

2022년 2학기 운영체제실습 9주차

CPU Scheduling

System Software Laboratory

School of Computer and Information Engineering

Kwangwoon Univ.

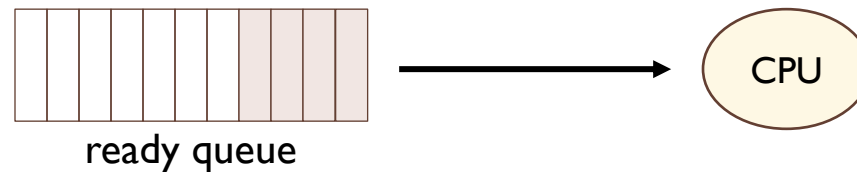
Contents

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

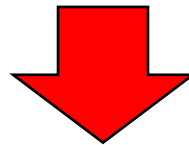
Scheduling

- **Scheduler**

- 한정된 자원을 다수의 client가 사용하려할 때
 - Ready queue에 client들이 대기



- 성능 향상을 위해 ready queue 내의 client들의 순서 조정 필요
 - Throughput, response time 등 ..



Scheduling

Scheduling

- **Scheduler**
 - OS에서의 Scheduling
 - CPU Scheduling
 - Task (process and thread)
 - Storage Scheduling
 - I/O requests
 - Network Scheduling
 - Packets

Linux Scheduling Policy (1/6)

■ Policy

- 스케줄러가 무엇을 언제 실행할 것인지를 정하는 동작
- 프로세서 시간의 사용을 최적화하는 책임이 있다

■ 목적

- 태스크 응답 시간을 빠르게 하는 것
- 시스템 사용률을 최대화 하는 것

■ 태스크 종류

- 프로세서 중심 태스크 (CPU-bound task)
- 입출력 중심 태스크 (I/O bound task)

Linux Scheduling Policy (2/6)

- **CPU-bound task vs I/O bound task**

- CPU의 사용 시간을 고려하여 분류
- CPU-bound
 - 대부분의 시간을 code를 실행하는 데 사용
 - 입출력 요청으로 중단되는 경우가 드물어 선점될 때까지 계속 실행된다
ex) compiler, video encoder
- I/O-bound
 - 대부분의 시간을 I/O 요청을 하고 기다리는 데 사용
 - 실제 실행시간은 아주 짧다.
ex) text editor

- **Linux에서는 I/O-bound task를 우선적으로 처리**

Linux Scheduling Policy (3/6)

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

Linux Scheduling Policy (4/6)

■ Time Slice

- 선점되기 전까지 작업을 얼마나 더 실행할 수 있는지 나타내는 값
- Time slice만큼 CPU를 사용 가능
 - 선점 가능
- Time slice를 너무 **길게** 잡으면
 - 시스템의 대화형 어플리케이션 성능 저하
- Time slice를 너무 **짧게** 잡으면
 - time slice를 소진한 프로세스를 다음 프로세스로 전환하는 데 빈번하게 시스템 시간 사용하게 됨. **context switching**
- Task의 priority에 따라 time slice를 배분
 - 높은 priority를 가진 task는 더 많은 time slice를 받는다

Linux Scheduling Policy (5/6)

- **Range of Priority**

- 2가지의 단위로 표현 가능
 - Nice, Real time priority(실시간 우선순위)

- **nice 값 (일반 task)**

- -20(highest) ~ +19(lowest)
- default value : 0

- **Real time priority (특수 task)**

- 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

Linux Scheduling Policy (6/6)

■ 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 교환

O(1) Scheduler (1/4)

■ O(1) Scheduler

- Linux kernel 2.6.22까지 사용하는 CPU Scheduler
 - time slice를 가지면서 가장 높은 priority를 가지는 task를 먼저 수행
 - Time slice 계산 과정에 상수시간 알고리즘 적용
 - 프로세서마다 별도의 실행대기 큐를 만듦
- 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 (2/4)

