

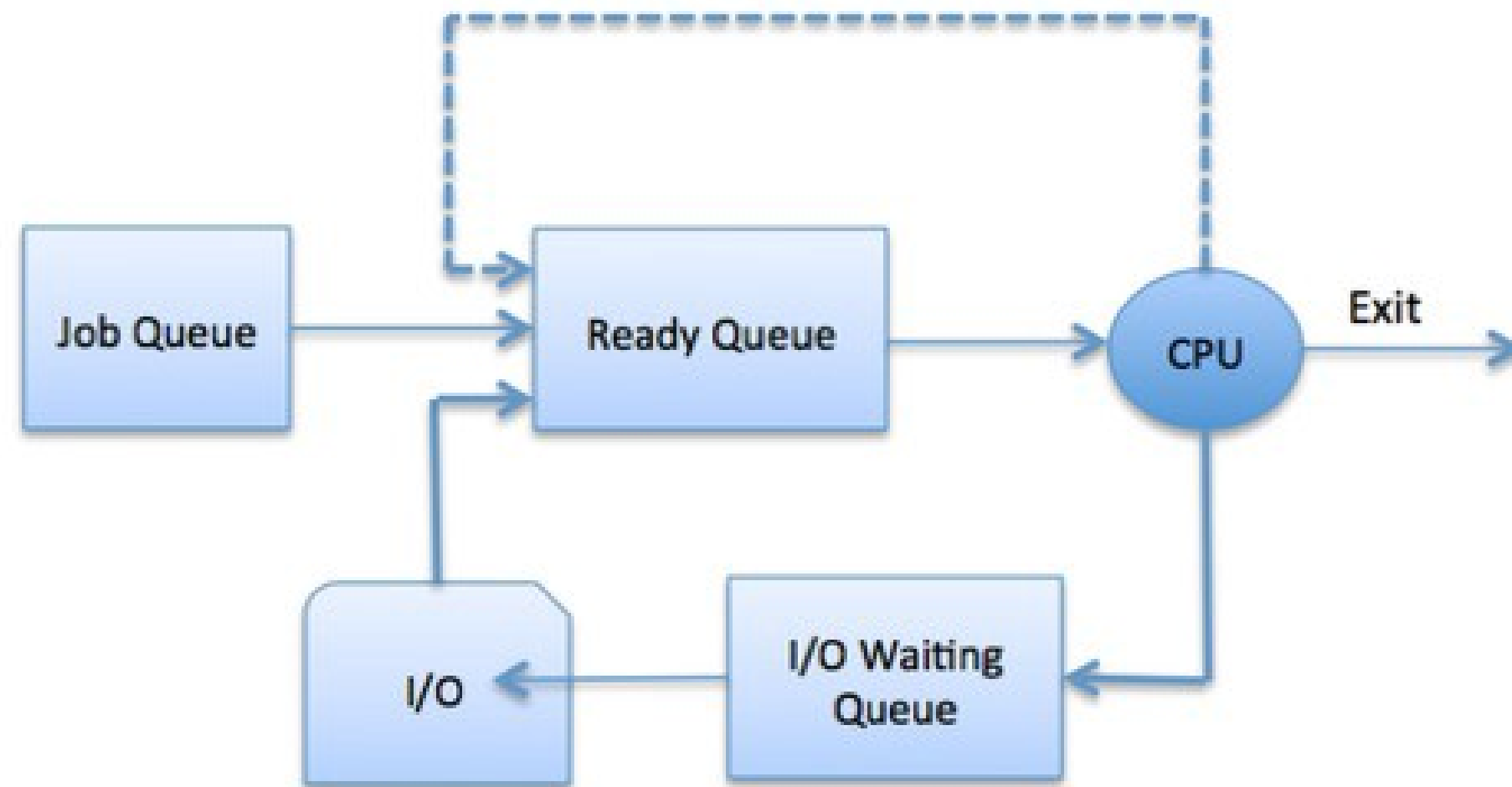
CPU Scheduling

목차

1. CPU 스케줄링이 왜 필요할까?
 2. 선점형과 비선점형 방식
 3. CPU 스케줄링 기법
 4. 리눅스에서 스케줄링 방법
-

CPU 스케줄링이 왜 필요할까?

