

운영체제

Lecture 05: 스케줄링



전남대학교 인공지능학부
박태준 (taejune.park@jnu.ac.kr)

지난 시간 복습

- ▶ 프로세스의 컨텍스트 스위칭 문제 → 쓰레드 개념 등장
 - 오늘날 프로세스는 쓰레드의 컨테이너! 프로그램의 실행 단위는 쓰레드!
 - 함수 하나가 쓰레드 하나라 보면 쉬움!
 - 주의! Call stack 쌓이는 것과, Thread로 생성되어 처리되는 것은 구분!
 - 쓰레드 주소 공간은 프로세스의 주소 공간 내에!
- ▶ 쓰레드의 生과 死
 - 쓰레드 생명 주기: 프로세스와 거의 유사!
 - 프로세스에 PCB가 있으면, 쓰레드엔 TCB가 있다!
- ▶ Multi-threading model
 - User-level thread
 - Kernel-level thread, Pure kernel-level thread

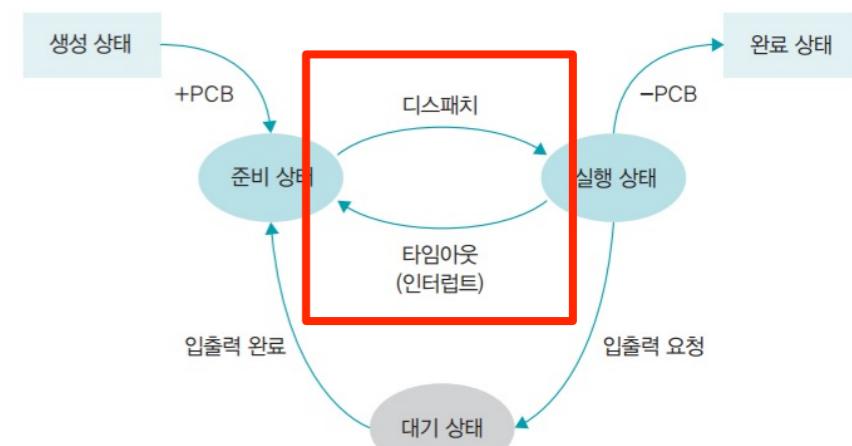
지난 시간 지금까지 배웠던 내용들 정리

- ▶ 운영체제가 하는 일: 컴퓨터라는 하드웨어를 '잘' 쓰기 위한 것
 - 효율적이면서도, 안정적이면서, 확장도 잘되고, 편리하게!
 - 그런데 컴퓨터는 한정된 자원을 두고 여럿이서 사용해야한다! → 그리고 다중 프로그래밍
 - 이걸 위해 프로세스, 메모리, 파일, 장치, 네트워크, 보안, 기타 등등의 관리 기능들이 있음
- ▶ 그래서 지금까지 배운 것: **프로세스 및 쓰레드 개념**
 - 폰노이만 구조에 의해서 프로그램은 → 프로세스가 되어야한다 (i.e., 메모리에 적재되어야 한다)
 - 이걸 잘 하기 위한 방법들:
 - 여러 프로세스를 처리하기 위해 오르락 내리락 → 프로세스 생명주기와 Context switching, 그리고 TCB
 - 메모리에 올릴 때는? 여러 프로세스들이 오르락 내리락 하다보니, 자리가 확정적이지 못하다!
 - → Loader를 통해서 → 메모리의 공간을 할당 받고 → Relocation
 - 그리고 각 프로세스는 혼자 통째로 메모리를 쓴다 (고 생각되게 만든다!): 가상메모리, 그리고 memory layout
 - 프로세스 만드는거, 너무 힘들던데? fork() and exec() 기반으로 복사 후 실행으로 대응해보자
- ▶ 프로세스도 매번 새로 만들어 올리기 너무 무겁더라.
 - 특히 여러 기능들을 구현할 때 Context switching 관련해서..
 - 그래서 **프로세스를 뜯개자! 쓰레드 개념**

그래서 요약하면

- 지금까지 배운 내용들은 여러 프로그램들을 한꺼번에 다루기 위해, '**메모리에 어떻게 잘 올리냐 내리냐, 어떻게 하면 빠르게 바꾸는가**' 하는 방법들에 대해 죽 배운 것!
 - 메모리에 프로세스를 적재하거나 내리기 위한 방법 자체에 대해 배운 것.
 - 즉, '공간'적 개념에 대해 주로 배웠다고 보면됨.
- 여기서 빠진 중요한 포인트 '**언제**', '**누구를**' 처리 할 것인가? / 뺄 것인가?
 - 음... 일단 time-sharing, 즉 시분할로 돌아가며 쓴다고도 했었고...
 - 또 timeout 일어날 때 interrupt 가 발생해서 running state → ready state가 된다고는 설명드렸습니다 (또는 terminated, blocked 경우도 있지만 이건 너무 자명하니...!)

- 그 Timeout 시간이 얼마만큼 이길래,
어떤 기준으로, 잘 돌아가던 프로세스를 중단시키는 걸까?



Goal

- ▶ CPU 스케줄링의 목표와 평가기준에 대한 이해
 - 스케줄링 시 고려사항들
 - 우선순위(Priority)의 개념
- ▶ 스케줄링의 작동 원리와 방식
- ▶ 스케줄링 종류
 - FCFS, SJF, HRN, RR, SRT, MultiLevel Queue Scheduling 등... (음... 좀 많아요...)
 - 선점과 비선점
 - 기아(Starvation), 에이징(aging)
- ▶ 너무 빨라서도, 너무 늦어서도 안되는 그 절묘함...!

CPU 스케줄링의 개요

운영체제의 꽃!

자원(Resource)이 부족한 곳에는 항상 스케줄링이 존재합니다.

- ▶ 일상 생활에서도...!
- ▶ 컴퓨터 시스템 안에서는?
 - 작업(job) 스케줄링
 - 대기중인 배치 작업 중에서 메모리에 적재할 작업
 - CPU 스케줄링
 - 프로세스/스레드 중 하나를 선택하여 CPU 할당
 - (오늘날 CPU 스케줄링은 스레드 스케줄링)
 - 디스크 스케줄링
 - 디스크 장치 내에서, 입출력 요청 처리 순서
 - 나중에 배움!
 - 프린터 스케줄링
 - 네트워크 스케줄링
 - 서버에서의 요청 처리 (e.g., 티켓팅)



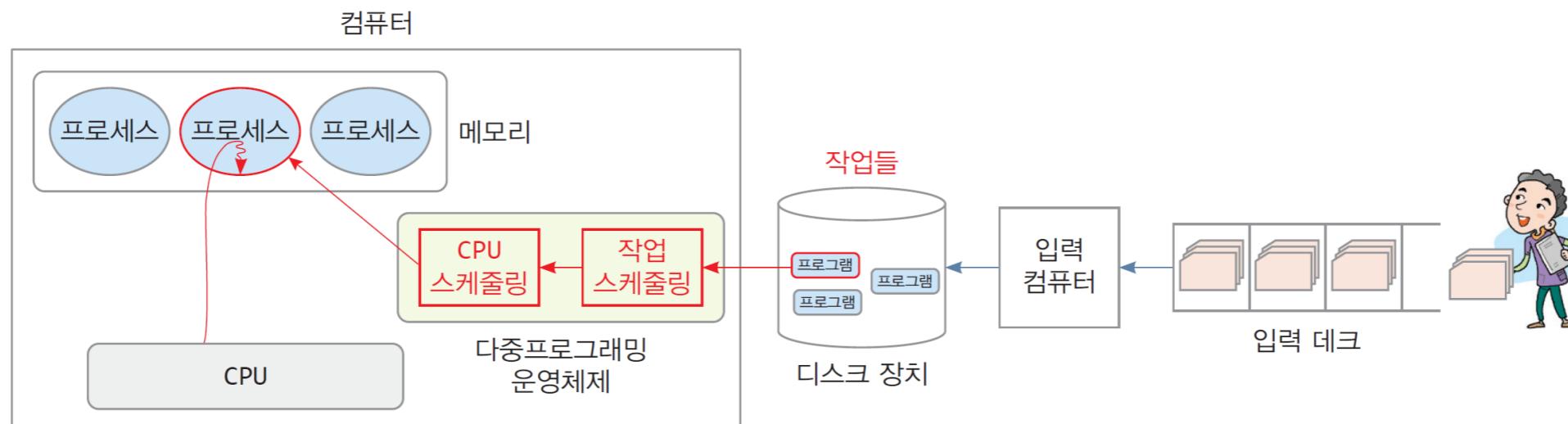
그림 4-1 식당 관리자의 역할



그림 4-2 조리 순서 변경

다중 프로그래밍과 스케줄링

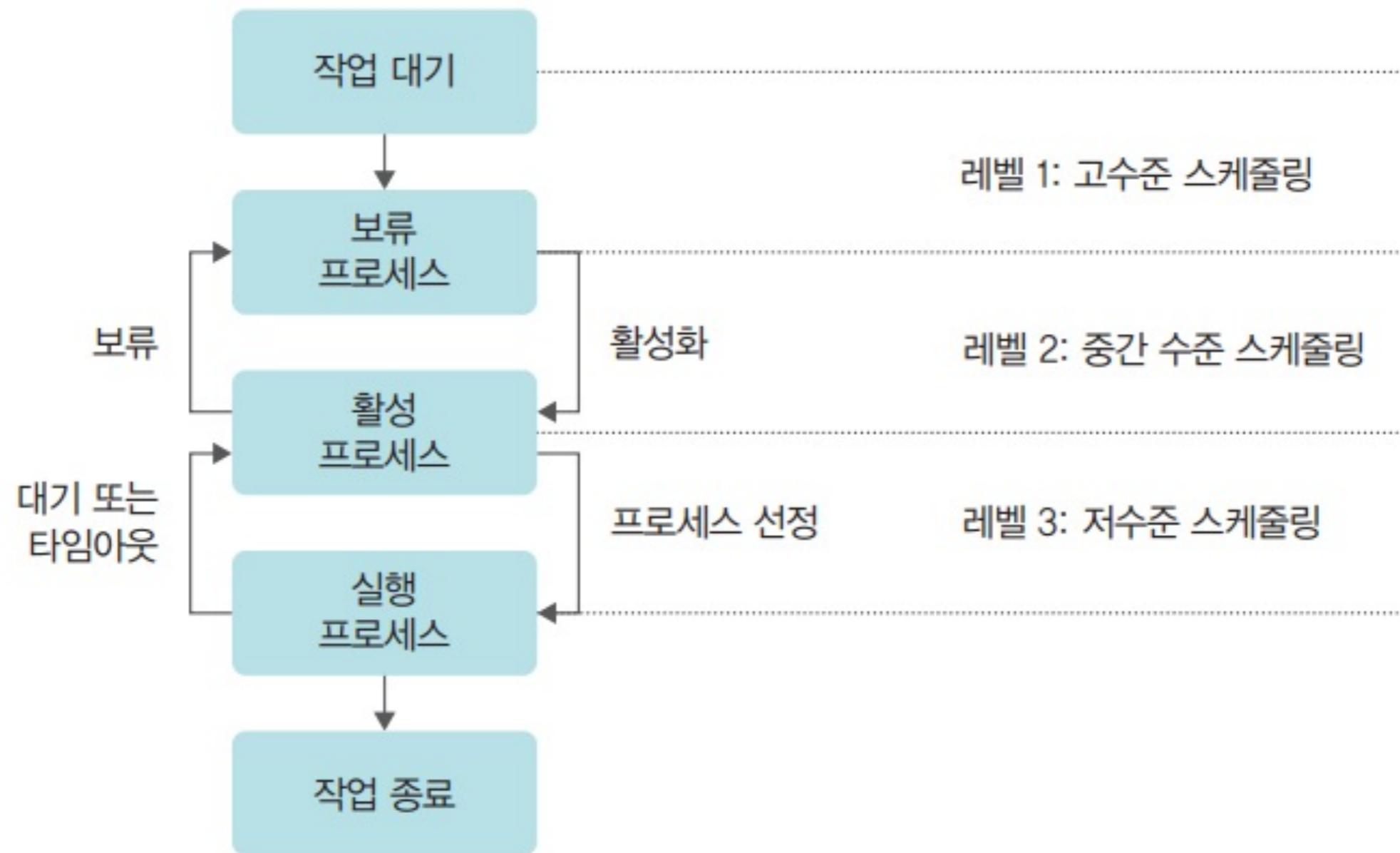
- ▶ 다중 프로그래밍의 도입 목적
 - 여러 프로세스가 하나의 CPU를 돌아가며 사용하여야 한다
 - 어떤 순서로 작업을 시킬지, 스케줄링 개념이 필요함
 - 이걸 잘하면 → CPU의 유휴(대기) 시간이 줄어들고 → CPU 활용률 향상
 - 예, 프로세스가 I/O를 요청하면 다른 프로세스에게 CPU 할당
- ▶ 그래서 다중 프로그래밍에는 두가지 스케줄링 개념이 필요합니다!
 - 작업 스케줄링(job scheduling): 디스크 장치로부터 메모리에 올릴 작업 선택하기
 - CPU 스케줄링(CPU scheduling): 메모리에 적재된 작업 중 CPU에 실행시킬 프로세스 선택



