

과제 #2-1
-리눅스 CPU 스케줄링 구조
수정

학번 : 201711032


이름 : 고영훈

커널 수정 내용

①fair와 idle 클래스 사이에 새로운 스케줄링 클래스 추가 (mysched.c)

```
static int __init init_mysched(void)
{
    const struct sched_class *class;


    CONFIG_MYSCHED = 1;
    classpointer = &my_sched_class;
    fair_sched_class.next = &my_sched_class;
```



기존의 커널의 스케줄링 매커니즘을 건드리지 않고 모듈이 삽입 됐을 때만 나의 스케줄링 클래스가 동작하게 하기 위해서 fair의 next에 나의 스케줄링 클래스를 넣어주었다.

```
static void __exit exit_mysched(void)
{
    const struct sched_class *class;

    CONFIG_MYSCHED = 0;
    classpointer = NULL;
    fair_sched_class.next = &idle_sched_class;
```



마찬가지로 모듈이 빠져 나왔을 때 fair 클래스의 next를 idle로 원상복구 시켜준다.
또한 아래와 같이 const 키워드를 제거해서 .next의 값을 수정 가능하도록 했다.

(sched/fair.c)

```
struct sched_class fair_sched_class = {  
    .next = &idle_sched_class,  
    .enqueue_task = enqueue_task_fair,  
    .dequeue_task = dequeue_task_fair,  
    .yield_task = yield_task_fair,  
    .yield_to_task = yield_to_task_fair,  
    .check_preempt_curr = check_preempt_wakeup,  
    .pick_next_task = pick_next_task_fair,  
    .put_prev_task = put_prev_task_fair,
```

②LKM이 load 되었을 때만 스케줄러가 작동하기 위한 방법
(sched/core.c)

```
struct sched_class* classpointer = NULL;
int CONFIG_MYSCHED;
EXPORT_SYMBOL(CONFIG_MYSCHED);
EXPORT_SYMBOL(classpointer);
```

위와 같이 core.c의 상단부분에 위와 같은 코드를 작성했다. 그리고 LKM 모듈에서 사용 할 수 있게끔 EXPORT_SYMBOL 해주었다.

위 두 개의 변수는 아래와 같이 사용된다.

```
static void __setscheduler(struct rq *rq, struct task_struct *p,
                           const struct sched_attr *attr, bool keep_boost)
{
    printk(KERN_DEBUG " __setscheduler call");
    __setscheduler_params(p, attr);

    /*
     * Keep a potential priority boosting if called from
     * sched_setscheduler().
     */
    p->prio = normal_prio(p);
    if (keep_boost)
        p->prio = rt_effective_prio(p, p->prio);

    if (dl_prio(p->prio))
        p->sched_class = &dl_sched_class;
    else if (rt_prio(p->prio))
        p->sched_class = &rt_sched_class;
    else
        p->sched_class = &fair_sched_class;
    if(p->rt_priority==0 && CONFIG_MYSCHED==1)//수정 ←
        p->sched_class = classpointer;
}
```

(sched/core.c)

__setscheduler는 스케줄러의 우선순위에 따라 task_struct, 즉 PCB의 스케줄러를 등록하는 함수이다. CONFIG_MYSCHED는 전역변수로 선언했기 때문에 0으로 초기화 된다.

```
static int __init init_mysched(void)
{
    const struct sched_class *class;

    CONFIG_MYSCHED = 1; ←
    classpointer = &my_sched_class;
    fair_sched_class.next = &my_sched_class;
```

(mysched.c)

모듈 코드에서 CONFIG_MYSCHED값을 1로 바꿔주어 모듈이 들어온 걸 커널이 알 수 있게 해준다. 또한 유저 프로그램 cpu.c에서 priority를 0으로 설정했기 때문에 priority가 0인지 구분할 수 있게 한다.

```
static void __exit exit_mysched(void)
{
    const struct sched_class *class;

    CONFIG_MYSCHED = 0;
    classpointer = NULL;
    fair_sched_class.next = &idle_sched_class;
}
```

모듈 exit시 CONFIG_MYSCHED값을 0으로 바꿔주어 모듈이 나갔음을 알린다.

