상태 전이와 실행 수준 변화

상태 전이
State transition

상태 전이 (State transition)

태스크가 자원요청을 했지만 해당 자원을 당장 제공해 줄 수 없는 상태일때,

1. 해당 태스크를 **대기 상태로 전환**시키고
2. 다른 태스크를 진행한 뒤
3. 사용가능해질 때 수행시켜줄 수 있는 특징

상태 전이 (State transition)

- TASK_RUNNING
- TASK_DEAD
- TASK_STOPPED
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_KILLABLE

태스크 생성 및 실행
태스크 종료
시그널
대기 상태

상태 전이 (State transition)

TASK_RUNNING - 프로세스가 CPU에서 실행 중이거나 실행되기를 기다리는 중

- TASK_RUNNING (ready)
  > CPU 점유하기까지 대기
- TASK_RUNNING (running)
  > CPU 점유 후 태스크가 실행 중

상태 전이 (State transition)

1. CPU 시간을 모두 사용 시
2. 다른 태스크가 높은 우선순위를 가질 시

상태 전이 (State transition)

1. 스케쥴링에 맞춰서

상태 전이 (State transition)

TASK_INTERRUPTIBLE - 프로세스가 어떤 조건을 기다리며 보류중.

- 하드웨어 인터럽트 발생
- 프로세스 자원 해지
- …

발생 시 TASK_RUNNING로 돌아 감.

상태 전이 (State transition)

TASK_UNINTERRUPTIBLE – TASK_INTERRUPTIBLE과 비슷하지만 시그널 반응 x

- 하드웨어 인터럽트 발생
- 프로세스 자원 해지
- …

발생 시 TASK_RUNNING로 돌아 감.

상태 전이 (State transition)

TASK_STOPPED – 프로세스 실행 중단 상태

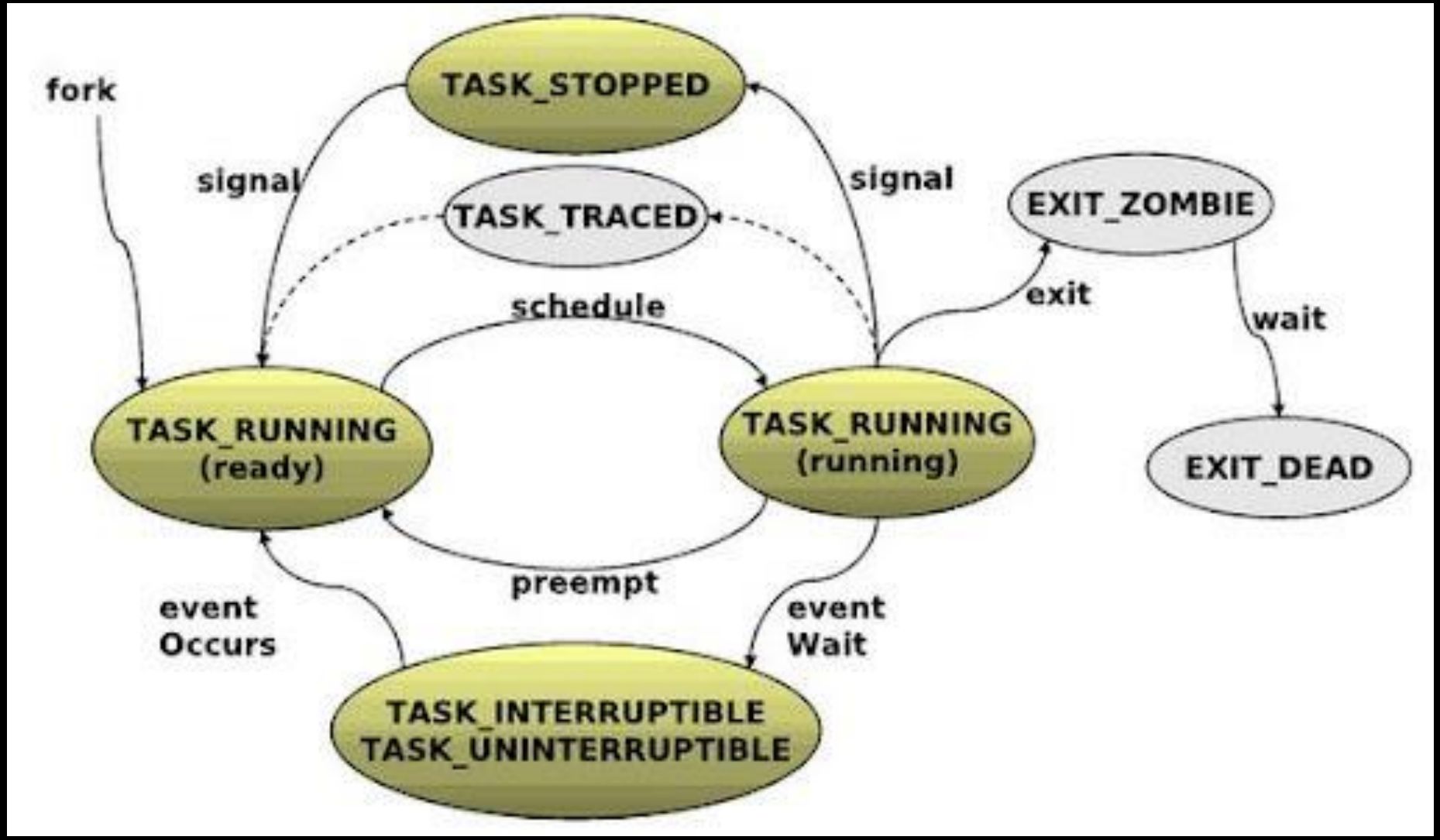
SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU 시그널 받으면 현재 상태가 됨.

