운영체제 기말 프로젝트

컴퓨터소프트웨어학부 2018008331 박민경

스케줄러 구현 – 설계

- priority 스케줄러를 구현한다.
- 자식 프로세스를 생성하고 각각 우선순위를 지정해주며, enqueue 될 때 queue 안의 태스크를 탐색하며 우선순위 변수에 대한 내림차순으로 저장될 수 있도록 한다. (우선순위 변수(정수) 값이 클 수록 우선순위가 높도록 할 것이다.) 우선순위가 동일한 태스크들에 대해서는 FCFS 방식을 적용한다.
- Dequeue 함수는 queue가 비어 있는 지 확인하고, 비어 있지 않다면 현재 태스크를 queue에서 제거한다.
- 다음 실행될 태스크를 선택하기 위해 pick_next_task 함수를 정의한다. queue에 있는 태스크들 중 가장 앞의 태스크 (우선순위가 가장 높은 태스크)를 선택한다.
- aging의 구현은 task_tick 함수에서 update_curr 함수를 호출하여 task_tick이 일어날 때마다 현재 태스크의 우선순위를 전체 프로세스에서 가장 낮은 단계인 10으로 만들어주어, 뒷 순위로 미루도록 한다.

스케줄러 구현 – 구현

스케줄러 우선순위는 rt > mysched > myrr > mypri > fait > idle 의 순서로 두었으며 Mypri 스케줄러를 사용하기 위해, 앞서 실습에서 했던 것과 같이 sched.h, core.c, Makefile 등에 환경설정을 해주었다. 여기서 sched_mypri_entity 구조체의 pri_num 멤버변수가 프로세스의 우선순위를 나타낼 것이다.

```
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;
struct sched_mysched_entity mysched;
struct sched_myrr_entity myrr;
struct sched_mypri_entity mypri;
```

```
struct sched_mypri_entity{
        struct list_head run_list;
        unsigned int pri_num;
};
```

<Include/linux/sched.h>

```
struct cfs_rq cfs;
struct rt_rq rt;
struct dl_rq dl;
struct mysched_rq mysched;
struct myrr_rq myrr;
struct mypri_rq mypri;
```

```
struct mypri_rq {
        struct list_head queue;
        unsigned int nr_running;
};
```

<kernel/sched/sched.h>

스케줄러 구현 – 결과

```
[   29.977975] ***[MYPRI] enqueue head: success cpu=1, nr_running=1, pid=1934, next p
ri_num=20
[   29.978111] ***[MYPRI] enqueue first: success cpu=0, nr_running=0, pid=1933
[   29.978120]          ***[MYPRI] show_queue: p->pid=1933, p->pri_num=20
[   29.978126]  ***[MYPRI] pick_next_task: cpu=0, prev->pid=1933, next_p->pid=1933, n
r_running=1, next_se->pri_num=20
[   29.978139]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1933
[   29.978148] ***[MYPRI] dequeue: success cpu=0, nr_running=0, pid=1933
[   29.978157] ***[MYPRI] enqueue tail: success cpu=1, nr_running=2, pid=1933, next p
ri_num=20
[   29.978488] ***[MYPRI] enqueue first: success cpu=0, nr_running=0, pid=1932
[   29.978499]          ***[MYPRI] show_queue: p->pid=1932, p->pri_num=40
[   29.978504]  ***[MYPRI] pick_next_task: cpu=0, prev->pid=1932, next_p->pid=1932, n
r_running=1, next_se->pri_num=40
[   29.978521]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1932
[   29.978532] ***[MYPRI] dequeue: success cpu=0, nr_running=0, pid=1932
[   29.978541] ***[MYPRI] enqueue head: success cpu=1, nr_running=3, pid=1932, next p
ri_num=30
```

생성되는 프로세스에 차례로 pri_num을 10, 40, 20, 30, 20으로 할당해준 결과이다. 프로세스가 enqueue될 때, 이미 queue에 있는 태스크들의 pri_num과 자신의 pri_num을 비교하여 내림차순을 지켜 enqueue된다. (enqueue tail, enqueue head로 표현. Enqueue first는 queue가 비어있을 경우 enqueue한 것이다.) 우선순위 순으로 정렬되며, 가장 큰 pri_num을 가진 태스크를 다음 태스크로 선정한다.