■ 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_PRIO]; /* priority queues */
};
```

O(1) Scheduler (3/4)

Time Slice 계산

- Priority array를 이용
- 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);
}
```

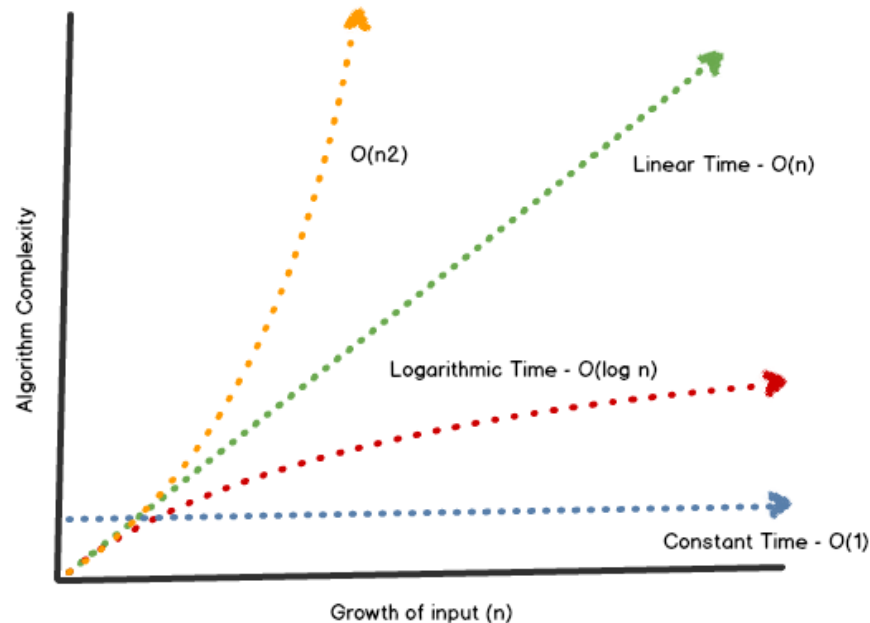
O(1) scheduler의 문제점

- 시스템 내 우선순위가 가장 높은 태스크를 항상 먼저 스케줄링
- 동작하는 태스크의 time slice가 만료되면, 해당 태스크의 time slice를 static하게 재계산한 후 expired list로 옮김
 - Task의 수, CPU의 상황 등을 고려하지 않고 절대적인 time slice값을 계산

O(1) Scheduler (4/4)

■ O(1) scheduler

- 장점 : 대화형(interactive) 프로세스가 없는 큰 서버 작업에는 이상적
- 단점 : 대화형 프로세스의 역할이 중요한 시스템에는 부적합



■ 2.6.23 커널 버전부터 CFS scheduler로 대체

CFS (Completely Fair Scheduler) (1/8)

■ 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 발생
 - expired array에 있는 task의 실행이 미뤄질 수 있음
- 우선 순위(priority)에 대한 가중치(weight)에 따라서 프로세스에 time slice 할당

CFS (Completely Fair Scheduler) (2/8)

■ CFS

- 가장 실행이 덜 된 프로세스를 다음에 실행할 프로세스로 선택
- Time slice 계산
 - weight(가중치)**에 기반한 time slice 계산
 - weight는 priority에 따라 할당
 - nice값이 높을 수록(우선순위가 낮을수록) 낮은 가중치 할당
 - nice값이 낮을 수록(우선순위가 높을수록) 높은 가중치 할당
 - 프로세스 실행 시간 = $\frac{\text{자신의 가중치}}{\text{전체 프로세스 가중치 총합}} \times (\text{기본 값})$
- priority가 낮더라도 좀 더 공평하게 time slice를 할당 받을 수 있음
- 각 프로세스에 할당하는 time slice의 최소 한계치가 존재
 - 기본값은 1ms
 - 최소 실행시간을 보장해 context switching 때문에 실행시간이 잠식되는 것을 방지

CFS (Completely Fair Scheduler) (3/8)