③사용자가 변경 요청한 클래스의 validity를 검사하는 코드

```
static inline int myclass_policy(int policy)
{
    return policy == 7;
}

static inline bool valid_policy(int policy)
{
    return idle_policy(policy) || fair_policy(policy) ||
           rt_policy(policy) || dl_policy(policy) || myclass_policy(policy);
}
```

my_sched_class가 valid 검사를 실시할 때 myclass도 검사를 실시한다.
추가하려는 MYCLASS의 policy는 7번으로 설정해두었다.

```
#define SCHED_NORMAL 0
#define SCHED_FIFO 1
#define SCHED_RR 2
#define SCHED_BATCH 3
/* SCHED_ISO: reserved but not implemented */
#define SCHED_IDLE 5
#define SCHED_DEADLINE 6
#define SCHED_MYCLASS 7
```

dmesg 결과

```
[786144.320606] sched_setattr syscall
[786144.320610] sched_setattr call
[786144.320611] __sched_setscheduler call
[786144.320617] __setscheduler call
[786144.320617] __setscheduler_params call
[786144.320635] MYMOD: enqueue_task_fifo CALLED task = 00000000a360bc78
[786144.320637] MYMOD: set_curr_task_fifo CALLED
[786144.320637] check_class_changed call
[786144.320640] MYMOD: switched_to_fifo CALLED new = 00000000a360bc78
[786148.324217] MYMOD: dequeue_task_fifo CALLED
[786148.324220] MYMOD: enqueue_task_fifo CALLED task = 00000000a360bc78
[786148.324221] MYMOD: set_curr_task_fifo CALLED
[786148.324239] MYMOD: dequeue_task_fifo CALLED
root@os201711032-2:/proj#
```

./cpu 2 2 수행 시 총 2개의 프로세스가 enqueue, dequeue 되는 것을 확인 할 수 있다.

과제 #2-2.

스케줄링 알고리즘 구현 &
Context switching overhead
분석

스케줄링 알고리즘의 구현

①FIFO(fifo.c)

FIFO 알고리즘을 구현하기 위해 list_head 자료구조를 사용한다.

list_head 자료구조는 리눅스 커널에서 제공하는 자료구조로 양방향, 순환 연결리스트가 가능하다.

자료구조는 <linux/list.h>에 구현되어있다.

첫 번째로, 연결리스트의 head를 설정하기 위해 전역변수로 아래와 같이 선언했다.

```
#include <linux/list.h>
#include <linux/slab.h>

static LIST_HEAD(my_list); //head 설정
```

그 다음으로 rt.c를 참고하여 list_head를 어떻게 사용하는지 알아보았다.

```
483 struct sched_rt_entity {
484     struct list_head run_list;
485     unsigned long timeout;
486     unsigned long watchdog_stamp;
487     unsigned int time_slice;
488     unsigned short on_rq;
489     unsigned short on_list;
490
491     struct sched_rt_entity *back;
492 #ifdef CONFIG_RT_GROUP_SCHED
493     struct sched_rt_entity *parent;
494     /* rq on which this entity is (to be) queued: */
495     struct rt_rq *rt_rq;
496     /* rq "owned" by this entity/group: */
497     struct rt_rq *my_rq;
498 #endif
499 } __randomize_layout;
```

(sched/rt.c)

(492~498행)에서는 그룹 스케줄링을 사용할 때 사용되는 자료구조들이다.

내가 만들려고 하는 FIFO는 그룹 스케줄링을 사용하지 않으므로 참고하지 않았고 entity 안에 list_head로 태스크가 스케줄링 클래스에 enqueue될 때 태스크를 동적할당 해서 list_head를 활용해 모든 entity를 양방향으로 연결하면 된다고 생각했다. entity는 태스크를 감싸는 컨테이너라고 보면 될 것 같다.

그래서 fifo.c에 아래와 같이 sched_my_entity를 구현했다.

```
struct sched_my_entity // 런 큐에 enqueue할 자료구조
{
    struct task_struct *ts;
    struct list_head run_list;
};
```

sched_my_entity에는 프로세스의 정보를 담고있는 task_struct 포인터를 넣어주었고 당연히 list_head를 넣어주었다.