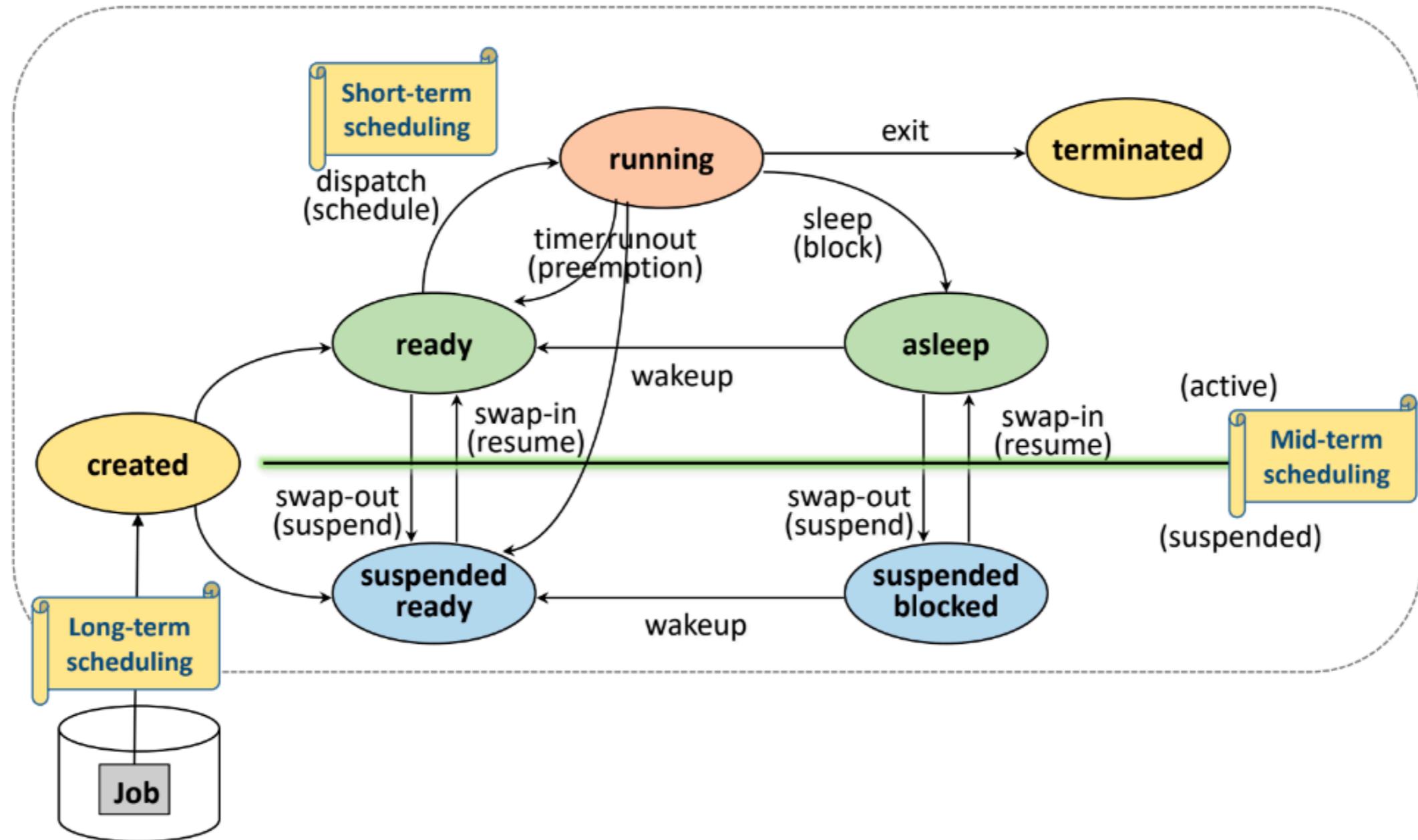
스케줄링 단계(Level)에 따른 구분

- ▶ 발생하는 빈도 및 할당 자원에 따른 구분
- ▶ 고수준 스케줄링 (**Long-term scheduling**) ~ **Job scheduling**
 - 시스템 내의 전체 작업 수를 조절하는 것, 시스템 내에서 동시에 실행 가능한 프로세스의 갯수 결정
 - 다중 프로그래밍의 정도(Degree)를 결정
 - 어떤 작업을 시스템이 받아들일지 또는 거부할지를 결정
 - 장기 스케줄링, 작업 스케줄링, 승인 스케줄링이라고도 함
- ▶ 중간 수준 스케줄링 (**Mid-term scheduling**) ~ **Swapping**
 - 중지와 활성화로 전체 시스템의 활성화된 프로세스 수를 조절하여 과부하를 막음
 - 일부 프로세스를 중지 상태로 옮김으로써 나머지 프로세스가 원만하게 작동하도록 지원
 - 메모리 할당 관련
- ▶ 저수준 스케줄링 (**Short-term scheduling**) ~ **Processor/CPU scheduling**
 - 어떤 프로세스에 CPU를 할당할지, 어떤 프로세스를 대기 상태로 보낼지 등을 결정
 - 아주 짧은 시간에 일어나기 때문에 단기 스케줄링이라고도 함

스케줄링 단계(Level)에 따른 구분 (그림)



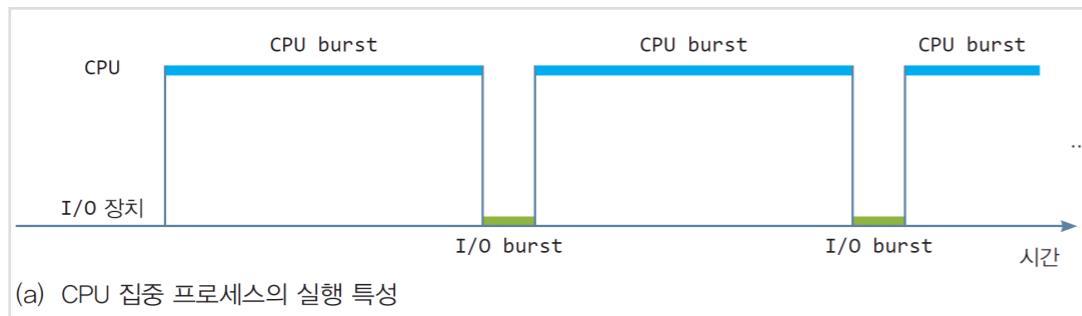
스케줄링 단계(Level)에 따른 구분 (그림)



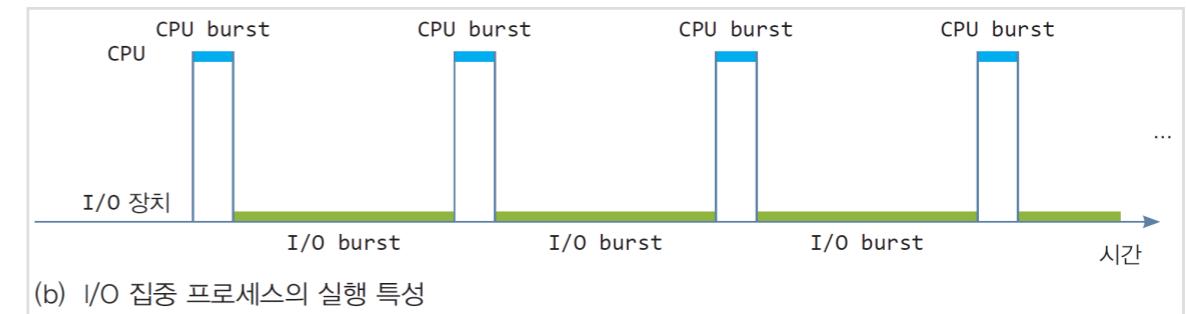
뭘 보고 정할까?

▶ 프로세스는 CPU 연산 작업과 I/O 작업의 연속!

- CPU burst vs I/O burst
 - CPU burst: CPU연산을 많이 필요로 하므로, CPU할당 시간을 많이 줘야함
 - I/O burst: 입출력작업이 많기 때문에 CPU 를 길게 줄 필요가 없음
- **프로세스 수행 = CPU 사용 + I/O 대기**



(a) CPU 집중 프로세스의 실행 특성



(b) I/O 집중 프로세스의 실행 특성

▶ CPU 스케줄링

- 실행 준비 상태의 스레드 중 하나를 선택하는 과정 → 누가누가 CPU를 많이 쓰나?
- CPU를 사용할 순서를 정함

▶ 기본 목표: CPU를 놀게 하지 말자!

- CPU 활용률 극대화 → 컴퓨터 시스템 처리율 향상

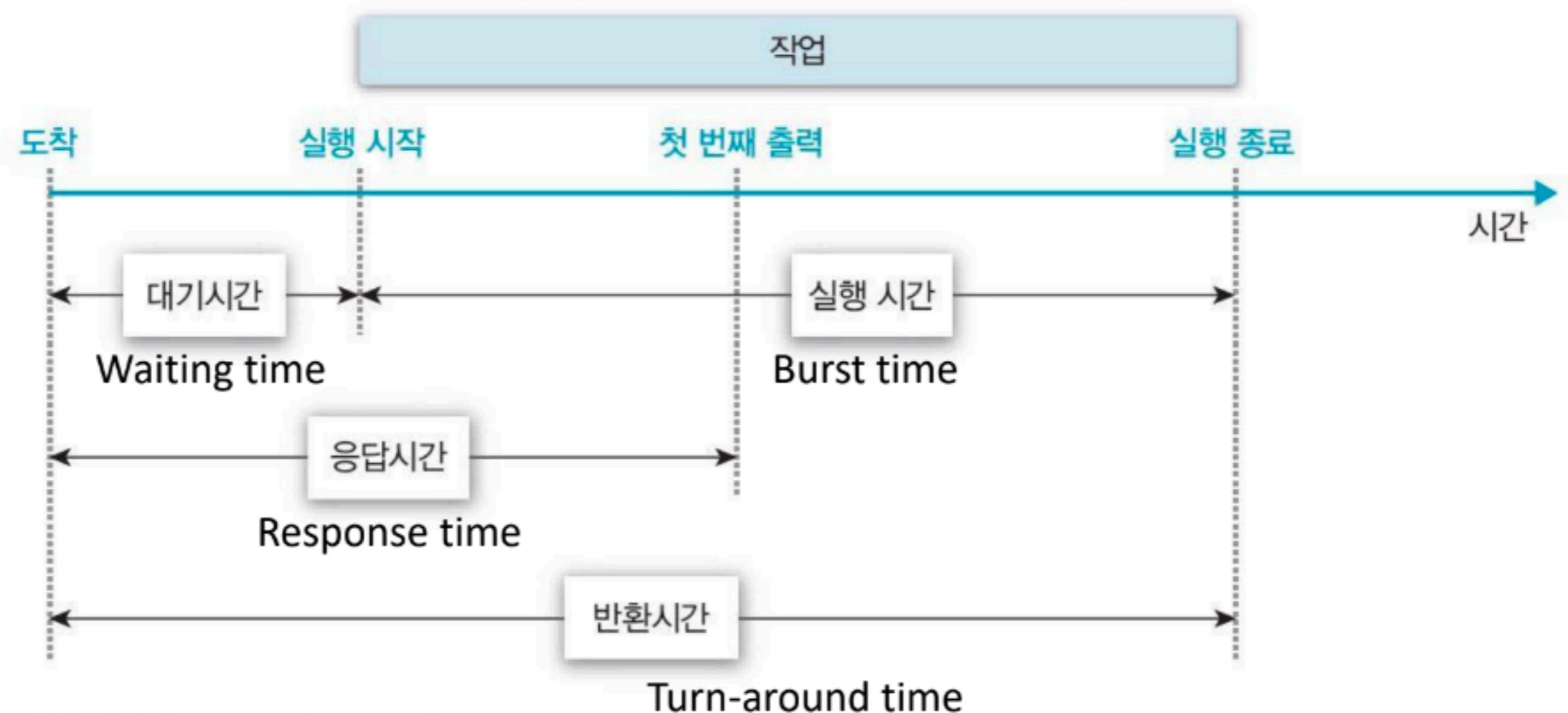
요런 것들을 타임 슬라이싱으로 달성합니다!

- ▶ 대부분 운영체제에서 하나의 스레드가 너무 오래 CPU를 사용하도록 허용하지 않음!
- ▶ **타임 슬라이스(time slice)**
 - 스케줄된 스레드에게 한 번 할당하는 CPU 시간
 - 커널이 스케줄을 단행하는 주기 시간
 - 타이머 인터럽트의 도움을 받아 타임 슬라이스 단위로 CPU 스케줄링
 - 현재 실행중인 스레드 강제 중단(preemption), 준비 리스트에 삽입
 - 타임 쿼텀(time quantum), 타임 슬롯(time slot)이라고도 함
- ▶ 그럼 시간을 얼마나 할당할 것인가?

이와 같은 기준(Criteria)을 고려하여 정해집니다!

