컴퓨터 소프트웨어학부2019062833 감유진

Priority Scheduler

Preemptive and Aging

Part 1

전체적인 스케줄러 흐름도 설계

STEP 1

프로세스가 생성되면 큐에 Enqueue하게 된다.
이때 우선순위에 맞게 순서대로 배치되어 들어가게 되고 가장 앞의 프로세스가 CPU를 획득하게 된다.

>>

STEP 2

CPU를 획득한 프로세스는 시간이지남에 따라 Aging으로 인해 우선순위가 낮아지게 된다.
다음 프로세스의 우선순위보다 낮아지게 되면 큐에서 재배치 되며 CPU를 뺏기게 된다.

>>

STEP 3

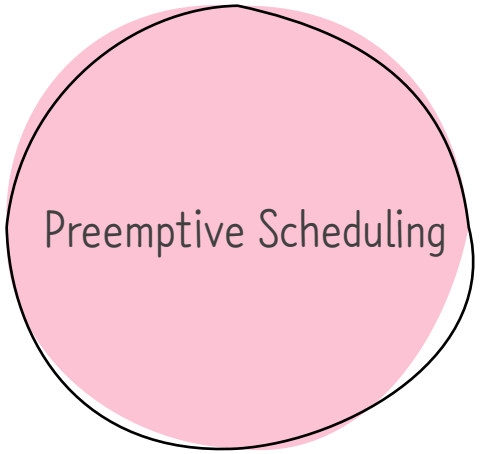
프로세스가 실행도중에 자신보다 우선 순위가 높은 프로세스가 큐에 들어오게 된다면 CPU를 뺏기게 되고 큐에서 밀려나게 된다.

>>

STEP 4

실행을 마친 프로세스는 Dequeue되어 큐에서 제거되며 이 후 큐가 다시 재배치 된다.

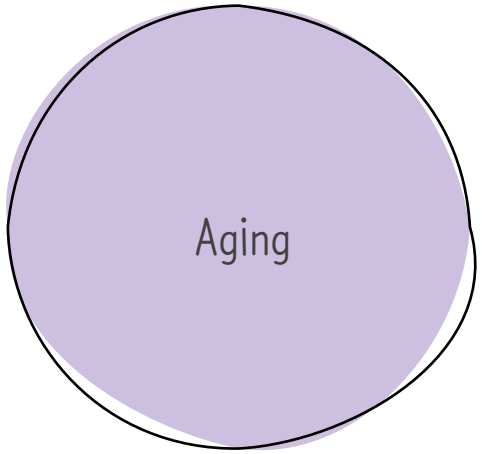
Preemptive Scheduling 과 Aging 설계



현재 실행 중인 것보다 우선 순위가 높은 프로세스가 있다면 CPU를 뺏기게 된다.

우선 순위가 더 높은 새 프로세스가 Enqueue 된 경우 큐의 맨 앞에 배치하고 스케줄링을 다시 실행한다.

현재 실행 중이던 프로세스가 Aging으로 인해 다음 프로세스보다 우선순위가 낮아지게 되면 그 프로세스에게 선점 당하게 된다. 즉, 현재의 프로세스를 큐에서 재배치 시키고 스케줄링을 다시 실행한다.



Entity는 priority 값을 추가로 보유하고 있으며 mypriority_rq는 next priority 값을 가져 두번째로 높은 우선순위를 알고 있다.

시간이 지남에 따라 Update_curr 함수가 불리게 되고 이 함수 내부에서 entity의 priority 값을 증가 시킨다(우선 순위 감소).

만약 증가시킨 그 값이 현재 두번째로 높은 우선순위보다 낮아지게(숫자가 커지게) 된다면 큐를 재배치하고 CPU를 넘겨준다.

