1. 스케줄러 설계에 대한 설명

- 설계
  - Scheduler 우선순위
    - RT > myprio > mysched > myrr > fair > idle
  - Priority 부여 방법
    - 해당 process의 pid를 우선순위로 부여
    - 낮은 번호일수록 우선순위가 높다
  - 선점형 Priority Scheduler
    - Enqueue되는 task 또는 rq에서 대기 중 우선순위가 증가한 task 중에서 current task 보다 높은 우선 순위를 가지는 task가 발견되면 즉시 current task를 해당 task로 교체한다
  - Aging 방법
    - 설정한 시간이 지나면 priority를 1만큼 감소시켜 우선순위를 높여준다
      - EX) 10초 경과 마다 우선순위가 증가하는 경우, 30초를 대기하게 되면 priority 값이 3만큼 감소
      - 즉, 우선순위가 3만큼 증가한다고 볼 수 있다
    - Starvation 때문에 우선순위가 높아진 task는 current task가 되면 이후 교체되어 나오는 순간 자신의 pid로 초기화된 우선순위를 가지게 된다
      - EX) pid = 1905의 task가 1800의 높아진 우선순위를 가지고 current task가 된 이후 교체되어 나오는 순간 다시 초기의 priority와 동일한 1905의 우선순위를 가지게 된다
  - enqueue_task_myprio
    - Task에게 기다리기 시작한(wait_time) 시간과 priority를 부여
      - Wait_time과 priority는 sched_myprio_entity에 존재
    - Enqueue된 task의 우선순위를 확인하여 current task의 우선순위 보다 높으면 즉시 rescheduling하여 enqueue된 task를 current task로 만들어 준다
  - dequeue_task_myprio
    - Rq에서 해당 task 삭제
  - pick_next_task_myprio
    - Rq에서 제일 앞의 task를 선택 후 return
  - task_tick_myprio
    - 각각의 task가 rq에서 기다린 시간 확인
    - 기다린 시간이 사전에 설정한 특정 값의 시간보다 큰 경우 우선순위를 증가시킨다
    - 증가된 우선순위가 current task의 우선순위 보다 높은 경우 update_curr_myprio 호출
  - update_curr_myprio
    - Rq를 순회하며 우선순위가 제일 높은 task 선택
      - 우선순위가 동일한 task가 존재할 경우 먼저 발견된 task를 선택
    - 선택된 task의 우선순위가 current task의 우선순위보다 높은 경우
      - 선택된 task의 위치를 rq의 제일 앞으로 변경
      - Current task의 priority를 current task의 pid로 변경
      - Current task의 기다리기 시작한 시각을 현재 시각으로 변경
      - Rescheduling 요청
    - 선택된 task의 우선순위가 current task의 우선순위보다 같거나 작은 경우
      - Current task가 변경되지 않았다는 메시지 출력

2. 스케줄러 구현에 대한 설명

- 구현
  - enqueue_task_myprio

```
myprio_se->wait_time = jiffies;
myprio_se->priority = p->pid;
```

    - wait_time(대기 시작 시각)은 linux에서 제공하는 jiffies를 활용
    - Priority로 자신의 pid 부여

```
list_add_tail(&p->myprio.run_list, &rq->myprio.queue);

//TODO: Check priority
update_curr_myprio(rq);
```

    - Queue에 enqueue된 task 추가 후 update_curr_myprio 호출

  - dequeue_task_myprio

```
list_del_init(&p->myprio.run_list);
```

  - pick_next_task_myprio

```
selected_se = list_entry(myprio_rq->queue.next, struct sched_myprio_entity, run_list);
selected_p = container_of(selected_se, struct task_struct, myprio);
```

    - Queue 안의 제일 앞에 위치한 task를 선택 후 return

  - task_tick_myprio

```
list_for_each_entry(myprio_se, &rq->myprio.queue, run_list) {
    if (time_after(jiffies - myprio_se->wait_time, time_to_raise_priority)) {
        temp_p = container_of(myprio_se, struct task_struct, myprio);
        if (temp_p->pid != p->pid) {
            printk(KERN_INFO "\t\t***[MYPRIO] priority raise: pid=%d, old prio
            myprio_se->priority--;
            myprio_se->wait_time = jiffies;
            if (myprio_se->priority < curr_se->priority)
                update_curr_myprio(rq);
        }
    }
}
```