- ▶ 컴퓨터 시스템들은 기본 목표 외에 서로 다른 스케줄링 목표를 가질 수 있음!
- ▶ CPU 활용률(CPU utilization) / Efficiency
 - 전체 시간 중 CPU의 사용 시간 비율, 운영체제 입장
- ▶ 처리율(throughput)
 - 단위시간당 처리하는 프로세스의 개수, 운영체제 입장
- ▶ 공평성(fairness) / Load balancing
 - CPU를 스레드들에게 공평하게 배분, 사용자 입장
 - 무한정 대기하는 기아 스레드(starving thread)가 생기지 않도록 스케줄
- ▶ 응답시간(response time)
 - 대화식 사용자의 경우, 명령에 응답하는데 걸리는 시간, 사용자 입장
- ▶ 대기시간(waiting time)
 - 스레드가 준비 큐에서 머무르는 시간, 운영체제와 사용자 입장
- ▶ 소요 시간(turnaround time)
 - 프로세스 (스레드) 가 컴퓨터 시스템에 도착한 후 완료될 때까지 걸린 시간, 사용자 입장
 - 배치 처리 시스템에서 주된 스케줄링의 기준
- ▶ 시스템 정책(policy enforcement) 우선
 - 컴퓨터 시스템의 특별한 목적을 달성하기 위한 스케줄링, 운영체제 입장
 - 예) 실시간 시스템에서는 스레드가 완료 시한(deadline) 내에 이루어지도록 하는 정책
 - 예) 급여 시스템에서는 안전을 관리하는 스레드 우선 정책 등
- ▶ 자원 활용률(resource efficiency), 우선순위(Priority), ...

대기시간, 실행시간, 응답시간, 반환시간



CPU 스케줄링의 기본

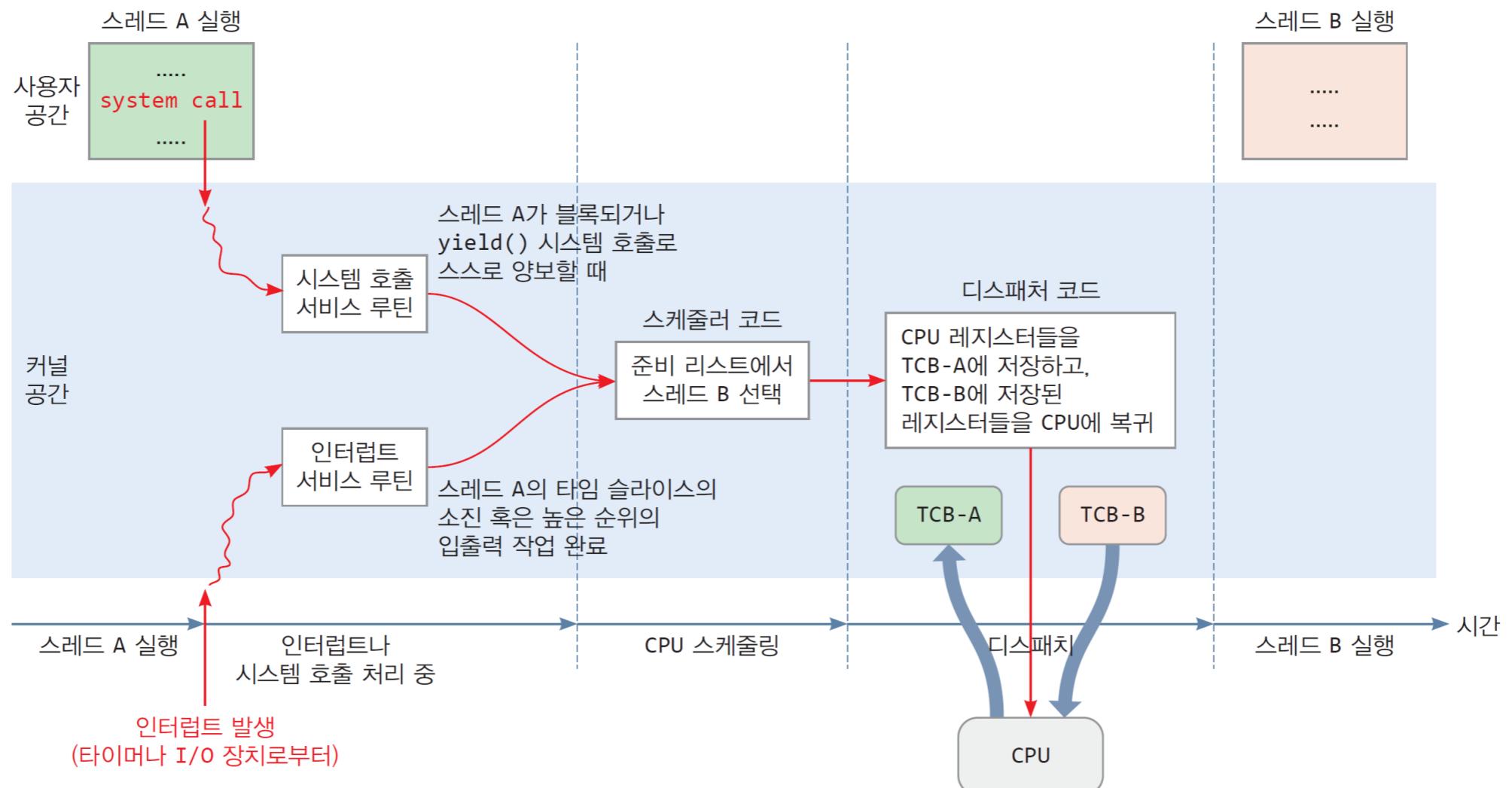
CPU 스케줄링이 실행되는 4가지 상황

- ▶ 1. 스레드가 시스템 호출 끝에 I/O를 요청하여 블록될 때
 - 스레드를 블록 상태로 만들고 스케줄링
 - (CPU의 활용률 향상 목적)
- ▶ 2. 스레드가 자발적으로 CPU를 반환할 때
 - yield() 시스템 호출 등을 통해 스레드가 자발적으로 CPU 반환
 - 커널은 현재 스레드를 준비 리스트에 넣고, 새로운 스레드 선택
 - (CPU의 자발적 양보)
- ▶ 3. 스레드의 타임 슬라이스가 소진되어 타이머 인터럽트 발생
 - (균등한 CPU 분배 목적)
- ▶ 4. 더 높은 순위의 스레드가 요청한 입출력 작업 완료, 인터럽트 발생
 - 현재 스레드를 강제 중단시켜 준비 리스트에 넣고
 - 높은 순위의 스레드를 깨워 스케줄링
 - (우선순위를 지키기 위한 목적)

Scheduler and Dispatcher

- ▶ 스케줄링 관련 코드는 커널 내 코드 형태로 있음! → **그만큼 중요하시는 거지! (진짜중요함)**
 - 별도로 실행되는 커널 프로세스나 스레드의 형태가 아님
 - 시스템 호출이나 인터럽트 서비스 루틴이 끝나는 마지막 단계에서 실행
- ▶ **스케줄러**
 - 스케줄러 타이머가 주기적으로 인터럽트를 발생시킴
 - 인터럽트 코드는 현재 실행 중인 프로세스의 실행 시간이 업데이트 시킴.
 - 스케줄링해야 하는 경우, 스케줄링 플래그 비트가 스케줄링 타이머 인터럽트에 설정됨
 - 인터럽트 종료(리턴)
 - 커널은 인터럽트의 리턴에 스케줄링 플래그 비트가 설정되었는지 여부를 판별, 되어있으면 디스패쳐 작동
- ▶ **디스패쳐**
 - 컨텍스트 스위칭을 실행하는 커널 코드
 - 스케줄러에 의해 선택된 스레드를 CPU가 실행하도록 하는 작업
 - 커널 모드에서 사용자 모드로 전환
 - 새로 선택된 스레드가 이전에 중단된 곳에서 실행하도록 점프
- ▶ **스케줄러와 디스패쳐 모두 실행 시간이 짧도록 작성**

Scheduling process



Linux code

- ▶ <https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c>

```
/ kernel / sched / core.c

1 // SPDX-License-Identifier: GPL-2.0-only
2 /*
3  * kernel/sched/core.c
4  *
5  * Core kernel scheduler code and related syscalls
6  *
7  * Copyright (C) 1991-2002 Linus Torvalds
8  */
9 #define CREATE_TRACE_POINTS
10 #include <trace/events/sched.h>
11 #undef CREATE_TRACE_POINTS
12
13 #include "sched.h"
14
15 #include <linux/nospec.h>
16 #include <linux/blkdev.h>
17 #include <linux/kcov.h>
18 #include <linux/scs.h>
19
20 #include <asm/switch_to.h>
21 #include <asm/tlb.h>
22
23 #include "../workqueue_internal.h"
24 #include "../../../fs/io-wq.h"
25 #include "../smptime.h"
26
27 #include "pelt.h"
28 #include "smp.h"
29
30 /*
31  * Export tracepoints that act as a bare tracehook (ie: have no trace event
32  * associated with them) to allow external modules to probe them.
33  */
34 EXPORT_TRACEPOINT_SYMBOL_GPL(pelt_cfs_tp);
35 EXPORT_TRACEPOINT_SYMBOL_GPL(pelt_rt_tp);
36 EXPORT_TRACEPOINT_SYMBOL_GPL(pelt_dl_tp);
37 EXPORT_TRACEPOINT_SYMBOL_GPL(pelt_irq_tp);
38 EXPORT_TRACEPOINT_SYMBOL_GPL(pelt_se_tp);
39 EXPORT_TRACEPOINT_SYMBOL_GPL(sched_cpu_capacity_tp);
40 EXPORT_TRACEPOINT_SYMBOL_GPL(sched_overutilized_tp);
41 EXPORT_TRACEPOINT_SYMBOL_GPL(sched_util_est_cfs_tp);
42 EXPORT_TRACEPOINT_SYMBOL_GPL(sched_util_est_se_tp);
43 EXPORT_TRACEPOINT_SYMBOL_GPL(sched_update_nr_running_tp);
44
45 DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
```

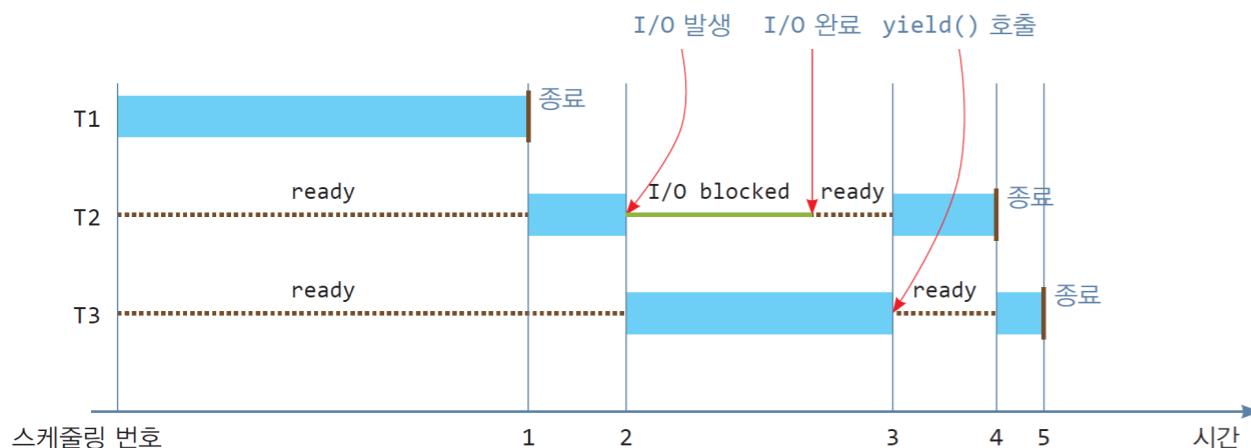
```
6142 /*
6143  * __schedule() is the main scheduler function.
6144  *
6145  * The main means of driving the scheduler and thus entering this function are:
6146  *
6147  * 1. Explicit blocking: mutex, semaphore, waitqueue, etc.
6148  *
6149  * 2. TIF_NEED_RESCHED flag is checked on interrupt and userspace return
6150  *    paths. For example, see arch/x86/entry_64.S.
6151  *
6152  * To drive preemption between tasks, the scheduler sets the flag in timer
6153  * interrupt handler scheduler_tick().
6154  *
6155  * 3. Wakeups don't really cause entry into schedule(). They add a
6156  *    task to the run-queue and that's it.
6157  *
6158  * Now, if the new task added to the run-queue preempts the current
6159  * task, then the wakeup sets TIF_NEED_RESCHED and schedule() gets
6160  * called on the nearest possible occasion:
6161  *
6162  * - If the kernel is preemptible (CONFIG_PREEMPTION=y):
6163  *
6164  *   - in syscall or exception context, at the next outmost
6165  *     preempt_enable(). (this might be as soon as the wake_up()'s
6166  *     spin_unlock()!)
6167  *
6168  *   - in IRQ context, return from interrupt-handler to
6169  *     preemptible context
6170  *
6171  * - If the kernel is not preemptible (CONFIG_PREEMPTION is not set)
6172  * then at the next:
6173  *
6174  *   - cond_resched() call
6175  *   - explicit schedule() call
6176  *   - return from syscall or exception to user-space
6177  *   - return from interrupt-handler to user-space
6178  *
6179  * * WARNING: must be called with preemption disabled!
6180  */
6181 static void __sched notrace __schedule(unsigned int sched_mode)
6182 {
6183     struct task_struct *prev, *next;
6184     unsigned long *switch_count;
6185     unsigned long prev_state;
6186     struct rq_flags rf;
6187     struct rq *rq;
6188     int cpu;
6189
6190     cpu = smp_processor_id();
6191     rq = cpu_rq(cpu);
6192     prev = rq->curr;
6193
6194     schedule_debug(prev, !!sched_mode);
6195
6196     if (sched_feat(HRTICK) || sched_feat(HRTICK_DL))
6197         hr tick_clear(rq);
6198
6199     local_irq_disable();
6200     rCU_note_context_switch(!!sched_mode);
6201
6202 }
```

