CPU Schedule

스케줄링 알고리즘

CPU 스케줄링 결정

1. 한 프로세스가 실행 상태에서 대기 상태로 전환될 때
   - ex) I/O 요청 or 자식 프로세스가 종료되기를 기다리기 위해 wait()을 호출할 때

2. 프로세스가 실행 상태에서 준비 완료 상태로 전환될 때
   - ex) 인터럽트가 발생할 때

CPU 스케줄링 결정

3. 프로세스가 대기 상태에서 준비 완료 상태로 전환될 때
   - ex) I/O의 종료 시

4. 프로세스가 종료할 때

CPU 스케줄링 결정(정리!)

1. 한 프로세스가 실행 상태에서 대기 상태로 전환될 때

2. 프로세스가 실행 상태에서 준비 완료 상태로 전환될 때

3. 프로세스가 대기 상태에서 준비 완료 상태로 전환될 때

4. 프로세스가 종료할 때

CPU 스케줄링 결정(정리!)

1. 한 프로세스가 실행 상태에서 대기 상태로 전환될 때

4. 프로세스가 종료할 때

=> 스케줄링 면에서는 선택의 여지가 없음,

실행을 위해 새로운 프로세스(준비 큐에 하나라도 존재할 경우)가
반드시 선택되어야 함

CPU 스케줄링 결정(정리!)

1. 한 프로세스가 실행 상태에서 대기 상태로 전환될 때

4. 프로세스가 종료할 때

=> **비선점, 협조적**

CPU 스케줄링 결정(정리!)

2. 프로세스가 실행 상태에서 준비 완료 상태로 전환될 때

3. 프로세스가 대기 상태에서 준비 완료 상태로 전환될 때

=> 여기서는 선택의 여지가 있음

CPU 스케줄링 결정(정리!)

2. 프로세스가 실행 상태에서 준비 완료 상태로 전환될 때

3. 프로세스가 대기 상태에서 준비 완료 상태로 전환될 때

=> 선점

비선점 vs 선점 - 비선점

일단 CPU가 한 프로세스에 할당되면?

프로세스가 종료하든지

or
대기 상태로 전환해

CPU를 방출할 때까지 점유함

비선점 vs 선점 - 선점

데이터가 Many Processes에 의해 share될 때
경쟁 조건을 초래할 수 있음

ex) 두 프로세스가 자료를 공유하는 경우,

한 프로세스가 자료를 갱신하고 있는 동안 선점되어
두 번째 프로세스가 실행 가능한 상태가 될 수 있음

비선점 vs 선점 - 선점

ex) 두 프로세스가 자료를 공유하는 경우,

한 프로세스가 자료를 갱신하고 있는 동안 선점되어

두 번째 프로세스가 실행 가능한 상태가 될 수 있음

This! 2가 데이터를 읽으려고 할 때, 데이터의 일관성은 이미 깨짐

비선점 vs 선점 - 선점

OS Kernel도 비선점 or 선점 방식으로 설계될 수 있음

선점형 커널에는 공유 커널 데이터 구조에 액세스할 때

경쟁 조건 방지하기 위해

mutex 락과 같은 기법 필요

비선점 vs 선점 - 선점

interrupt - 어느 시점에서건 일어날 수 있고, 커널에 의해서 항상 무시될 수는 없음

인터럽트에 의해 영향을 받는 코드 부분은 반드시 동시 사용으로부터 보호되어야 함

OS는 거의 항상 인터럽트를 받아들일 필요가 있음
(입력을 잃어버리거나 or 출력이 겹쳐서 쓰일 수 있음)

비선점 vs 선점 - 선점

OS는 거의 항상 인터럽트를 받아들일 필요가 있음
(입력을 잃어버리거나 or 출력이 겹쳐서 쓰일 수 있음)

