



2016년 2학기 운영체제실습 9주차 (1/2)

CPU Scheduling

Dept. of Computer Engineering,
Kwangwoon Univ.

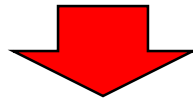
Contents

- ▶ **Scheduling**
- ▶ **Linux Scheduling Policy**
- ▶ **O(1) Scheduler**
- ▶ **CFS (Completely Fair Scheduler)**
- ▶ **CFS Scheduler**

Scheduling

▶ Scheduler

- ▶ 한정된 자원을 다수의 client가 사용하려할 때
 - ▶ Ready queue에 client들이 대기
 - ▶ Throughput, response time 등의 성능 향상을 위해
 - ready queue 내의 client들의 순서 조정 필요



Scheduling

- ▶ OS에서의 Scheduling
 - ▶ CPU Scheduling
 - Process(task and thread)
 - ▶ Storage Scheduling
 - I/O requests
 - ▶ Network Scheduling
 - Packets

Linux Scheduling Policy (1/3)

▶ Preemptive Scheduling

- ▶ 실행 중인 task가 CPU를 뺏길 수 있음
 - ▶ time slice의 만료
 - ▶ 더 높은 priority를 가지는 task의 실행

▶ I/O bound task vs CPU-bound task

- ▶ CPU의 사용 시간을 고려하여 분류
- ▶ I/O-bound
 - ▶ 대부분의 시간을 I/O 요청을 함
 - ▶ e.g. text editor
- ▶ CPU-bound
 - ▶ 대부분의 시간을 code 실행
 - ▶ e.g. compiler, video encoder
- ▶ Linux에서는 I/O-bound task를 우선적으로 처리

Linux Scheduling Policy (2/3)

▶ Range of Priority

- ▶ nice 값
 - ▶ -20(highest) ~ +19(lowest)
 - ▶ default value : 0
- ▶ Real time priority
 - ▶ 0 ~ 99
 - ▶ 모든 실시간 task는 일반 task보다 더 높은 priority를 가진다
- ▶ 높은 priority를 가지는 task가 먼저 수행
- ▶ Linux에서는 task 실행 중 priority값이 동적으로 변경됨
 - ▶ I/O-bound task가 높은 priority를 부여 받음

▶ Time Slice

- ▶ Task의 priority에 따라 time slice를 배분
 - ▶ 높은 priority를 가진 task는 더 많은 time slice를 받는다
- ▶ Time slice만큼 CPU를 사용 가능
 - ▶ 선점 가능

Linux Scheduling Policy (3/3)

▶ Context Switch

- ▶ 하나의 task context(문맥)에서 다른 task의 context로 교환

▶ Context

- ▶ Address space, Stack pointer, CPU register

```
static inline struct task_struct
*context_switch(struct rq *rq, struct task_struct *prev, struct task_struct *next)
{
    struct mm_struct *mm = next->mm;
    struct mm_struct *oldmm = prev->active_mm;
    ...
    switch_mm(oldmm, mm, next);
    ...
    switch_to(prev, next, prev);
}
```

▶ switch_mm(), switch_to()

- ▶ Task의 address space, stack pointer, register를 switch할 task의 값들과 교환을 담당하는 함수

O(1) Scheduler (1/5)

▶ O(1) Scheduler

- ▶ Linux kernel 2.6.22까지 사용하는 CPU Scheduler
 - ▶ time slice를 가지면서 가장 높은 priority를 가지는 task를 먼저 수행
- ▶ 시간 복잡도
 - ▶ $O(1)$: 여러 task가 ready queue에 존재하더라도 scheduling 시 발생하는 overhead가 $O(1)$
- ▶ 특징
 - ▶ SMP(Symmetric Multi Processor) 지원
 - ▶ Response time의 감소
 - ▶ Fairness 제공
- ▶ CPU 당 하나의 Run queue를 가짐
 - ▶ 실행 가능(TASK_RUNNING)한 task들의 목록
 - ▶ task는 하나의 run queue에만 존재

O(1) Scheduler (2/5)