CPU 스케줄링이 왜 필요할까?



한정된 CPU 자원을 여러
프로세스가 효율적으로
사용하기 위해서

CPU 스케줄링 목표

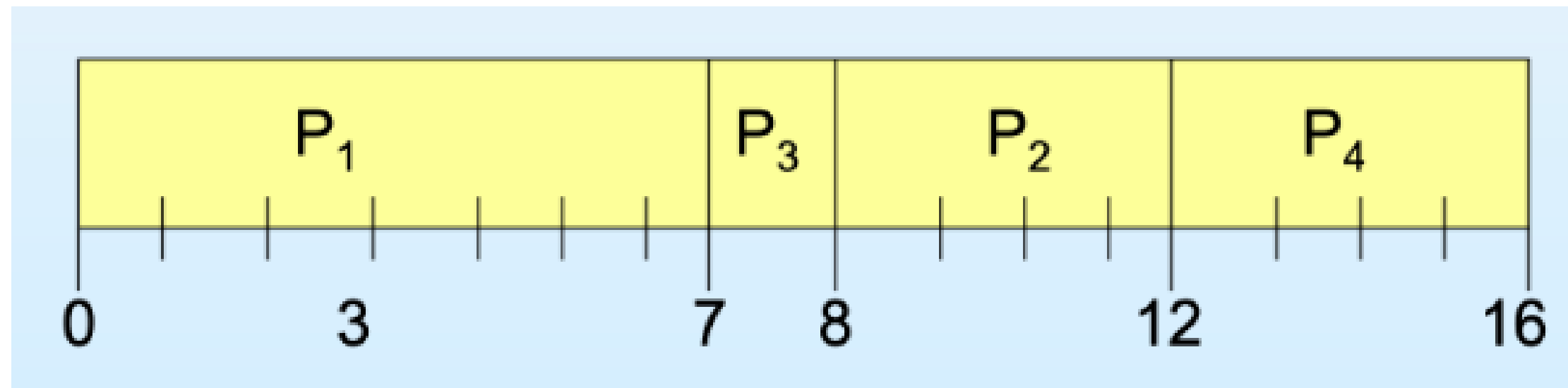
- **CPU Utilization** : CPU를 쉬지 않고 사용하는가
- **Throughput** : 단위시간 당 처리량
- **Turnaround Time** : 수행을 요청한 후 작업이 끝날 때까지 걸릴 시간
- **Waiting Time** : 프로세스가 Ready Queue에서 기다린 총 시간
- **Response Time** : 수행을 요청한 후 최초로 CPU를 할당받기 까지 걸린 시간