Part 2

스케줄러 구현 1

```
static void queueing_mypriority(struct sched_mypriority_entity *entity, struct mypriority_rq *mypriority_rq, pid_t pid) {
    struct list_head *head = &mypriority_rq->queue.next;
    struct list_head *now_list = mypriority_rq->queue.next;

    struct task_struct *next_p = NULL;
    struct sched_mypriority_entity *next_se = NULL;
    struct sched_mypriority_entity * now_entity;
    printk("+++[MYPRIORITY] queuing mypriority - START pid=%d\n", pid);
    //Find a place to get in and put it in the queue.
    while (now_list->next != head) {
        now_entity = container_of(now_list, struct sched_mypriority_entity, run_list);
        if (now_entity->priority > entity->priority) {
            //Connect to Linked List
            entity->run_list.next = now_list;
            entity->run_list.prev = now_list->prev;
            now_list->prev->next = &entity->run_list;
            now_list->prev = &entity->run_list;

            next_se = container_of(now_list, struct sched_mypriority_entity, run_list);
            next_p = container_of(next_se, struct task_struct, mypriority);
            printk("+++[MYPRIORITY] queuing mypriority - LOCATE!!!! pid=%d my next=%d priority=%d\n", pid, next_p->pid, entity->priority);
            break;
        }
        now_list = now_list->next;
    }
    if (now_list->next == head) {
        printk("+++[MYPRIORITY] queuing mypriority - LOCATE TAIL pid=%d priority=%d\n", pid, entity->priority);
        list_add_tail(&entity->run_list, &mypriority_rq->queue);
    }

    //Update information of the second highest priority value
    if (mypriority_rq->nr_running > 1) {
        struct sched_mypriority_entity *find_entity;
        struct list_head *head = &mypriority_rq->queue.next;
        find_entity = container_of(head->next, struct sched_mypriority_entity, run_list);
        mypriority_rq->next_priority = find_entity->priority;
        printk("+++[MYPRIORITY] queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=%d\n", mypriority_rq->next_priority);

    }
}
```

받아온 entity가 큐에 들어갈 자리를 탐색한다. 큐는 우선순위 순서로 오름차순으로 정렬되어 있어 가장 앞의 요소가 가장 작은 값을 가져 가장 높은 우선순위를 갖는다.
자리 배치가 완료 되었으면 두번째로 높은 우선순위 값을 갱신하여 둔다.

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

    struct task_struct *curr = rq->curr;
    struct mypriority_rq *mypriority_rq = &rq->mypriority;
    struct sched_mypriority_entity *entity = &curr->mypriority;
    //우선순위 값을 줄인다.
    entity->priority++;
    printk("+++[MYPRIORITY] aging_mypriority AGING ++   pid=%d priority=%d\n", curr->pid, entity->priority);
    //만약 두번째로 높은 우선순위보다 자신의 우선 순위가 낮다면(숫자가 더 크다면) 큐를 재배치 시킨다.
    if (mypriority_rq->nr_running > 1 && (entity->priority > mypriority_rq->next_priority)) {
        printk("+++[MYPRIORITY] AGING -- > REPLACE BEFORE pid=%d priority=%d\n", curr->pid, entity->priority);
        //큐에서의 재배치 작업 - 우선 제거
        list_del_init(&curr->mypriority.run_list);
        mypriority_rq->nr_running--;
        //큐에서의 재배치 작업 - 순서에 맞게 다시 넣기
        queueing_mypriority(entity, mypriority_rq, curr->pid);
        mypriority_rq->nr_running++;
        //다시 스케줄링을 실시
        resched_curr(rq);
        printk("+++[MYPRIORITY] AGING -- > REPLACE AFTER pid=%d\n", curr->pid);
    }

}
```

```
void task_tick_mypriority(struct rq *rq, struct task_struct *p, int queued) {
    //Over time, AGING is used to lower priorities.
    update_curr_mypriority(rq);

}
```

일정한 시간마다 task_tick 함수가 호출되어 update_curr 함수를 호출하게 된다. Update_curr 함수 내부에서는 우선 현재의 프로세스의 우선순위 값을 줄이고(숫자는 크게) 변경 후의 값이 다음 프로세스의 우선순위 값보다 커지게 되면 큐에서 위치를 재배치 시키고 스케줄링을 재실행 한다.

Part 2

스케줄러 구현 2

```
static void enqueue_task_mypriority(struct rq *rq, struct task_struct *p, int flags) {

    struct mypriority_rq *mypriority_rq = &rq->mypriority;
    struct sched_mypriority_entity *task = &p->mypriority;
    struct task_struct *curr = rq->curr;

    //Prioritize with pid values.
    task->priority = p->pid % PRIORITY_NUM;
    //Queue in order of priority.
    rq->mypriority.nr_running++;
    queueing_mypriority(task, mypriority_rq, p->pid);
    printk(KERN_INFO"+++[MYPRIORITY] Enqueue: success cpu=%d, nr_running=%d, pid=%d\n", cpu_of(rq), rq->mypriority.nr_running, p->pid);
    //  list_add_tail(&task->run_list, &mypriority_rq->queue);

