Assignment 2

201610674 이영훈

2-1) LKM을 사용하며 스케줄링 구조 학습하자.

주어진 mysched.c을 LKM으로 적재하여 fair 와 idle 스케줄러 사이에 끼어 넣도록 커널 프로그래밍을 한다.

Kernel에서 fair와 idle을 연결해주기 위해서 커널 코드를 먼저 수정해주었다.

Core.c mysched_used -> mysched모듈이 적재되었는지 확인하는 변수 1이면 적재 0이면 적재 x

set_class_my 함수 포인터로 mysched에서 myclass 함수를 적재하여 사용했다.

밑에는 idle fair rt를 EXPORT_SYMBOL을 통해서 보냈다. 또한 dl과 stop스케쥴러도 보내기 위해서 각각 stop_task.c와 deadline.c에서도 보냈다. 캡처는 하지 않았지만 소스로 존재한다. Core.c에서 __setscheduler함수에서 policy을 지정해주기 때문에 이 부분에서 모듈이 적재되어 있으며, 정책이 SCHED_MY로 잡혀 있을 때 함수포인터로 연결된 set_class_my로 잡아서 해당 함수를 부르게 된다.

```
int mysched_used;
void (*set_class_my) (struct task_struct *p);
EXPORT_SYMBOL(set_class_my);
EXPORT_SYMBOL(mysched_used);

EXPORT_SYMBOL(idle_sched_class);
EXPORT_SYMBOL(fair_sched_class);
EXPORT_SYMBOL(rt_sched_class);
```

core.c

```
void myclass (struct task_struct *p) {
    printk(KERN_INFO "MYMOD: myclass CALLED\n");
    p->sched_class = &my_sched_class;
}
```

mysched.c

```
static void __setscheduler(struct rq *rq, struct task_struct *p,
                           const struct sched_attr *attr, bool keep_boost)
{
    printk("__setscheduler() call!");
    __setscheduler_params(p, attr);
    if((mysched_used == 1) && (p->policy == SCHED_MY))
    {
        set_class_my(p);
        return;
    }
    /*
     * 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;
}
```

core.c

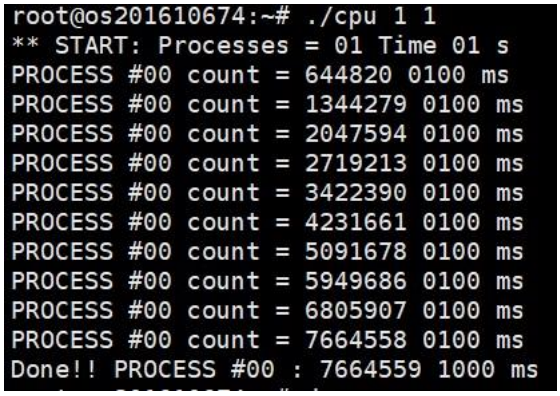
그 뒤로는 fair.c와 sched.h에서 코드를 수정해주었다.

Fair.c에서는 fair_sched_class에서 next을 수정해주기 위해서 const을 지워주기 위해서 선언문과 정의문의 const을 지워주며, sched.h에서 extern해서 받아온 변수도 const을 지워주었다. 또한 sched.h에서 valid_policy() 정책이 올바른가 확인해주는 함수에 mysched도 검사해주도록 넣어주었다. 또한 valid_policy함수에 들어가는 my_policy을 작성해주었다. 또한 그 곳에서 필요한 상수를 위해서 /include/uapi/linux/sched.h에 #define SCHED_MY 7 값으로 넣어줬다. 그리고 mysched.c을 모듈이 적재되었을 때 적재가 해제되었을 때 필요한 것들을 정의했다.

```
static int __init init_mysched(void)
{
    const struct sched_class *class;
    mysched_used = 1;
    fair_sched_class.next = &my_sched_class;
    task = NULL;
    set_class_my = &myclass;
    for_each_class(class)
        printk(KERN_INFO "INIT_MOD: class = %p\n", class);
    return 0;

}

static void __exit exit_mysched(void)
{
    const struct sched_class *class;
    mysched_used = 0;
    task = NULL;
    fair_sched_class.next = &idle_sched_class;
    for_each_class(class)
        printk(KERN_INFO "EXIT_MOD: class = %p\n", class);

    return;

}
```

