운영체제 기말 프로젝트
선점형 + Aging Priority 스케줄러의 구현

2016025469 서건식

https://drive.google.com/file/d/1_8QhagXLT1oaWoiW03E3LgvJFx7gbE9j/view?usp=sharing

요약

프로세스마다 부여된 Priority에 대한 MultiLevel Queue를 선언했습니다.
각 Queue안에서 스케줄링 되는 방식은 FCFS 알고리즘을 사용했습니다.
우선순위를 부여하는 방식은 system call을 사용해 유저모드에서 직접 프로세스에 원하는 우선순위를 부여하는 방식으로 구현하였습니다.
Aging기법은 정해진 시간을 초과하면 우선순위를 0으로 만들어 0 우선순위 queue에 삽입해 가장 먼저 스케줄링 되도록 구현했습니다.

커널 설정 과정

1. kernel/sched/sched.h

```
#define SCHEDH_MAX_MYPRI     10
struct myprio_rq {
    struct list_head queue[SCHEDH_MAX_MYPRI];
    unsigned int nr_running_total;
    unsigned int nr_running[SCHEDH_MAX_MYPRI];
};
```

Rq의 경우 우선순위 별로 rq를 만들어줬으며, 최대로 낮은 우선순위의 값은 10입니다. 또한 그 큐 마다 nr_running 값을 다르게 측정하며, 전체값은 nr_running_total에 저장합니다.

2. kernel/sched/core.c

```
static void __setscheduler_params(struct task_struct *p,
        const struct sched_attr *attr)
{
    int policy = attr->sched_policy;

    if (policy == SETPARAM_POLICY)
        policy = p->policy;

    p->policy = policy;

    if (dl_policy(policy))
        __setparam_dl(p, attr);
    else if (fair_policy(policy))
        p->static_prio = NICE_TO_PRIO(attr->sched_nice);

    /*
     * __sched_setscheduler() ensures attr->sched_priority == 0 when
     * !rt_policy. Always setting this ensures that things like
     * getparam()/getattr() don't report silly values for !rt tasks.
     */
    p->rt_priority = attr->sched_priority;
    p->myprio = attr->sched_myprio;
    p->normal_prio = normal_prio(p);
    set_load_weight(p);
}
```

3번에서 언급하지만 mypriority에 attr 구조체의 멤버 변수인 sched_myprio를 저장하는 모습입니다. Mypriority는 task_struct에 선언 되어 있습니다.

3. include/linux/sched.h

root@2020osclass: /usr/src/linux-3.18.27/include/linux

```
struct sched_attr {
    u32 size;

    u32 sched_policy;
    u64 sched_flags;

    /* SCHED_NORMAL, SCHED_BATCH */
    s32 sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    u32 sched_priority;

    /* SCHED_DEADLINE */
    u64 sched_runtime;
    u64 sched_deadline;
    u64 sched_period;

    /* SCHED_MYPRI*/
    u32 sched_myprio;
};
```

Sched_attr 구조체에 myprio라는 값을 선언했습니다. 이는 user level에서 parameter를 전달하여(system call 사용) core.c의 __setschedular_params에서 저장합니다.

```
struct sched_myprio_entity{
    struct list_head run_list;
    unsigned int myprio;
};
```

Sched_myprio_entity 구조체입니다. myprio 라는 우선순위 번호를 멤버변수로 가지게 했습니다.

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;

#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
    struct task_struct *last_wakee;
    unsigned long wakee_flips;
    unsigned long wakee_flip_decay_ts;

    int wake_cpu;
#endif
    int on_rq;

    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;
    unsigned int mypriority;
    struct sched_myprio_entity myprio;
#ifdef CONFIG_CGROUP_SCHED
```

task_struct 구조체입니다. Mypriority(우선순위)와 entity를 추가로 선언 하였습니다.

Priority 스케줄러 구현과 Aging 기법

1. 상수값

```
#define MAX_MYTIME      4500000011 //about 4.5 sec
#define MAX_MYPRI       10
```

MYTIME을 보시면 4500000011으로 설정하였으며, 실행시간이 이 시간을 초과하면 에이징 기법을 사용해 해당 프로세스의 우선순위를 높입니다. 약 4.5초 이전은 에이징 기법이 적용되지 않은 우선순위 스케줄러이고, 4.5초 이후는 에이징이 적용된다고 보시면 됩니다.

2. init_myprio_rq