▶ O(1) Scheduler

▶ Priority Arrays

▶ 하나의 Run queue는 두 개의 priority array를 가짐

- Active : time slice가 남은 task들의 목록
- Expired : time slice가 만료된 task들의 목록

```
struct runqueue{
    spinlock_t lock;                /* spin lock that protects this runqueue */
    unsigned long nr_running;        /* number of runnable tasks */
    unsigned long nr_switches;      /* context switch count */
    unsigned long expried_timestamp; /* time of last array swap */
    unsigned long nr_uninterruptible; /* uninterruptible tick */
    unsigned long long timestamp_last_tick; /* last scheduler tick */
    struct task_struct *curr;        /* currently running task */
    struct task_struct *idle;        /* this processor's idle task */
    struct mm_struct *prev_mm;       /* mm_struct of last ran task */
    struct prio_array *active;        /* active priority array */
    struct prio_array *expired;      /* the expired priority array */
    struct prio_array arrays[2];     /* the actual priority arrays */
    struct task_struct *migration_thread; /* migration thread */
    struct list_head migration_queue; /* migration queue */
    atimic_t nr_iowait;              /* number of tasks waiting on I/O */
};
```

▶ 각 priority array는 하나의 queue를 가짐

▶ Priority bitmap

- 가장 높은 priority를 가진 task를 빠르게 검색

```
struct prio_array{
    int nr_active;                  /* number of tasks in the queues */
    unsigned long bitmap[BITMAP_SIZE]; /* priority bitmap */
    struct list_head queue[MAX_PRI0]; /* priority queues */
};
```


O(1) Scheduler (3/5)

▶ Time Slice 계산

- ▶ Priority array를 이용
- ▶ task의 time slice 만료 시
 - ▶ task의 priority에 따라 동적으로 time slice 계산
 - `task_timeslice(struct task_struct *p)`

```
static inline unsigned int task_timeslice(struct task_struct *p)
{
    return static_prio_timeslice(p->static_prio);
}
```

- ▶ Expired array로 이동되어 queue에 삽입

- ▶ Active array의 모든 task가 expired array로 이동 시
 - ▶ Active array와 expired array의 pointer를 교환

```
struct prio_array *array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
}
```

- ▶ 다수의 task가 존재하더라도 처리하는 시간은 O(1)만 걸림

O(1) Scheduler (4/5)

▶ schedule()

- ▶ 프로세스 스케줄링과 관련된 함수
- ▶ 아래의 경우 호출
 - ▶ 프로세스가 휴면(sleep)하는 경우
 - ▶ 프로세스가 선점(preemption)되는 경우

```
struct task_struct *prev, *next;  
struct list_head *queue;  
struct prio_array *array;  
int idx;
```

```
prev = current;  
array = rq->active;
```

```
idx = sched_find_first_bit(array->bitmap);
```

```
queue = array->queue + idx;
```