마지막으로 cpu.c에서 policy을 mysched으로 바꿔주고 우선순위를 0으로 바꾸어서 실행하였다. 이때 나온 결과가 밑과 같다.

```
[ 1102.776938] mysched: loading out-of-tree module taints kernel.
[ 1102.777237] mysched: module verification failed: signature and/or required key missing - tainting kernel
[ 1102.781463] INIT_MOD: class = 000000001a7ced19
[ 1102.781464] INIT_MOD: class = 00000000386862dd
[ 1102.781464] INIT_MOD: class = 00000000268c41d4
[ 1102.781465] INIT_MOD: class = 00000000b168ae6f
[ 1102.781465] INIT_MOD: class = 00000000d65471e0
[ 1122.968829] sys_sched_setattr() call!
[ 1122.968833] sched_setattr() call!
[ 1122.968833] __sched_setscheduler() call!
[ 1122.968837] __setscheduler() call!
[ 1122.968838] __setscheduler_params() call!
[ 1122.968839] MYMOD: myclass CALLED
[ 1122.968842] MYMOD: enqueue_task_fifo CALLED task = 0000000016593409
[ 1122.968842] MYMOD: set_curr_task_fifo CALLED
[ 1122.968843] MYMOD: switched_to_fifo CALLED new = 0000000016593409
[ 1123.969565] MYMOD: dequeue_task_fifo CALLED
[ 1123.969567] MYMOD: enqueue_task_fifo CALLED task = 0000000016593409
[ 1123.969567] MYMOD: set_curr_task_fifo CALLED
[ 1123.969572] MYMOD: dequeue_task_fifo CALLED
```



해당 스케줄러에 태스크가 들어갔다 나오는 걸 보아 성공했다는 것을 알 수 있다.

2-2) MYSCHED 클래스를 FIFO와 Round Robin, Weight Round Robin을 구현해야 한다.

먼저 FIFO의 경우 런큐에 테스크가 들어간 순서대로 pick_next_task함수에서 뽑혀서 실행되는게 핵심이다. 그래서 mysched 구조체에서 enqueue와 dequeue, pick_next_task함수를 만들어서 구현했다. 핵심으로는 list_head를 만들어줘서 런큐처럼 이용했다. List_head 포인터를 하나 kmalloc으로 동적할당해준 뒤 task마다의 rt_entity에 들어있는 list_head type의 run_list을 연결하여 런큐를 만들었다. List_head을 enqueue에서 tail로 연결하고 dequeue에서는 del_init을 해준다. 그리고 list_head 크기를 가늠하기 위해서 list_num변수를 둬서 enqueue와 dequeue때 숫자를 1을 더하거나 빼서 list_head의 크기를 유지했다. 크기가 1미만일 때는 pick_next_task가 돌아가지 않게 했다. 그 다음 pick_next_task함수에서 현재 my_list의 next의 rt_entitiy을 찾고 그 entity을 싸고 있는 task_struct을 찾아서 반환해준다. Rt_entity을 찾을 때는 list_entry함수를 사용하고 task_struct을 찾을 때는 container_of을 사용했다. 이렇게 하고 cpu.c을 fork이후 스케쥴러를 설정하게 한 후 돌리니 FIFO로 스케쥴러가 잡혔으나, 어떤 이유인지 모르게 2번째 프로세스가 조금 실행되고 꺼지게 되어 나오게 되면 sleep을 둬서 FIFO시에 정확히 나오도록 했다. List_head타입의 run_list ⊂ rt_entity, entity ⊂ Task_struct을 이용해서 구현했다.