```
void init_myprio_rq (struct myprio_rq *myprio_rq)
{
    unsigned int i;
    myprio_rq->nr_running_total=0;
    for (i = 0; i < MAX_MYPRI; i++) {
        INIT_LIST_HEAD(&myprio_rq->queue[i]);
        myprio_rq->nr_running[i] = 0;
    }
    printk(KERN_INFO "***[MYPRI] Myprio class is online \n");
}
```

런큐를 초기화하는 함수이며, 해당 멤버변수들을 모두 초기화해줍니다.

Priority 스케줄러 구현과 Aging 기법

3. enqueue_task_myprio & dequeue_task_myprio

```
static void enqueue_task_myprio(struct rq *rq, struct task_struct *p, int flags) {
    p->myprio.myprio = p->mypriority;
    list_add_tail(&p->myprio.run_list, &rq->myprio.queue[p->myprio.myprio]);
    rq->myprio.nr_running[p->myprio.myprio]++;
    rq->myprio.nr_running_total++;

    printk(KERN_INFO"***[MYPRI] enqueue: success pri=%u, nr_running=%d, pid=%d\n",p->mypr
;
}
static void dequeue_task_myprio(struct rq *rq, struct task_struct *p, int flags)
{
    if(rq->myprio.nr_running_total>0)
    {
        p->mypriority = p->myprio.myprio;
        list_del_init(&p->myprio.run_list);
        rq->myprio.nr_running[p->myprio.myprio]--;
        rq->myprio.nr_running_total--;

        printk(KERN_INFO"\t***[MYPRI] dequeue: success pri=%u, nr_running=%d, pid=%d\n",p
p->pid);
    }
    else{
    }

}
```

1) enqueue_task_myprio

Task_struct 구조체인 p의 멤버변수 mypriority는 시스템콜을 통해 유저레벨로 부터 우선순위 값을 부여 받습니다.
부여받은 우선순위를 task_struct상에 선언된 sched_myprio_entit의 myprio에 저장합니다.
그 우선순위를 바탕으로 런큐에 해당 우선순위에 해당하는 큐에 삽입합니다. 삽입한 후에는 우선순위에 해당하는 nr_runnin값과 total값을 update 합니다.

2) dequeue_task_myprio

전체 nr_runnin값이 0보다 클 때, task_struct의 mypriority의 값을 지울 entity의 우선순위 값으로 저장해준 후 list_del_init으로 큐에서 삭제하고 nr_running 값을 수정합니다.

Priority 스케줄러 구현과 Aging 기법

4. pick_next_task_myprio

```
struct task_struct *pick_next_task_myprio(struct rq *rq, struct task_struct *prev)
{
    int i;
    int highest_prio = MAX_MYPRI;
    struct task_struct* next_p;
    struct sched_myprio_entity* myprio_se = NULL;

    if(rq->myprio.nr_running_total == 0){
        return NULL;
    }
    else {
         for(i=MAX_MYPRI-1 ; i>=0 ; i--){
             if(rq->myprio.nr_running[i] >0) {
                highest_prio = i;
             }
         }

        myprio_se = list_entry(rq->myprio.queue[highest_prio].next, struct sched_myprio_entity, run_list);
        next_p = container_of(myprio_se, struct task_struct, myprio);
        printk(KERN_INFO "\t***[MYPRI] pick_next_task: pri=%u, prev->pid=%d,next_p->pid=%d,nr_running=%d\n",highe
ext_p->pid, rq->myprio.nr_running_total);
        return next_p;
    }
        return NULL;
}
```

먼저 highest_prio는 현재 런큐에 삽입된 프로세스 중 가장 높은 우선순위를 가진 프로세스가 무엇인지 판별하기 위함입니다. 가장 낮은 우선순위로 초기화 해준 후에 런큐의 nr_running 값이 0보다 큰 경우 highest_prio의 값을 업데이트 해주면 반복문을 나왔을 때 가장 높은 우선순위가 배정이 됩니다. 배정된 우선순위를 기반으로, myprio_se에 entity를 저장한 후 container_of 함수를 통해 task_struct로 변환하여 return합니다.

우선순위가 가장 높은 queue에 있는 프로세스는 FCFS방식으로 스케줄링 했습니다. 이는 Next_p를 구하기 위해 sched_myprio_entity를 가장 높은 우선순위 queue의 next로 설정하는 것을 통해 확인할 수 있습니다.

Priority 스케줄러 구현과 Aging 기법

5. update_curr_myprio (aging 구현)

