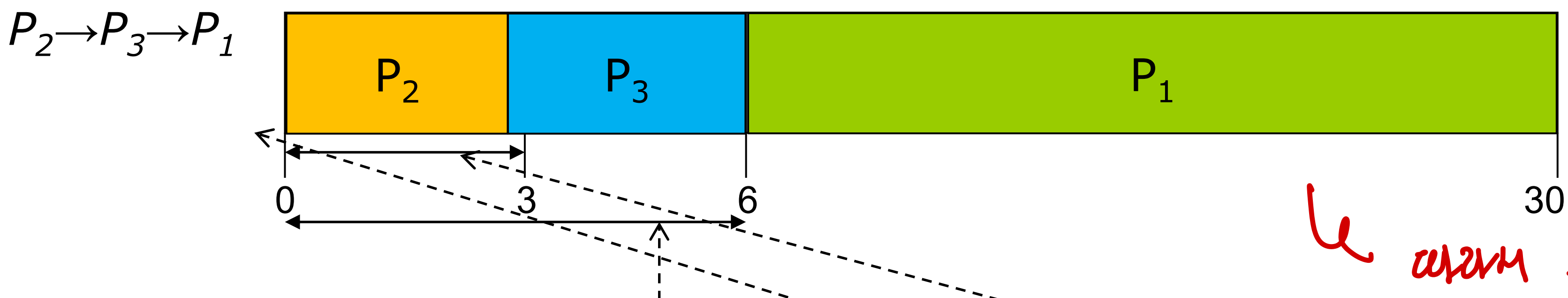
- **Convoy Effect** → 앞에 차가 느리면, 뒤 차도 같은 속도로 가야함
  - Larger **waiting time** for I/O bound process
    - CPU-bound jobs will hold CPU until exit or I/O



FCFS Scheduling

Suppose that the processes arrive in the order: P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



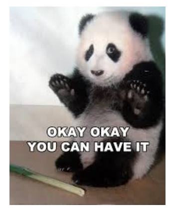
따라서 I/O bound 작업은 먼저 해야 한다!
그래야 평균 waiting time 가 크게 줄어들기 때문이다

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time (WT): (6 + 0 + 3)/3 = **3**
  - Turnaround time (TT): (30+3+6)/3 = 13
- Lesson: scheduling algorithm can reduce WT and TT

Shortest-Job-First (SJF) Scheduling

- Schedule the process with the shortest time
- Gives **minimum average waiting time**
- Two schemes:
  - **Nonpreemptive SJF** 중간에 CPU를 뺏을 수 없음
    - Once CPU given to the process it cannot be preempted until completes its CPU burst

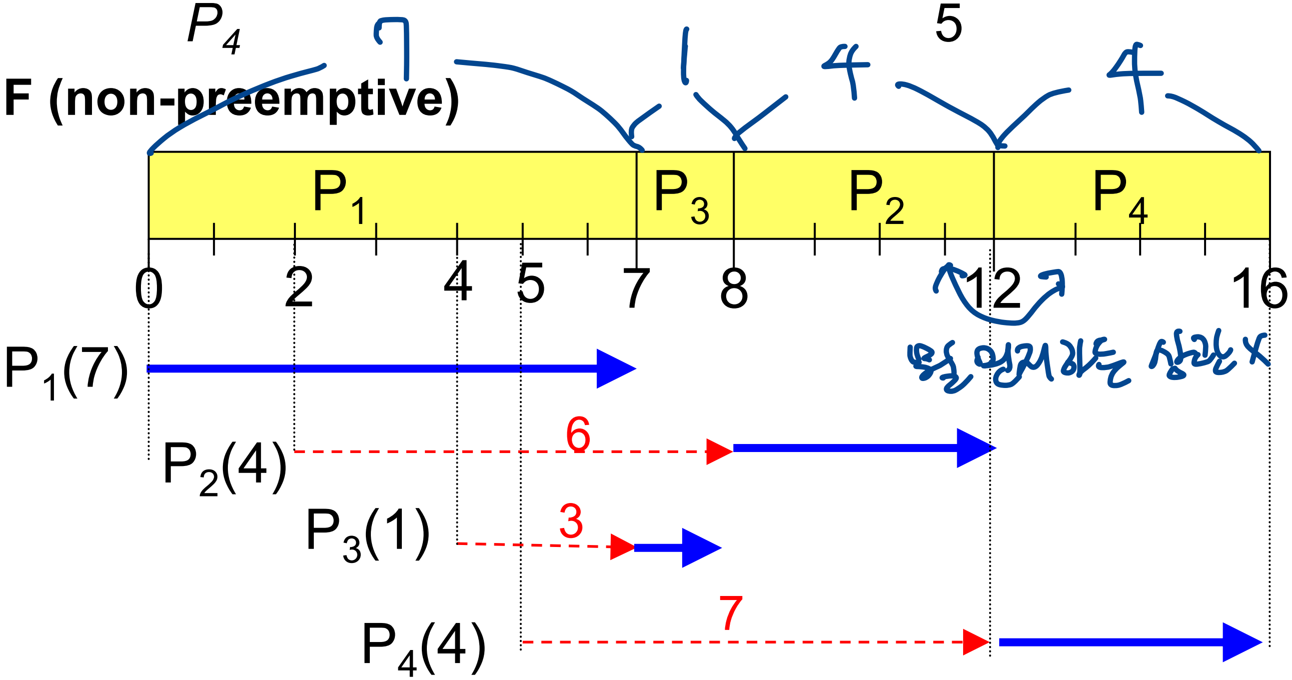
  - **Preemptive SJF**
    - If a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
    - This scheme is known as the **Shortest-Remaining-Time-First (SRTF)**



Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **SJF (non-preemptive)**



P_1's wating time = 0

P_2's wating time = 6

P_3's wating time = 3

P_4's wating time = 7

Average WT = (0 + 6 + 3 + 7)/4 = 4

Average TT = (7 + 10 + 4 + 11)/4 = 8

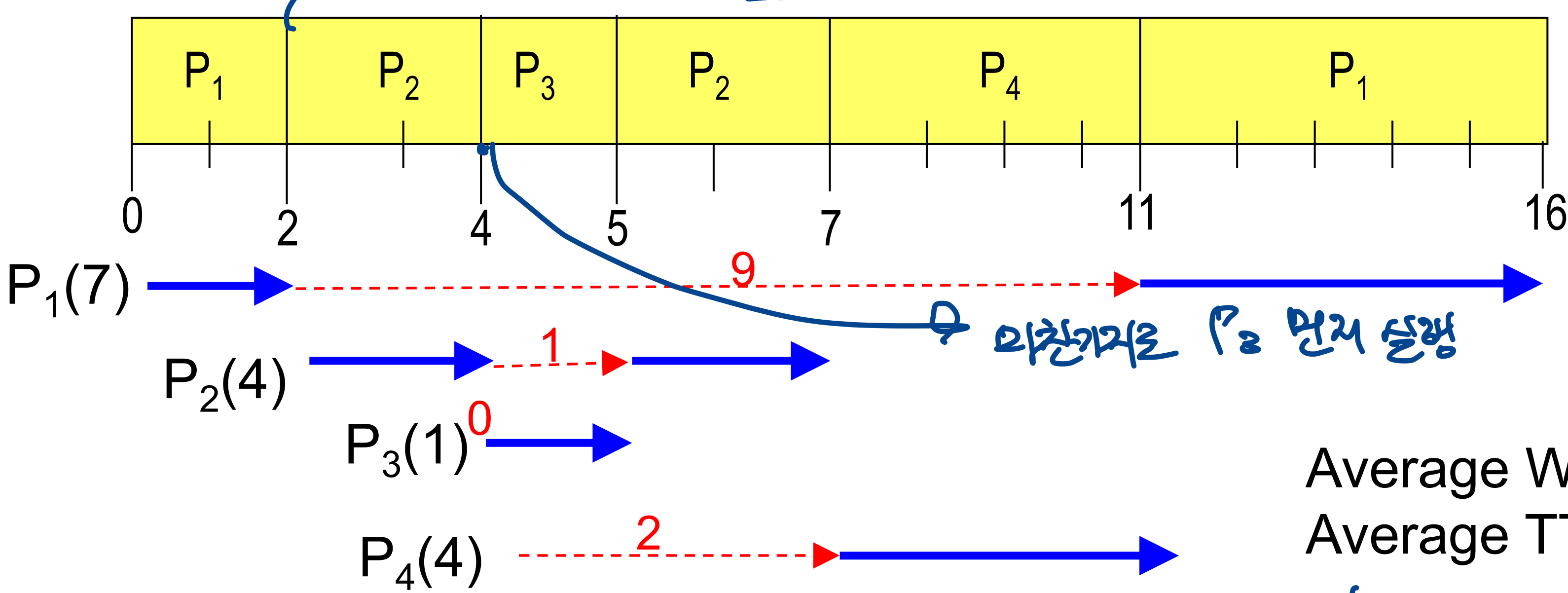
Example of Preemptive SJF (SRTF)

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

★ 문제!: CPU bound job들이 CPU 할당을 못받을 수 있다!

- **SRTF (preemptive SJF)**

P_1, P_2를 비교 남은 burst time < P_1 7-2=5, P_2 4 즉 P_2 먼저 실행



마찬가지로 P_2 먼저 실행

P_1's wating time = 9

P_2's wating time = 1

P_3's wating time = 0

P_4's wating time = 2

Average WT = (9 + 1 + 0 +2)/4 = 3

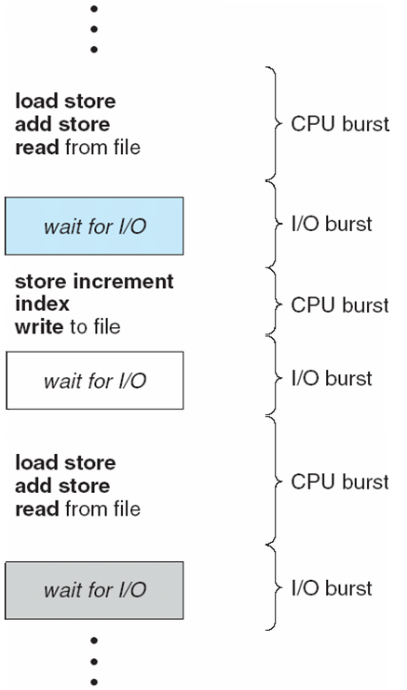
Average TT = (16 + 5 + 1 +6)/4 = 7

↳ Average 값 작아진다!!

CPU Burst Prediction

- How do we know the next CPU burst length?
  - We cannot know for sure
  - But we may ***predict*** the CPU burst length
    - How? – "History is a mirror to the future"
    - The next CPU burst is generally predicted as an ***exponential moving average*** of the measured lengths of previous CPU bursts:
      - From n CPU bursts from the past: $(t_1, t_2, \ldots, t_n)$
      - Predict the future $(n+1)^{\text{th}}$ CPU burst: (τ_{n+1})

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



CPU Burst Prediction

- Predicted value for the next CPU burst using ***exponential moving average***

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

예측 CPU burst

- τ_{n+1}: predicted value for the next CPU burst.
- t_n: length of the nth CPU burst, $0 \le \alpha \le 1$,

과거 CPU burst

- $0 \le \alpha \le 1$ controls the weight of recent and past
  - What does it mean
    - if $\alpha = 0$? : $\tau_{n+1} = \tau_n$
    - if $\alpha = 1$? : $\tau_{n+1} = t_n$

α을 기준으로 비중을 어디에 더 많이 잡을건지 정할수 있다

α, 1−α 는 모두 소수이므로 많이 곱할수록 값이 작아진다.

따라서, 최근 값이 가장 반영이 많이 된다.