설명한 FIFO 구현할 때 수정한 함수

```
static int __init init_mysched(void)
{
    const struct sched_class *class;
    my_list = kmalloc(sizeof(struct list_head), GFP_KERNEL);
    INIT_LIST_HEAD(my_list);
    mysched_used = 1;
    fair_sched_class.next = &my_sched_class;
    task = NULL;
    set_class_my = &myclass;
    for_each_class(class)
        printk(KERN_INFO "INIT_MOD: class = %p\n", class);
    return 0;

}
```

모듈이 적재될 때 my_list 를 kmalloc 해주고 INIT_LIST_HEAD 로 초기화시켜주었다.

```
static struct task_struct *
pick_next_task_fifo(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    //carefully use printk() in this function !!
    //printk(KERN_DEBUG "MYMOD: pick_next_task_fifo CALLED\n");
    if(list_num < 1) return NULL;
    struct sched_rt_entity *_next = NULL;
    struct task_struct *p;
    put_prev_task(rq, prev);
    _next = list_entry(my_list->next,struct sched_rt_entity,run_list);
    if(_next == NULL) return NULL;

    p = container_of(_next, struct task_struct, rt);
    p->se.exec_start = rq_clock_task(rq);
    //printk(KERN_DEBUG "MYMOD: pick_next_task_fifo return task p\n");
    return p;
}

static void
enqueue_task_fifo(struct rq *rq, struct task_struct *p, int flags)
{
   struct sched_rt_entity *rt_entity = &p->rt;
   printk(KERN_INFO "MYMOD: enqueue_task_fifo CALLED task = %p\n", p);
   list_add_tail(&rt_entity->run_list, my_list);
    list_num++;
   add_nr_running(rq, 1);

   task = p;
}

static void
dequeue_task_fifo(struct rq *rq, struct task_struct *p, int flags)
{
   printk(KERN_INFO "MYMOD: dequeue_task_fifo CALLED\n");


   list_del_init(&p->rt.run_list);
    list_num--;
   sub_nr_running(rq, 1);
   //task = NULL;
}
```

그 후 설명한 내용과 같이 pick_next_fifo 에서 rt_entity 을 list_entry 로 찾고 그 rt_entity 로 해당 task_struct 을 찾아서 리턴해준다.

enqueue 에서는 my_list 의 tail 에 추가해주며 enqueue 시에는 list_num 을 늘려준다.

dequeue 에서는 my_list 에서 버리고 list_num 을 줄여준다.