이러한 코드는 Many Processes들이 병행으로 접근할 수 없도록

그 진입점에서 인터럽트를 불능화 & 출구에서 인터럽트를 다시 가능화함

Dispatcher

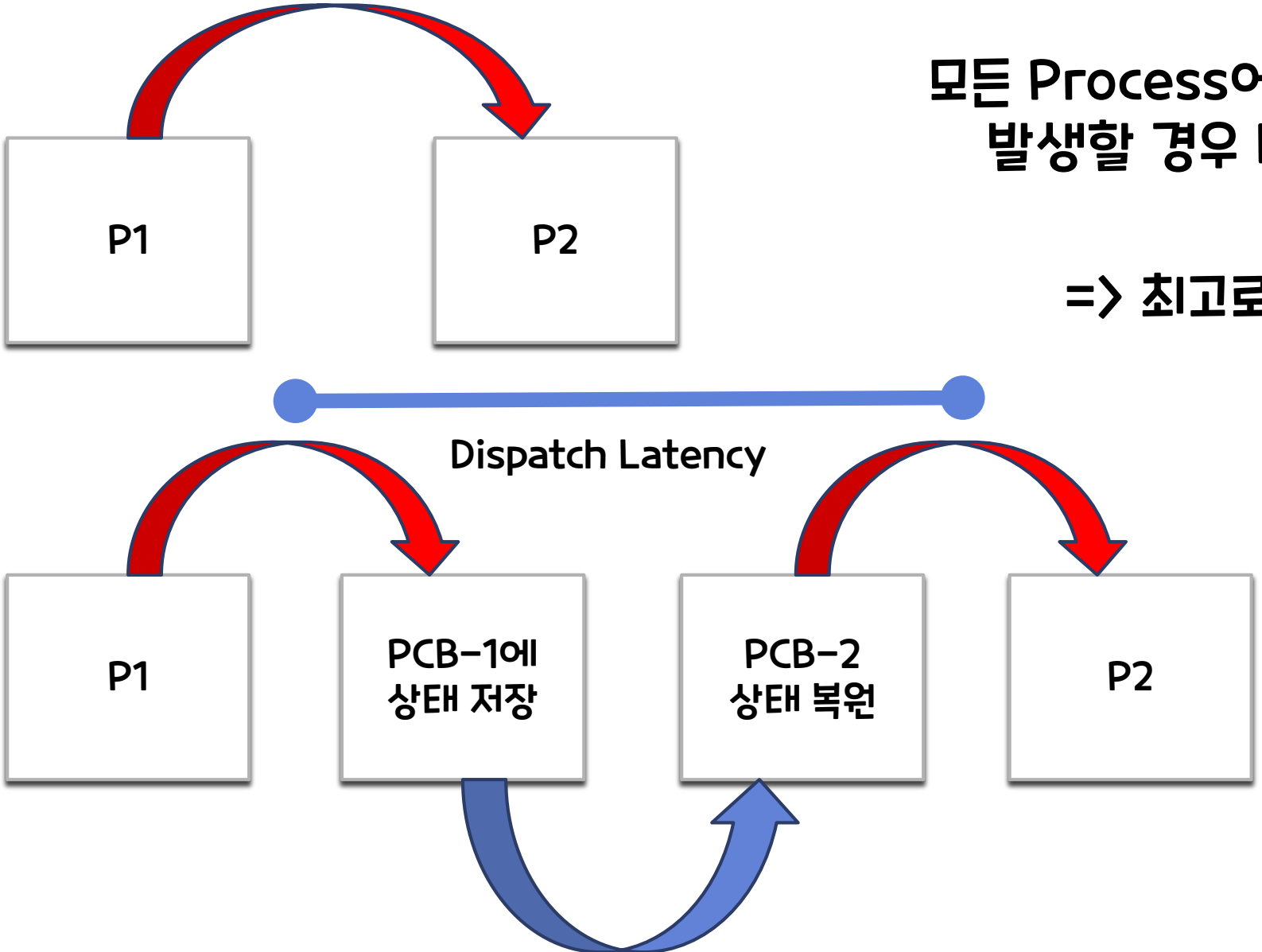
- CPU 코어의 제어를 CPU 스케줄러가 선택한
프로세스에 주는 모듈

1. 한 프로세스에서 다른 프로세스로 Context Switch

2. User Mode로 전환

3. 프로그램을 다시 시작하기 위해 사용자 프로그램의 적절한 위치로
JUMP하는 일

Context Switch?



모든 Process에서 Context Switch가 발생할 경우 Dispatcher가 호출됨

=> 최고로 빨리 수행되어야

모든 Process에서 Context Switch가 발생할 경우 Dispatcher가 호출됨

=> 최고로 빨리 수행되어야

Context Switch는 얼마나 자주 발생?

Linux의 vmstat 명령어로 알 수 있음

system의 cs 항목

- 46 : 시스템 부팅 이후 1초 단위의 평균 Context Switch 횟수

- 직전 1초 동안 384번의 Context Switch,
그 이전 1초 동안 339번의 Context Switch가 이루어졌음

Context Switch는 얼마나 자주 발생?

```
root@2ab97aa7e977:/# ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 11:34 pts/0    00:00:00 /bin/bash
root          17     1  0 11:42 pts/0    00:00:00 ps -ef
root@2ab97aa7e977:/# cat /proc/1/status | grep "ctxt_switches"
voluntary_ctxt_switches:        395
nonvoluntary_ctxt_switches:     3
root@2ab97aa7e977:/#
```

자발적 vs 비자발적

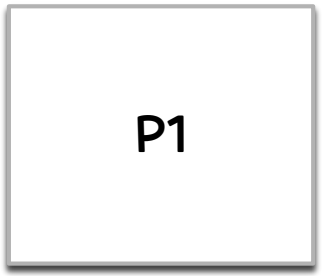
- Voluntary : 현재 사용 불가능한 자원(ex - I/O 기다리며 봉쇄됨)을 요청했기 때문
  - 프로세스가 CPU 제어를 포기한 경우

- NonVoluntary : Time slice 만료 or 우선순위가 더 높은 Process에 의해 선점된 경우
  - CPU를 빼앗겼을 때 발생