- If we expand the above equation,

CPU bursts in time order: $(t_1, t_2, \ldots, t_{n-1}, t_n)$

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

$$\ldots = \alpha t_n + \alpha(1-\alpha)t_{n-1} + \alpha(1-\alpha)^2 t_{n-2} + \alpha(1-\alpha)^3 t_{n-3} + \cdots$$

CPU Burst Prediction

- Example

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

P3's previous three CPU burst samples are: 8, 16, 8 (in time order; 시간순서). What should be P3's next predicted CPU burst value of P3? Assume we use *exponential moving average* with *α*=0.5 and initial average τ_0 = 8. [OS midterm exam'19]

$$\tau_1 = 0.5 \times \overset{8}{t_0} + 0.5 \times \overset{8}{\tau_0} = 8$$

$$\tau_2 = 0.5 \times \overset{16}{t_1} + 0.5 \times \overset{8}{\tau_1} = 12$$

$$\tau_3 = 0.5 \times \overset{8}{t_2} + 0.5 \times \overset{12}{\tau_2} = 10$$

P3's CPU burst = <u>10</u>

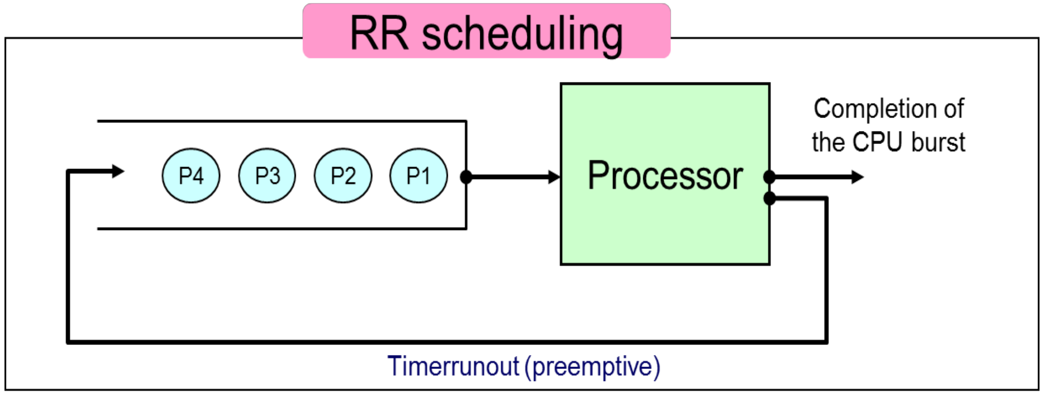
SJF Scheduling (Cont.)

- Pros
  - Gives **minimum average waiting time** for a given set of processes
  - Minimizes the number of processes in the system
    - Less average person waits in line (=waiting time), the shorter the line (=number of processes)
- Cons
  - **CPU burst time** can only be **predicted**

    CPU burst time을 우리가 predict해서 알아내야 하기 때문에...
    - We can only approximate SJF scheduling
    - Thus, may not be optimal if prediction is inaccurate.

Round Robin (RR)

- **RR (Round-Robin) scheduling**



- User **Timer**
  - Preempt job after **time quantum** (**time slice**)
  - When preempted move back to Ready queue

RR (Round-Robin) scheduling

- RR (Round-Robin) scheduling

  단위시간마다 process가 바뀜

  - **Time quantum** (=**time slice**) for each process
    - System parameter (generally from 10 to 100 ms in length)
  - The CPU scheduler sets a **timer** to 1 **time quantum**
    - An interrupt will occur after 1 time quantum
    - The (running) process that has exhausted his time quantum releases the CPU and goes to the ready state (timerrunout)
    - A context switch occurs!

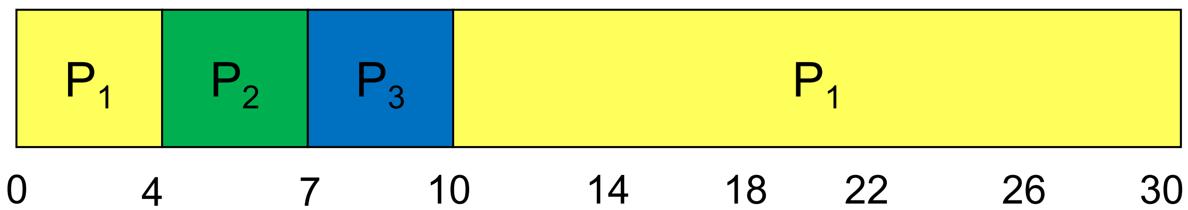
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

convoy effect case

Convoy effect를 줄여준다.

- The Gantt chart for RR is:

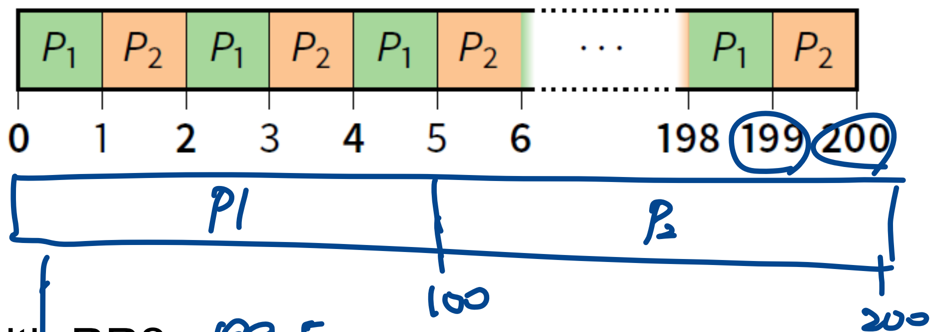


예측이 아니기 때문에 정확하다.

- Typically, higher average **waiting time** than SJF, but fair and no starvation

RR (Round-Robin) scheduling

- Pros
  - Low response time for all jobs → 내 차례가 자주 돌아옴
  - Low average waiting time for I/O bound jobs
  - No starvation for CPU bound jobs → CPU bound jobs가 CPU를 할당받지 못하는 사태는 없음
- Cons
  - Bad for *same-sized* jobs 같은 길이 jobs는 오히려 손해가 크다.