        //preempt if it has the highest priority.
    if (mypriority_rq->nr_running > 1 && (&task->run_list == mypriority_rq->queue.next)) {
        resched_curr(rq);
        printk(KERN_INFO"+++[MYPRIORITY] enqueue - PREEMPT: success cpu=%d, nr_running=%d, now_pid=%d\n", cpu_of(rq), rq->mypriority.nr_runn

    }
}
```

```
struct task_struct *pick_next_task_mypriority(struct rq *rq, struct task_struct *prev)
{
    struct task_struct *next_p = NULL;
    struct sched_mypriority_entity *next_se = NULL;
    struct mypriority_rq *mypriority_rq = &rq->mypriority;

    if (rq->mypriority.nr_running == 0) {
        return NULL;
    }

    //If there is more than one task in the current queue, determine the task in the head as the following task
    next_se = container_of(mypriority_rq->queue.next, struct sched_mypriority_entity, run_list);
    next_p = container_of(next_se, struct task_struct, mypriority);

    printk(KERN_INFO "+++[MYPRIORITY] pick_next_task: cpu=%d, prev->pid=%d,next_p->pid=%d,nr_running=%d\n", cpu_o
    return next_p;
}
```

```
static void dequeue_task_mypriority(struct rq *rq, struct task_struct *p, int flags)
{
    struct mypriority_rq *mypriority_rq = &rq->mypriority;
    if (rq->mypriority.nr_running > 0)
    {

        //If there is more than one task in the current queue, remove it from the queue
        list_del_init(&p->mypriority.run_list);
        //Reduce the number of internal tasks in the queue
        rq->mypriority.nr_running--;
        printk(KERN_INFO"+++[MYPRIORITY] Dequeue: success cpu=%d, nr_running=%d, pid=%d\n", cpu_of(rq), rq->mypriority.nr_running, p->pid);
        if (mypriority_rq->nr_running > 1) {
            struct sched_mypriority_entity *find_entity;
            struct list_head *head = mypriority_rq->queue.next;
            find_entity = container_of(head->next, struct sched_mypriority_entity, run_list);
            mypriority_rq->next_priority = find_entity->priority;
            printk("+++[MYPRIORITY]  Dequeue - NEXT PRIORITY VALUE mypriority_rq->next_priority=%d\n", mypriority_rq->next_priority);
        }
    }
    else {
    }
}
```

1. Enqueue
Pid를 이용해서 task에 우선 순위를 부여한 후 그 우선순위에 맞게 큐에 배치시킨다. 만약 가장 높은 우선 순위인 프로세스라면 CPU를 선점하여 빼앗아 올 수 있다.

2. Dequeue
넘겨받은 프로세스를 큐에서 제거한다. 만약 제거 이후에도 2개 이상의 프로세스가 존재하는 경우 두번째로 높은 우선순위 값을 재설정 한다.

3. Pick next task
현재 대기 중인 프로세스가 존재한다면 다음으로 실행 될 프로세스를 골라서 넘겨주게 되는데, 큐의 가장 맨 앞의 프로세스가 그 대상이 된다.

Part 3

테스트 프로그램 동작 흐름

STEP 1

Fork함수 호출을 통해 정해진 수만큼의 프로세스를 생성한다.

>>

STEP 2

메인에서 받은 인자 값을 확인해서 사용할 스케줄러를 확인하고 설정한다.
.

>>

STEP 3

각 프로세스 들은 주어진 업무를 실행하게 된다. 이번의 경우 result 변수에 1더하는 작업을 4000000번 반복하게 된다.

>>

STEP 4

해당 스케줄러 위에서 프로세스들이 실행되게 되고 각각의 프로세스 들은 일을 끝마치고 종료하게 된다.

Part 3

테스트 프로그램과 스케줄러 실행 결과 1

```
cpuset at [lst] cpu in child process(pid=1883) is succeed
cpuset at [lst] cpu in child process(pid=1882) is succeed
cpuset at [lst] cpu in child process(pid=1881) is succeed
cpuset at [lst] cpu in child process(pid=1880) is succeed
```

```
queuing mypriority - START pid=1881
queuing mypriority - LOCATE TAIL pid=1881 priority=31
Enqueue: success cpu=1, nr_running=1, pid=1881
pick_next_task: cpu=1, prev->pid=0,next_p->pid=1881,nr_running=1
aging_mypriority AGING ++        pid=1881 priority=32
aging_mypriority AGING ++        pid=1881 priority=33
aging_mypriority AGING ++        pid=1881 priority=34
aging_mypriority AGING ++        pid=1881 priority=35
```