사실 task_struct 포인터를 넣는게 맞는지 잘 모르겠다. 왜 넣고자 했는지 이유를 설명하자면 아래와 같이 task_struct의 구현 코드에 sched_rt_entity가 존재한다.

```
632 struct task_struct {
633     #ifdef CONFIG_THREAD_INFO_IN_TASK
634         /*
635          * For reasons of header soup (see current_thread_info()), this
636          * must be the first element of task_struct.
637
638         const struct sched_class *sched_class;
639         struct sched_entity *se;
640         struct sched_rt_entity rt; ←
641     #ifdef CONFIG_CGROUP_SCHED
```

그러면 rt 스케줄러의 pick_next_task()는 어떻게 entity에 따른 task를 찾아서 반환하는지 알아보았다.

```
113 static inline struct task_struct *rt_task_of(struct sched_rt_entity *rt_se)
114 {
115     #ifdef CONFIG_SCHED_DEBUG
116         WARN_ON_ONCE(!rt_entity_is_task(rt_se));
117     #endif
118     return container_of(rt_se, struct task_struct, rt); ←
119 }
```

(sched/rt.c)

pick_next_task_rt함수에서 위의 rt_task_of 함수를 호출해서 task를 반환한다.

보면 container_of()를 사용하는데 container_of()는 구조체의 멤버변수의 주소값을 알고 있으면 그 구조체의 주소를 얻어 올 수 있다. 즉 entity의 주소값을 알고 있으면 task_struct를 얻을 수 있다. 왜냐하면 task_struct에는 sched_rt_entity 변수가 존재하기 때문이다.

내가 만든 FIFO스케줄러에서 pick_next_task()를 만났을 때 위의 매커니즘대로 구현하려면 task_struct에 sched_my_entity를 따로 추가시켜야 한다고 생각했다. 하지만 그에 따른 시간적 비용과 리눅스의 프로세스 갯수만큼 task가 존재하기 때문에 internal fragmentation이 심화될 수 있다고 생각한다.

그래서 sched_my_entity에 task_struct를 담았다.

enqueue_task_myfifo(fifo.c)

```
static void enqueue_task_myfifo(struct rq *rq, struct task_struct *p, int flags)
{
    struct sched_my_entity *my_se;
    my_se = kmalloc(sizeof(my_se), GFP_KERNEL);
    my_se->ts = p; //entity에 task_struct 정보 저장 할 수 있게끔
    enqueue_my_entity(my_se); //entity enqueue
    printk(KERN_INFO "MYMOD: enqueue_task_fifo CALLED task = %p\n", p);
}
```

enqueue_task_myfifo는 위와 같이 작성했다.

task가 enqueue 되면 sched_my_entity를 kmalloc을 이용해 동적할당 한다.

그 다음으로 my_se의 task_struct에 task의 정보를 저장한다.

enqueue_my_entity에서는 아래와 같이 리스트의 꼬리에 entity의 리스트를 저장한다.

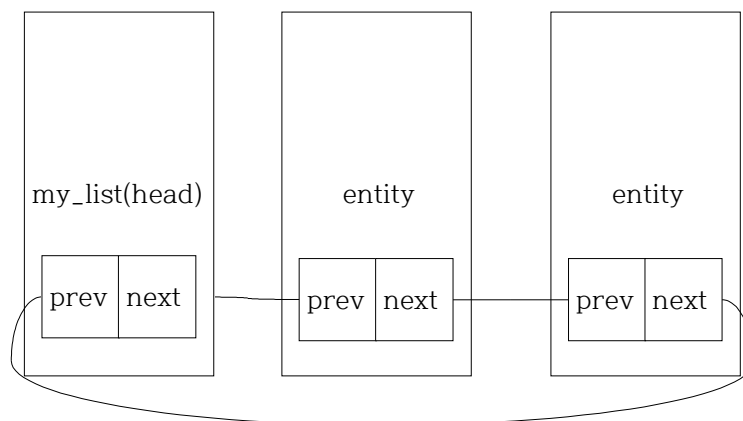
```
static void enqueue_my_entity(struct sched_my_entity *my_se)
{
    list_add_tail(&my_se->run_list, &my_list);
}
```

pick_next_task_myfifo(fifo.c)

```
static struct task_struct *pick_next_task_myfifo(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    struct sched_my_entity *my_se;
    my_se = list_entry(my_list.next, struct sched_my_entity, run_list);
    struct task_struct *task = my_se->ts;
    return task;
}
```

반환할 태스크를 선택할 때 위의 화살표 부분처럼 head list의 next를 선택해서 task를 반환한다.

그림으로 보면



my_list(head)의 next를 참조해서 바로 다음 entity를 알아낼 수 있다.