    - Assume 2 jobs of time=100 each:
    - Even if context switches were free...
      - What would average turnaround time be with RR? 199.5
      - How does that compare to FCFS? 150
  - Context switching overhead due to preemptions

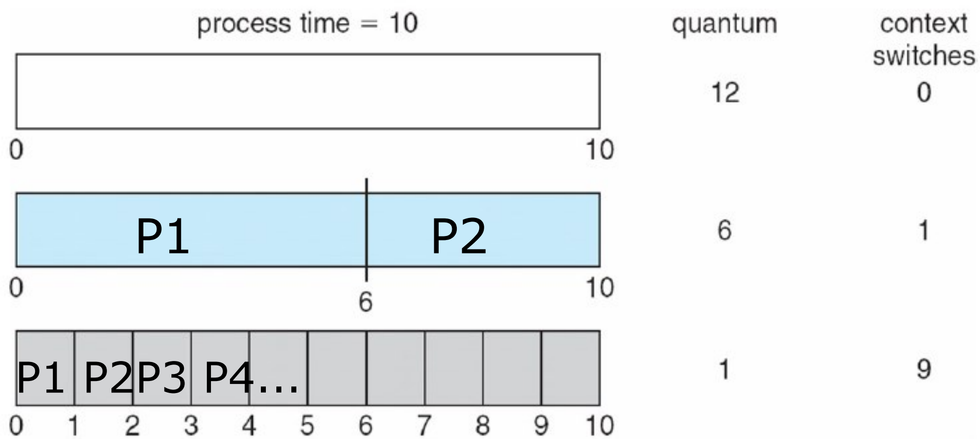
너무 자주 switch가 일어나면 overhead ↑↑

Round Robin Scheduling

q가 작으면, time-sharing이 좋아지지만
context-switch overhead가 커진다

- Performance
  - quantum (q) extremely large = FCFS
  - q small ⇒ **time-sharing** ⇒ q must be large with respect to **context switch time**, otherwise overhead is too high.

따라서 q 길이를
적당히 잡아준다!



- Number of Context Switching
  - quantum = 12 ➔ FCFS
  - quantum = 6 ➔ 2 quanta, 1 Context Switching
  - quantum = 1 ➔ 10 quanta, 9 Context Switching

너무 많음!

In practice,
- Quantum: 10~100ms
- Context switch time: <10us
 (0.1~0.01% overhead)

Priority Scheduling

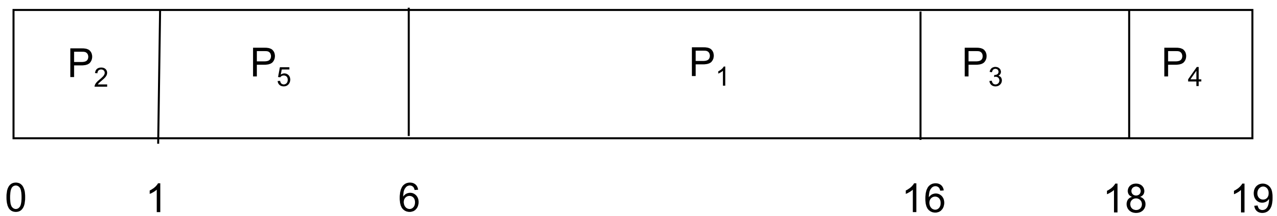
- A numeric **priority** is associated with each process
  - sometimes smaller number means higher priority (e.g., Unix/BSD)
  - or sometimes smaller number means lower priority (e.g., Pintos)
  - but you will be clearly given the definition in your problems 기준이 주어질 것임
- The CPU is allocated to the process with the **highest priority**
  - Preemptive or Nonpreemptive
- Note **Shortest Job First (SJF)** is a **priority scheduling** where priority = predicted next CPU burst time ㄴ사실상 priority scheduling의 일종이다.
- Problem → SJF과 마찬가지로 우선순위 낮은 프로세스는 실행되기 힘들다.
  - **Starvation** – low priority processes may never execute
- Solution
  - **Aging** – as time progresses increase the priority of the process
    ㄴ 우선순위를 점점 높여준다.

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Lower number is higher priority

- Priority scheduling Gantt Chart



사실 priority scheduling
의 특수한 케이스이다

- Average waiting time = 8.2 msec

- SJF : Special case of the general priority scheduling
- FCFS : Equal priority

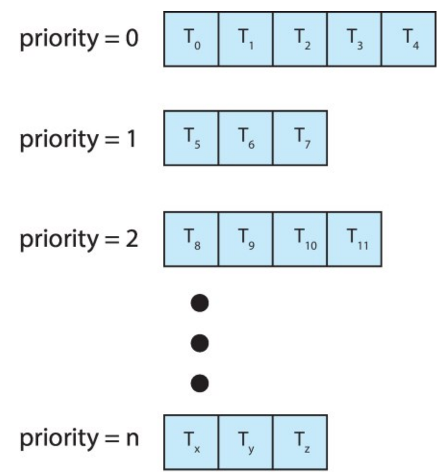
큐를 정하고
큐에서 우선순위를 정함
ㄴ 실제 OS에서 사용하는 방식이다.

Multilevel Queue

- **Multilevel Queue:**
  - Priority + round-robin scheduling
  - Ready queue is partitioned into separate queues, eg:
  - Kernel runs process on highest-priority non-empty queue
  - Round-robins among processes on same queue

같은 priority일 경우 RR 방식으로
우선순위를 정함

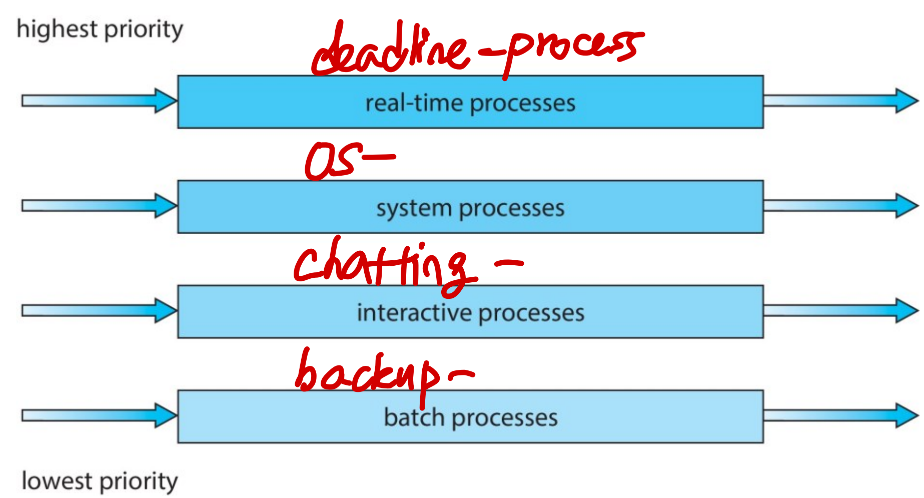
우선 서비스



큐 안에 priority에
따라 process가
들어간다.