```
aging_mypriority AGING ++        pid=1881 priority=71
aging_mypriority AGING ++        pid=1881 priority=72
queuing mypriority - START pid=1882
queuing mypriority - LOCATE!!!! pid=1882 my next=1881 priority=32
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=72
Enqueue: success cpu=1, nr_running=2, pid=1882
enqueue - PREEMPT: success cpu=1, nr_running=2, now_pid=1881
check_preempt_curr_mypriority
aging_mypriority AGING ++        pid=1881 priority=73
AGING -- > REPLACE BEFORE pid=1881 priority=73
queuing mypriority - START pid=1881
queuing mypriority - LOCATE TAIL pid=1881 priority=73
AGING -- > REPLACE AFTER pid=1881
pick_next_task: cpu=1, prev->pid=1881,next_p->pid=1882,nr_running=2
```

1. 테스트 프로그램이 동작하면서 4개(1883, 1882, 1881, 1880)의 프로세서를 생성한다.

2. 먼저 우선순위 31을 가진 1881번 프로세스가 빈 큐로 들어오게 되고 CPU를 획득하게 되고 이후 Aging 방식으로 인해 우선순위가 낮아지게 된다.

3. 우선 순위가 72까지로 떨어지게 될 무렵에 우선순위 32를 가진 1882 프로세스가 큐에 들어오게 되고 현재 가장 우선순위가 높기 때문에 큐의 맨 앞으로 배치된다.

이 때 Next Priority value 값이 갱신 되어 현재 큐에서 두번째로 높은 우선순위의 값을 담아둔다.

현재 실행 중이던 1881번 프로세스 보다 우선순위가 높은 프로세스가 들어왔기 때문에 선점이 이루어진다. Enqueue 되는 시점에서 바로 교체가 이루어지지는 않기 때문에 enqueue 된 후에도 한동안 1881번 프로세스가 실행되게 되어 Aging이 그대로 적용된다.

우선 Aging으로 인해 우선순위가 Next Priority value 보다 낮아지게 되면(숫자가 커지게) 큐에서 재배치(REPLACE 부분)를 시키는 방식을 택하였다. 그러나 선점을 당하는 경우 우선순위가 높은 프로세스가 자신의 앞쪽에 이미 배치되어 있는 상황이며 CPU를 실제로 뺏길 때까지는 딜레이도 존재하므로 불가피하게 불필요한 재배치가 동작하게 된다.

이 후 1881번 프로세스는 우선순위가 더 높은 1882번 프로세스에게 CPU를 뺏기게 된다.

테스트 프로그램과 스케줄러 실행 결과 2

```
aging_mypriority AGING ++        pid=1882 priority=68
aging_mypriority AGING ++        pid=1882 priority=69
queuing mypriority - START pid=1883
queuing mypriority - LOCATE!!!! pid=1883 my next=1882 priority=33
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=69
Enqueue: success cpu=1, nr_running=3, pid=1883
enqueue - PREEMPT: success cpu=1, nr_running=3, now_pid=1882
check_preempt_curr_mypriority
pick_next_task: cpu=1, prev->pid=1882,next_p->pid=1883,nr_running=3
```

4. 1882 프로세스의 우선순위가 69로 낮아지게 되었고 이 시점에 33 우선순위를 가진 1883번 프로세스가 큐에 들어온다. 그 결과 현재 큐의 우선순위는 head->33->69->72 이므로 Next Priority value 값이 69로 갱신된다.

이 경우도 선점을 당한 상황이므로 앞의 상황이 반복된다.

이 후 1883번 프로세스가 CPU를 획득하게 된다.