Scheduling Algorithm

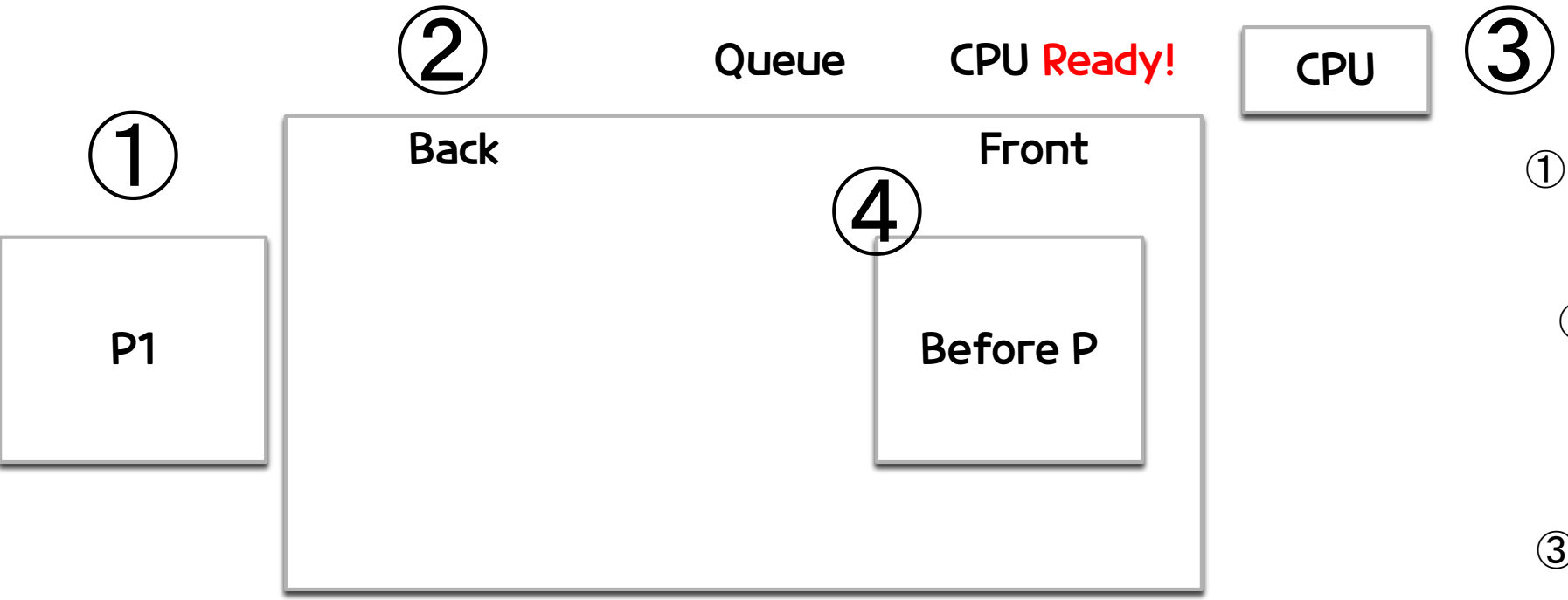
- FCFS – 선입 선처리 스케줄링
- SJF – 최단 작업 우선 스케줄링
- RR – 라운드 로빈 스케줄링
- Priority Scheduling - 우선순위 스케줄링

FCFS – First Come, First Serve



- CPU를 먼저 요청하는 Process가 CPU를 먼저 할당받음
  - Queue이용

FCFS – First Come, First Serve



평균 대기 시간은 종종 대단히 길 수 있음

① 프로세스가 준비 큐에 진입

② 이 Process의 PCB를
Queue 끝에 연결

③ CPU가 가용상태가 되면,
준비 큐의 앞부분에 있는 프로세스에 할당됨

④ 이 실행 상태의 프로세스는 이어
준비 큐에서 제거됨

FCFS – First Come, First Serve

Process	Burst Time
P1	24
P2	3
P3	3

평균 대기 시간 : (0 + 24 + 27) / 3 = 17ms

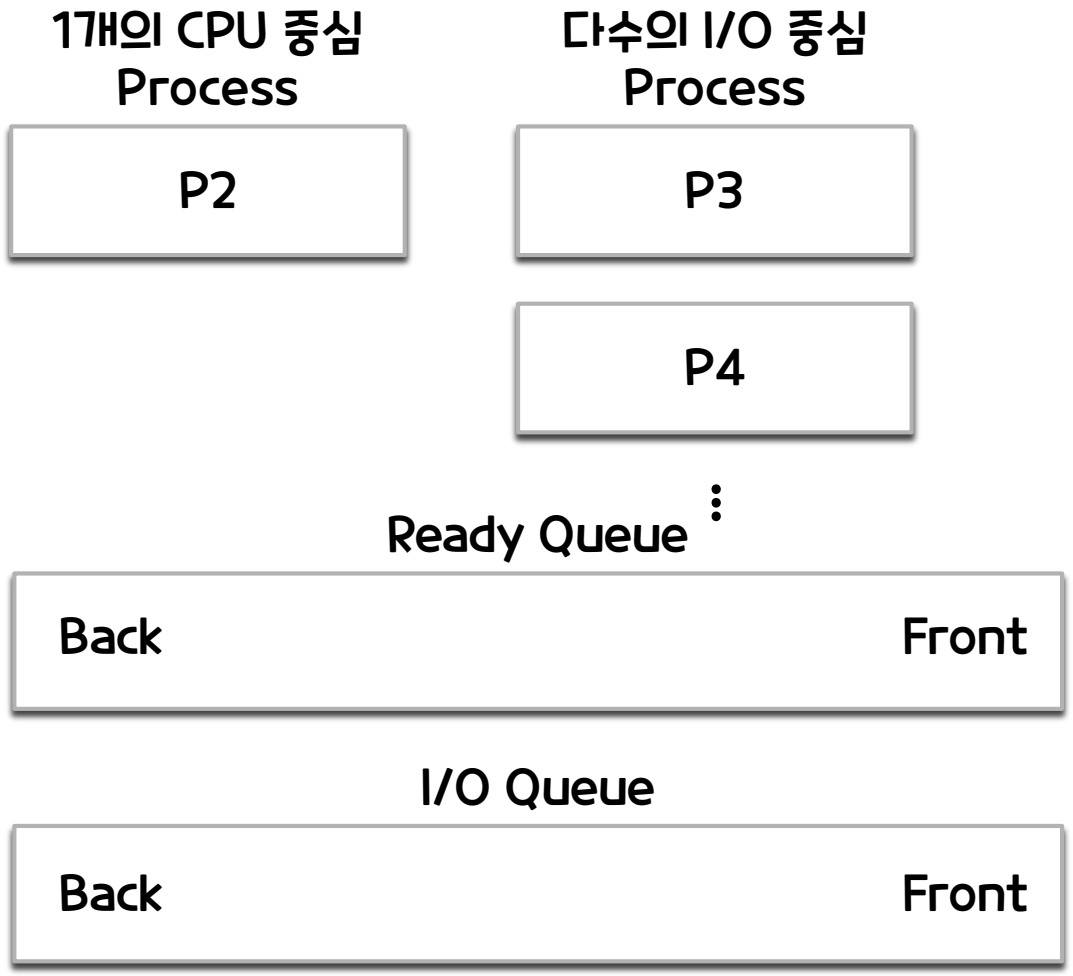
FCFS – First Come, First Serve

Process	Burst Time
P1	24
P2	3
P3	3

평균 대기 시간 : (6 + 0 + 3) / 3 = 3ms

- FCFS에서 평균 대기 시간은 일반적으로 최소가 아님

FCFS - First Come, First Serve(동적)