큐에 priority가 정해져 있음

priority의 기준,
하나의 예시
프로세스의 특징에
따라 결정된다.



Midterm Exam'16

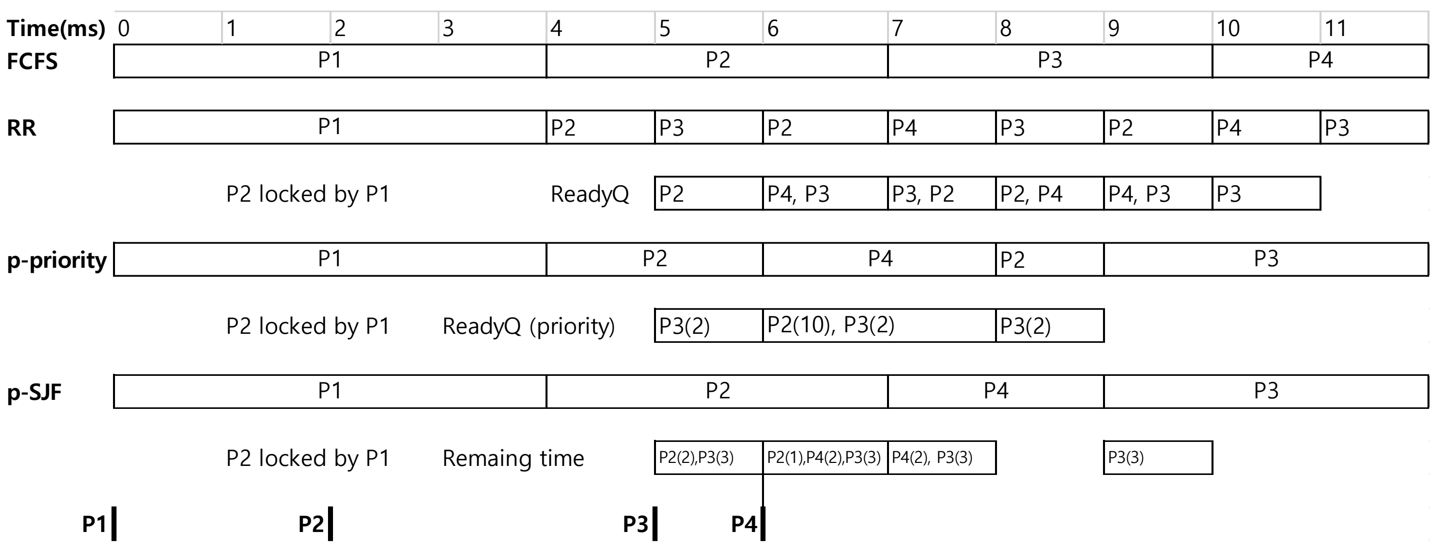
4. Process Scheduling: Consider the right processes, arrival times, waiting, priority, and CPU processing requirements . For each scheduling algorithm, draw the **GANTT chart** and compute the **average waiting time** for each scheduling algorithm (**FCFS, RR, preemptive-priority, and preemptive shortest -job-first (SJF)**)

Process	Arrival Time	Priority	Waits on Lock held by	CPU burst (ms)
P1	0	1	None	4
P2	2	10	P1	3
P3	5	2	None	3
P4	6	20	None	2

with the thread that is running on the CPU (for time-quantum (slice) based algorithms, assume a 1ms time quantum). [Total 2+5+5+5 = 17pts] **Notes:**

- For RR and Priority, assume that a newly arriving thread can run at the beginning of its arrival time, if the scheduling policy allows it. (스케쥴링 규칙이 허용한다면, 도착 후 바로 실행 가능)
- A larger integer priority number indicates higher priority. If two threads have same priority, then FCFS is applied.
- If thread A tries to acquire a lock, but the lock is held by thread B, then thread A will be sent to WAITING state, and return to READY when thread A finishes the CPU burst . If thread A waits for a lock (formerly) held by thread B and B has already finished, it does not wait and acquires the lock immediately.
- You must show your work (계산과정) when computing each average waiting time.

Solution



FCFS: (0+2+2+4)/4 = 2ms RR: (0+5+4+3)/4 = 3ms

p-priority: (0+4+4+0)/4 = 2ms p-SJF: (0+2+4+1)/4 = 1.75ms

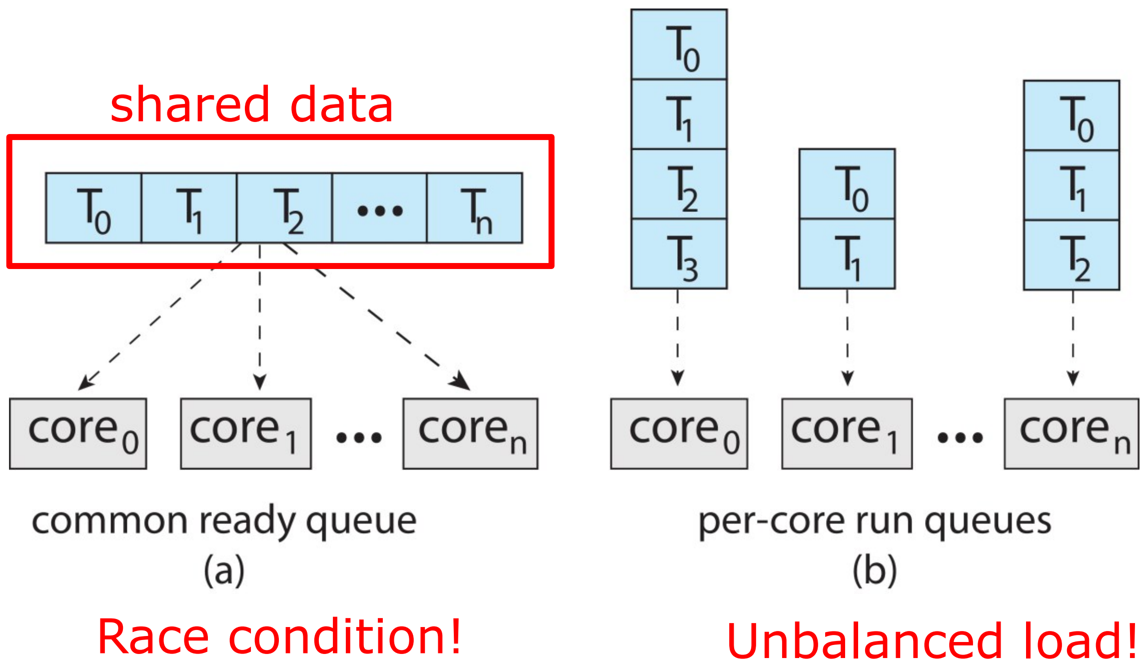
(1/4/4/4 pts for correct Gantt chart, 1pt for each correct computation/waiting time)