선점과 비선점

- ▶ 실행중인 스레드의 강제 중단 여부에 따른 CPU 스케줄링 타입
- ▶ **비선점 스케줄링(non-preemptive scheduling)**
 - 현재 실행중인 스레드를 강제로 중단시키지 않음
 - 일단 스레드가 CPU를 할당받아 실행을 시작하면, 완료되거나 CPU를 더 이상 사용할 수 없는 상황이 될 때까지 스레드 강제 중단시키지 않고 스케줄링도 하지 않는 방식
 - 스케줄링 시점
 - CPU를 더 이상 사용할 수 없게 된 경우 : I/O로 인한 블록 상태, sleep 등
 - 자발적으로 CPU 양보할 때
 - 실행 중 종료할 때
- ▶ **선점 스케줄링(preemptive scheduling)**
 - 현재 실행중인 스레드 강제 중단시키고 다른 스레드 선택, CPU 할당
 - 스케줄링 시점
 - 타임슬라이스가 소진되어 타이머 인터럽트가 발생될 때
 - 인터럽트나 시스템 호출 종료 시점에서, 더 높은 순위의 스레드가 준비 상태일 때
- ▶ 오늘날 범용 운영체제: 선점 스케줄링 타입 사용

선점과 비선점 (2)

비선점



선점

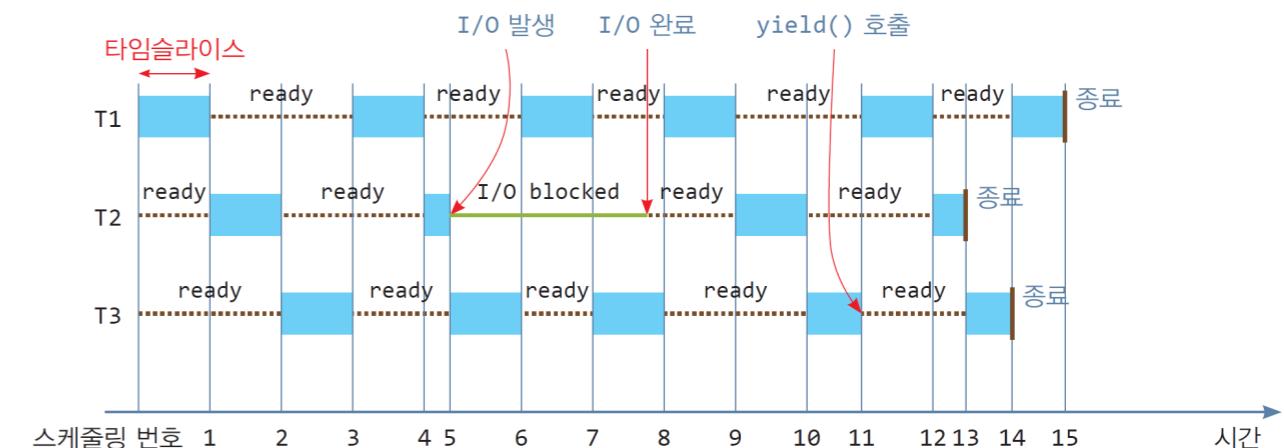


표 4-1 선점형 스케줄링과 비선점형 스케줄링의 비교

구분	선점형	비선점형
작업 방식	실행 상태에 있는 작업을 중단시키고 새로운 작업을 실행할 수 있다.	실행 상태에 있는 작업이 완료될 때까지 다른 작업이 불가능하다.
장점	프로세스가 CPU를 독점할 수 없어 대화형이나 시분할 시스템에 적합하다.	CPU 스케줄러의 작업량이 적고 문맥 교환의 오버헤드가 적다.
단점	문맥 교환의 오버헤드가 많다.	기다리는 프로세스가 많아 처리율이 떨어진다.
사용	시분할 방식 스케줄러에 사용된다.	일괄 작업 방식 스케줄러에 사용된다.
중요도	높다.	낮다.

우선순위

- ▶ 우선순위가 높은 프로세스가 CPU를 먼저, 더 오래 차지!
 - 시스템에 따라 높은 숫자가 높은 우선순위를 나타내기도 하고, 낮은 숫자가 높은 우선순위를 나타내기도 함 (e.g., 리눅스: 낮을 수록 우선순위 높음)
- ▶ 우선순위 경우들...
 - 일반적으로 Kernel process > User-level process
 - 일반적으로 I/O burst process > CPU burst process
 - 일반적으로 Foreground process > background process

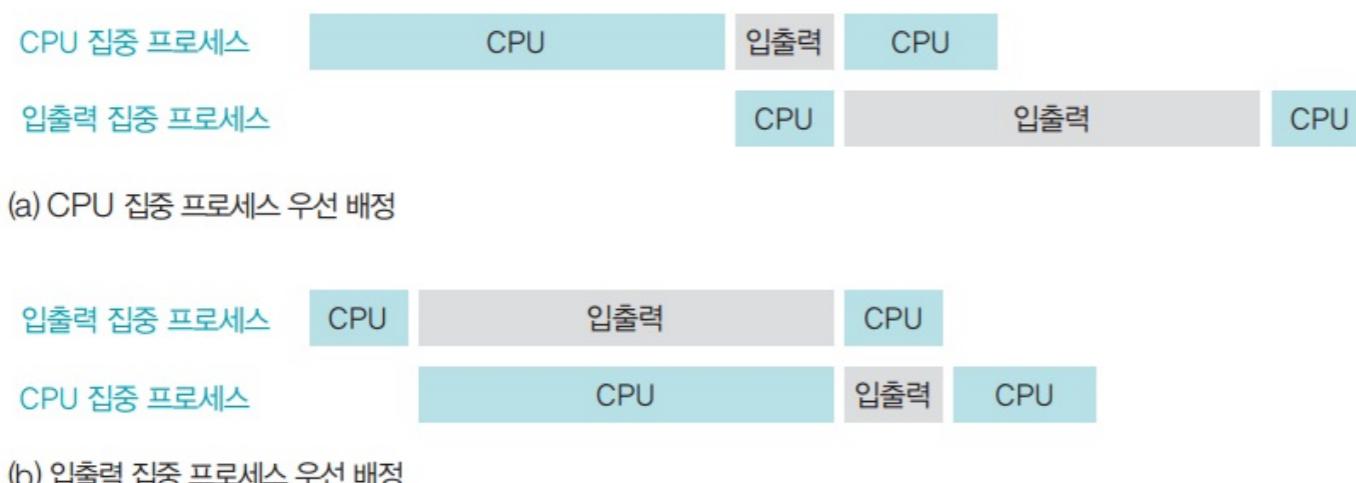


그림 4-6 CPU 집중 프로세스와 입출력 집중 프로세스의 우선 배정 결과

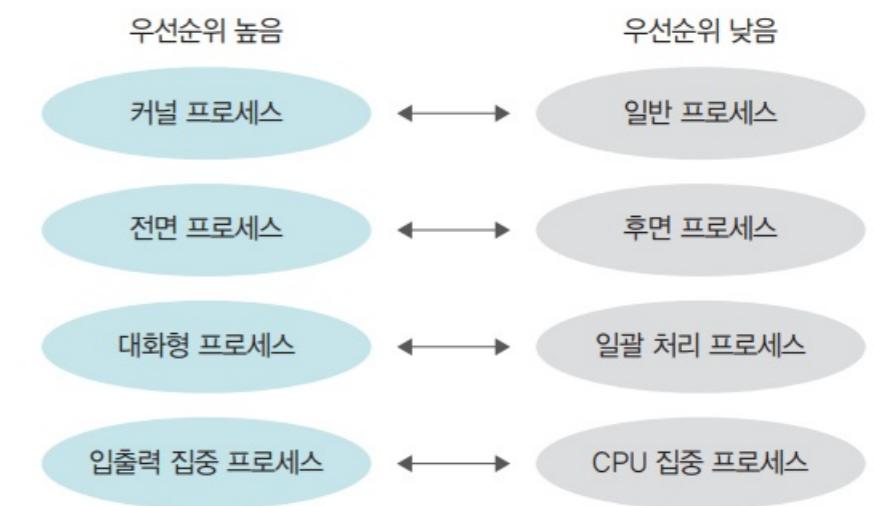


그림 4-8 CPU 스케줄링 시 고려 사항

프로세스의 우선순위를 배정하는 방식

▶ 정적 스케줄링(Static Scheduling)

- 운영체제가 프로세스에 우선순위를 부여하면 프로세스가 끝날 때까지 바뀌지 않는 방식
- 프로세스가 작업하는 동안 우선순위가 변하지 않기 때문에 구현하기 쉽지만, 시스템의 상황이 시시각각 변하는데 우선순위를 고정하면 시스템의 변화에 대응하기 어려워 작업 효율이 떨어짐

▶ 동적 스케줄링(Dynamic Scheduling)

- 프로세스 생성 시 부여받은 우선순위가 프로세스 작업 중간에 변하는 방식
- 구현하기 어렵지만 시스템의 효율성을 높일 수 있음

Starvation and aging

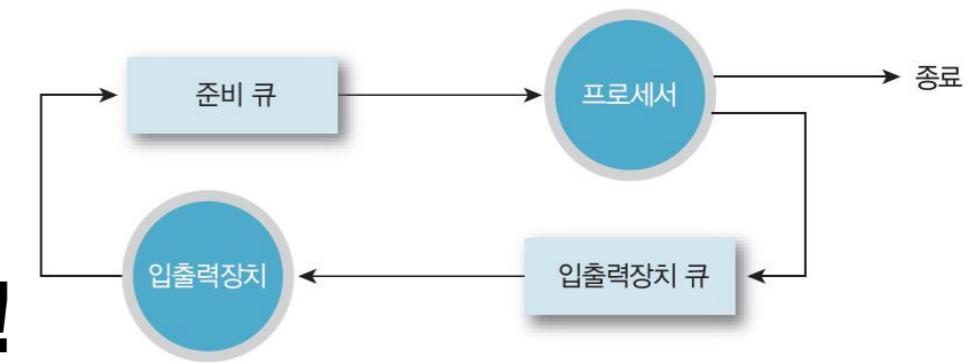
▶ 기아(starvation)

- 스레드가 스케줄링에서 선택되지 못한 채 오랜 동안 준비 리스트에 있는 상황
- 사례
 - 우선순위를 기반으로 하는 시스템에서, 더 높은 순위의 스레드가 계속 시스템에 들어오는 경우
 - 짧은 스레드를 우선 실행시키는 시스템에서, 자신보다 짧은 스레드가 계속 도착하는 경우
- 스케줄링 알고리즘 설계 시 기아 발생을 면밀히 평가
 - 기아가 발생하지 않도록 설계하는 것이 바람직함

▶ 에이징(aging)

- 기아의 해결책
- 스레드가 준비 리스트에 머무르는 시간에 비례하여 스케줄링 순위를 높이는 기법
- 오래 기다릴 수는 있지만 언젠가는 가장 높은 순위에 도달하는 것 보장

스케줄링은 큐(Queue)에 관한 이야기!



- ▶ 프로세스는 준비 상태에 들어올 때마다 자신의 우선순위에 해당하는 큐의 마지막에 삽입
 - CPU 스케줄러는 우선순위가 가장 높은 큐의 맨 앞에 있는 프로세스에 CPU 할당
- ▶ 대기상태에서는 시스템의 효율을 높이기 위해 같은 입출력끼리 모아둠