이렇게 다음 entity를 알아낸 후 entity에 저장되어있는 task_struct의 정보를 반환하면 된다.

dequeue_task_myfifo(fifo.c)

```
static void dequeue_task_myfifo(struct rq *rq, struct task_struct *p, int flags)
{
    dequeue_my_entity();
}
```

dequeue_task_myfifo는 dequeue_my_entity를 호출한다.

dequeue_my_entity

```
static void dequeue_my_entity(void)
{
    kfree(my_list.next);
    list_del(my_list.next);
}
```

head의 next의 동적할당을 해제하고 list에서 지워준다.

이렇게 FIFO 알고리즘을 구현했다.

./cpu 2 2 수행 시

```
root@os201711032-2:/proj# ./cpu 2 2
Creating Process: #0
Creating Process: #1
PROCESS #01 count = 125 100
PROCESS #01 count = 265 100
PROCESS #01 count = 422 100
PROCESS #01 count = 580 100
PROCESS #01 count = 734 100
PROCESS #01 count = 893 100
PROCESS #01 count = 1047 100
PROCESS #01 count = 1209 100
PROCESS #01 count = 1369 100
PROCESS #01 count = 1511 100
PROCESS #01 count = 1607 100
PROCESS #01 count = 1704 100
PROCESS #01 count = 1867 100
PROCESS #01 count = 2026 100
PROCESS #01 count = 2185 100
PROCESS #01 count = 2344 100
PROCESS #01 count = 2500 100
PROCESS #01 count = 2658 100
PROCESS #01 count = 2817 100
PROCESS #01 count = 2970 100
DONE!! PROCESS #01 : 002970 2000
```

1번 프로세스가 실행하지만 0번 프로세스가 실행하지 못한 상태로 있다가 커널이 꺼지는 현상이 발생한다. pick_next_task 혹은 enqueue, dequeue하는 코드를 잘못 짰 것 같다.

②Round robin, ③Weighted round robin의 경우 시간이 부족하기도 했고, FIFO를 구현하려고 시도한 게 생각보다 시간이 오래 걸렸다. 거기다 FIFO가 제대로 동작하지 않아 다른 알고리즘을 구현해보지 못했다.

Context switching overhead 분석을 실험을 통해 보이지는 못했지만 어느 정도 추측해보자면

①FIFO의 경우

NO-SMP 환경에서 Context switching이 일어나지 않기 때문에 context switching overhead가 발생하지 않을 것이다.

②Round robin의 경우

```
root@hcpark:~/proj2# ./cpu 2 1
** START: Processes = 02 Time = 01 s
Creating Process: #0
PROCESS #01 count = 0112 100 ms
PROCESS #01 count = 0243 100 ms
PROCESS #00 count = 0001 201 ms
PROCESS #00 count = 0133 100 ms
PROCESS #01 count = 0252 202 ms
PROCESS #01 count = 0414 100 ms
PROCESS #00 count = 0259 295 ms
PROCESS #00 count = 0421 100 ms
PROCESS #01 count = 0574 298 ms
PROCESS #01 count = 0737 100 ms
PROCESS #00 count = 0582 299 ms
PROCESS #00 count = 0744 100 ms
DONE!! PROCESS #00 : 000744 1097 ms
PROCESS #01 count = 0899 200 ms
DONE!! PROCESS #01 : 000899 1104 ms
```

time slice 200ms

```
root@hcpark:~/proj2# ./cpu 2 1
** START: Processes = 02 Time = 01 s
Creating Process: #0
PROCESS #01 count = 0082 100 ms
PROCESS #01 count = 0173 100 ms
PROCESS #01 count = 0264 100 ms
PROCESS #01 count = 0355 100 ms
PROCESS #00 count = 0001 498 ms
PROCESS #00 count = 0110 100 ms
PROCESS #00 count = 0214 105 ms
PROCESS #00 count = 0324 100 ms
PROCESS #00 count = 0435 100 ms
PROCESS #01 count = 0465 600 ms
DONE!! PROCESS #01 : 000465 1003 ms
root@hcpark:~/proj2# PROCESS #00 count = 0542 100 ms
DONE!! PROCESS #00 : 000542 1005 ms
```

time slice 500ms

직접 Round robin을 구현하지 않아 교수님께서 주신 pdf파일의 결과를 보면 time slice를 200ms로 설정하고 수행 시 PROCESS #01이 200ms만큼 수행 할 동안 PROCESS #0은 running 상태가 되길 기다리고 있다가 PROCESS #1을 preemption 시키고 수행한다. 이 때 200ms동안 cpu연산은 하지 못했지만 time slice가 증가하고 있었고 time slice를 500ms로 설정할 때도 동일하게 preemption된다.

time slice 200ms의 경우 context switching이 5번 일어났고 500ms의 경우 2번 일어났다. time slice(=time quantum)을 작게 설정할수록 라운드가 더 잘게 쪼개지기 때문에 context switching이 더 많이 일어난다. 수행 시간을 봐도 200ms, 500ms 비교 했을 때 수행 시간에 확연한 차이가 나는 걸 볼 수 있다.