```
static void update_curr_myprio(struct rq *rq){

    struct task_struct *curr = rq->curr;
    struct sched_myprio_entity *myprio_se = &curr -> myprio;
    unsigned int beforePri;
    unsigned int afterPri;
    unsigned int frontPri;
    unsigned int highest_prio = MAX_MYPRI;
    unsigned int i;
    u64 delta_exec;
    if (curr->sched_class != &myprio_sched_class)
        return;

    for(i=MAX_MYPRI-1 ; i>0 ; i--){
        if(rq->myprio.nr_running[i] >0) {
            highest_prio = i;
        }
    }

    //update_rq_clock(rq);

    delta_exec = rq_clock_task(rq) - curr->se.exec_start;
    frontPri = myprio_se->myprio-1;
    //printk(KERN_INFO"***[MYPRIO] update_curr_myrr     delta_exec  = %llu\n",delta_exec);
    if(delta_exec > MAX_MYTIME && frontPri == highest_prio)
    {

        beforePri = myprio_se->myprio;
        list_del_init(&curr->myprio.run_list);
        rq->myprio.nr_running[myprio_se->myprio]--;
        myprio_se->myprio = 0;
        curr->mypriority = myprio_se->myprio;
        afterPri = myprio_se -> myprio;
        list_add_tail(&curr->myprio.run_list, &rq->myprio.queue[myprio_se->myprio]);
        rq->myprio.nr_running[myprio_se->myprio]++;
        resched_curr(rq);
    }
```

Update_curr_myprio함수에서 aging을 구현하였습니다.
Delta_exec 변수는 실행시간을 계산하기 위한 변수이며, rq_clock_task(rq) 함수를 사용해 rq->clock_task 멤버변수를 받아와 현재 실행중인 시간을 받아오고, task_struct에 sched_entity 구조체로 선언되어 있는 se 변수의 멤버변수인 exec_start의 값을 빼줍니다. 이 값은 프로세스가 시작한 시간입니다. 그 차이는 현재 프로세스가 얼마나 실행했는지를 파악할 수가 있습니다.

우선 순위 증가의 기준은 rq에서 우선순위가 가장 높은 숫자를 highest_prio에 저장합니다. 그리고 현재 curr의 priority의 바로 앞 priority를 frontPri에 저장하고 그것이 만약 가장 높은 우선순위의 숫자와 같다면 우선순위를 감소시켜 aging을 실시합니다. 이는 가장 높은 우선순위의 다음 순위가 curr인지 검사하는 과정이라고 볼 수 있습니다.

따라서 limit으로 정한 MAX_MYTIME값을 초과한 프로세스들 중에서 highest priority를 가지는 프로세스를 제외하고 가장 우선순위가 높은 프로세스를 먼저 처리하겠다는 의도입니다.

Aging 과정은 다음과 같습니다.
먼저 현재 우선순위에 해당하는 큐에서 해당 프로세스를 뺀 후, 해당 우선순위 큐의 nr_running 값을 감소시켜 줍니다. 이후 우선순위를 0으로 높여준 후에 curr의 mypriority 값에도 반영합니다. list_add_tail 함수를 통해 우선순위 0에 해당하는 런큐에 해당 프로세스를 삽입하며, nr_running 값을 update 해줍니다. 이후 resched_curr(rq) 함수를 통해 다시 스케줄링을 실시하는 방식으로 aging을 구현하였습니다.

테스트 유저 프로그램

```c
struct sched_attr {
    __u32 size;

    __u32 sched_policy;
    __u64 sched_flags;

    /* SCHED_NORMAL, SCHED_BATCH */
    __s32 sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    __u32 sched_priority;

    /* SCHED_DEADLINE */
    __u64 sched_runtime;
    __u64 sched_deadline;
    __u64 sched_period;

    /* SCJED_MYPRI */
    __u32 sched_myprio;
};
```

커널에 전달할 sched_attr 구조체이며, sched_myprio 값을 선언해줬습니다.

```c
else if (c == 'p')
{
    printf("***[NEWCLASS] Select mypri scheduling class \n");
    /* set attributes for SCHED_MYPRI */
    attr.size = sizeof(attr);
    attr.sched_policy = SCHED_MYPRI;
    attr.sched_flags = 0;

    attr.sched_period = 0;
    attr.sched_runtime = 0;
    attr.sched_deadline = 0;

    attr.sched_nice = 0;        // for SCHED_NORMAL and SCHED_BATCH
    attr.sched_priority = 0;    // for SCHED_FIFO and SCHED_RR

    attr.sched_myprio = 9 - i;      //for SCHED_MYPRI
    ret = sched_setattr(my_pid, &attr, flags);
    if (ret != 0) {
        perror("sched_setattr");
        exit(1);
    }
}
```

입력값이 p일때 Priority 스케줄러가 작동하며, 우선순위는 들어오는 프로세스부터 9로 시작하여 순차적으로 우선순위를 높여주는 방식으로 부여하였습니다.