테스트 유저 프로그램 – 결과

```
root@2020osclass:~/newclass# ./mynewclass p
cpuset at [0th] cpu in parent process(pid=2186) is succeed
Child's PID = 2187
Child's PID = 2188
Child's PID = 2189
Child's PID = 2190
Child's PID = 2191
forking 5 tasks is completed
root@2020osclass:~/newclass# ***[NEWCLASS] Select mypri scheduling class
***[NEWCLASS] Select mypri scheduling class
***[NEWCLASS] Select mypri scheduling class
***[NEWCLASS] Select mypri scheduling class
***[NEWCLASS] Select mypri scheduling class
cpuset at [1st] cpu in child process(pid=2188) is succeed
pid=2188:       result=1
pid=2188:       result=2
pid=2188:       result=3
pid=2188:       result=4
pid=2188:       result=5
pid=2188:       result=6
pid=2188:       result=7
pid=2188:       result=8
pid=2188:       result=9
pid=2188:       result=10
pid=2188:       result=11
pid=2188:       result=12
pid=2188:       result=13
pid=2188:       result=14
pid=2188:       result=15
pid=2188:       result=16
pid=2188:       result=17
pid=2188:       result=18
pid=2188:       result=19
pid=2188:       result=20
pid=2188:       result=21
```

테스트 유저 프로그램인 mynewclass.c 를 실행시켰을 경우의 출력문이다. 각 프로세스가 result값에 1을 더하는 과정을 100번씩 반복한다. 의도대로라면, 가장 먼저 생성된 프로세스가 먼저 실행이 되고 있어야 하는데, 모든 프로세스가 enqueue 되고 나서 우선순위 순으로 정렬이 된 후에야 각자의 작업을 수행하는 모습을 볼 수 있다.

Priority 스케줄러 dmesg – 결과

```
[ 2217.316670]          ***[MYPRI] show_queue: p->pid=2188, p->pri_num=40
[ 2217.316698]          ***[MYPRI] show_queue: p->pid=2190, p->pri_num=30
[ 2217.316717]          ***[MYPRI] show_queue: p->pid=2191, p->pri_num=20
[ 2217.316732]          ***[MYPRI] show_queue: p->pid=2189, p->pri_num=20
[ 2217.316744]          ***[MYPRI] show_queue: p->pid=2187, p->pri_num=10
[ 2217.316755]  ***[MYPRI] pick_next_task: cpu=1, prev->pid=0, next_p->pid=2188,
 nr_running=5, next_se->pri_num=40
[ 2217.317427] ***[MYPRI] dequeue: success cpu=1, nr_running=4, pid=2188
[ 2217.317439]  ***[MYPRI] put_prev_task: do_nothing, p->pid=2188
[ 2217.317455] ***[MYPRI] enqueue head: success cpu=1, nr_running=4, pid=2188, n
ext pri_num=30
[ 2217.317630] ***[MYPRI] dequeue: success cpu=1, nr_running=4, pid=2188
[ 2217.317641]          ***[MYPRI] show_queue: p->pid=2190, p->pri_num=30
[ 2217.317651]          ***[MYPRI] show_queue: p->pid=2191, p->pri_num=20
[ 2217.317661]          ***[MYPRI] show_queue: p->pid=2189, p->pri_num=20
[ 2217.317670]          ***[MYPRI] show_queue: p->pid=2187, p->pri_num=10
[ 2217.317681]  ***[MYPRI] pick_next_task: cpu=1, prev->pid=2188, next_p->pid=21
90, nr_running=4, next_se->pri_num=30
[ 2217.318381] ***[MYPRI] dequeue: success cpu=1, nr_running=3, pid=2190
[ 2217.318411]  ***[MYPRI] put_prev_task: do_nothing, p->pid=2190
[ 2217.318427] ***[MYPRI] enqueue head: success cpu=1, nr_running=3, pid=2190, n
ext pri_num=20
[ 2217.318525] ***[MYPRI] dequeue: success cpu=1, nr_running=3, pid=2190
[ 2217.318536]          ***[MYPRI] show_queue: p->pid=2191, p->pri_num=20
[ 2217.318545]          ***[MYPRI] show_queue: p->pid=2189, p->pri_num=20
[ 2217.318554]          ***[MYPRI] show_queue: p->pid=2187, p->pri_num=10
[ 2217.318564]  ***[MYPRI] pick_next_task: cpu=1, prev->pid=2190, next_p->pid=21
91, nr_running=3, next_se->pri_num=20
[ 2217.319213] ***[MYPRI] dequeue: success cpu=1, nr_running=2, pid=2191
[ 2217.319223]  ***[MYPRI] put_prev_task: do_nothing, p->pid=2191
[ 2217.319239] ***[MYPRI] enqueue head: success cpu=1, nr_running=2, pid=2191, n
ext pri_num=10
```

테스트 유저 프로그램인 mynewclass.c 를 실행시켰을 경우의 스케줄러의 실행 결과(dmesg)이다. 앞 장에서 언급한 문제처럼, 결국 작업이 수행될 때는 모든 태스크가 enqueue된 상태에서 수행을 시작한다. 우선순위가 같은 경우에는 FCFS 방식으로 동작하긴 한다.