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Advanced Scheduling
  - Multiple-Processor Scheduling
  - Real-Time CPU Scheduling

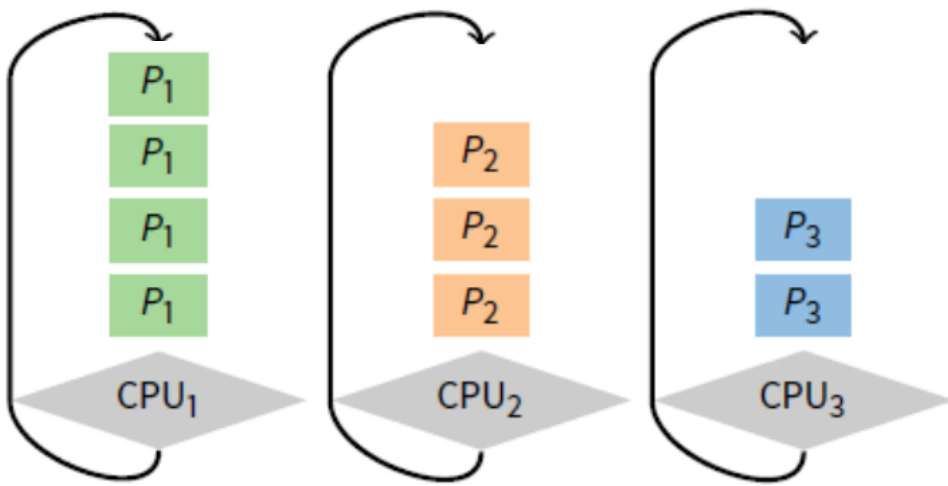
Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
   a. All threads may be in a common ready queue
   b. Each processor may have its own private queue of threads



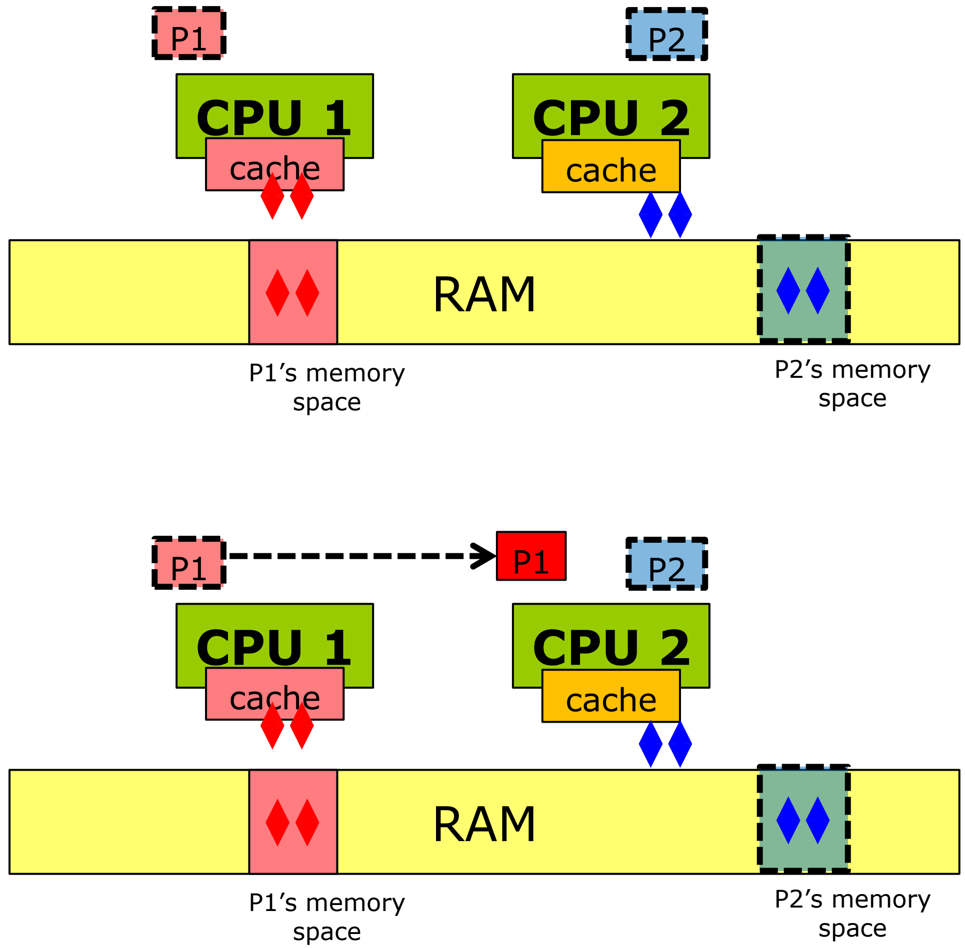
Load balancing

- **Load balancing** needed for per-core ready queues
  - Keep the workload between processors balanced
  - Migration
    - Move tasks from a busy processor to an idle processor



Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread. (*aka* **warm cache**)
  - A thread has **processor affinity**
  - A lot of **cache hits**!
- Moving processes between processors has costs
  - The new processor cache must be repopulated - more **cache misses**!
  - e.g., common ready queue
  - e.g., load balancing

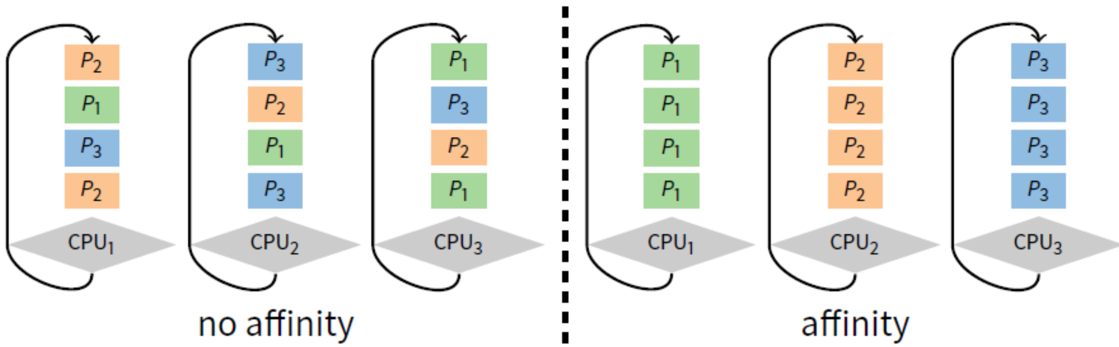


Processor Affinity

- To avoid **cache misses**, try to keep processes on same processor

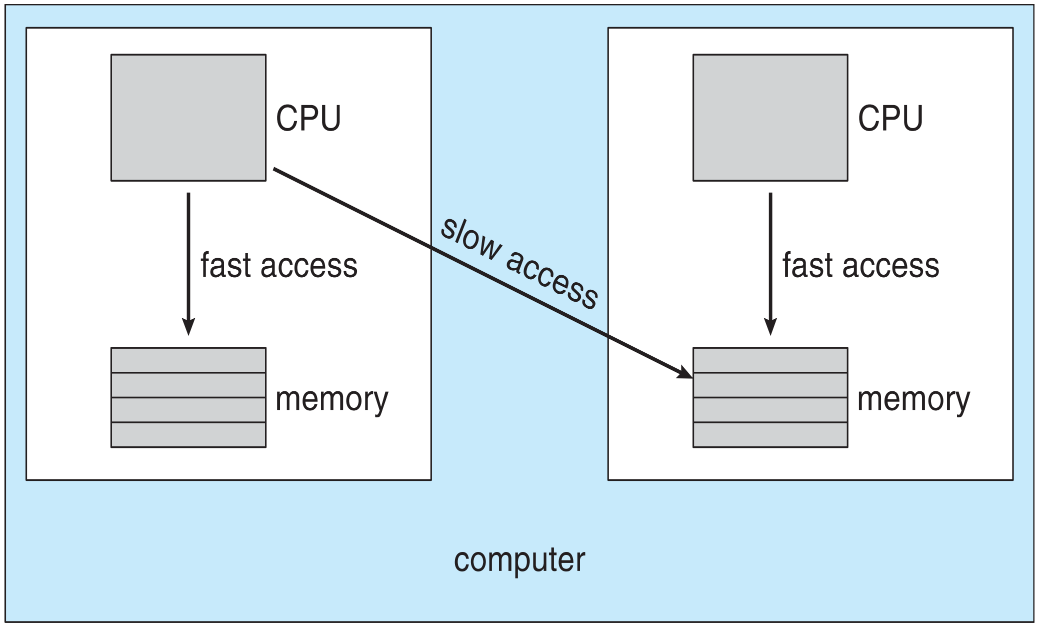
  = **Processor Affinity**

- Load balancing vs. Affinity Scheduling
  - Scheduler needs to balance between two



NUMA and CPU Scheduling

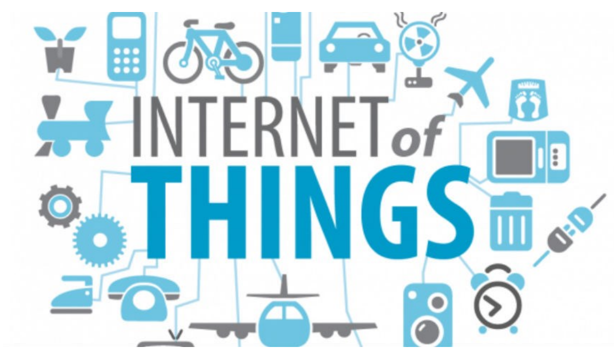
- If the operating system is Non-uniform memory access (**NUMA**)**-aware**, it will assign memory closes to the CPU the thread is running on.
- NUMA systems also need **Processor affinity** for faster memory access!



Chapter 5: CPU Scheduling

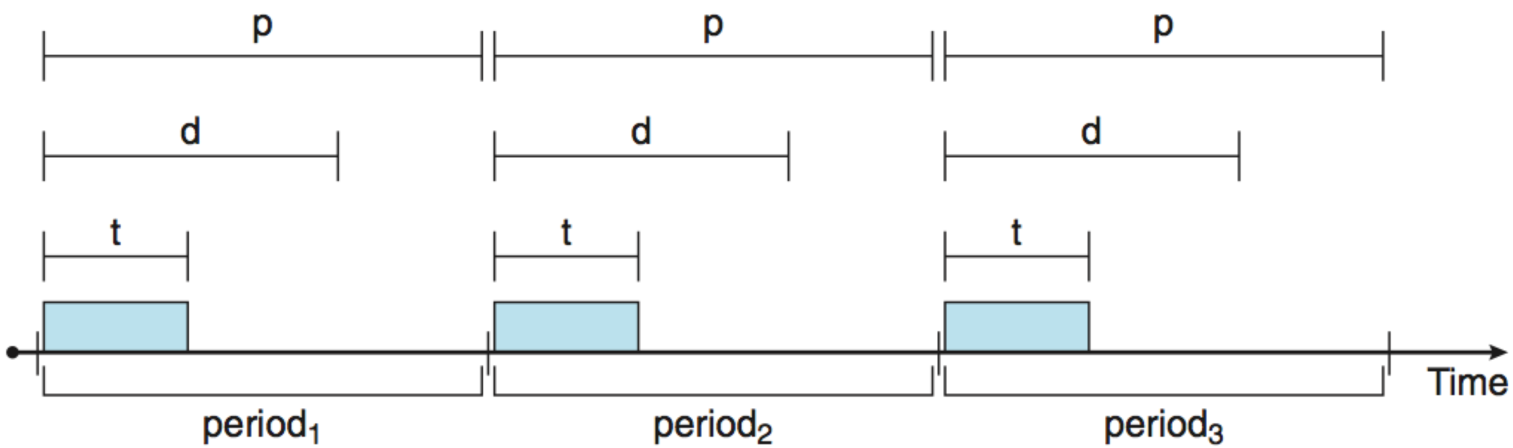
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Advanced Scheduling
  - Multiple-Processor Scheduling
  - Real-Time CPU Scheduling