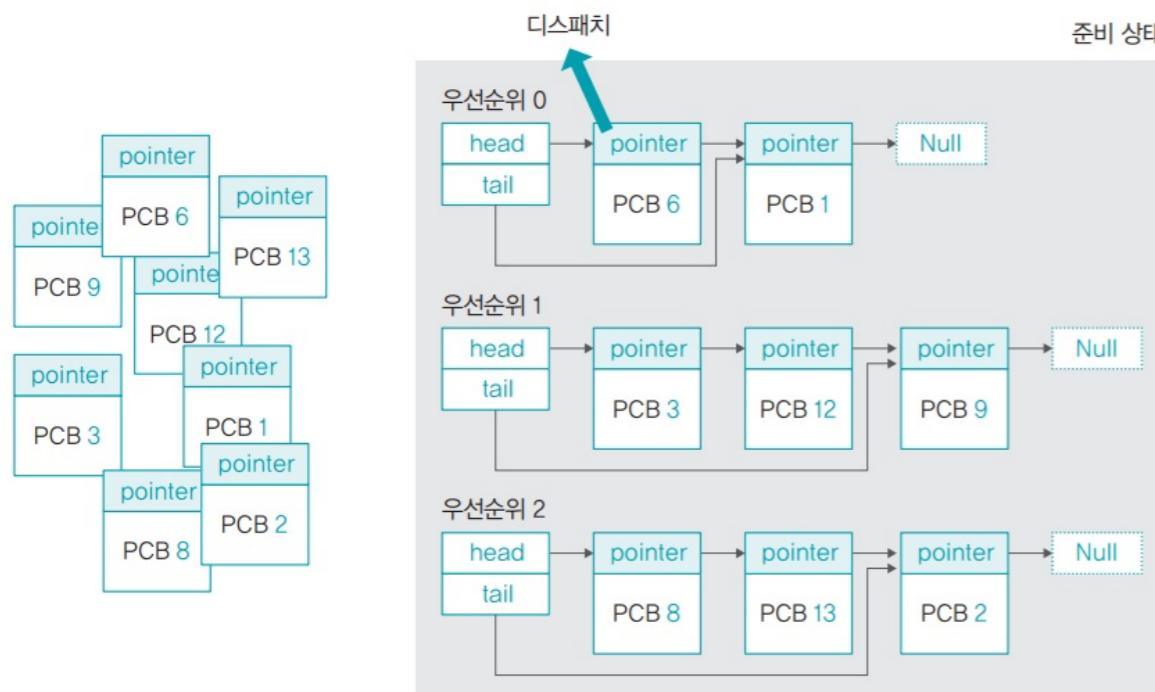


그림 4-9 준비 상태의 다중 큐

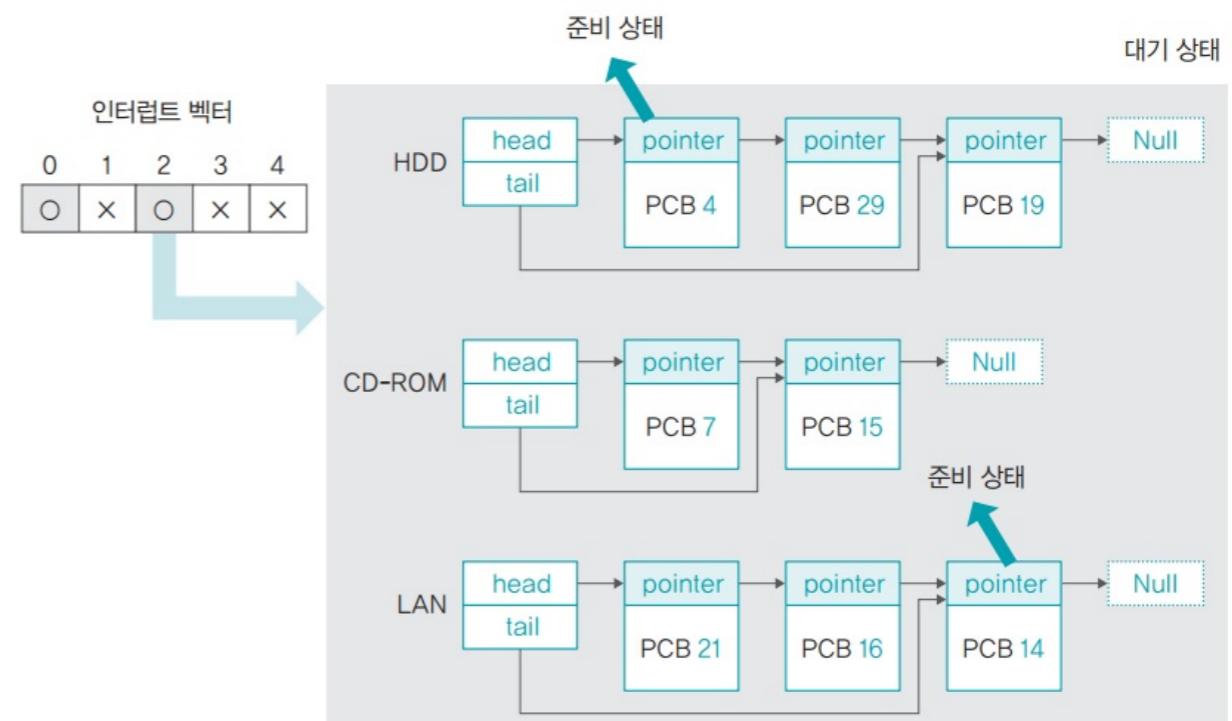


그림 4-10 대기 상태의 다중 큐

스케줄링 알고리즘

큐를 어떻게 구성할 것인가?!

평가기준

- ▶ 대기 시간(**Waiting Time**): 프로세스가 준비 큐 내에서 대기하는 시간의 총합.
- ▶ 반환 시간(**Turn-around Time**): 프로세스가 시작해서 끝날 때까지 걸리는 시간.
- ▶ 응답 시간(**Response Time**): '첫번째' 응답이 오기 시작할 때까지의 시간.
- ▶ CPU 사용률(**CPU Utilization**): 전체 시스템 시간 중 CPU가 작업을 처리하는 시간의 비율
- ▶ 처리량(**Throughput**): CPU가 단위 시간당 처리하는 프로세스의 개수.

그리고 평균대기시간/평균반환시간

- 모든 프로세스의 대기/반환 시간을 합한 뒤 프로세스의 수로 나눈 값
 - 이것들이 빨라야 일반적으로 좋은 알고리즘

도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9

평균 대기시간: $(0+27+42)/3=23$
평균 반환시간: $(30+48+57)/3=45$

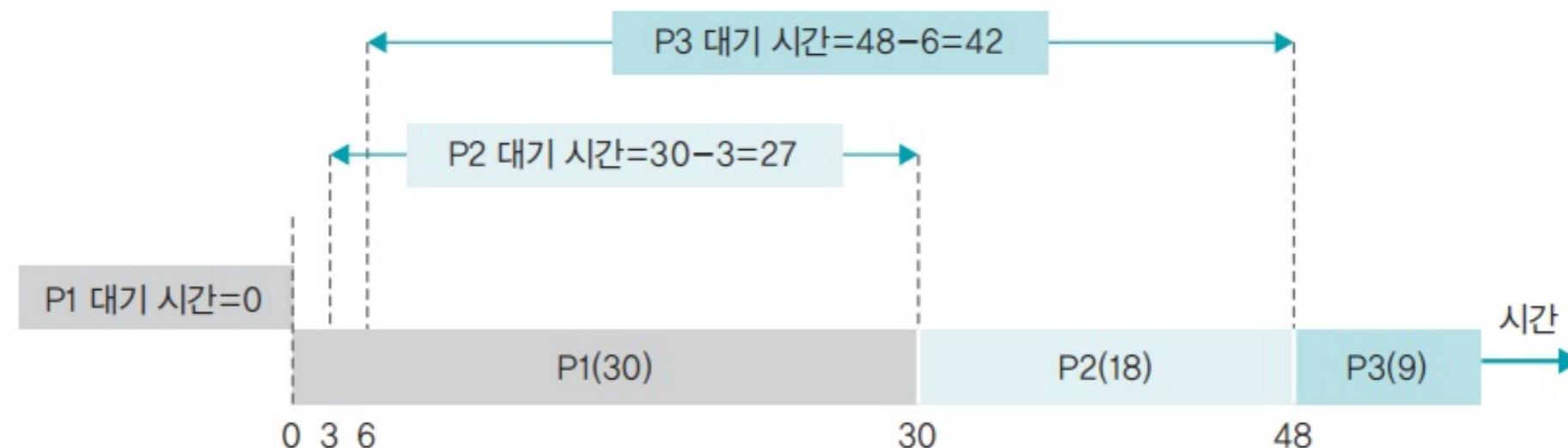


표 4-2 스케줄링 알고리즘의 종류

구분	종류
비선점형 알고리즘	FCFS 스케줄링, SJF 스케줄링, HRN 스케줄링
선점형 알고리즘	라운드 로빈 스케줄링, SRT 스케줄링, 다단계 큐 스케줄링, 다단계 피드백 큐 스케줄링
둘 다 가능	우선순위 스케줄링

그리고 알고리즘들...

- ▶ FCFS(First Come First Served) ([비선점 스케줄링](#))
 - 도착한 순서대로 스레드를 준비 큐에는 넣고 도착한 순서대로 처리
- ▶ SJF(Shortest Job First) ([비선점 스케줄링](#))
 - 가장 짧은 스레드 우선 처리
- ▶ Shortest remaining time first ([선점 스케줄링](#))
 - 남은 시간이 짧은 스레드가 준비 큐에 들어오면 이를 우선 처리
- ▶ Round-robin ([preemptive](#))
 - 스레드들을 돌아가면서 할당된 시간(타임 슬라이스)만큼 실행
- ▶ Priority Scheduling ([선점/비선점 스케줄링 둘다 구현 가능](#))
 - 우선 순위를 기반으로 하는 스케줄링. 가장 높은 순위의 스레드 먼저 실행
- ▶ Multilevel queue scheduling ([선점/비선점 스케줄링 둘다 구현 가능](#))
 - 스레드와 큐 모두 n개의 우선순위 레벨로 할당, 스레드는 자신의 레벨과 동일한 큐에 삽입
 - 높은 순위의 큐에서 스레드 스케줄링, 높은 순위의 큐가 빌 때 아래 순위의 큐에서 스케줄링
 - 스레드는 다른 큐로 이동하지 못함
 - 예) background process, Foreground process
- ▶ Multilevel feedback queue scheduling ([선점/비선점 스케줄링 둘다 구현 가능](#))
 - 큐만 n개의 우선순위 레벨을 둠. 스레드는 레벨이 없이 동일한 우선순위
 - 스레드는 제일 높은 순위의 큐에 진입하고 큐타임슬라이스가 다하면 아래 레벨의 큐로 이동
 - 낮은 레벨의 큐에 오래 있으면 높은 레벨의 큐로 이동

FCFS (First Come First Served); 선입선출

- ▶ 가장 단순한 형태. 그냥 큐 그 자체. Queue
- ▶ 먼저 도착한(큐의 맨 앞에 있는) 스레드 먼저 스케줄링

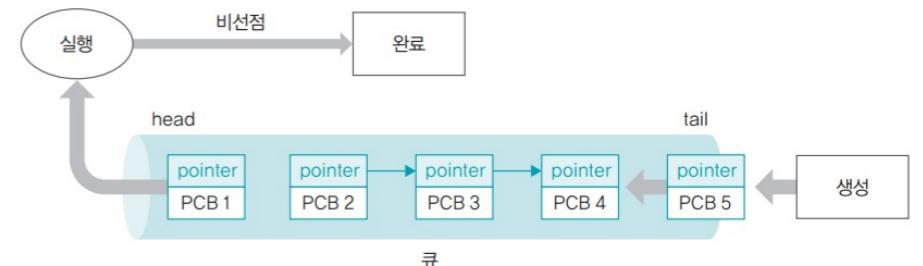


그림 4-14 FCFS 스케줄링의 동작

스레드	도착 시간	실행 시간(ms)
T1	0	4
T2	1	3
T3	2	1
T4	5	3

FCFS Summary

- ▶ Summary
 - 스케줄링 파라미터 : 스레드 별 도착 시간
 - 스케줄링 타입 : 비선점 스케줄링
 - 스레드 우선 순위 : 없음
 - 기아 : 발생하지 않음
 - 스레드가 오류로 인해 무한 루프를 실행한다면 뒤 스레드의 기아 발생
- ▶ 성능 이슈
 - 처리율 낮음
 - 호위 효과(convoy effect) 발생
 - 긴 스레드가 CPU를 오래 사용하면, 늦게 도착하면 짧은 스레드는 오래 대기

SJF (Shortest Job First); 최단 작업 우선 스케줄링

- ▶ 예상 실행 시간이 가장 짧은 스레드 선택
- ▶ 스레드가 도착할 때, 예상 실행 시간이 짧은 순으로 큐 삽입, 큐의 맨 앞에 있는 스레드 선택

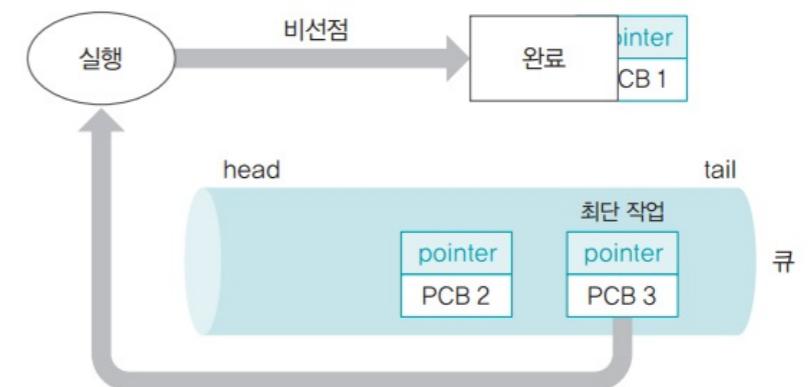


그림 4-17 SJF 스케줄링의 동작

Case 1

스레드	도착 시간	실행 시간(ms)
T1	0	4
T2	1	3
T3	2	1
T4	5	3

Case 2

프로세스	도착시간	CPU burst 시간
P1	0	14
P2	4	8
P3	8	2
P4	10	8

SJF summary

▶ Summary

- 스케줄링 파라미터 : 스레드 별 예상 실행 시간 (남은실행시간)
 - 하지만 스레드별 실행 시간을 예측하는 것이 가능할까?
- 스케줄링 타입 : 비선점 스케줄링
 - **but 선점도 가능!!** 이런 경우 **SRTF (Shortest Remaining Time First)**라고 부름.
- 스레드 우선 순위 : 없음
- **기아 : 발생 가능**
 - 지속적으로 짧은 스레드가 도착하면, 긴 스레드는 언제 실행 기회를 얻을지 예측할 수 없음

▶ 성능 이슈

- 짧은 스레드가 먼저 실행되므로 평균 대기 시간 최소화

▶ 문제점

- 실행 시간의 예측이 불가능하므로 현실에서는 거의 사용되지 않음

라운드 로빈 (Round-robin, RR)

- ▶ 스레드들에게 공평한 실행 기회를 주기 위해 큐에 대기중인 스레드들을 타임 슬라이스 주기로 돌아가면서 선택
 - 도착하는 순서대로 스레드들을 큐에 삽입
 - 타임 슬라이스가 지나면 큐 끝으로 이동