상태 전이 (State transition)

TASK_DEAD – 프로세스가 제거되는 중

- TASK_DEAD (EXIT_ZOMBIE)
  > 할당된 자원 대부분을 커널에게 반환
  > 종료된 이유, 자원 통계정보 유지 (wait() 호출을 위해)

- TASK_DEAD (EXIT_DEAD)
  > 부모 프로세스가 wait() 등의 함수 호출 시 전이
  > 자식 태스크는 모든 것을 반환 및 종료


fork
TASK_STOPPED
signal
signal
TASK_TRACED
EXIT_ZOMBIE
schedule
exit
wait
TASK_RUNNING
(ready)
TASK_RUNNING
(running)
EXIT_DEAD
preempt
event
Occurs
event
Wait
TASK_INTERRUPTIBLE
TASK_UNINTERRUPTIBLE

실행 수준 변화
Change of Execution level

실행 수준 변화 (Change of Execution Level)

- User Level Running

  사용자 수준에서 제작된 응용 프로그램 & 라이브러리 코드 수행 상태

- Kernel Level Running

  CPU에서 커널 코드를 수행하고 있는 상태

실행 수준 변화 (Change of Execution Level)

User Level → **Kernel Level**

1 .System call 사용

Task의 system call → Kernel Interrupt → Kernel level running

2. Interrupt 발생

Task에서 Interrupt 발생 → Kernel Interrupt → Kernel level running

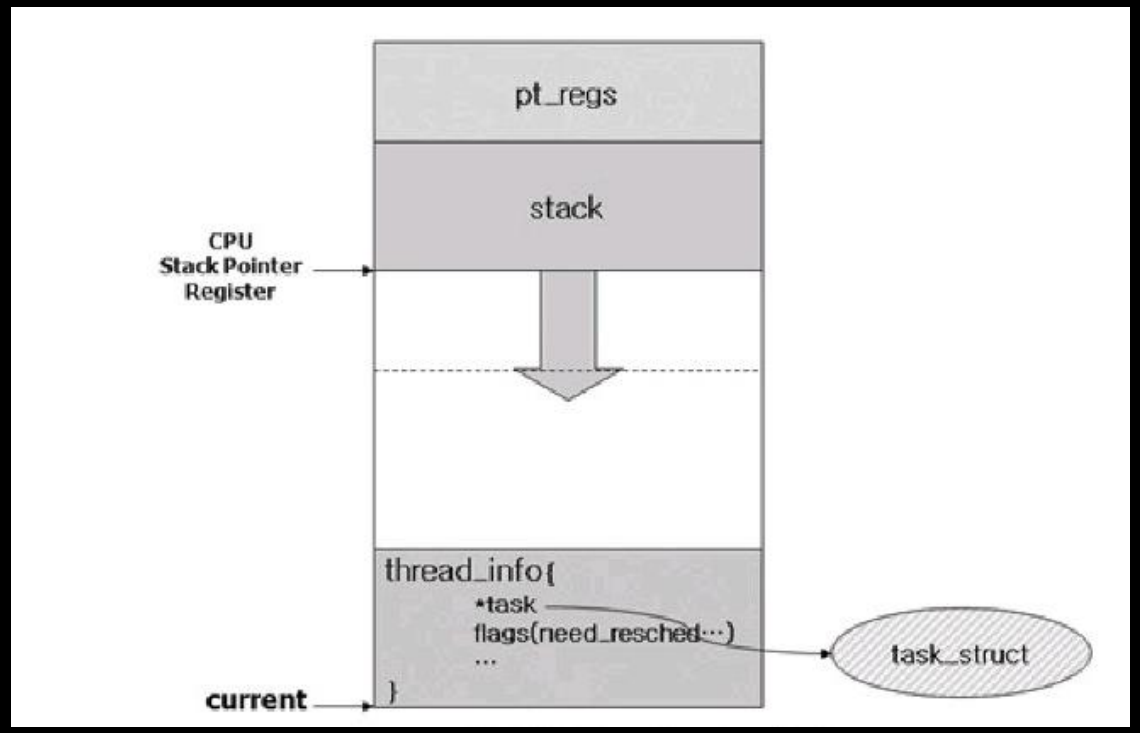
실행 수준 변화 (Change of Execution Level)

Kernel Level → **User Level**