    - time_to_raise_priority : 초기에 설정된 전역 변수
    - 대기 시간은 (현재 시각(jiffies) – 대기 시작 시각(wait_time))으로 확인.
    - Time_after(a, b) : a가 더 큰 경우 true를 반환
    - temp_p->pid != p->pid : queue를 순회하는 중 자신(current task)과 마주하면 skip한다
    - task의 대기 시간이 time_to_raise_priority 보다 큰 경우 우선순위를 증가시키고 대기 시작 시각(wait_time)을 현재 시각(jiffies)로 변경한다
    - 증가된 우선순위가 current task의 우선순위 보다 높은 경우 update_curr_myprio 호출

  - update_curr_myprio

```
int temp_priority = 10000;
list_for_each_entry(temp_se, &rq->myprio.queue, run_list) {
if (temp_priority > temp_se->priority) {
    temp_priority = temp_se->priority;
    highest_prio_se = temp_se;
    }
}

highest_prio_p = container_of(highest_prio_se, struct task_struct, myprio);
if (curr_se->priority > highest_prio_se->priority) {
    // Step 2
    list_del(&highest_prio_p->myprio.run_list);
    list_add(&highest_prio_p->myprio.run_list, &rq->myprio.queue);

    printk(KERN_INFO "\t***[MYPRIO] update_curr_myprio: Old Curr=%

    // Step 3
    curr_se->wait_time = jiffies;
    curr_se->priority = curr_p->pid;

    // Step 4
    resched_curr(rq);
}
```

    - highest_prio_p : 우선순위가 가장 높은 task의 task_struct
    - highest_prio_se : 우선순위가 가장 높은 task의 entity
    - curr_p : current task의 task_struct
    - curr_se : current task의 entity

3. 스케줄러 결과에 대한 설명

- 결과
  - enqueue_task_myprio

```
[   29.885673] ***[MYPRIO] Enqueue Start
[   29.885679]  ***[MYPRIO] update_curr_myprio: Old Curr=1871, Old Prio=1871, New Curr=1870, New Prio=1870
[   29.885742] ***[MYPRIO] Enqueue: success cpu=1, nr_running=4, pid=1870, prio=1870
[   29.885748] ***[MYPRIO] Enqueue End
```

    - Enqueue 시 current task보다 자신의 우선순위가 높은 경우 즉시 current task 교체
    - Current task : pid = 1871, priority = 1871
    - Enqueue task : pid = 1870, priority = 1870
    - Current task를 enqueue된 task로 즉시 교체

  - dequeue_task_myprio

```
***[MYPRIO] Dequeue: start
***[MYPRIO] dequeue: success cpu=1, nr_running=1, pid=1876, prio=1876
***[MYPRIO] Dequeue: end
```

```
[   42.436134]  ***[MYPRIO] update_curr_myprio: Old Curr=1870, Old Prio=1865, New Curr=1871, New Prio=1864
[   42.436232] ***[MYPRIO] Pick_next_task: cpu=1, prev->pid=1870, next_p->pid=1871, next_p prio=1864, nr_running=4
[   42.436254] ***[MYPRIO] Pick Next Task Myprio End
[   43.436309]          ***[MYPRIO] priority raise: pid=1870, old prio=1870, new prio=1869
[   43.436313]          ***[MYPRIO] priority raise: pid=1873, old prio=1872, new prio=1871
[   43.436316]          ***[MYPRIO] priority raise: pid=1872, old prio=1868, new prio=1867
[   44.437045]          ***[MYPRIO] priority raise: pid=1870, old prio=1869, new prio=1868
[   44.437092]          ***[MYPRIO] priority raise: pid=1873, old prio=1871, new prio=1870
[   44.437132]          ***[MYPRIO] priority raise: pid=1872, old prio=1867, new prio=1866
[   45.437525]          ***[MYPRIO] priority raise: pid=1870, old prio=1868, new prio=1867
[   45.437563]          ***[MYPRIO] priority raise: pid=1873, old prio=1870, new prio=1869
[   45.437598]          ***[MYPRIO] priority raise: pid=1872, old prio=1866, new prio=1865
[   46.437778]          ***[MYPRIO] priority raise: pid=1870, old prio=1867, new prio=1866
[   46.437782]          ***[MYPRIO] priority raise: pid=1873, old prio=1869, new prio=1868
[   46.437784]          ***[MYPRIO] priority raise: pid=1872, old prio=1865, new prio=1864
[   47.438324]          ***[MYPRIO] priority raise: pid=1870, old prio=1866, new prio=1865
[   47.438328]          ***[MYPRIO] priority raise: pid=1873, old prio=1868, new prio=1867
[   47.438331]          ***[MYPRIO] priority raise: pid=1872, old prio=1864, new prio=1863
[   47.438333]  ***[MYPRIO] update_curr_myprio: Old Curr=1871, Old Prio=1864, New Curr=1872, New Prio=1863
[   47.438344] ***[MYPRIO] Pick_next_task: cpu=1, prev->pid=1871, next_p->pid=1872, next_p prio=1863, nr_running=4
[   47.438347] ***[MYPRIO] Pick Next Task Myprio End
```