```c
    }
    sleep(i*1);

    /* child process work */
    int j=0;
    for(j=0; j<8; j++) {

        int i=0;
        int result = 0;
        for(i=0; i<200000000; i++)
        {
            result+=1;
        }
        printf("pid=%d:\tresult=%d\n", my_pid, result);
        usleep(58*1000);
    }
    exit(1);
}
printf("Child's PID = %d\n", pids);
//sleep(1);
```

좀 더 정확한 동작을 보기 위해 반복 횟수를 늘리고 usleep으로 ms단위로 프로그램의 시간을 늘려줬습니다.

실행결과(dmesg) – Aging 적용 X

```
root@2020osclass:~/newclass# ./newclass3 p
cpuset at [0th] cpu in parent process(pid=1891) is succeed
Child's PID = 1892
Child's PID = 1893
Child's PID = 1894
Child's PID = 1895
```

```
[  202.614003]  ***[MYPRI] pick_next_task: pri=6, prev->pid=1895,next_p->pid=1895,nr_running=1
[  202.614015]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1895
[  202.614027]  ***[MYPRI] dequeue: success pri=6, nr_running=0, pid=1895
[  202.614036] ***[MYPRI] enqueue: success pri=6, nr_running=1, pid=1895
[  202.614158]  ***[MYPRI] pick_next_task: pri=6, prev->pid=0,next_p->pid=1895,nr_running=1
[  202.614162] ***[MYPRI] set_curr_task_myprio
[  202.614171] ***[MYPRI] enqueue: success pri=7, nr_running=1, pid=1894
[  202.614179]  ***[MYPRI] pick_next_task: pri=7, prev->pid=1894,next_p->pid=1894,nr_running=1
[  202.614190]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1894
[  202.614199]  ***[MYPRI] dequeue: success pri=7, nr_running=0, pid=1894
[  202.614208] ***[MYPRI] enqueue: success pri=7, nr_running=2, pid=1894
[  202.614213] ***[MYPRI] check_preempt_curr_myprio
[  202.614219]  ***[MYPRI] dequeue: success pri=6, nr_running=1, pid=1895
[  202.614219]  ***[MYPRI] pick_next_task: pri=7, prev->pid=1895,next_p->pid=1894,nr_running=1
[  202.614219] ***[MYPRI] set_curr_task_myprio
[  202.614219] ***[MYPRI] enqueue: success pri=8, nr_running=1, pid=1893
[  202.614219]  ***[MYPRI] pick_next_task: pri=8, prev->pid=1893,next_p->pid=1893,nr_running=1
[  202.614219]  ***[MYPRI] dequeue: success pri=7, nr_running=0, pid=1894
[  202.614219]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1894
[  202.614220]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1893
[  202.614220]  ***[MYPRI] dequeue: success pri=8, nr_running=0, pid=1893
[  202.614220] ***[MYPRI] enqueue: success pri=8, nr_running=1, pid=1893
[  202.614220]  ***[MYPRI] pick_next_task: pri=8, prev->pid=0,next_p->pid=1893,nr_running=1
[  202.614220] ***[MYPRI] set_curr_task_myprio
[  202.614220]  ***[MYPRI] dequeue: success pri=8, nr_running=0, pid=1893
[  202.614220]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1893
[  202.614763] ***[MYPRI] enqueue: success pri=9, nr_running=1, pid=1892
[  202.614918]  ***[MYPRI] pick_next_task: pri=9, prev->pid=1892,next_p->pid=1892,nr_running=1
[  202.614941]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1892
[  202.614957]  ***[MYPRI] dequeue: success pri=9, nr_running=0, pid=1892
[  202.614973] ***[MYPRI] enqueue: success pri=9, nr_running=1, pid=1892
[  202.615068]  ***[MYPRI] pick_next_task: pri=9, prev->pid=0,next_p->pid=1892,nr_running=1
[  203.537536]  ***[MYPRI] dequeue: success pri=9, nr_running=0, pid=1892
[  203.537571]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1892
[  203.595193] ***[MYPRI] enqueue: success pri=9, nr_running=1, pid=1892
[  203.595301]  ***[MYPRI] pick_next_task: pri=9, prev->pid=0,next_p->pid=1892,nr_running=1
[  203.615529] ***[MYPRI] enqueue: success pri=8, nr_running=2, pid=1893
```

생성된 프로세스는 다음과 같이 4개이며 우선순위는 각각
1892 - 9 1893 - 8
1894 - 7 1895 - 6
으로 설정됩니다.

Aging 기법은 4.5초 이후에 적용이 됩니다. 따라서 dmesg에 찍히는 시간 기준으로 4.5초 전과 후의 dmesg를 살펴보도록 하겠습니다.
먼저 1895 process부터 enqueue가 되고, 그 뒤의 우선 순위를 가진 프로세스들이 차례대로 처리가 되는 점을 볼 때 Priority 스케줄러가 잘 작동함을 알 수 있습니다. 선점형 우선순위 스케줄러로 구현하였기 때문에 예를 들어 자신의 우선순위 보다 높은 프로세스가 enqueue되면 pick_next_task 함수는 그 프로세스를 선택할 것입니다.

그 내용은 다음 슬라이드에서 Aging 기법과 함께 보이겠습니다.

실행결과(dmesg) – Aging 적용 O

빨간색 밑줄은 선점형 우선순위 스케줄러의 특징을 보여줍니다. 우선순위가 8인 프로세스가 런큐에 먼저 들어왔지만 이후 우선순위가 6인 프로세스가 enqueue 되고, pick_next_task에서 선택되는 것은 우선순위가 높은 1895 프로세스가 선택됨을 확인할 수 있습니다.