User → Kernel → User 변화에서 Kernel 진입 전 작업 상황을 저장해야 함.
Kernel stack 상단부에 `pt_regs` 구조체를 통해 저장.

시그널 처리 핸들러 호출
스케줄러 재호출 필요 여부에 따라 스케줄러 호출
연기된 루틴들 존재시 실행

실행 수준 변화 (Change of Execution Level)



- `pt_regs`
  문맥 교환 시 현재 레지스터 값을 저장
- `stack`
  커널 스택
- `thread_info`
  `task_struct` 포인터,
  스케줄링 플래그 등을 포함

Run queue & Scheduling

Run queue & Scheduling

- Scheduling

- 여러 task중 **다음 수행시킬 task에 cpu를 할당하는 과정.**
- 리눅스의 task는 실시간, 일반 task로 나뉘며
  이를 위해 **별도의 스케줄링 알고리즘**이 구현되어 있음.
- 140단계의 우선순위 중 실시간 task는 0~99 단계,
  일반 task는 100~139 단계까지 사용.
  (숫자가 낮을수록 높은 우선순위)

Run queue & Scheduling

- Scheduling

태스크	단계	알고리즘
실시간	0 ~ 99	CFS
일반	100 ~ 139	FIFO, RR, DEADLINE

Run queue & Scheduling

- Run queue

- 스케줄링의 수행을 위해 **수행 가능한 상태의 task를 관리하는 자료구조**
- 복수의 CPU를 가진 시스템은 각 CPU가 각자의 런 큐를 가지고 있음.
- task_list에 연결된 task 중 TASK_RUNNING 상태로 전이된 task는 부모 task가 존재하던 런 큐로 삽입 (task_struct의 cpus_allowed 필드를 이용)

Run queue & Scheduling

- Run queue

task_list (init_task)