1. CPU 중심 P가 CPU를 할당 받아 점유

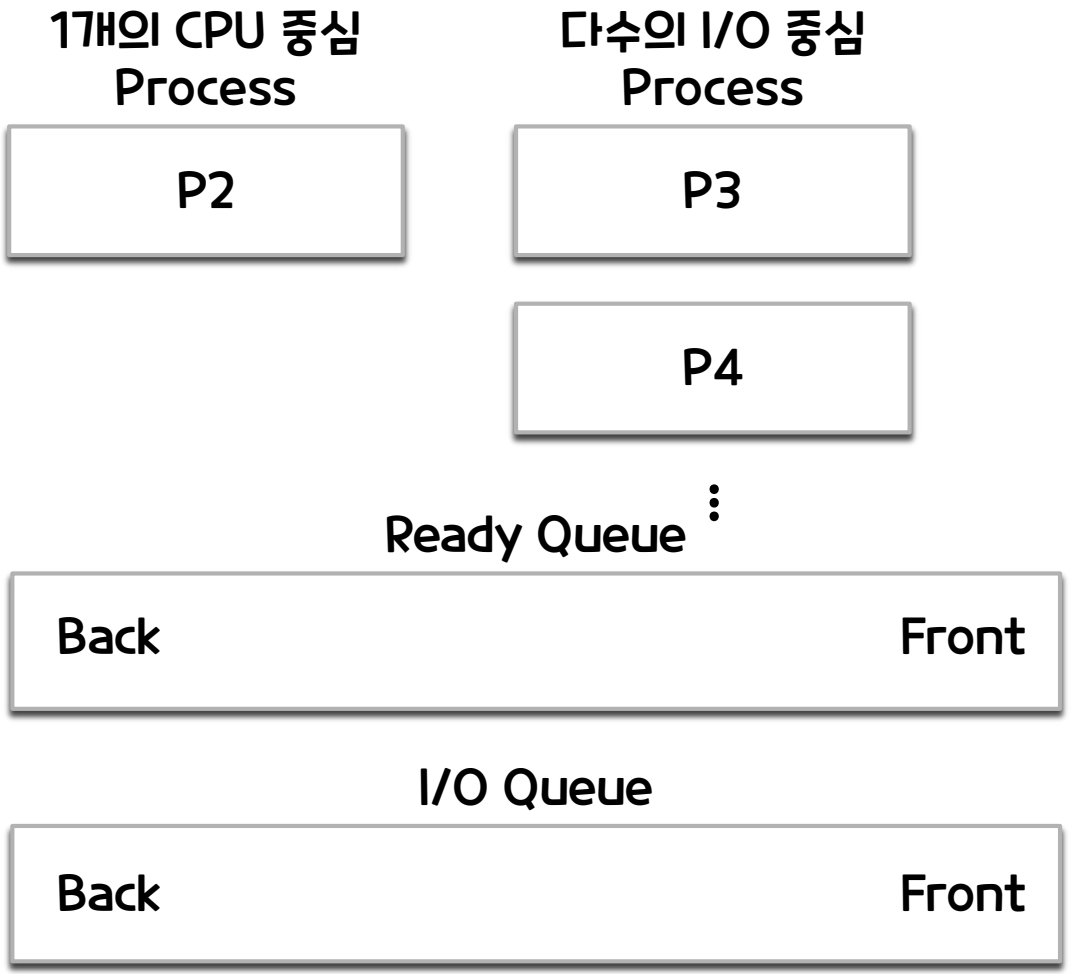
2. 그 동안 다른 P들은 I/O 끝내고 준비 큐로 이동해 CPU를 기다림

- 이 순간! P들이 준비 큐에서 기다리는 동안 I/O장치들 쉬고 있음

3. CPU 중심 P가 자신의 CPU Burst를 끝내고 I/O 장치로 이동

- 모든 I/O 중심의 프로세스들은 매우 짧은 CPU Burst를 갖고 있음

FCFS - First Come, First Serve(동적)