CPU 스케줄링 목표

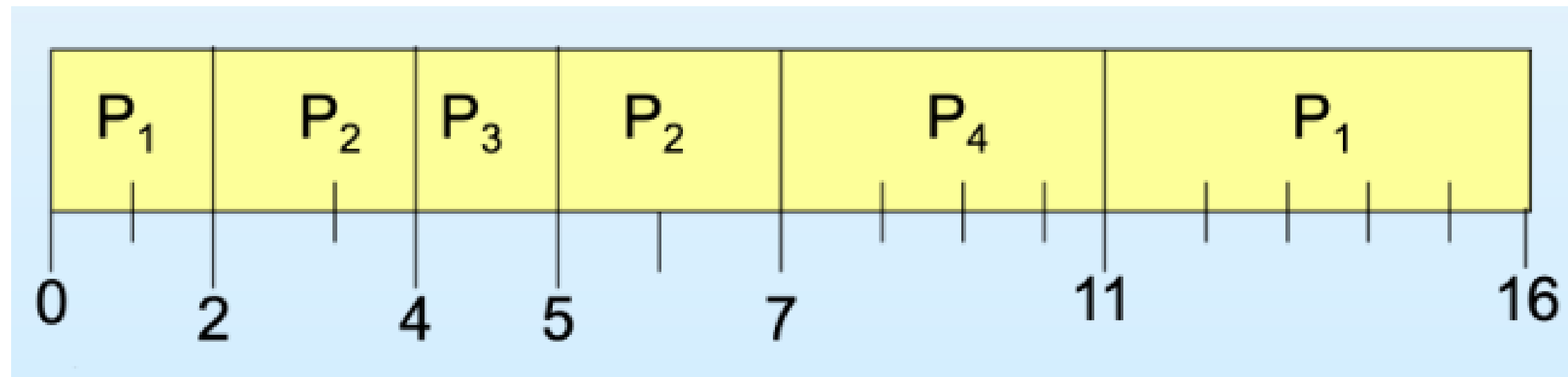
- CPU Utilization, Throughput은 늘리고, Time 지표들을 줄이는 것이 목표
- 하지만, 이러한 성능 척도들은 서로 상충 관계이다.
 - ex) 빠른 응답성을 보장하기 위해 짧은 시간 할당을 준 스케줄링 알고리즘은 잦은 Context Switching으로 인해 Throughput이 감소하게 된다.
- **단 하나 최고의 스케줄링 알고리즘을 찾는 것이 아닌, 시스템 혹은 프로세스의 목적에 맞춰 각각에 맞는 스케줄링 방식을 적용해야 한다.**

선점형 방식과 비선점형 방식

비선점형 방식



선점형 방식



CPU 스케줄링 기법

CPU 스케줄링 기법

1.FCFS

- a.Queue로 구현되어서, 먼저 Ready Queue에 도착한 순서대로 프로세스 처리
- b.비선점형
- c.Convey Effect : 만약 실행 시간이 매우 긴 작업 먼저 도착하면, 그 뒤의 작업들 대기가 길어진다.

2.Round Robin

- a.FCFS + 선점형 방식
- b.time slice 선택이 중요
 - i.너무 클 경우 : FCFS와 다를바가 없다.
 - ii.너무 작을 경우 : context swithcing에 따른 오버헤드 증가로 처리량 감소