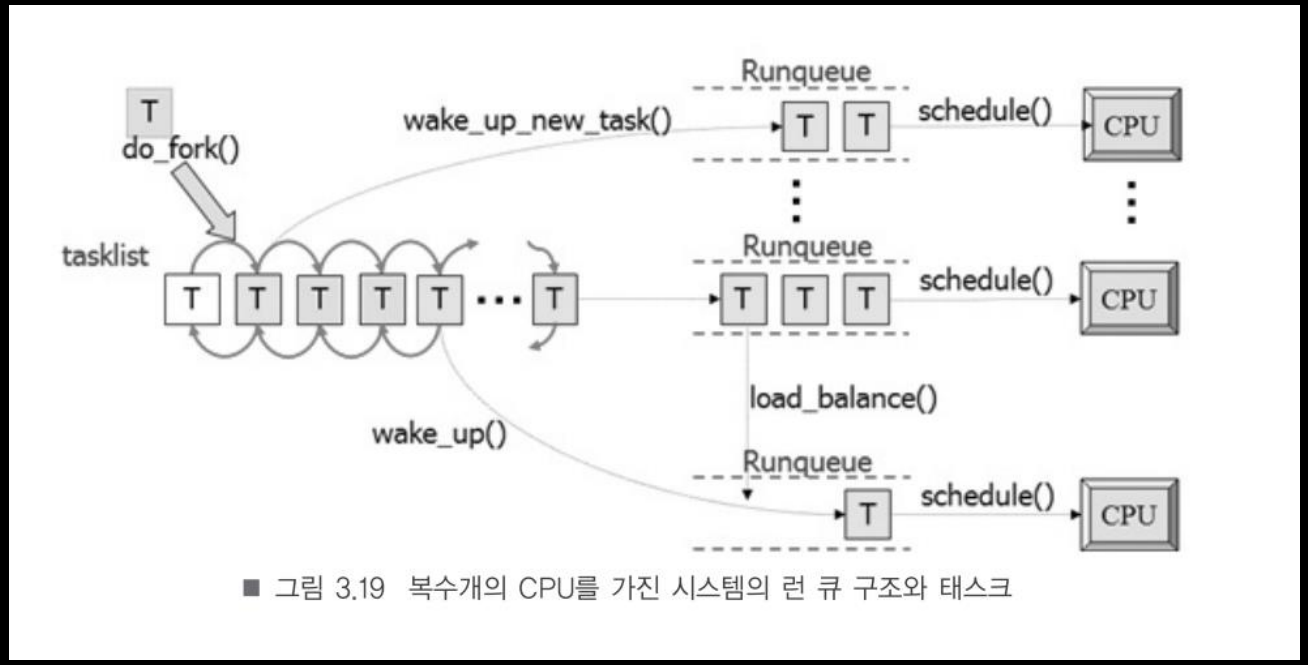
- 이중 연결 리스트.
- `init_task`가 header.
- 모든 태스크가 linked.
- `TASK_RUNNING` 상태의 task는 Run queue중 하나에 소속됨.

Run queue & Scheduling

- Run queue

- wake_up_new_task()
  > 새롭게 생성 & TASK_RUNNING 상태

- wake_up()
  > 이벤트 대기 중 활성화



■ 그림 3.19 복수개의 CPU를 가진 시스템의 런 큐 구조와 태스크

Run queue & Scheduling

- Run queue

1. 새롭게 생성된 태스크가 런 큐에 들어가는 경우
   > 부모 태스크를 따라간다.
   > 캐시 친화력 활용

2. 대기 상태에서 깨어난 태스크가 런 큐에 들어가는 경우
   > 이전에 수행되던 CPU의 런 큐에 삽입
   > 캐시 친화력 활용

※ `task_struct`의 `cpus_allowed` field → CPU번호 저장, 런 큐 분배시 사용

Run queue & Scheduling

- Run queue

/kernel/sched/sched.h

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
          /* runqueue lock: */
          raw_spinlock_t                lock;

          ...