모든 I/O 중심의 프로세스들은 매우 짧은 CPU Burst를 갖고 있음

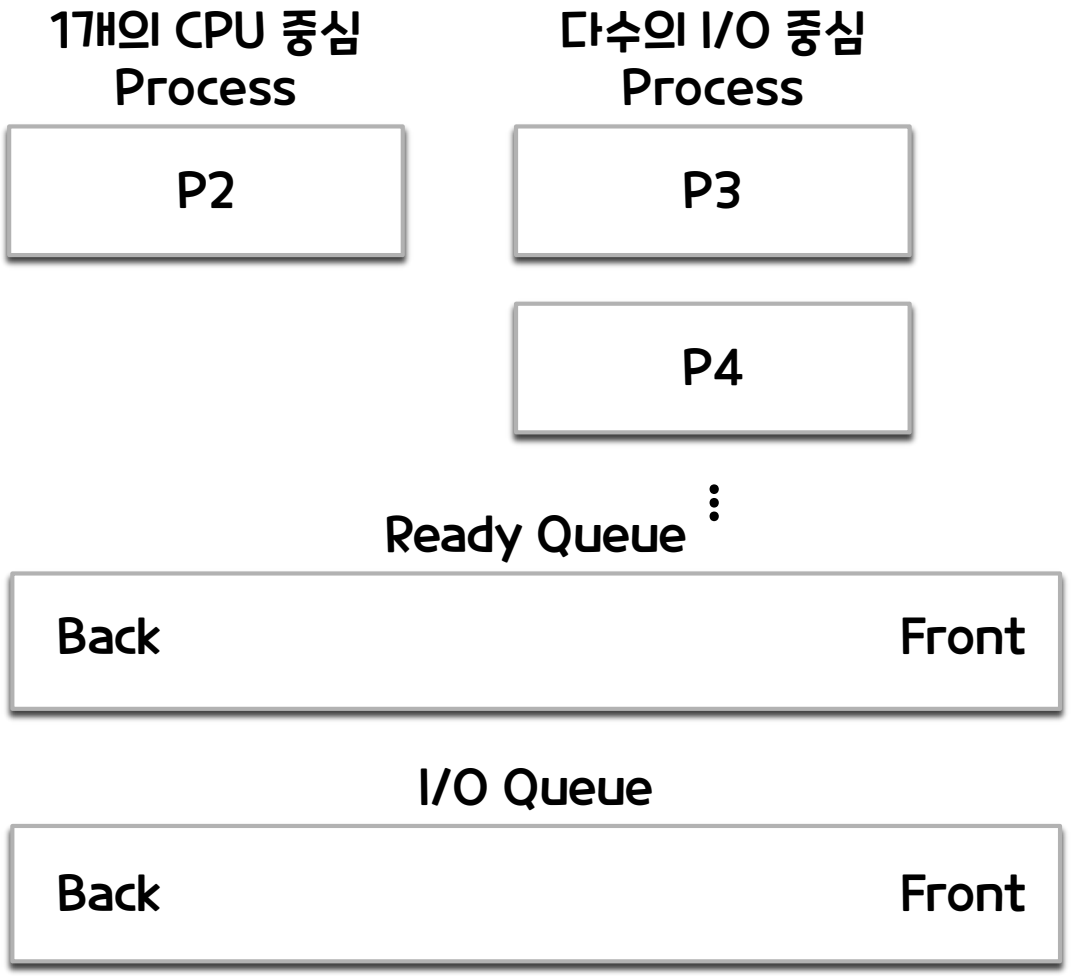
CPU 작업을 신속하게 끝내고 다시 I/O 큐로 이동

(이 시점에 CPU가 쉬게 됨)

CPU 중심 P는 다시 준비 큐로 이동해 CPU를 할당 받음

CPU 중심 프로세스가 끝날 때까지, 모든 I/O 프로세스들은 다시 준비 큐에서 기다리게 됨

FCFS - First Come, First Serve(동적)



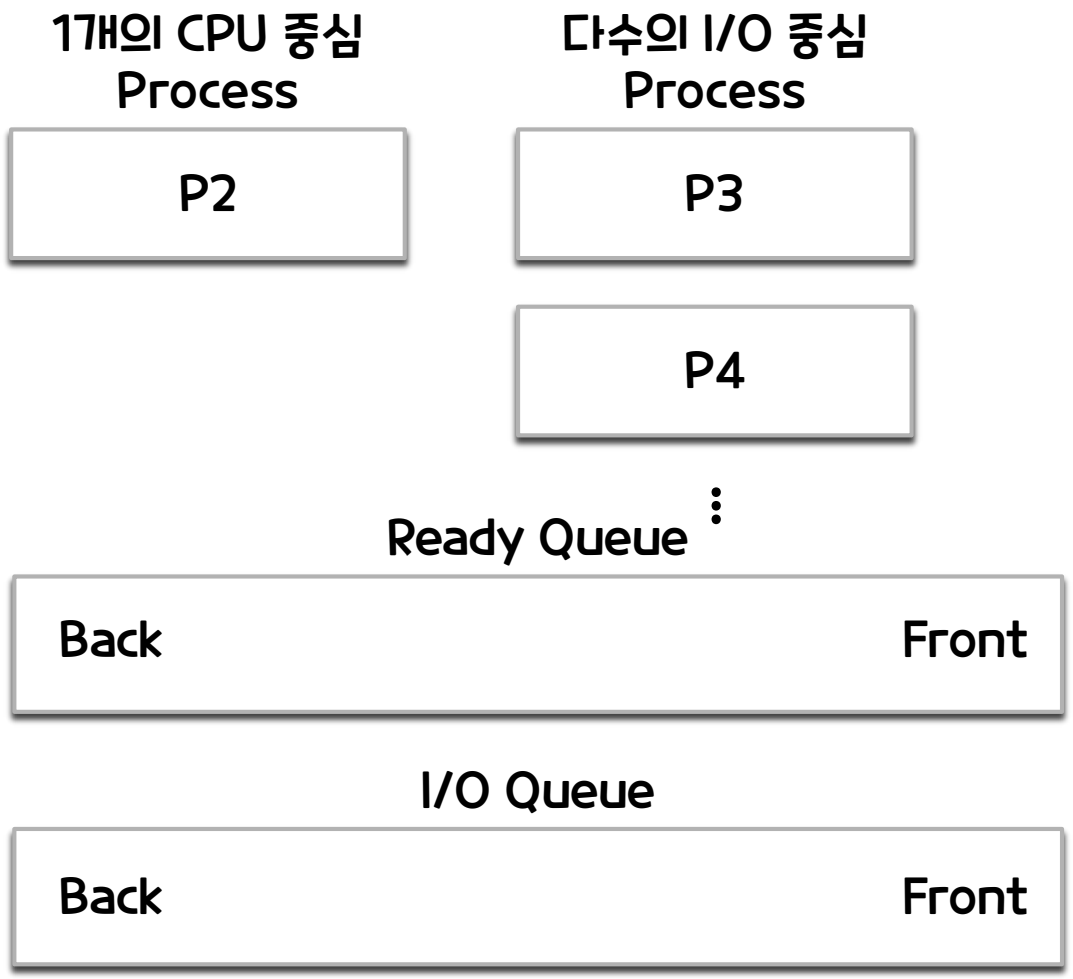
Convoy effect : 호위 효과

모든 다른 프로세스들이 하나의 긴 프로세스가 CPU를 양도하기를 기다리는 것

짧은 프로세스들이 먼저 처리되도록

허용될 때보다 CPU, 장치 이용률이 저하됨

FCFS - First Come, First Serve(동적)