FIFO 실행화면

```
** START: Processes = 03 Time 01 s
Creating Process: #0
Creating Process: #1
PROCESS #00 count = 650111 0100 ms
PROCESS #00 count = 1353853 0100 ms
PROCESS #00 count = 2055215 0100 ms
PROCESS #00 count = 2822210 0100 ms
PROCESS #00 count = 3683792 0100 ms
PROCESS #00 count = 4523597 0100 ms
PROCESS #00 count = 5370869 0100 ms
PROCESS #00 count = 6235813 0100 ms
PROCESS #00 count = 7093113 0100 ms
PROCESS #00 count = 7950626 0100 ms
Done!! PROCESS #00 : 7950627 1000 ms
PROCESS #01 count = 821266 0100 ms
PROCESS #01 count = 1678298 0100 ms
PROCESS #01 count = 2512188 0100 ms
PROCESS #01 count = 3366720 0100 ms
PROCESS #01 count = 4217347 0100 ms
PROCESS #01 count = 5072503 0100 ms
PROCESS #01 count = 5919750 0100 ms
PROCESS #01 count = 6754864 0100 ms
PROCESS #01 count = 7609521 0100 ms
PROCESS #01 count = 8464085 0100 ms
Done!! PROCESS #01 : 8464086 1000 ms
PROCESS #02 count = 856677 0100 ms
PROCESS #02 count = 1714138 0100 ms
PROCESS #02 count = 2558146 0100 ms
PROCESS #02 count = 3414620 0100 ms
PROCESS #02 count = 4271255 0100 ms
PROCESS #02 count = 5122458 0100 ms
PROCESS #02 count = 5976061 0100 ms
PROCESS #02 count = 6835419 0100 ms
PROCESS #02 count = 7698343 0100 ms
PROCESS #02 count = 8560062 0100 ms
Done!! PROCESS #02 : 8560063 1000 ms
```

```
[38344.569292] MYMOD: myclass CALLED
[38344.569292] MYMOD: enqueue_task_fifo CALLED task = 00000000f231c0f3
[38344.569293] MYMOD: set_curr_task_fifo CALLED
[38344.569294] MYMOD: switched_to_fifo CALLED new = 00000000f231c0f3
[38344.569297] MYMOD: dequeue_task_fifo CALLED
[38345.569308] MYMOD: enqueue_task_fifo CALLED task = 00000000a35ab28b
[38345.569309] MYMOD: check_preempt_curr_fifo CALLED
[38345.569450] MYMOD: dequeue_task_fifo CALLED
[38346.569362] MYMOD: enqueue_task_fifo CALLED task = 00000000f231c0f3
[38346.569364] MYMOD: check_preempt_curr_fifo CALLED
[38346.569613] MYMOD: dequeue_task_fifo CALLED
[38347.570389] MYMOD: dequeue_task_fifo CALLED
[38347.570392] MYMOD: enqueue_task_fifo CALLED task = 00000000f231c0f3
[38347.570392] MYMOD: set_curr_task_fifo CALLED
[38347.570395] MYMOD: enqueue_task_fifo CALLED task = 00000000a35ab28b
[38347.570396] MYMOD: check_preempt_curr_fifo CALLED
[38347.570398] MYMOD: dequeue_task_fifo CALLED
[38347.570454] MYMOD: dequeue_task_fifo CALLED
[38347.570454] MYMOD: enqueue_task_fifo CALLED task = 00000000a35ab28b
[38347.570454] MYMOD: set_curr_task_fifo CALLED
[38347.570455] MYMOD: enqueue_task_fifo CALLED task = 000000001583e537
[38347.570456] MYMOD: check_preempt_curr_fifo CALLED
[38347.570456] MYMOD: dequeue_task_fifo CALLED
[38347.570510] MYMOD: dequeue_task_fifo CALLED
[38347.570510] MYMOD: enqueue_task_fifo CALLED task = 000000001583e537
[38347.570510] MYMOD: set_curr_task_fifo CALLED
[38347.570513] MYMOD: dequeue_task_fifo CALLED
```

Round Robin은 타임슬라이스마다 context switch하는 방식대로 생각을 해 보았다. 주어진 task_tick_fifo에서 시간마다 list_head에서 빼고 다시 넣는 작업을 해주기 때문에 위에서 구현한 FIFO대로 enqueue와 dequeue을 진행하였다. 처음에는 다른 함수들을 구현할지 막막했으나, rt에서 round robin을 뽑는 함수를 보며 FIFO를 구현하였기 때문에 그와 같이 하되 주어진 task_tick_fifo을 사용하면 FIFO와 다르게 타임슬라이스마다 list_head인 my_list의 맨 앞이 바뀌기 때문에 똑같이 구현을 해보았다. Put_prev_task의 역할은 원래 이전 테스크를 빼고 뒤로 넣어주는 역할이지만 task_tick_fifo에서 해주므로 넣으면 오히려 안 돌아가는 것을 확인했다. 그리고 fifo와 다르게 sleep을 지워서 바로바로 context switch하게 했다.

```
** START: Processes = 02 Time 01 s
Creating Process: #0
PROCESS #00 count = 634231 0100 ms
PROCESS #01 count = 17523 0100 ms
PROCESS #00 count = 661370 0100 ms
PROCESS #00 count = 1348906 0100 ms
PROCESS #00 count = 2038153 0100 ms
PROCESS #00 count = 2844938 0100 ms
PROCESS #00 count = 3682407 0100 ms
PROCESS #01 count = 673462 0595 ms
PROCESS #01 count = 1514588 0100 ms
PROCESS #01 count = 2353744 0100 ms
PROCESS #01 count = 3190345 0100 ms
PROCESS #01 count = 4034546 0100 ms
Done!! PROCESS #01 : 4034547 1095 ms
PROCESS #00 count = 4507859 0499 ms
Done!! PROCESS #00 : 4507860 1099 ms
```