          /* capture load from *all* tasks on this CPU: */
          struct load_weight  load;
          unsigned long                 nr_load_updates;
          u64                           nr_switches;

          struct cfs_rq                 cfs;
          struct rt_rq                  rt;
          struct dl_rq                  dl;
...
```

■ 일반 태스크 스케줄링
■ 실시간 태스크 스케줄링

Run queue & Scheduling

- 실시간 태스크 스케줄링

- 실시간 태스크의 우선순위

  `rt_priority` = 0~99까지의 우선순위 설정 가능

  예외를 제외하고는 `rt_priority`에 따름

- 예외

  1. 태스크의 수행 종료
  2. 태스크의 자체 중지
  3. 태스크의 타임슬라이스 전체 소모 (SCHED_RR 정책에만 해당)

- 효율적인 우선순위 판별

  bitmap 도입

Run queue & Scheduling

- 실시간 태스크 스케줄링 → DEADLINE

Deadline에 가장 가까운 task를 스케줄링 대상으로 선정.

Run queue & Scheduling

- 실시간 태스크 스케줄링 → DEADLINE

- 완료시간 : deadline
- 작업량 : runtime
- 주기성 : period

Run queue & Scheduling

- 실시간 태스크 스케줄링 → DEADLINE

- Runtime의 합은 CPU의 최대 처리량을 넘길 수 없음.
- Runtime과 Period를 이용해 성공적인 완료 여부를 확정적 결정 가능
- 우선 순위가 의미가 없음
  - 기아 현상에 효율적임
  - 영상, 음성, 스트리밍에 효과적 (제약시간을 가지는 것들)

Run queue & Scheduling

- 실시간 태스크 스케줄링 → DEADLINE

- Red-Black Tree (`rbtree`)
  - `SCHED_DEADLINE`을 채택한 태스크들은 `rbtree`라는 구조체에 정렬.
- 스케줄러 호출 시
  - 가장 가까운 deadline을 가진 태스크를 스케줄링 대상으로 선정
- `struct dl_rq` 사용
  - 태스크 정렬을 위한 `rbtree` 자료구조 존재

Run queue & Scheduling

- 실시간 태스크 스케줄링 → RR / FIFO

- 우선순위를 위한 근거
  - policy
  - prio
  - rt_priority
- `struct rt_rq` 사용
  - 우선순위별로 태스크 관리 위해 비트맵 & 큐 존재

```c
/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
        struct rt_prio_array active;
        unsigned int rt_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
        struct {
                int curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
                int next; /* next highest */
#endif
        } highest_prio;
#endif
#ifdef CONFIG_SMP
        unsigned long rt_nr_migratory;
        unsigned long rt_nr_total;
        int overloaded;
        struct plist_head pushable_tasks;
#endif
        int rt_queued;

        int rt_throttled;
        u64 rt_time;
        u64 rt_runtime;
        /* Nests inside the rq lock: */
        raw_spinlock_t rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
        unsigned long rt_nr_boosted;

        struct rq *rq;
        struct task_group *tg;
#endif
};
```

```c
/* Deadline class' related fields in a runqueue */
struct dl_rq {
        /* runqueue is an rbtree, ordered by deadline */
        struct rb_root rb_root;
        struct rb_node *rb_leftmost;

        unsigned long dl_nr_running;

#ifdef CONFIG_SMP
        /*
         * Deadline values of the currently executing and the
         * earliest ready task on this rq. Caching these facilitates
         * the decision wether or not a ready but not running task
         * should migrate somewhere else.
         */
        struct {
                u64 curr;
                u64 next;
        } earliest_dl;

        unsigned long dl_nr_migratory;
        int overloaded;

        /*
         * Tasks on this rq that can be pushed away. They are kept in
         * an rb-tree, ordered by tasks' deadlines, with caching
         * of the leftmost (earliest deadline) element.
         */
        struct rb_root pushable_dl_tasks_root;
        struct rb_node *pushable_dl_tasks_leftmost;
#else
        struct dl_bw dl_bw;
#endif
};
```

Run queue & Scheduling

- 일반 태스크 스케줄링 → CFS

완벽하게 공평한 (Completely fair) 스케줄링을 추구하는 기법

```c
/* CFS-related fields in a runqueue */
struct cfs_rq {
        struct load_weight load;
        unsigned int nr_running, h_nr_running;

        u64 exec_clock;
        u64 min_vruntime;
#ifndef CONFIG_64BIT
        u64 min_vruntime_copy;
#endif

        struct rb_root tasks_timeline;
        struct rb_node *rb_leftmost;