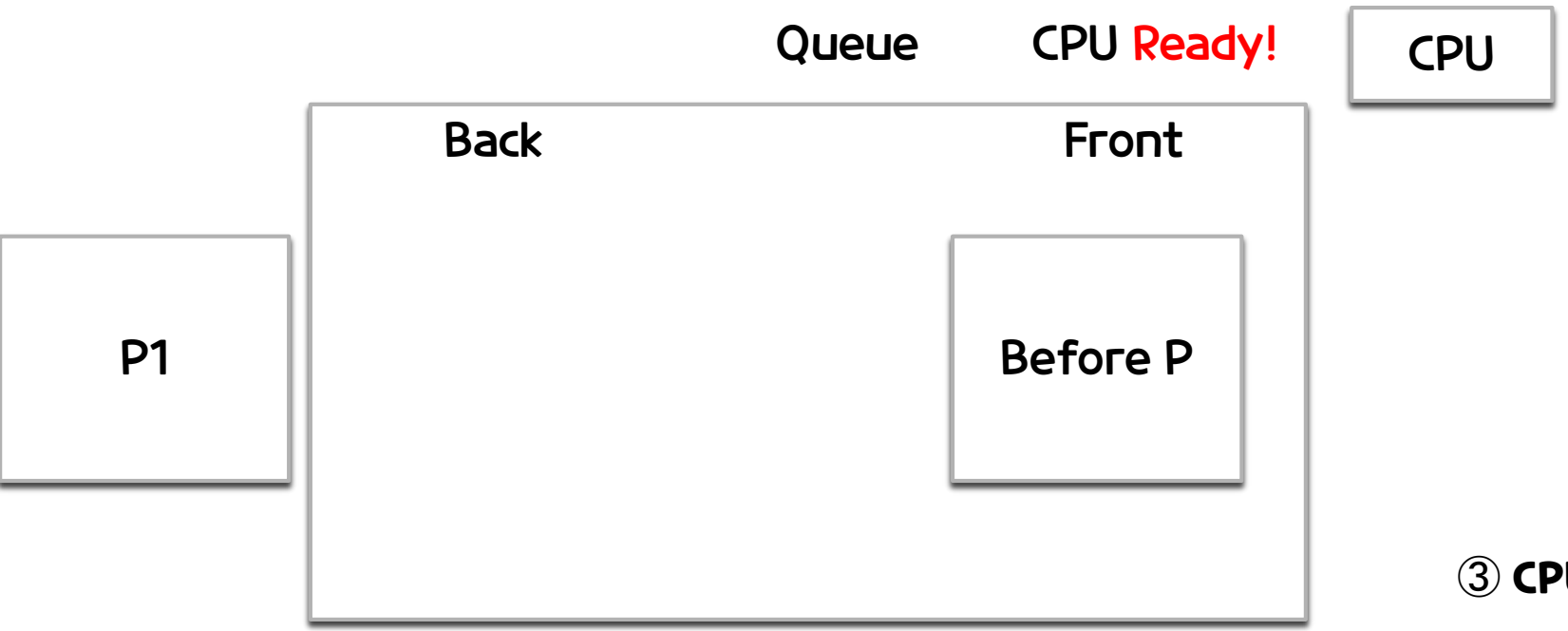
FCFS = 비선점

일단 CPU가 할당되면, 그 P가 종료 or I/O 처리를 요구해 CPU를 방출할 때까지 CPU를 점유

대화형 시스템에서 문제!

why? 각 프로세스가 규칙적인 간격으로 CPU의 몫을 얻는 것이 매우 중요!

FCFS – First Come, First Serve



평균 대기 시간은 종종 대단히 길 수 있음

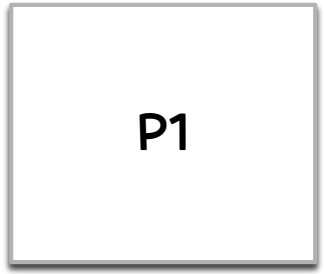
① 프로세스가 준비 큐에 진입

② 이 Process의 PCB를
Queue 끝에 연결

③ CPU가 가용상태가 되면, 준비 큐의 앞부분에 있는
프로세스에 할당됨

④ 이 실행 상태의 프로세스는 이어
준비 큐에서 제거됨

SJF - Shortest Job First Scheduling



- 각 Process의 다음 CPU Burst 길이를 연관시킴
- CPU가 이용 가능해지면, 가장 작은 다음 CPU Burst를 가진 Process에 할당함
- if 두 Process가 동일한 길이의 다음 CPU Burst를 가진다면?