Case 1 (Time slice: 2ms, 1ms)

스레드	도착 시간	실행 시간(ms)
T1	0	4
T2	1	3
T3	2	1
T4	5	3

Case 2 (Time slice: 10ms)

도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9

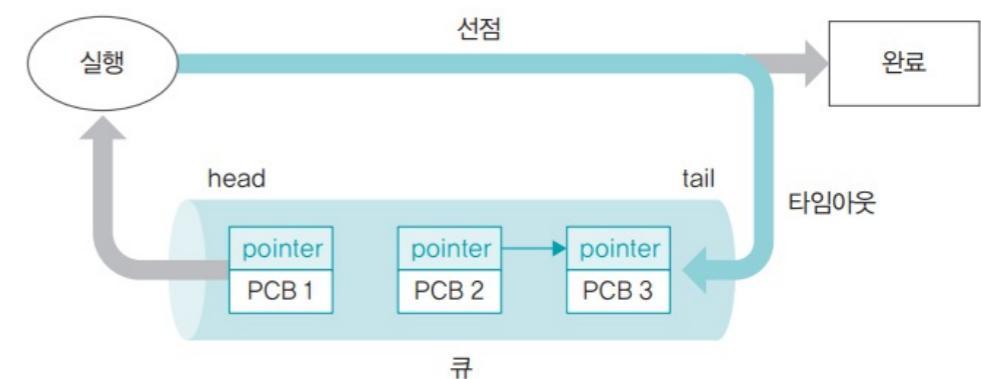


그림 4-20 라운드 로빈 스케줄링의 동작

RR Summary

▶ Summary

- 스케줄링 파라미터 : 타임 슬라이스
- **스케줄링 타입 : 선점 스케줄링**
- 스레드 우선 순위 : 없음
- 기아 : 없음
 - 스레드의 우선순위가 없고, 타임 슬라이스가 정해져 있어, 일정 시간 후에 스레드는 반드시 실행

▶ 성능 이슈

- 공평하고, 기아 현상 없고, 구현이 쉬움
- 짧은 스케줄링으로 전체 스케줄링 오버헤드 큼. **특히 타임 슬라이스가 작을 때 더욱 큼**
 - 처리 시간에 표현되지 않은 Context switching 시간을 고려해본다면...!!
- 균형된 처리율 : 타임슬라이스가 크면 FCFS에 가까움, 적으면 SJF/SRTF에 가까움
- 늦게 도착한 짧은 프로세스는 FCFS보다 빨리 완료되고, 긴 프로세스는 SJF보다 빨리 완료됨



우선순위 (Priority)

- ▶ 우선순위에 따라 스레드를 실행시킴. 가장 높은 순위의 스레드 선택
 - 현재 스레드가 종료되거나 더 높은 순위의 스레드가 도착할 때, 가장 높은 순위의 스레드 선택
 - 고정 우선순위 알고리즘
 - 한 번 우선순위를 부여받으면 종료될 때까지 우선순위가 고정
 - 단순하게 구현할 수 있지만 시시각각 변하는 시스템의 상황을 반영하지 못해 효율성이 떨어짐
 - 변동 우선순위 알고리즘
 - 일정 시간마다 우선순위가 변하여 일정 시간마다 우선순위를 새로 계산하고 이를 반영
 - 복잡하지만 시스템의 상황을 반영하여 효율적인 운영 가능
- ▶ 도착하는 스레드는 우선 순위 순으로 큐에 삽입

도착 순서	도착 시간	작업 시간	우선순위
P1	0	30	3
P2	3	18	2
P3	6	9	1

Priority summary

- ▶ Summary
 - 스케줄링 파라미터 : 스레드 별 고정 우선 순위
 - 스케줄링 타입 : 선점 스케줄링/비선점 스케줄링
 - 선점 스케줄링 : 더 높은 스레드가 도착할 때 현재 스레드 강제 중단하고 스케줄링
 - 비선점 스케줄링 : 현재 실행 중인 스레드가 종료될 때 비로소 스케줄링
 - 스레드 우선 순위 : 있음
 - 기아 : 발생 가능
 - 지속적으로 높은 순위의 스레드가 도착하는 경우 언제 실행 기회를 얻을 수 예상할 수 없음
 - 큐 대기 시간에 비례하여 일시적으로 우선순위를 높이는 에이징 방법으로 해결 가능
- ▶ 성능 이슈
 - 높은 우선순위의 스레드일수록 대기 혹은 응답시간 짧음
 - Real-time OS에서 유리!

HRN 스케줄링 (Highest Response ratio Next)

- ▶ 기아현상을 완화 하기 위해 만들어진 알고리즘 (기본적으로 SJF의 변형입니다!)
- ▶ 서비스를 받기 위해 기다린 시간과 CPU 사용 시간을 고려하여 스케줄링을 하는 방식
- ▶ 우선순위 = $\frac{\text{대기시간} + \text{CPU사용시간}}{\text{CPU사용시간}}$

Case 1 (각 우선순위를 구하면?)

작업	대기시간	서비스시간
A	5	20
B	40	20
C	15	45
D	20	20

Case 2

도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9

HRN Summary

▶ Summary

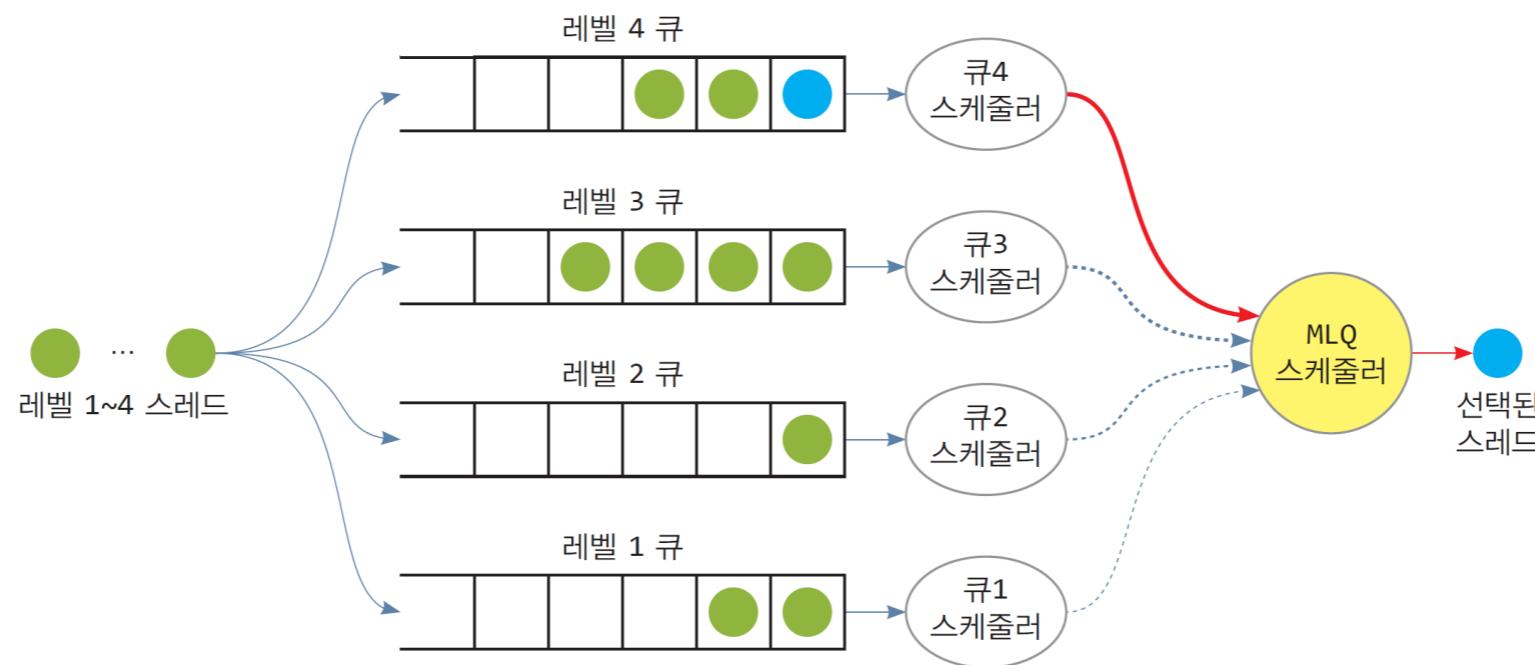
- 스케줄링 파라미터: 스레드별 작업시간 + 대기시간
- 스케줄링 타입 : 비선점 스케줄링
 - 선점이라면? 핑퐁핑퐁핑퐁~
- 스레드 우선 순위 : 있음
- 기아 : 발생 낮음
 - 알고리즘에 의해 거의 발생 하지 않음.
 - 다만 여전히 공평하다고는 말하기 어려움.

▶ 성능 이슈

- 대기시간이 긴 스레드일 수록 우선순위가 높아짐

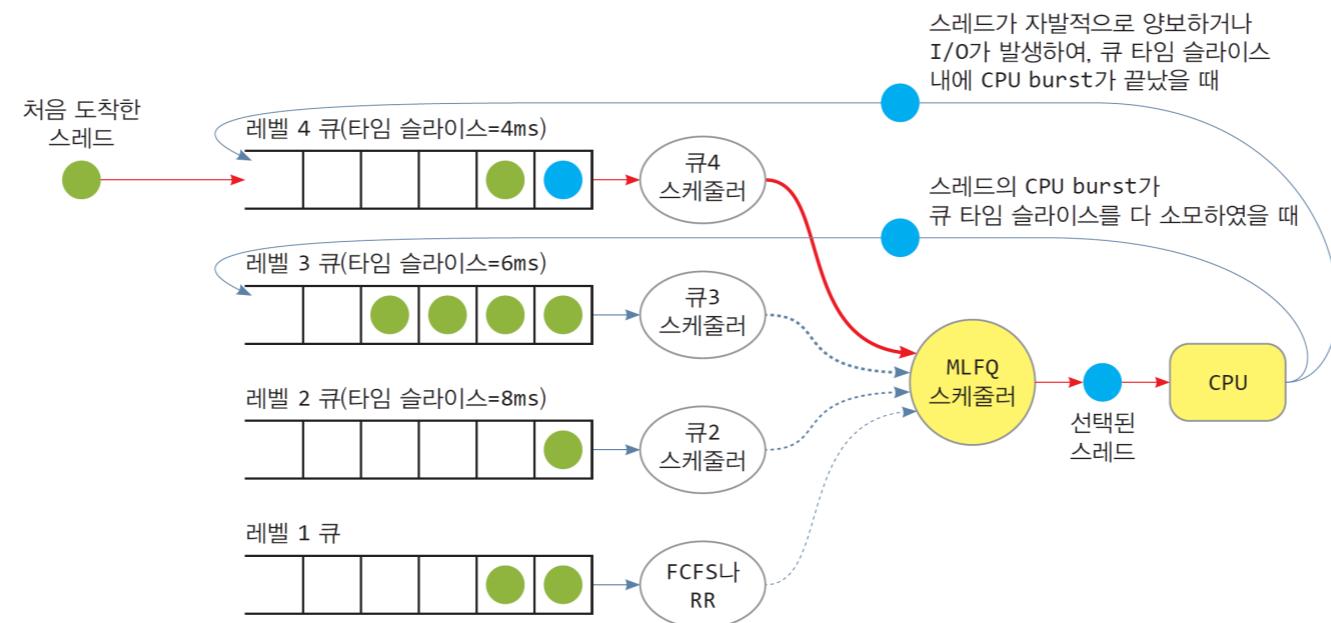
MLQ(Multi-level Queue)

- ▶ 스레드들을 n개의 우선순위 레벨로 구분, 레벨이 높은 스레드들을 우선적으로 처리
 - 고정된 n 개의 큐 사용, 각 큐에 고정 우선순위 할당
 - 스레드들의 우선순위도 n개의 레벨로 분류
 - 각 큐는 나름대로의 기법으로 스케줄링
 - 스레드는 도착 시 우선 순위에 따라 해당 레벨 큐에 삽입. 다른 큐로 이동할 수 없음
 - 가장 높은 순위의 큐가 빌 때, 그 다음 순위의 큐에서 스케줄링



MLFQ(Multi-level Feedback Queue)

- ▶ 기아를 없애기 위해 여러 레벨의 큐 사이에 스레드 이동 가능하도록 설계
 - n개의 고정 큐. 큐마다 서로 다른 스케줄링 알고리즘
 - 큐마다 스레드가 머무를 수 있는 큐 타임슬라이스 있음. 낮은 레벨의 큐일수록 더 긴 타임 슬라이스
 - 스레드는 도착 시 최상위 레벨의 큐에 삽입
 - 가장 높은 레벨의 큐에서 스레드 선택. 비어 있으며 그 아래의 큐에서 스레드 선택
 - 스레드의 CPU-burst가 큐 타임 슬라이스를 초과하면 강제로 아래 큐로 이동시킴
 - 스레드가 자발적으로 중단한 경우, 현재 큐 끝에 삽입
 - 스레드가 I/O로 실행이 중단된 경우, I/O가 끝나면 동일한 레벨 큐 끝에 삽입
 - 큐에 있는 시간이 오래되면 기아를 막기 위해 하나 위 레벨 큐로 이동