■ CFS

- 각 프로세스 별로 공정하게 할당된 몫만큼만 프로세서를 사용해야 한다
 - 프로세스의 실행시간을 기록해두어야 함
 - <linux/sched.h>에 정의된 struct sched_entity 를 이용해 정보 저장

```
struct sched_entity {  
    struct load_weight    load;           /* for load-balancing */  
    struct rb_node        run_node;  
    struct list_head      group_node;  
    unsigned int          on_rq;  
  
    u64                   exec_start;  
    u64                   sum_exec_runtime;  
    u64                   vruntime;  
    u64                   prev_sum_exec_runtime;  
  
    u64                   nr_migrations;  
};
```

CFS (Completely Fair Scheduler) (4/8)

■ Scheduler entity structure

- CFS는 관리중인 모든 프로세스에 대해 sched_entity라는 구조체 유지

- task_struct->sched_entity

```
struct task_struct {  
    int prio, static_prio, normal_prio;  
    unsigned int rt_priority;  
    const struct sched_class *sched_class;  
    struct sched_entity se;  
    struct sched_rt_entity rt;  
    ...  
};
```

- 이 구조체는 CFS 스케줄링 작업을 위한 여러 정보를 포함

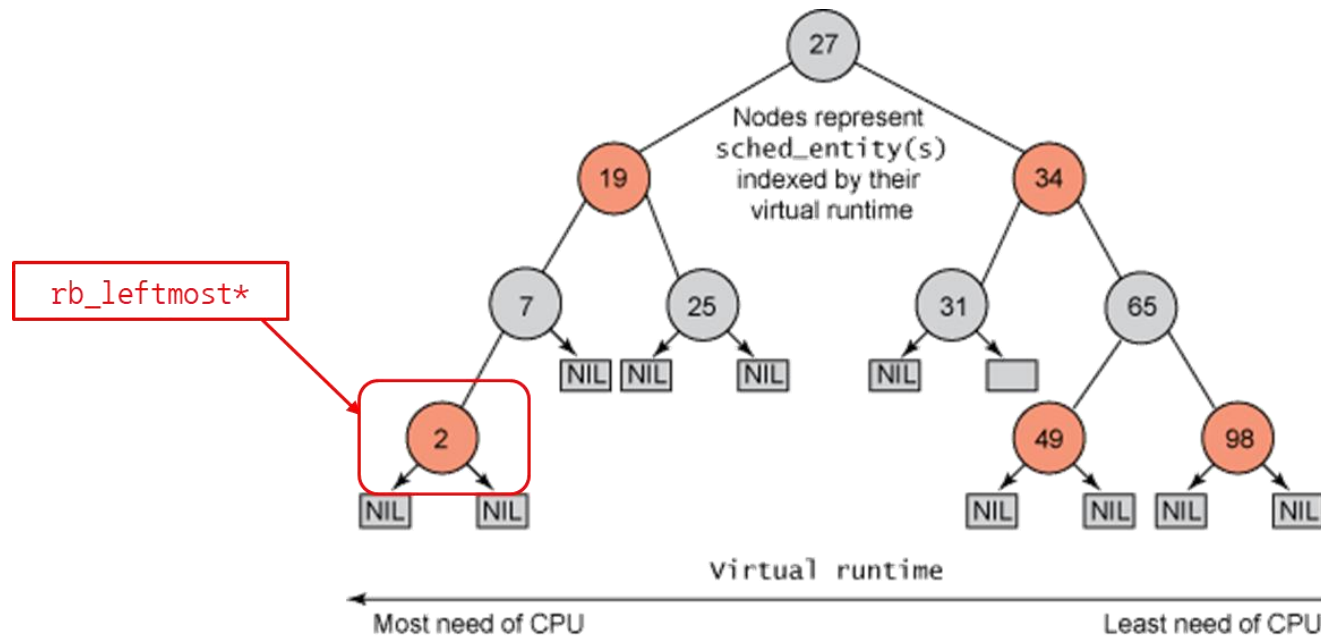
- task_struct->sched_entity.vruntime

```
struct sched_entity {  
    struct load_weight    load;           /* for load-balancing */  
    struct rb_node        run_node;  
    struct list_head      group_node;  
    unsigned int          on_rq;  
  
    u64                   exec_start;  
    u64                   sum_exec_runtime;  
    u64                   vruntime;  
    u64                   prev_sum_exec_runtime;  
  
    u64                   nr_migrations;  
};
```