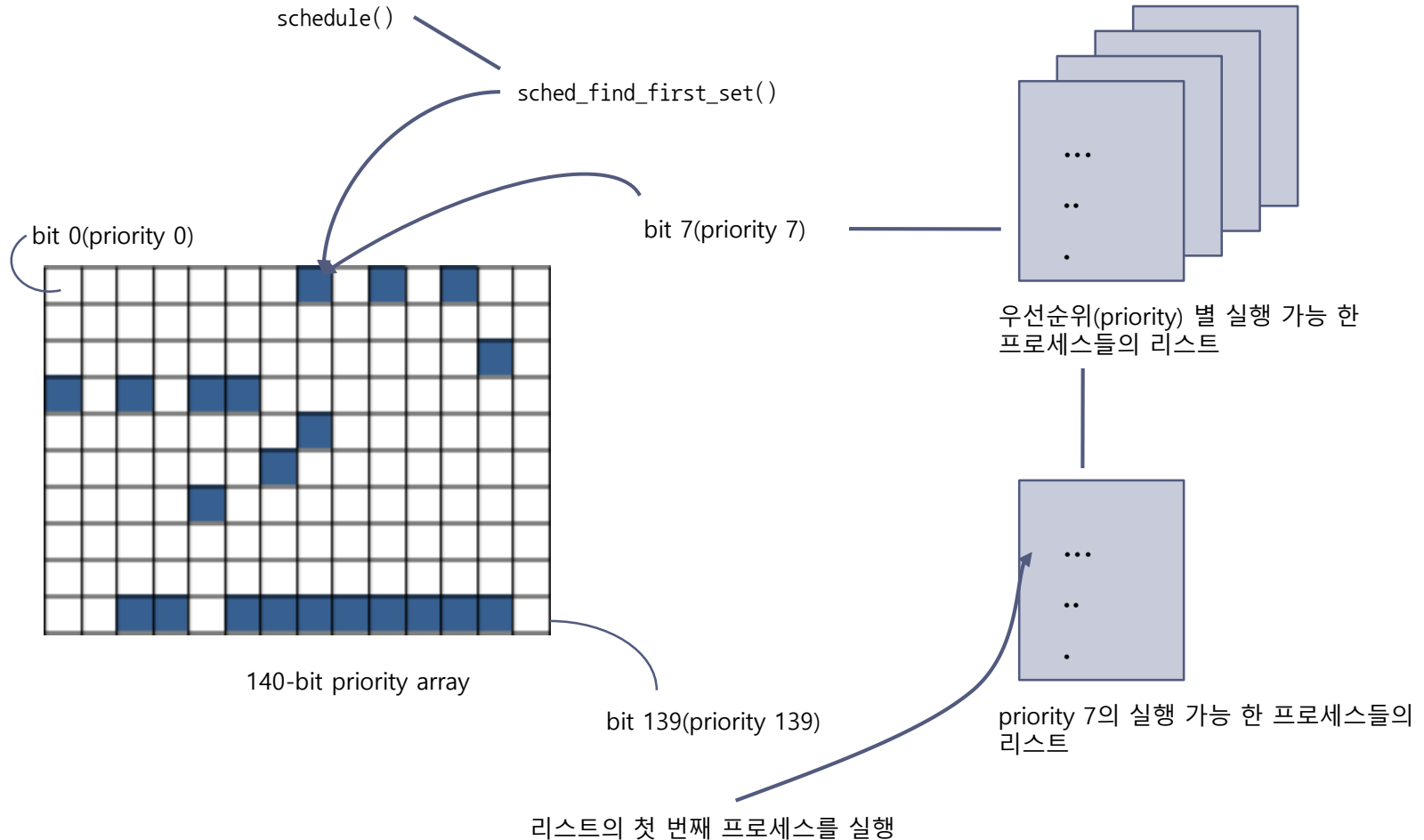
```
next = list_entry(queue->next, struct task_struct, run_list);
```

Active array에서
첫 번째로 설정된 bit를 찾는다

prev와 next가 다르면 새로운 task 실행
(context_switch() 호출)
실행 권한이 prev에서 next로 넘겨짐

O(1) Scheduler (5/5)

▶ O(1) Scheduler의 예



CFS (Completely Fair Scheduler) (1/6)