freeRTOS



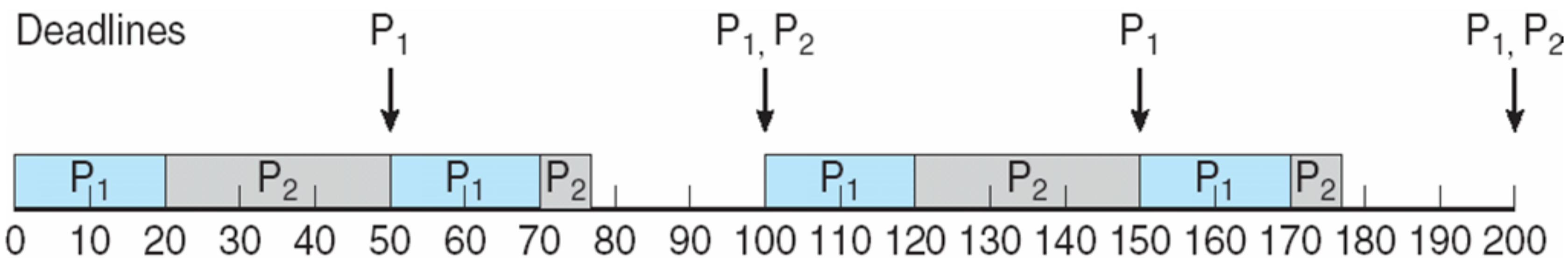
Real-time scheduling

- **Real-time scheduling** applications must provide ability to meet **deadlines**
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has fixed **processing time *t***, **deadline *d*, period *p***
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



Rate Montonic Scheduling

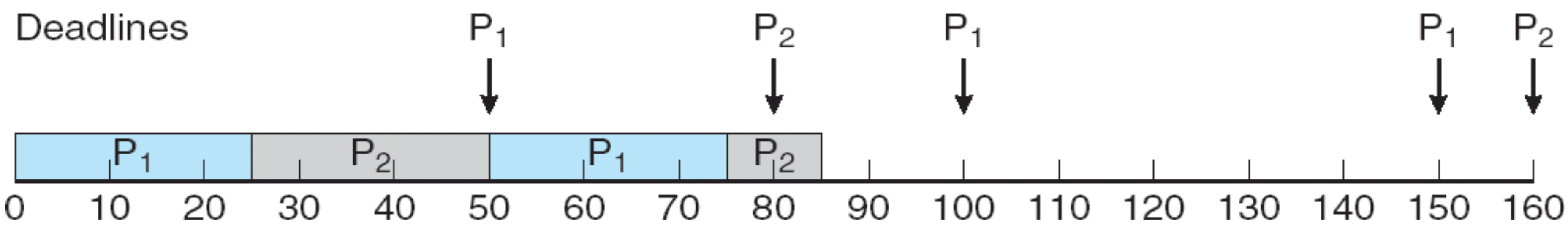
- A priority is assigned based on the inverse of its period (=rate)
  - Shorter periods = higher priority
  - Longer periods = lower priority
- Example
  - periods: p_1 = 50, p_2 = 100, CPU burst t_1 = 20, t_2 = 35
  - deadline = complete CPU burst before start of next period
  - P_1 is assigned a higher priority than P_2.



Missed Deadlines with Rate Monotonic Scheduling

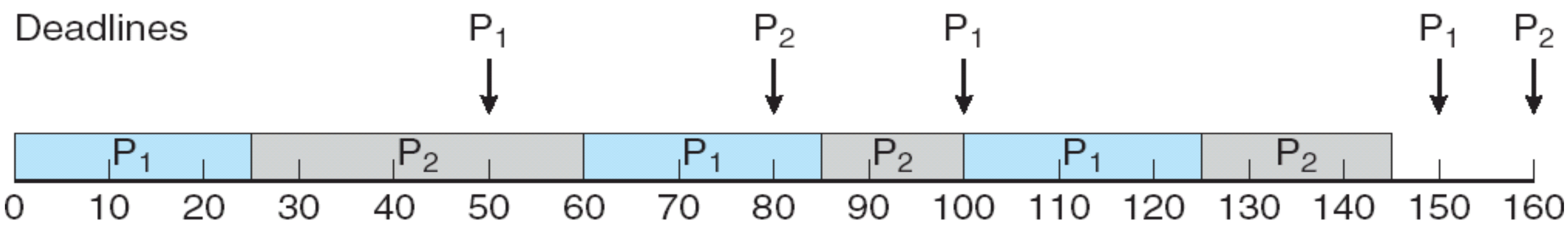
periods: $p_1 = 50$, $p_2 = 80$, CPU burst $t_1 = 25$, $t_2 = 35$

deadline = complete CPU burst before start of next period



Earliest Deadline First Scheduling (EDF)

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



  - periods: p_1 = 50, p_2 = 80, CPU burst t_1 = 25, t_2 = 35
  - deadline = complete CPU burst before start of next period

https://www.searchenginejournal.com/google-expands-rich-results-for-qa-pages-in-search/281355/