  - 순위를 정하기 위해 FCFS 스케줄링 적용

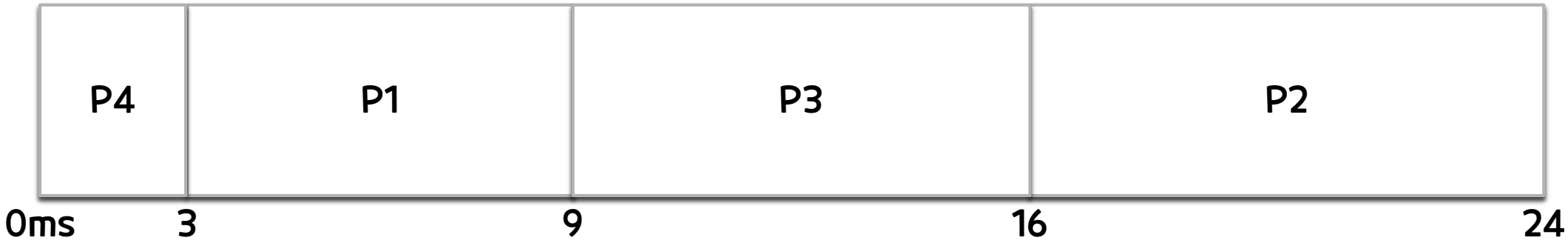
SJF - Shortest Job First Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

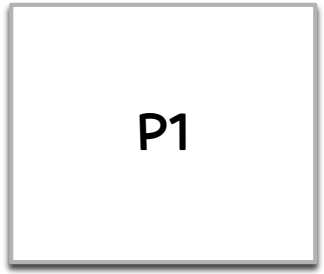
평균 대기 시간 : (3 + 16 + 9 + 0) / 4 = 7ms

만약 FCFS였다면? (0+6+14+21) / 4 = (41) / 4

= 10.25



SJF - Shortest Job First Scheduling



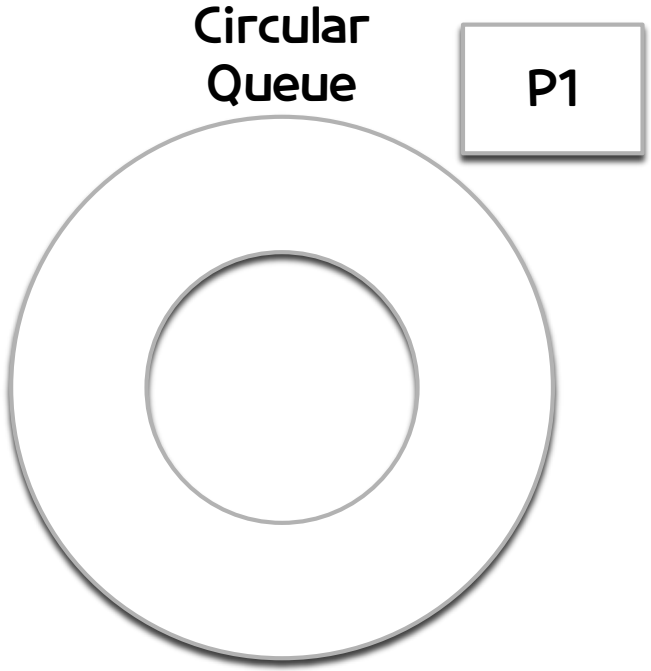
- 주어진 Process 집합에 대해 최소의 평균대기 시간을 가진다는 점 - 증명
- why 최적일까?

짧은 프로세스를 긴 프로세스의 앞으로 이동함으로써,

짧은 프로세스의 대기 시간을 긴 프로세스의 대기 시간이 증가하는 것보다 더 많이 줄일 수 있음

=> 평균 대기 시간 줄어듬

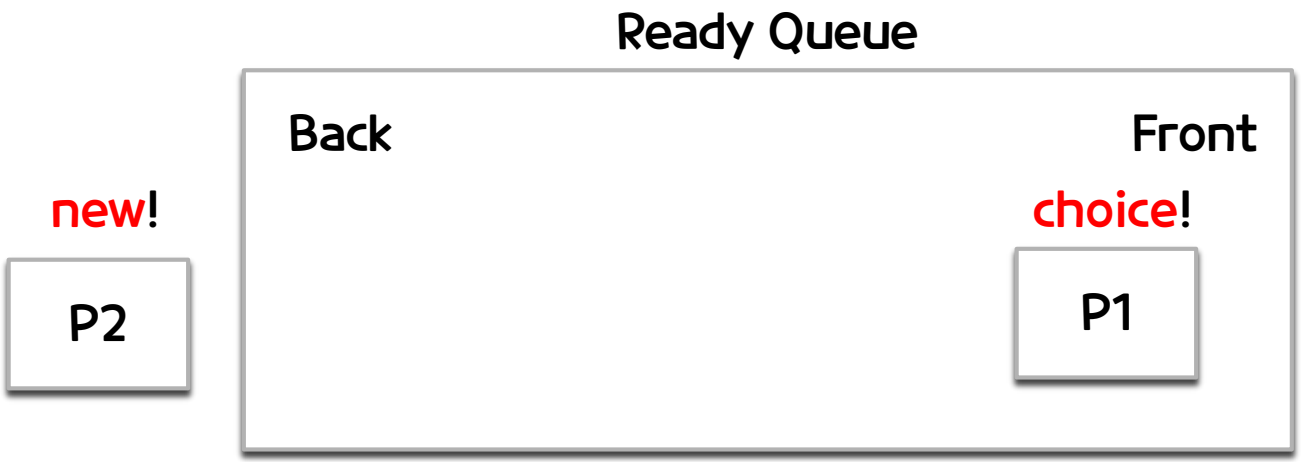
RR - Round-Robin Scheduling



- FCFS와 유사하지만, 시스템이 Process들 사이로 옮겨 다닐 수 있도록 선점이 추가됨
- Time Quantum(시간 할당량) or Time Slice(타임슬라이스)라고 하는 작은 단위의 시간 정의

  (일반적으로 10~100ms)
  - 준비 Queue는 원형 큐로 동작
- CPU Scheduler는 Queue를 돌면서 한 번에 한 Process에 한 번의 Time slice 동안 CPU를 할당

RR - Round-Robin Scheduling



- Ready Queue가 FIFO Queue라고 생각
  - 새로운 P는 RQ의 Back에 추가
- CPU 스케줄러는 RQ에서 1번째 P를 선택

=> 한 번의 Time Slice 이후에 Interrupt를 걸도록 Timer를 설정하고 P를 Dispatch

RR - Round-Robin Scheduling

2가지 경우!