CFS (Completely Fair Scheduler) (5/8)

■ CFS

- 프로세스들을 효율적으로 관리하기 위해 Red-Black tree 사용
 - 다음에 실행하고자 하는 프로세스를 선택하기 쉽다
 - rb_leftmost 포인터를 유지 -> 굳이 tree 를 순회하지 않아도 되도록 최적화 됨
 - virtual runtime이 제일 작은 프로세스에 해당하는 포인터
 - tree의 가장 왼쪽에 있는 node에 해당하는 프로세스
- $O(\log N)$ 으로 $O(1)$ 에 비해 느리지만 성능 상의 큰 차이는 없음



CFS (Completely Fair Scheduler) (6/8)

■ Virtual runtime

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

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

```
static inline void __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
unsigned long delta_exec)
{
    ...
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    ...
}
```

CFS (Completely Fair Scheduler) (7/8)

■ Adding a process to the tree

- 프로세스를 Run queue에 등록 시,
 - enqueue_entity() 함수 호출.
 - 몇 가지 사전작업 후, __enqueue_entity() 함수 호출.
 - rbtree에 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_cached(&se->run_node, &cfs_rq->tasks_timeline,
leftmost);
}
```

rb_link_node : 새로 추가할 프로세스를 자식 node로 만든다.

rb_insert_color : tree의 자가 균형 속성 유지하게 한다.

CFS (Completely Fair Scheduler) (8/8)

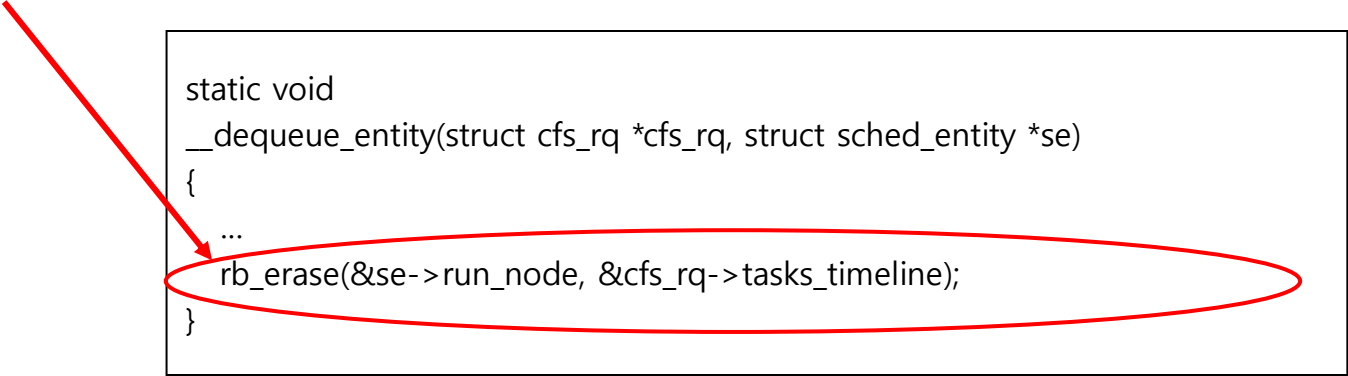
■ Removing a process to the tree

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

static void

deque_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)

```
{  
    ...  
    __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);  
}
```

rb_erase : tree에서 프로세스 제거

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,  
    ...  
};
```

Assignment 3

- 제출 기한: 2022. 10.06(목) ~ 2022.11.10(목) 23:59:59
- Delay 없음
- 업로드 양식에 어긋날 경우 감점 처리
- Hardcopy 제출하지 않음