```
aging_mypriority AGING ++        pid=1883 priority=70
AGING -- > REPLACE BEFORE pid=1883 priority=70
queuing mypriority - START pid=1883
queuing mypriority - LOCATE!!!! pid=1883 my next=1881 priority=70
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=70
AGING -- > REPLACE AFTER pid=1883
queuing mypriority - START pid=1880
queuing mypriority - LOCATE!!!! pid=1880 my next=1882 priority=30
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=69
Enqueue: success cpu=1, nr_running=4, pid=1880
enqueue - PREEMPT: success cpu=1, nr_running=4, now_pid=1883
check_preempt_curr_mypriority
aging_mypriority AGING ++        pid=1883 priority=71
AGING -- > REPLACE BEFORE pid=1883 priority=71
queuing mypriority - START pid=1883
queuing mypriority - LOCATE!!!! pid=1883 my next=1881 priority=71
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=69
AGING -- > REPLACE AFTER pid=1883
pick_next_task: cpu=1, prev->pid=1883,next_p->pid=1880,nr_running=4
aging_mypriority AGING ++        pid=1880 priority=31
```

5. 83번 프로세스가 70까지 우선순위가 낮아지게 되어 Next Priority value 을 초과하는 상황이 발생한다. 즉 REPLACE 가 일어난다.

이 시점에 마지막 프로세스인 우선순위 30의 1880 이 큐에 들어와 선점하게 된다. 이 후 현재 큐의 우선순위 상황은 head ->30(1880)->69(1882)->70(1883)->72(1881) 가 되고 Next Priority value 의 값은 69로 갱신된다.

가장 앞 프로세스인 1880 프로세스가 이제 CPU를 차지하게 된다.

테스트 프로그램과 스케줄러 실행 결과 3

```
aging_mypriority AGING ++        pid=1880 priority=69
aging_mypriority AGING ++        pid=1880 priority=70
AGING -- > REPLACE BEFORE pid=1880 priority=70
queuing mypriority - START pid=1880
queuing mypriority - LOCATE!!!! pid=1880 my next=1883 priority=70
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=70
AGING -- > REPLACE AFTER pid=1880
Dequeue: success cpu=1, nr_running=3, pid=1880
```

```
pick_next_task: cpu=1, prev->pid=1880,next_p->pid=1882,nr_running=3
aging_mypriority AGING ++        pid=1882 priority=70
aging_mypriority AGING ++        pid=1882 priority=71
aging_mypriority AGING ++        pid=1882 priority=72
AGING -- > REPLACE BEFORE pid=1882 priority=72
queuing mypriority - START pid=1882
queuing mypriority - LOCATE!!!! pid=1882 my next=1881 priority=72
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=72
AGING -- > REPLACE AFTER pid=1882
```

6. 1880 프로세스가 Aging 으로 인해 70까지 우선순위가 낮아지게 되어 다시 재배치가 일어난다. 그 결과 head ->69(1882)-> 70(1880)-> 70(1883)->72(1881)로 큐내부가 배치되고 Next Priority value 의 값은 70로 갱신된다.

이때 아직 CPU를 보유중이던 1880 프로세스가 종료되어 Dequeue 작업이 일어난다.

7. 1882 번 프로세스가 CPU를 차지하게 되고 Aging으로 인해 우선순위가 72까지 낮아지게 되어 REPLACE 작업이 일어난다.

현재 큐의 상태는 head -> 70(1883) ->72(1882)->72(1881) 가 되고 Next Priority value 의 값은 72로 갱신된다.

8. 이 후 모든 프로세스가 일을 마칠 때까지 이 과정들을 반복한다.

Aging 따른 차이

	선점형
Aging 미사용	현재 실행 중인 것보다 높은 우선순위의 프로세스가 들어오게 되면 현재 실행 중인 프로세스의 작업을 중단하고 CPU를 획득할 수 있다. Aging을 사용하지 않기때문에 현재 프로세스가 일을 끝마치기 전까지 큐의 프로세스들은 기다려야 한다.

```
queuing mypriority - START pid=1875
queuing mypriority - LOCATE TAIL pid=1875 priority=25
Enqueue: success cpu=1, nr_running=1, pid=1875
pick_next_task: cpu=1, prev->pid=0,next_p->pid=1875,nr_running=1
queuing mypriority - START pid=1874
queuing mypriority - LOCATE!!!! pid=1874 my next=1875 priority=24
queuing mypriority - NEXT PRIORITY VALUE mypriority_rq->next_priority=25
Enqueue: success cpu=1, nr_running=2, pid=1874
enqueue - PREEMPT: success cpu=1, nr_running=2, now_pid=1875
check_preempt_curr_mypriority
pick_next_task: cpu=1, prev->pid=1875,next_p->pid=1874,nr_running=2
Dequeue: success cpu=1, nr_running=1, pid=1874
pick_next_task: cpu=1, prev->pid=1874,next_p->pid=1875,nr_running=1
```

Ova 파일 링크 https://drive.google.com/file/d/1lSEuo0_VyjpR5nvT5AA8O8vkVlZj0ksq/view?usp=sharing