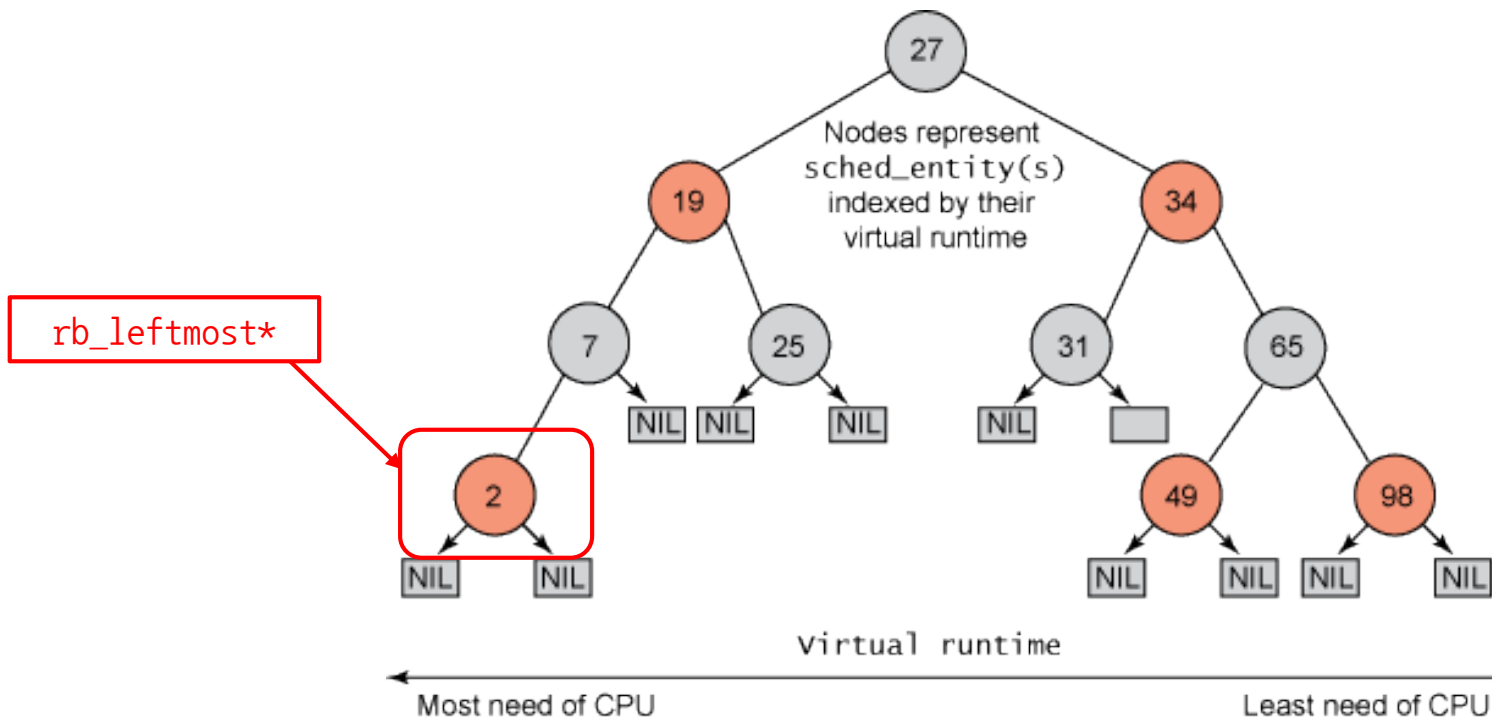
▶ CFS

- ▶ Linux kernel 2.6.23부터 사용되는 CPU scheduler
- ▶ O(1) scheduler의 문제점 해결
 - ▶ 빈번한 context switching의 발생 가능성
 - 적은 time slice를 갖는 task들이 많을 경우
 - context switching으로 인하 overhead 증가
 - Throughput 및 response time에 악영향
 - ▶ Priority에 따른 time slice 배분으로 인한 Unfair
 - expired array에 있는 task의 실행이 미뤄질 수 있음
- ▶ Time slice 계산
 - ▶ weight(가중치)에 기반한 time slice 계산
 - weight는 priority에 기반
 - ▶ priority가 낮더라도 좀 더 fair하게 time slice를 할당 받을 수 있음
- ▶ Red-Black tree를 사용하여 run queue를 관리
 - ▶ $O(\log N)$ 으로 $O(1)$ 에 비해 느리지만 성능 상의 큰 차이는 없음

CFS (Completely Fair Scheduler) (2/6)

▶ Run queue

- ▶ CFS는 Run queue 관리를 위해 Red-black tree 구조를 사용.
- ▶ 제일 작은 vruntime을 가진 프로세스를 찾기에 가장 효율적인 구조.
- ▶ rb_leftmost 포인터를 유지, 굳이 트리를 순회하지 않아도 되도록 최적화 됨.



CFS (Completely Fair Scheduler) (3/6)

▶ Scheduler entity structure

- ▶ CFS는 관리중인 모든 프로세스에 대해 sched_entity라는 구조체 유지.
 - ▶ `task_struct->sched_entity`
- ▶ 그리고 이 sched_entity들이 Red black tree 구조로 관리됨.
 - ▶ `task_struct->sched_entity.rb_node`
- ▶ 이 구조체는 CFS 스케줄링 작업을 달성하기 위해 충분한 정보를 포함.
 - ▶ `task_struct->sched_entity.vruntime`

```
struct sched_entity {  
    ...  
    struct rb_node run_node;  
    ...  
    u64 vruntime;  
    ...  
};
```

CFS (Completely Fair Scheduler) (4/6)

▶ Virtual runtime

- ▶ CFS는 각 우선순위마다 가중치를 부여.
- ▶ Virtual runtime은 가중치에 따라 real runtime을 정규화한 값.
- ▶ CFS는 이 virtual runtime이 제일 작은 프로세스를 실행.