큐 타임 슬라이스 : Q3=4ms, Q2=6ms, Q1=무제한

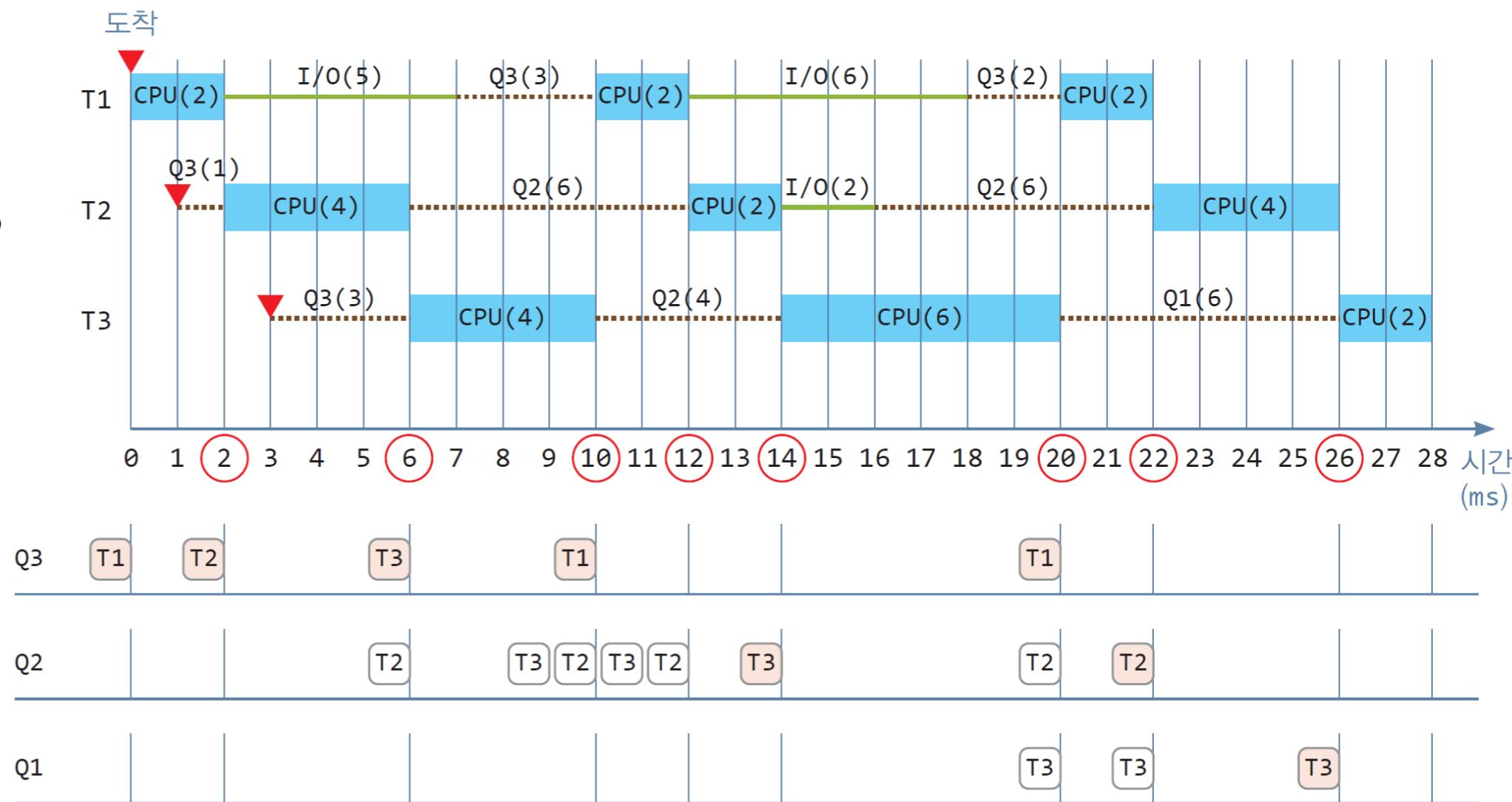
e.g.,

▶ 스케줄링 사례

- 3개의 레벨 큐
- 3개의 스레드

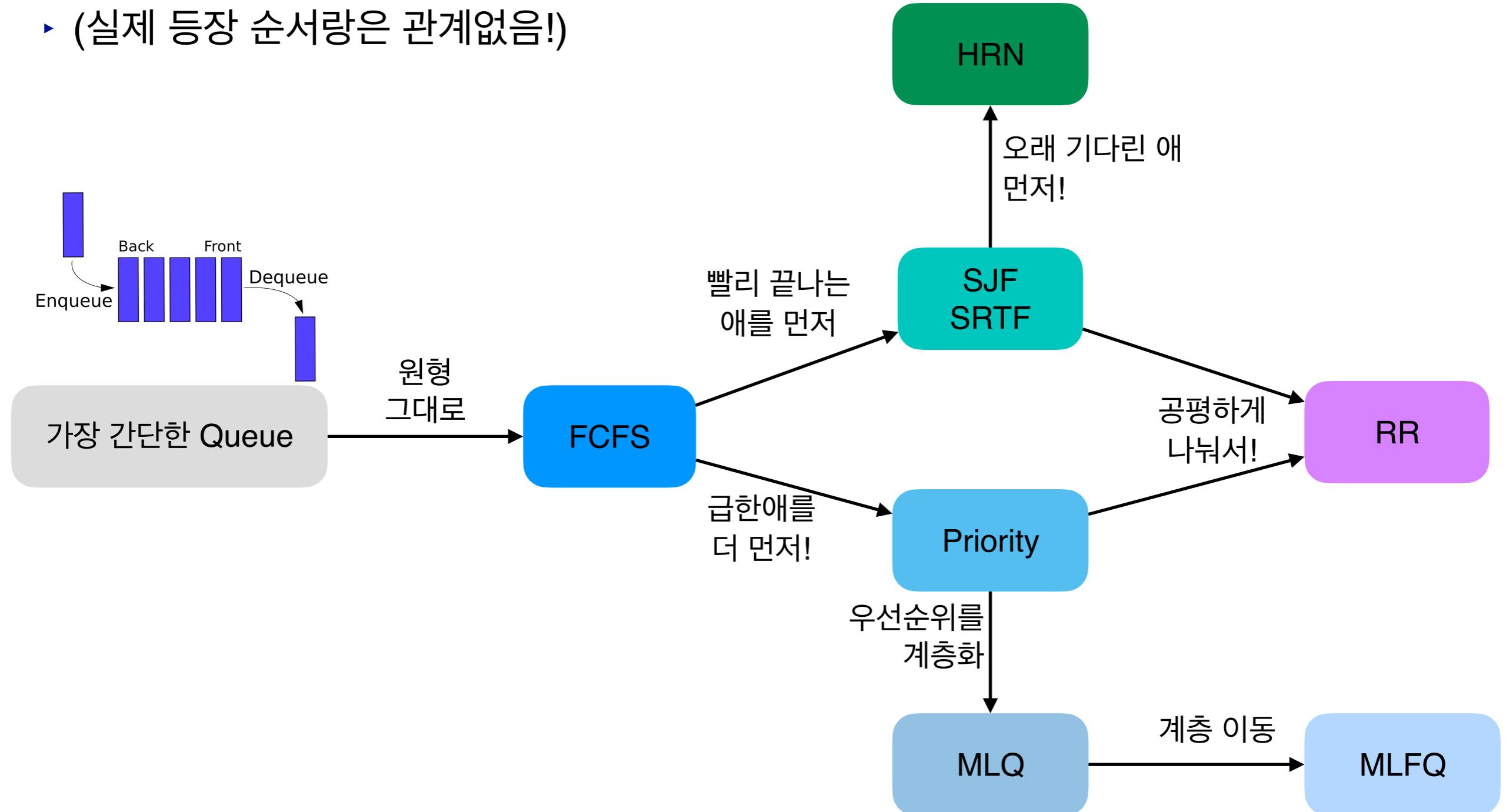
스레드	도착 시간	실행 시간(ms) (CPU-I/O-CPU-I/O-CPU)
T1	0	2 - 5 - 2 - 6 - 2
T2	1	6 - 3 - 4
T3	3	12

- ▶ 평균 대기 시간 :
 $(5 + 13 + 13)/3$
 $= 31/3 = 10.3$



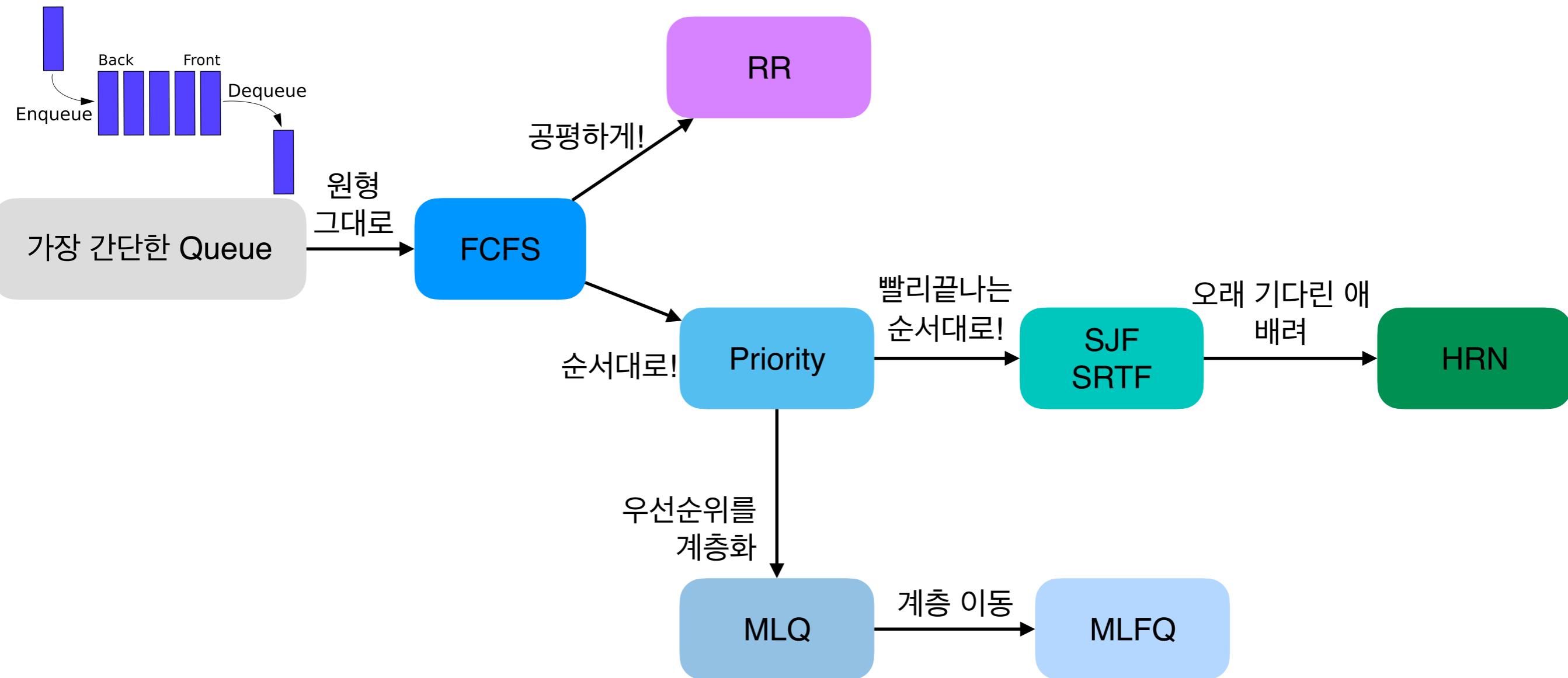
이해를 돋기 위한 도표(1)

- (실제 등장 순서랑은 관계없음!)



이해를 돋기 위한 도표(2)

- (실제 등장 순서랑은 관계없음!)