```
[  205.442775] ***[MYPRI] enqueue: success pri=8, nr_running=3, pid=1893
[  205.442850] ***[MYPRI] check_preempt_curr_myprio
[  205.615155] ***[MYPRI] enqueue: success pri=6, nr_running=4, pid=1895
[  205.615199] ***[MYPRI] check_preempt_curr_myprio
[  206.222080]  ***[MYPRI] dequeue: success pri=7, nr_running=3, pid=1894
[  206.222141]  ***[MYPRI] pick_next_task: pri=6, prev->pid=1894,next_p->pid=1895,nr_running=3
[  206.280388] ***[MYPRI] enqueue: success pri=7, nr_running=4, pid=1894
[  206.280557] ***[MYPRI] check_preempt_curr_myprio
[  206.995573]  ***[MYPRI] dequeue: success pri=6, nr_running=3, pid=1895
[  206.995590]  ***[MYPRI] pick_next_task: pri=7, prev->pid=1895,next_p->pid=1894,nr_running=3
[  207.054143] ***[MYPRI] enqueue: success pri=6, nr_running=4, pid=1895
[  207.054248] ***[MYPRI] check_preempt_curr_myprio
[  207.115413]  ***[MYPRI] pick_next_task: pri=0, prev->pid=1894,next_p->pid=1894,nr_running=4
[  207.804718]  ***[MYPRI] dequeue: success pri=0, nr_running=3, pid=1894
[  207.804739]  ***[MYPRI] pick_next_task: pri=6, prev->pid=1894,next_p->pid=1895,nr_running=3
[  207.862976] ***[MYPRI] enqueue: success pri=0, nr_running=4, pid=1894
[  207.863040] ***[MYPRI] check_preempt_curr_myprio
[  208.605268]  ***[MYPRI] dequeue: success pri=6, nr_running=3, pid=1895
[  208.605313]  ***[MYPRI] pick_next_task: pri=0, prev->pid=1895,next_p->pid=1894,nr_running=3
[  208.663524] ***[MYPRI] enqueue: success pri=6, nr_running=4, pid=1895
[  208.663605] ***[MYPRI] check_preempt_curr_myprio
[  209.051581]  ***[MYPRI] dequeue: success pri=0, nr_running=3, pid=1894
[  209.051589]  ***[MYPRI] put_prev_task: do_nothing, p->pid=1894
```

초록색 밑줄을 통해 Aging 기법이 구현되었음을 알 수 있습니다. 전 슬라이드를 보시면 202.614003 부근에서 시작이 되었기 때문에, 207.115413 부근이 실행된지 약 4.5초가 지난 시점입니다. 이 쯤에서 update_curr함수가 실행되었으며, 앞서 코드 구현 설명 부분에서 말씀드렸듯이 1985(6) 다음으로 높은 우선순위를 가진 1894(7) 프로세스의 우선순위를 가장 높은 0으로 만들어 먼저 처리하는 것을 확인할 수 있습니다.