$$curr_vruntime += delta_exec \times \left(\frac{NICE_0_LOAD}{curr_load_weight} \right)$$

```
static inline void __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
unsigned long delta_exec)
{
    ...
    delta_exec_weighted = delta_exec;
    if (unlikely(curr->load.weight != NICE_0_LOAD)) {
        delta_exec_weighted = calc_delta_fair(delta_exec_weighted, &curr->load);
    }

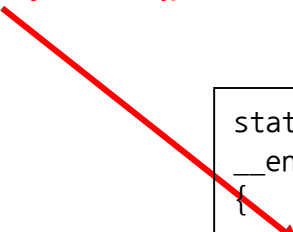
    curr->vruntime += delta_exec_weighted;
    ...
}
```

CFS (Completely Fair Scheduler) (5/6)

▶ Adding a process to the tree

- 프로세스를 Run queue에 등록 시,
 - enqueue_entity() 함수 호출.
- → 몇 가지 사전작업 후, __enqueue_entity() 함수 호출.
- → Red black tree에 해당 프로세스의 sched_entity 삽입.

```
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    ...
    __enqueue_entity(cfs_rq, se);
}
```



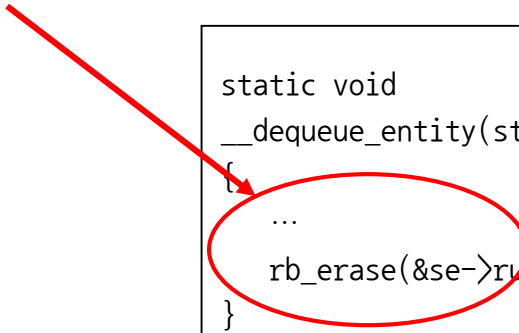
```
static void
__enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```


CFS (Completely Fair Scheduler) (6/6)

▶ Removing a process to the tree

- 프로세스를 Run queue에서 해제 시,
- -> dequeue_entity() 함수 호출.
- -> 몇 가지 사전작업 후, __dequeue_entity() 함수 호출.
- -> Red black tree로부터 해당 프로세스의 sched_entity 해제.

```
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int sleep)
{
    ...
    __dequeue_entity(cfs_rq, se);
    ...
}
```



```
static void
__dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    ...
    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

CFS Scheduler

▶ CFS Overview

```
struct task_struct {  
    ...  
    const struct sched_class *sched_class;  
    struct sched_entity se;  
    unsigned int policy;  
    unsigned int time_slice;  
    ...  
};
```

SCHED_NORMAL
SCHED_BATCH
SCHED_IDLE

SCHED_FIFO
SCHED_RR

**sched_class를 변경하여
스케줄링 정책 변경 가능.(유지보수)**

```
struct sched_entity {  
    struct load_weight load;  
    struct rb_node run_node;  
    ...  
#ifdef CONFIG_FAIR_GROUP_SCHED  
    struct sched_entity *parent;  
    struct cfs_rq *cfs_rq;  
    struct cfs_rq *my_rq;  
#endif  
};
```

**pick_next_task 함수는
다음에 스케줄링 될 태스크를 반환.**

```
static const struct sched_class fair_sched_class  
={  
    .next = &idle_sched_class,  
    .enqueue_task = enqueue_task_fair,  
    .dequeue_task = dequeue_task_fair,  
    .pick_next_task = pick_next_task_fair,  
    .put_prev_task = put_prev_task_fair,  
    ...  
#ifdef CONFIG_SMP  
    .load_balance = load_balance_fair,  
    .move_one_task = move_one_task_fair,  
#endif  
    .set_curr_task = set_curr_task_fair,  
    ...  
};
```

```
const struct sched_class rt_sched_class = {  
    .next = &fair_sched_class,  
    .enqueue_task = enqueue_task_rt,  
    .dequeue_task = dequeue_task_rt,  
    .pick_next_task = pick_next_task_rt,  
    .put_prev_task = put_prev_task_rt,  
    ...  
#ifdef CONFIG_SMP  
    .load_balance = load_balance_rt,  
    .move_one_task = move_one_task_rt,  
#endif  
    .set_curr_task = set_curr_task_rt,  
    ...  
};
```