Aging 구상

```
static void update_curr_mypri(struct rq *rq){
        struct mypri_rq *mypri_rq = &rq->mypri;
        struct list_head *pos = NULL;
        struct task_struct *pos_p = NULL;
        struct sched_mypri_entity *pos_se = NULL;
        struct task_struct *p = rq->curr;
        struct list_head *tmp = &rq->curr->mypri.run_list;
        int i=0;

        if (p->mypri.pri_num > 10)
        {
                p->mypri.pri_num = p->mypri.pri_num - 10;
                printk(KERN_INFO "***[MYPRI] update_curr_mypri: pid=%d, pri_
num=%d\n", p->pid, p->mypri.pri_num);
        }

        if(mypri_rq->nr_running > 1)
        {
                for(pos = mypri_rq->queue.next; pos!= &mypri_rq->queue; pos=
pos->next)
                {
                        pos_se = container_of(pos, struct sched_mypri_entity
, run_list);
                        pos_p = container_of(pos_se, struct task_struct, myp
ri);

                        if(p->mypri.pri_num >= pos_se->pri_num)
                        {
                                list_del_init(tmp);
                                printk(KERN_INFO "[MYPRI] update_curr_mypri_
reschedule: success cpu=%d, nr_running=%d, pid=%d\n", cpu_of(rq), mypri_rq->
nr_running, p->pid);
                                __list_add(tmp, pos->prev, pos);
                                resched_curr(rq);
                        }
                }
        }
}
```

(해당 코드를 포함하여 실행했을 때 kernel_panic 문제가 생겨 주석처리 하였습니다)

Update_curr_mypri는 task_tick 함수에서 호출된다. 이 때 현재 프로세스의 pri_num이 10보다 크다면 (10단위) pri_num을 10 감소시키고,, 다시 queue 안의 task들을 sorting하는 작업을 거친 후 resched_curr함수를 호출하는 과정이다.

Aging 적용 결과

```
static void update_curr_mypri(struct rq *rq){
        struct mypri_rq *mypri_rq = &rq->mypri;
        struct list_head *pos = NULL;
        struct task_struct *pos_p = NULL;
        struct sched_mypri_entity *pos_se = NULL;
        struct task_struct *p = rq->curr;
        struct list_head *tmp = &rq->curr->mypri.run_list;
        int i=0;

        if (p->mypri.pri_num > 10)
        {
                p->mypri.pri_num = p->mypri.pri_num - 10;
                printk(KERN_INFO "***[MYPRI] update_curr_mypri: pid=%d, pri_
num=%d\n", p->pid, p->mypri.pri_num);
        }

        if(mypri_rq->nr_running > 1)
        {
                for(pos = mypri_rq->queue.next; pos!= &mypri_rq->queue; pos=
pos->next)
                {
                        pos_se = container_of(pos, struct sched_mypri_entity
, run_list);
                        pos_p = container_of(pos_se, struct task_struct, myp
ri);

                        if(p->mypri.pri_num >= pos_se->pri_num)
                        {
                                list_del_init(tmp);
                                printk(KERN_INFO "[MYPRI] update_curr_mypri_
reschedule: success cpu=%d, nr_running=%d, pid=%d\n", cpu_of(rq), mypri_rq->
nr_running, p->pid);
                                __list_add(tmp, pos->prev, pos);
                                resched_curr(rq);
                        }
                }
        }
}
```

(실제 적용X)
Aging을 적용하였을 경우 예상되는 결과 (의도한 결과)는, 실행중인 태스크의 우선순위를 10씩 감소시켜 resorting 한 다음 그 결과에 따라 next_task를 선택하기 때문에 모든 task들이 골고루, queue 안에서 순서가 바뀌며 실행될 수 있도록 하는 것이다.

Aging을 적용하지 않으면 높은 우선순위의 프로세스들이 계속 queue안에 들어오고, next_task로 선정됨에 따라 낮은 우선순위를 갖는 프로세스가 작업을 하지 못하는 starving 현상이 일어날 수 있다.

링크

https://drive.google.com/file/d/1dnboYURMpL60WsqjB3FvUzwMu4Z217ZB/view?usp=sharing