타임슬라이스를 500ms

```
** START: Processes = 02 Time 01 s
Creating Process: #0
PROCESS #00 count = 648684 0100 ms
PROCESS #01 count = 14808 0100 ms
PROCESS #00 count = 676220 0100 ms
PROCESS #00 count = 1379124 0100 ms
PROCESS #01 count = 668157 0295 ms
PROCESS #01 count = 1504544 0100 ms
PROCESS #00 count = 2072059 0297 ms
PROCESS #00 count = 2914763 0100 ms
PROCESS #01 count = 2366634 0299 ms
PROCESS #01 count = 3226001 0100 ms
PROCESS #00 count = 3775903 0300 ms
PROCESS #00 count = 4637806 0100 ms
Done!! PROCESS #00 : 4637807 1097 ms
PROCESS #01 count = 4069563 0200 ms
Done!! PROCESS #01 : 4069564 1095 ms
```

200ms로

각각 돌린 결과이다.200ms일 때 context siwtch가 일어나기 때문에 시간이 더 많이 들어야하나, 1초가 짧아서 그런지 비슷하게 나왔다. FIFO와 비교하면 FIFO가 훨씬 카운트를 많이 세는 것을 볼 수 있다.

```
** START: Processes = 02 Time 01 s
Creating Process: #0
PROCESS #00 count = 658265 0100 ms
PROCESS #00 count = 1359328 0100 ms
PROCESS #00 count = 2044377 0100 ms
PROCESS #00 count = 2751965 0100 ms
PROCESS #00 count = 3431503 0100 ms
PROCESS #00 count = 4272164 0100 ms
PROCESS #00 count = 5122389 0100 ms
PROCESS #00 count = 5980627 0100 ms
PROCESS #00 count = 6849757 0100 ms
PROCESS #00 count = 7717527 0100 ms
Done!! PROCESS #00 : 7717528 1000 ms
PROCESS #01 count = 866614 0100 ms
PROCESS #01 count = 1710946 0100 ms
PROCESS #01 count = 2574079 0100 ms
PROCESS #01 count = 3436773 0100 ms
PROCESS #01 count = 4264660 0100 ms
PROCESS #01 count = 5119561 0100 ms
PROCESS #01 count = 5978215 0100 ms
PROCESS #01 count = 6823832 0100 ms
PROCESS #01 count = 7686482 0100 ms
PROCESS #01 count = 8532989 0100 ms
Done!! PROCESS #01 : 8532990 1000 ms
```

따라서 Context switch 가 많이 일어나면 같은 시간 속에 Count 값이 달라지는 것을 볼 수 있다. WRR 은 구현하지 못했지만 FIFO 와 RR 사이의 overhead 를 비교할 수 있었다.

어려웠던 점 : 1-3 에서 했던 코드 리뷰를 해당 스케쥴링 클래스에 맞춰서 하니 생각보다 어려웠고 많은 조건들이 있어서 원하는 부분을 얻기가 어려웠다. 특히 2-2 에서 스케줄링 클래스에서 언제 어떠한 함수가 수행되는 지 몰라서 상당히 긴 시간이 필요했고, 모듈을 적재하고 유저 어플리케이션을 실행할 때마다 커널이 멈춰서 계속 리붓이 필요했다. 하지만 이번 운영체제 과제로 커널코드를 조금이나마 이해할 수 있었고 추가적으로 CPU 가 스케줄링을 어떤 자료구조로 어떤 방식으로 하는가에 대해 크게 이해할 수 있는 시간이었다.