CPU 스케줄링 기법

3. SJF

- 다음 CPU 버스트가 가장 짧은 프로세스 선택

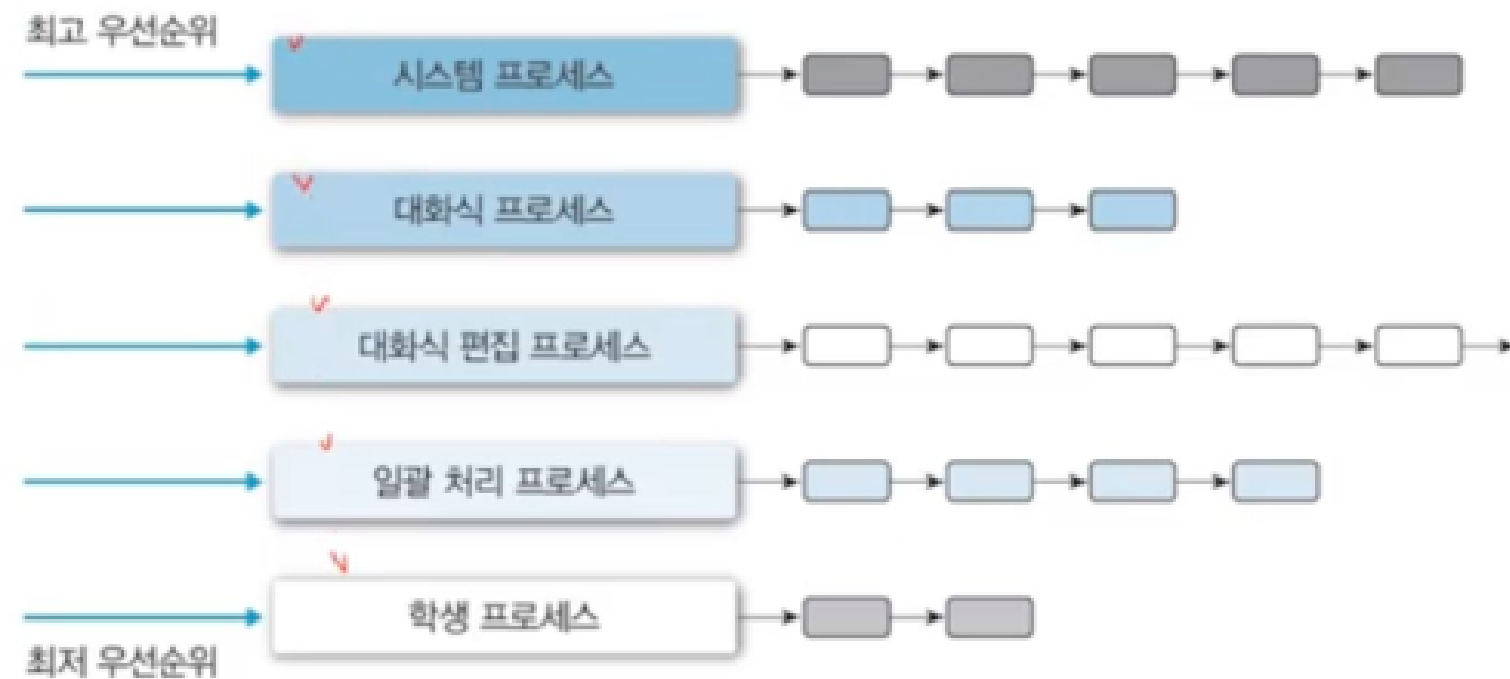
4. SRT

- SJF + 선점형 방식

5. 우선순위 스케줄링

- 우선순위를 부여하고, 가장 높은 우선순위 갖는 프로세스에게 CPU 할당
- **starvation**
- **aging**

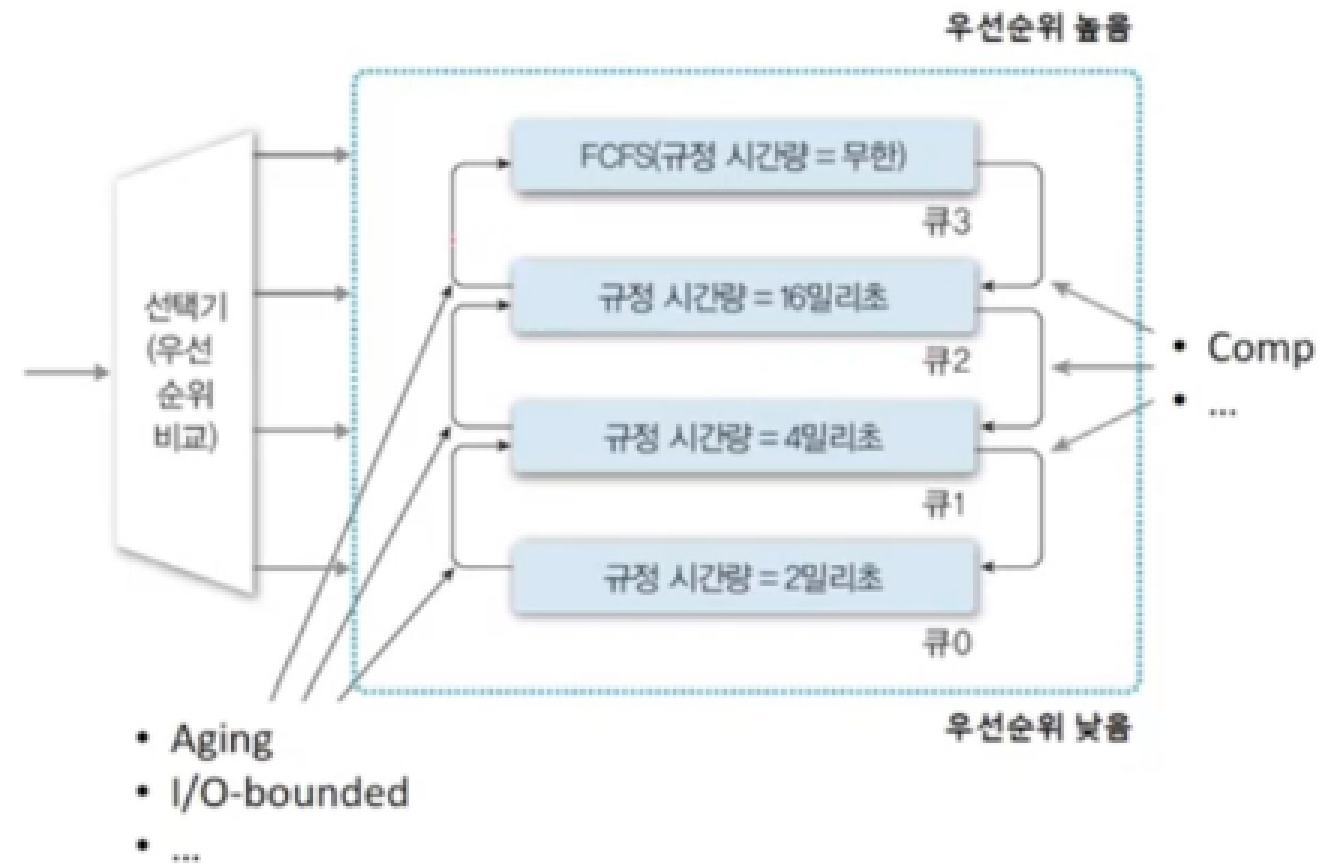
CPU 스케줄링 기법



Multilevel Queue(다단계 큐)

- 각 Queue는 자신만의 스케줄링 기법 갖을 수 있다.
- 단점
 - 프로세스가 큐에 할당되면 다른 큐로 이동 불가능 하다.
 - 우선순위가 높은 큐에 작업이 계속 유입 될 경우 starvation 상태 빠질 수 있다.

CPU 스케줄링 기법



Multilevel Feedback Queue

- MLQ와 컨셉은 동일하나 Queue간 이동 가능.
- 단점
 - 구현 복잡성
 - 높은 오버헤드
 - 예측 불가능성

리눅스의 스케줄링 기법

리눅스 스케줄링 기법

CFS(현재는 발전해서 EEVDF)

- 공정성을 척도로 동작
- 특정 시간안에 n개의 프로세스가 모두 동일한 시간동안 실행되는 컨셉
- vruntime
 - 프로세스가 지금까지 CPU를 얼마나 사용했는지를 나타내는 가상의 시간(우선순위가 반영된)
 - Red-Black Tree 에 vruntime 저장하여 프로세스 별 vruntime 기준 정렬
- ready queue에 있는 프로세스들 중에서 vruntime이 가장 작은 프로세스를 다음에 실행한다.
 - vruntime이 작다는 것은 그동안 CPU를 가장 적게 사용했다는 의미
 - 가장 CPU를 할당받지 못한 프로세스에게 할당해서 vruntime 최대한 비슷하게 유지

리눅스 스케줄링 기법

targeted latency(sched latency)

- CFS가 설정한 가장 작은 스케줄 단위
- ex) target latency가 20ms이면 2개의 작업이 있으면 각각 10ms, 20개의 작업이 있으면 1ms 씩 수행.

Niceness(우선순위)

- process마다 우선순위를 나타내는 nice 레벨 존재.
- 우선순위에 따라서 각각 프로세스가 할당받을 time slice 계산에 영향을 미침
- 기본값이 주어지고, 필요에 따라 개발자, 사용자가 조절가능.

리눅스 스케줄링 기법

time slice

$$\text{time_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \cdot \text{sched_latency}$$