        /*
         * 'curr' points to currently running entity on this cfs_rq.
         * It is set to NULL otherwise (i.e when none are currently running).
         */
        struct sched_entity *curr, *next, *last, *skip;

#ifdef  CONFIG_SCHED_DEBUG
        unsigned int nr_spread_over;
#endif

#ifdef CONFIG_SMP
        /*
         * CFS Load tracking
         * Under CFS, load is tracked on a per-entity basis and aggregated up.
         * This allows for the description of both thread and group usage (in
         * the FAIR_GROUP_SCHED case).
         */
        unsigned long runnable_load_avg, blocked_load_avg;
        atomic64_t decay_counter;
        u64 last_decay;
        atomic_long_t removed_load;
```

Run queue & Scheduling

- 일반 태스크 스케줄링 → CFS

1. **공평의 기준은 '시간'으로서 CPU 사용량이 공평해야 함.**
2. vruntime을 이용해 우선순위를 처리 함.
3. vruntime의 문제점에 대한 timeslice 솔루션 존재.

Run queue & Scheduling

- 일반 태스크 스케줄링 → CFS

- vruntime은 우선순위에 따른 가중치가 존재한다. (prio_to_weight[])

- Timer interrupt handler에서 주기적으로 scheduler_tick() 함수 호출
  → vruntime 값을 갱신

- **vruntime = vruntime 갱신 + 우선순위에 따른 가중치 (공식 존재)**
  - 가장 작은 vruntime 값을 가지는 태스크가 가장 과거 사건
  - rbtree 자료구조를 이용해서
    가장 작은 vruntime을 가지는 태스크 찾아 다음 스케줄로 선정
    - 공평한 스케줄링

Run queue & Scheduling

- 일반 태스크 스케줄링 → timeslice

- 리눅스는 시간 단위를 우선순위에 기반하여 각 task에 분배.
- 시간 단위는 잦은 스케줄링으로 인한 오버헤드를 최소화하기 위해 존재.
  `__sched_period()` 함수에서 계산.

Run queue & Scheduling

- 일반 태스크 스케줄링 → timeslice

- vruntime에 의하면 vruntime 값이 계속 갱신 될 때마다 스케줄링 발생.
  - 선점되지 않고 CPU를 사용할 수 있는 시간, 즉 timeslice가 지정되어 있음.
  - sched_slice() 함수로 timeslice 계산
  - __sched_period() 함수로 오버헤드 최소화

- timeslice가 작은 태스크가 존재할 수 있음.
  - 스케줄링간 최소 지연 시간이 정의

Run queue & Scheduling

- 일반 태스크 스케줄링 → scheduler 호출

- When : 4가지 경우
- How : 2가지 경우

> 직접적으로 `schedule()` 함수 호출

> 수행중인 태스크의 `thread_info` 내부의 `flag` 중 **`need_resched` 필드 설정**

Run queue & Scheduling

- 일반 태스크 스케줄링 → scheduler 호출

WHEN

1. Timer interrup의 루틴이 종료되는 시점에 **task의 need_resched 확인**해 결과에 따라 rescheduling
2. 현재 수행중인 태스크가 **자신의 timeslice 모두 사용, 혹은 이벤트를 대기**할 때 rescheduling
3. **새롭게 태스크 생성, 혹은 대기상태의 태스크가 활성화**될 때 rescheduling
4. 해당 태스크가 **스케줄링 관련 system call**을 할 때 rescheduling (`sched_setscheduler()`, …)

Run queue & Scheduling

- 일반 태스크 스케줄링 → timeslice

- 그룹 스케줄링 기법
  - 사용자 ID 기반 그룹 스케줄링
  - **특정 사용자간에 공평하게 CPU 배분**

- Cgroup 가상 파일시스템 기반 그룹 스케줄링
  - 사용자가 지정한 태스크들을 하나의 그룹으로 취급,
    그룹 간에 공평하게 CPU 배분