1. Process CPU Burst < Time Slice

- Process 자신이 CPU를 자발적으로 방출

=> scheduler는 RQ에 있는 next P로 진행

2. Process CPU Burst > Time Slice

- Timer가 끝나고 OS에 interrupt를 발생

- Context Switch가 일어나고 실행 중인 P는 RQ의 Back에 Push

=> scheduler는 RQ에 있는 next P로 진행

RR - Round-Robin Scheduling

유일하게 실행 가능한 P가

아니라면 연속적으로 두 번 이상의

Time slice을 할당받는 P는 X

Time Slice size가 너무 작다

=> 1ms, 매우 많은 Context Switch

너무 크면, FCFS와 같음

CPU Burst가

한 번의 시간 할당량을 초과하면?

- P는 선점되고 RQ로 돌아감

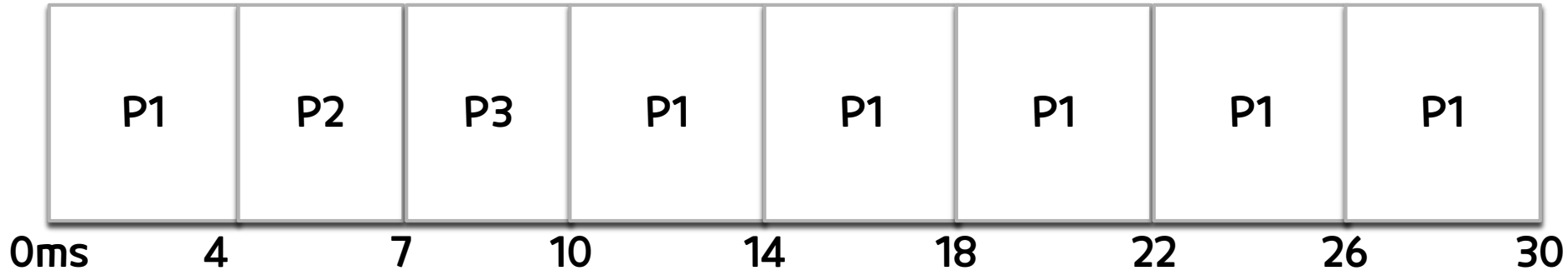
-> RR은 선점형

RR - Round-Robin Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

Time Slice를 4ms로 한다고 가정!

P1 : 10-4 = 6ms, P2 : 4ms, P3 : 7ms



Priority Scheduling

- SJF는 우선순위 스케줄링 알고리즘의 특별한 경우

- 우선순위가 각 P에 연관, CPU는 가장 높은 우선순위를 가진 P에 할당됨

- SJF = CPU Burst가 클수록 우선순위가 낮음(역도 성립)

- 선점형, 비선점형 모두 가능
- 선점형 – P가 RQ에 도착하면, new P와 executing P의 우선순위 비교
- 비선점형 – 단순히 RQ에 new P 추가

Priority Scheduling - Problem!

- 무한 봉쇄(indefinite Blocking)
- 기아 상태(starvation)

  - 실행 준비는 되어 있지만 CPU를 사용하지 못하는 P => CPU를 기다리며 Blocking
  - 낮은 우선순위 P들이 CPU를 무한히 대기하는 경우가 발생

- if 부하가 많은 Computer 시스템에선, 높은 우선순위 알고리즘이 꾸준히 들어와 낮은 우선순위 P들이 CPU를 얻지 못할 수 있음

1. P가 결국 실행 or 2. 시스템이 결국 crash해 낮은 P들을 잃어버리는 경우

Priority Scheduling - Solution!

1. 노화(aging)

- 오랫동안 시스템에서 대기하는 P들의 우선순위를 점진적으로 증가시킴
  - ex) 127(낮음) ~ 0(높음) 범위의 경우,
    - 주기적으로 대기 중인 P들의
    - 우선순위를 1씩 증가시킬 수 있을 것
  - 1초마다 증가시킨다면 2분을 겨우 넘는 시간만 걸림

2. RR + Priority Scheduling

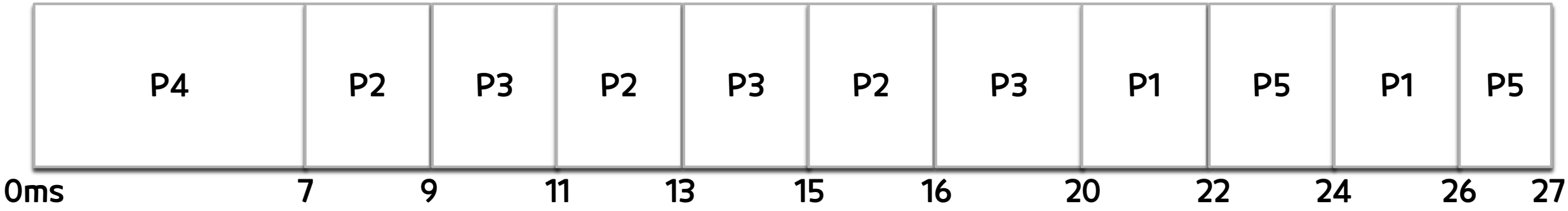
- 우선순위가 가장 높은 P를 실행하고
- 우선순위가 같은 P들은 RR를 사용해 스케줄

Priority Scheduling

Process	Burst Time	Arrived Time
P1	4	3
P2	5	2
P3	8	2
P4	7	1
P5	3	3

Time Slice를 2ms로 한다고 가정!

P4 - P2 - P3 - P1 - P5 순으로 들어옴



@STOP@

39ghwjd@naver.com