vruntime

$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \cdot \text{runtime}_i$$

리눅스 스케줄링 기법

- 스케줄링 클래스
 - 리눅스 커널은 단일 스케줄러가 아닌, 여러 스케줄링 정책을 계층적으로 운영한다.
 - 각 정책을 '스케줄링 클래스'라 부르며, 각각의 자신만의 스케줄링 알고리즘 갖는다.
- 실시간(Real-Time) 스케줄링 클래스:
 - **SCHED_DEADLINE** : 가장 높은 우선순위를 가지며, **EDF(Earliest Deadline First)** 알고리즘을 기반으로 마감 시간이 가장 가까운 작업을 먼저 처리
 - **SCHED_FIFO** : 정적 우선순위를 가지며, 같은 우선순위 내에서는 **FCFS(First-Come, First-Served)** 방식으로 작동하는 실시간 스케줄러. 한번 **CPU**를 잡으면 스스로 **놓거나**, 더 높은 우선순위의 프로세스가 나타나기 전까지 계속 실행.
 - **SCHED_RR** : **SCHED_FIFO**와 기본적으로 같지만, 같은 우선순위 내의 **프로세스들을 라운드 로빈(Round Robin)** 방식으로 정해진 시간 할당량(**time quantum**)만큼 번갈아 실행.
- 일반(Normal) 스케줄링 클래스:
 - **SCHED_NORMAL (CFS/EEVDF)**: 우리가 흔히 아는 대부분의 일반 프로세스들이 이 클래스에 속한다. 이 클래스는 **EEVDF(Earliest Eligible Virtual Deadline First)** 스케줄러(커널 6.6 이전에는 CFS)를 사용하여 모든 프로세스에게 '완벽하게 공정한' CPU 시간을 배분하려고 노력.

리눅스 스케줄링 기법

```
/*
 * pick_next_task() - 다음에 실행할 태스크를 선택한다.
 *
 * 이 함수는 우선순위가 가장 높은 스케줄러 클래스부터 순회하며
 * 실행 가능한 태스크가 있는지 확인한다.
 */
static __always_inline struct task_struct *
pick_next_task(struct rq *rq)
{
    const struct sched_class *class;
    struct task_struct *p;

    /*
     * 최적화 경로: 실행 큐에 있는 모든 태스크가 일반(fair) 클래스에
     * 속해 있다면, 굳이 다른 클래스를 확인할 필요 없이 바로 fair 클래스의
     * pick_next_task를 호출한다. 대부분의 시스템에서 이 조건이 참이다.
     */
    if (rq->nr_running == rq->cfs.h_nr_running) {
        p = fair_sched_class.pick_next_task(rq);
        if (p)
            return p;
    }


    /*
     * 일반 경로: 가장 높은 우선순위의 스케줄러 클래스부터 시작하는 루프.
     * sched_class 구조체들은 next 포인터를 통해 연결 리스트로 구현되어 있다.
     */
    for_each_sched_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }


    /* 이 코드는 실행되어서는 안 된다. 항상 idle 태스크가 존재하기 때문 */
    BUG();
    return NULL; /* 컴파일러 경고를 피하기 위함 */
}

// for_each_sched_class(class) 매크로는 다음과 같이 정의되어 있습니다.
// for (class = sched_class_highest; class; class = class->next)
```


torvalds/linux

Linux kernel source tree




 17k


Contributors

 1


Used by

 200k

Stars


 57k

Forks



linux/kernel/sched/core.c at master · torvalds/linux

Linux kernel source tree. Contribute to torvalds/linux development by creating an account on GitHub.

 GitHub

MLQ(Multi level Queue)와는 약간 다르다.

- 개념적으로는 비슷하지만, MLQ와는 다르다고 한다.
- 공통점
 - 커널은 가장 우선순위가 높은 큐부터 돈다는 점
 - 각 큐들이 여러개의 큐 구조로 나뉘어져 있고, 각 큐는 각자의 스케줄러를 갖고 있다는점
- 차이점
 - 과거 MLQ에서는 프로세스가 어떤 특성을 갖는지 휴리스틱하게 추측하는 과정이 있다.
 - 위의 추측을 바탕으로 특성에 따라 큐에 배치
 - 현대 리눅스에서는 추측하는 과정 없이 vruntime을 계산하는 방식으로 바뀌었다.(CFS)
 - 어떤 큐에 들어갈지는 '개발자의 명시', '시스템 개입'(커널의 시스템 안정성 유지, 데드락 방지 등)으로만 판단.
 - 프로세스에 대해 파악하는 과정이 없어졌다.