③WRR: priority-based proportional sharing

```
root@hcpark:~/proj2# ./cpu 5 5
** START: Processes = 05 Time = 05 s
    Creating Process: #3
    Creating Process: #2
    Creating Process: #1
    Creating Process: #0
DONE!! PROCESS #02 : 001076 5024 ms
PROCESS #01 count = 0691 271 ms
DONE!! PROCESS #01 : 000691 5028 ms
PROCESS #03 count = 1491 103 ms
DONE!! PROCESS #03 : 001491 5079 ms
PROCESS #00 count = 0344 387 ms
DONE!! PROCESS #00 : 000344 5086 ms
PROCESS #04 count = 1896 100 ms
DONE!! PROCESS #04 : 001896 5100 ms
root@hcpark:~/proj2#
```

proportional sharing을 하기 때문에 먼저 생성된 프로세스 순서대로 실행이 되며
preemption 없이 수행하기 때문에 context switching이 FIFO와 같이 나타나지 않음.

특히 어려웠던 점 해결 방법

① 2-2의 과제를 처음 봤을 때 어떤 방식으로 알고리즘을 만들어야하는지 잘 이해를 하지 못했다.

RT스케줄러도 FIFO와 Round robin 알고리즘이 사용되는데 rt.c라는 파일만 존재할 뿐이지 알고리즘이 구현된 파일은 찾지 못했다. 지금도 정확하게 rt 스케줄러가 어떤 방식으로 알고리즘을 제공하는지 잘 모르겠다. 처음엔 SCHED_MYFIFO, SCHED_MYRR, 스케줄링 클래스 번호를 등록하고 lkm 모듈 안에서 모든 알고리즘이 동작하게 하려고 했지만 유지 보수에는 좋겠지만, context switching overhead를 측정하는 목표이기 때문에 실험 할 때마다 일일이

```
const struct sched_class my_sched_class = {  
    .next = &idle_sched_class,  
  
    .enqueue_task = enqueue_task_fifo,  
    .dequeue_task = dequeue_task_fifo,  
    .yield_task = yield_task_fifo,  
  
    .check_preempt_curr = check_preempt_curr_fifo,  
  
    .pick_next_task = pick_next_task_fifo,  
    .put_prev_task = put_prev_task_fifo,  
  
    .set_curr_task = set_curr_task_fifo,  
    .task_tick = task_tick_fifo,  
  
    .get_rr_interval = get_rr_interval_fifo,  
  
    .prio_changed = prio_changed_fifo,  
    .switched_to = switched_to_fifo,  
    .update_curr = update_curr_fifo,  
};
```

수정해서 실험하는 방법을 선택했다.

②2-1에서 모듈과 연계해서 커널 코드를 수정하는 중에 굳이 수정 안해도 되는 부분들을 수정해서 커널이 올바르게 작동하지 않는 현상이 계속 발생했다. pick_next_task 코드를 보면 Optimization으로 fair 클래스 먼저 검사를 실시하는데 fair가 아니면 그 다음은 my_sched로

```
/* Assumes fair_sched_class->next == idle_sched_class */  
if (unlikely(!p))  
    p = idle_sched_class.pick_next_task(rq, prev, rf);
```

동작해야하기 때문에

/* Assumes fair_sched_class->next == idle_sched_class */ 이 문장에 현혹돼서 저 코드를 수정하려고 시도했다. 역시나 커널을 컴파일하고 다시 시작했을 때 커널이 정상 작동하지 않았다. unlikely(!p)의 경우 fair를 기준으로 fair가 아닐 경우 idle로 수행하겠다는 것이지 my_sched_class가 들어왔다고 해서 그 다음 class로 수행하겠다는 의미가 아닌 것이다. 이 코드 뿐만 아니라, 다른 커널 영역에도 코드를 꽤 넣었는데 줄이고 줄이다보니 모듈과 연계하여 커널이 잘 돌아가게 되었다.

감사합니다.