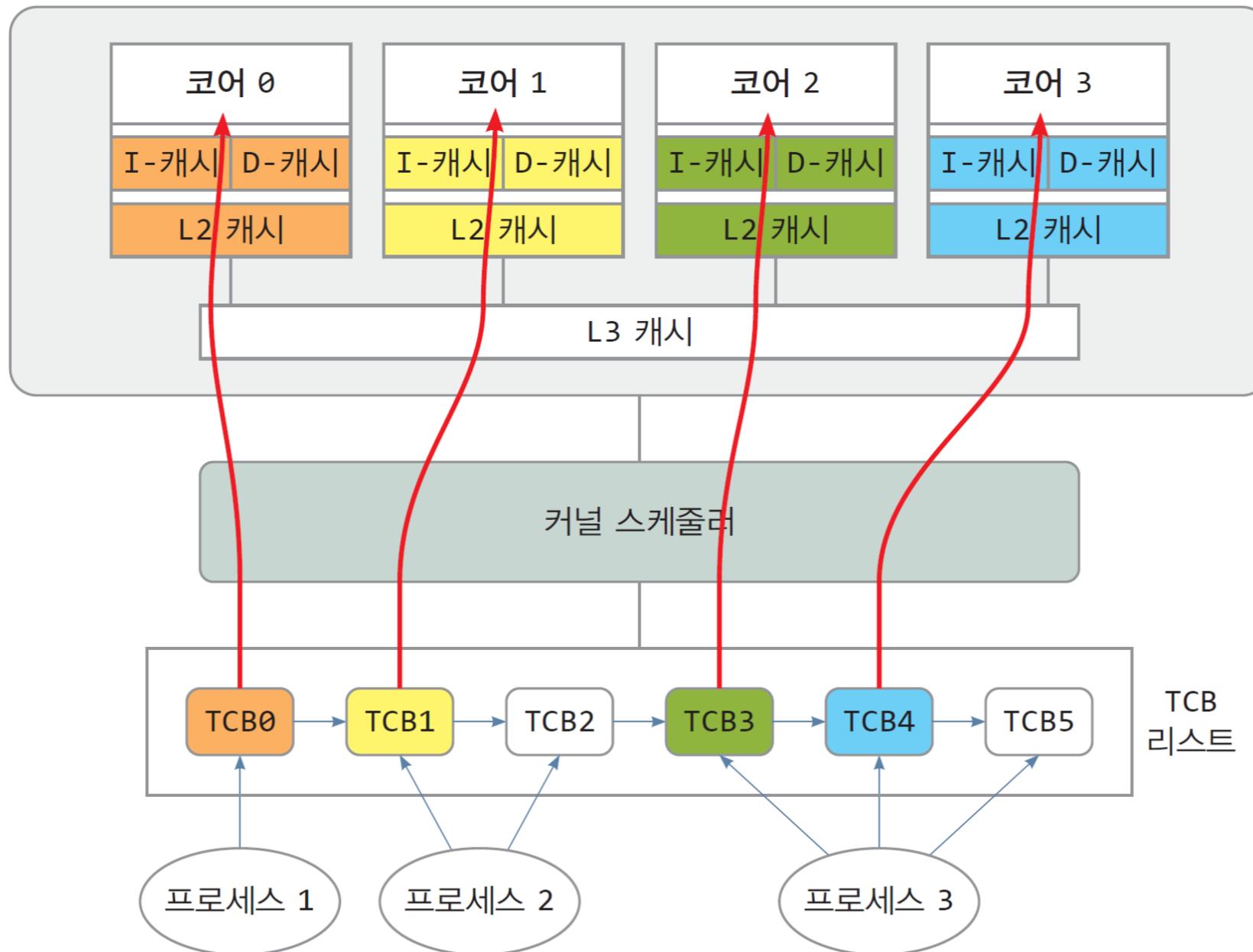
짬뽕도 있음!

- ▶ 각 스케줄링은 서로 섞여서 사용될 수 있음!
 - RR+SJF
 - MLQ + FCFS
 - MLQ + RR
 - ...
- ▶ 꼭 Queue만 사용하는 것은 아님!
 - 리눅스 CFS (Completely Fair Scheduler): Red-black tree 구조 활용!
- ▶ 우선순위마다 다른 스케줄링이 사용되기도 함!
 - 리눅스의 경우 (숫자가 낮을 수록 높은 우선순위)
 - Priority 0-99: Real-time process, static priority (RR 사용)
 - Priority 100-139: Ordinary process, dynamic priority (CFS 사용)

멀티코어 CPU 스케줄링

멀티 코어 시스템에서의 멀티스레딩

4개의 코어를 가진 Intel Core-i7 CPU



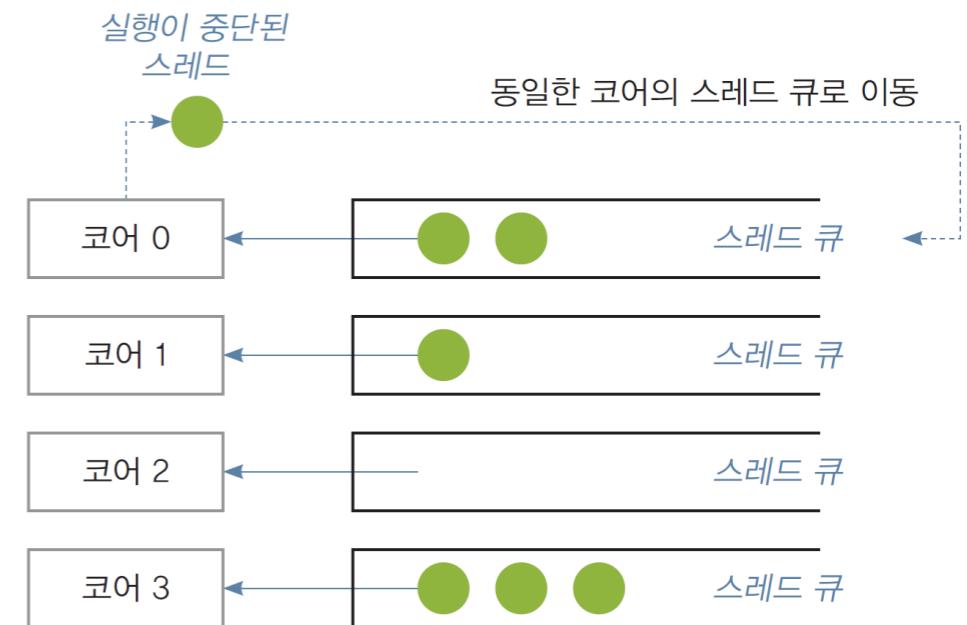
그런데... 이게 마냥 쉽지 않아요!

- ▶ 스케줄링을 통해 프로세스는 지정된 순서에 따라 CPU를 들락날락 합니다.
- ▶ 그런데, 멀티코어에서는 스케줄링이 여러 CPU에 걸쳐서 발생하게 됩니다!
 - 서로 다른 코어에 걸쳐서 **Context switching**이 발생한다면?
- ▶ **C1. 컨텍스트 스위칭 오버헤드 증가 문제**
 - 이전에 실행된 적이 없는 코어에 스레드가 배치될 때
 - e.g., Core1 → Core3
 - 캐시에 새로운 스레드의 코드와 데이터로 채워지는 긴 경과 시간
- ▶ **C2. 코어별 부하 불균형 문제**
 - 스레드를 무작위로 코어에 할당하면, 코어마다 처리할 스레드 수의 불균형 발생
 - P1 → Core1, P2 → Core1, P3 → Core1, P4 → Core1, ...
 - Core2, Core3, ... !!

해결책

▶ S1. 코어간 Context switching 문제 → CPU 친화성(CPU affinity)

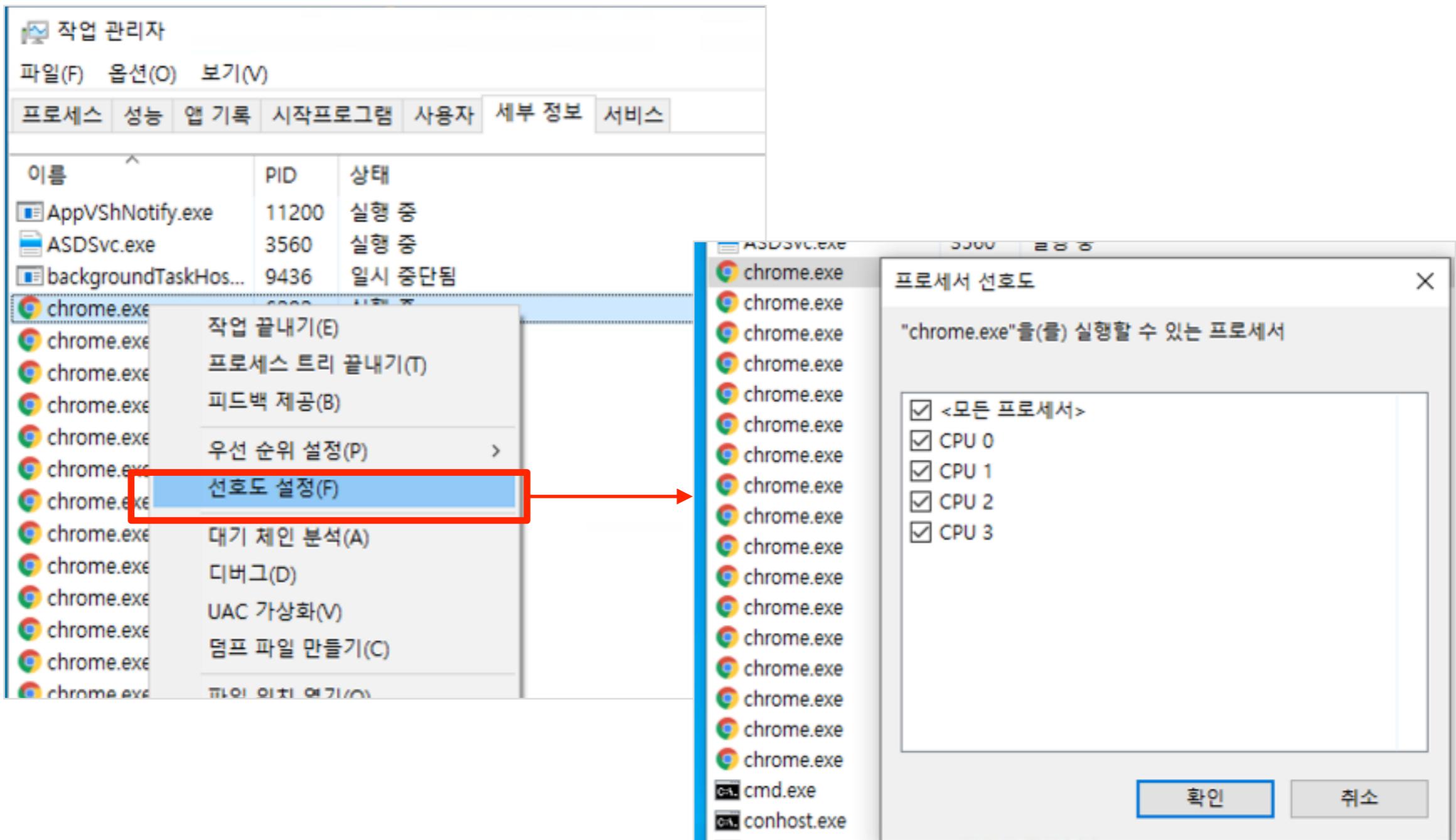
- a.k.a., 코어 친화성(Core affinity),
CPU 피닝(pinning), 캐시 친화성(cache affinity)
- 스레드를 동일한 코어에서만 실행하도록 스케줄링
- 코어 당 스레드 큐 사용



▶ S2. 코어별 부하 불균형 문제 → 부하 균등화 기법

- 코어 사용량 감시 스레드가 작업의 재할당(migration)하여 부하를 분산
- 푸시 마이그레이션(push migration)
 - 짧거나 빈 큐를 가진 코어에 다른 큐의 스레드를 옮겨놓는 기법
- 풀 마이그레이션(pull migration)
 - 코어가 처리할 스레드가 없게 되면, 다른 코어의 스레드 큐에서 자신이 큐에 가져와 실행시키는 기법

윈도우에서 볼 수 있음!



애초에 멀티 코어/스레드를 고려를 한 프로그래밍이 필요합니다!

- ▶ 아니면 특정 CPU만 혹사를... 시킬 수도..ㅠㅠ



그런데... 이거도 참 쉽지 않아요!

- ▶ 하나의 프로세스에서 일어나는 작업을 여러 코어/스레드에 걸쳐서??!
- ▶ 만약 공유 데이터에 '두' 함수가 아무 생각 없이 임의의 순서로 접근한다면....
 - 공유데이터가 훼손되는 문제 발생! (쓰레드에서 봤던 그것)
 - 공유데이터에 다수의 스레드가 접근시에는 서로 합을 맞추는 것이 필요!
- ▶ 그래서 다음시간에 배우는 것 → **동기화(Synchronization)!**

```
myThread1's tid: 2465000
myThread2's tid: 24E8000
    myThread 2 starts
    myThread 1 starts
myThreads have been finished
sum = -40164
ret1 = 200000
ret2 = 200000
```

```
myThread1's tid: 4AE0000
myThread2's tid: 4B63000
    myThread 1 starts
    myThread 2 starts
myThreads have been finished
sum = 32517
ret1 = 200000
ret2 = 200000
```

```
1 #include <pthread.h> // pthread lib
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int sum = 0; // global variable
6
7 void* myThread1(void *p) { // for the thread 1
8     printf("\t myThread 1 starts\n");
9
10    int *i = (int*)malloc(sizeof(int));
11    for(i = 0; i < (*(int*)p); i++) sum += 1;
12
13    return (void*)i;
14 }
15
16 void* myThread2(void *p) { // for the thread 2
17     printf("\t myThread 2 starts \n");
18
19    int *i = (int*)malloc(sizeof(int));
20    for(i = 0; i < (*(int*)p); i++) sum -= 1;
21
22    return (void*)i;
23 }
24
```

Summary

- ▶ 다중 프로그래밍 → 여러 프로세스가 하나의 CPU를 공유
 - 어떤 순서로 작업을 시킬지, 스케줄링 개념이 필요함
 - 이걸 잘하면 → CPU의 유휴(대기) 시간이 줄어들고 → CPU 활용률 향상
- ▶ 스케줄링의 단계: 고수준 / 중간수준 / 저수준
- ▶ 스케줄링의 요소들
 - 타임 슬라이싱! → 결국은 시분할
 - 처리량, 활용율, 공평성...
 - 대기시간, 실행시간, 응답시간, 반환시간...
 - 우선순위
- ▶ 각종 스케줄링 알고리즘들....! → 조금 많아보이지만, 잘 알아두시면 좋습니다!
 - CS분야 전반에서 진짜 정말 두루두루두루두루두루두루두루두루 응용됨
- ▶ 멀티코어환경에서의 스케줄링은 Context switching 오버헤드를 고려야해요!