  - [42.436134]
    - Pid=1871의 우선순위가 1864로 증가하여 기존의 current task (pid=1870, 우선순위 1865)보다 우선순위가 높아져 즉시 pid=1871이 current task로 교체됨
  - [47.438333]
    - Pid=1872의 우선순위가 1863으로 증가하여 기존의 current task (pid=1871, 우선순위 1864)보다 우선순위가 높아져 즉시 pid=1872가 current task로 교체됨

  - Task_tick_myprio
    - 1초마다 호출되어 대기중인 task들의 우선순위를 1씩 높여준다
    - Current task보다 우선순위가 높아진 task가 등장 하는 경우 update_curr_myprio 호출

  - update_curr_myprio
    - Queue에서 제일 높은 우선순위의 task를 선택하여 queue의 제일 앞으로 이동 시킨 뒤 rescheduling 한다.

  - pick_next_task_myprio
    - Queue에서 제일 앞에 위치한 task의 task struct를 return
    - update_curr_myprio에서 우선순위가 가장 높은 task를 queue 제일 앞으로 보냈기 때문에 의도한 task가 선택됨

4. Aging 기법 상세 설명

- Jiffies
  - Linux에서 제공하는 jiffies를 사용하여 Aging 구현

- time_to_raise_priority 변수
  - 사전에 미리 정해 놓은 값 (현재 코드에서는 HZ로 설정)
  - Time_to_raise_priority의 값 이상 대기 시 우선순위를 높인다

```
unsigned long time_to_raise_priority = HZ;
```

- Task가 queue에서 대기한 시간 측정
  - wait_time 변수 사용
    - Entity 안에 선언되어 있는 unsigned long 타입의 변수
    - 기다리기 시작한 시각으로 해석
    - Jiffies로 초기화 되는 순간
      - Enqueue되는 순간
      - Current task로 동작하다가 queue로 돌아가 대기하기 시작하는 순간
      - 우선순위가 증가하는 순간

```
struct sched_myprio_entity {
        struct list_head run_list;
        unsigned int priority;
        unsigned long wait_time;
};
```

  - Jiffies로 초기화 되는 순간들의 이유
    - Enqueue되는 순간
      - Current task보다 낮은 우선순위로 enqueue되는 경우 바로 대기상태로 진입하기 때문에 enqueue되는 순간 wait_time을 jiffies로 초기화

```
static void enqueue_task_myprio(struct
    struct sched_myprio_entity *myprio_

    printk(KERN_INFO "***[MYPRIO] Enqu

    myprio_se->wait_time = jiffies;
```

    - Current task로 동작하다가 queue로 돌아가 대기하기 시작하는 순간
      - Current task가 된 순간부터 entity에 있는 wait_time의 값은 의미가 없게 된다
      - 따라서 Current task에서 queue로 돌아와 대기하기 시작하는 순간을 wait_time으로 초기화

```
if (curr_se->priority > highest_prio_se->priority) {
    // Step 2
    list_del(&highest_prio_p->myprio.run_list);
    list_add(&highest_prio_p->myprio.run_list, &rq->myprio.queue);

    printk(KERN_INFO "\t***[MYPRIO] update_curr_myprio: Old Curr=%

    // Step 3
    curr_se->wait_time = jiffies;
    curr_se->priority = curr_p->pid;
```

    - 우선순위가 증가하는 순간
      - 우선순위가 증가한 뒤 다시 time_to_raise_priority 만큼 대기한 뒤 우선순위를 증가 시키기 때문

```
void task_tick_myprio(struct rq *rq, struct task_struct *p, int queued) {
    struct sched_myprio_entity *myprio_se = NULL;
    struct sched_myprio_entity *curr_se = &p->myprio;
    struct task_struct *temp_p;

    list_for_each_entry(myprio_se, &rq->myprio.queue, run_list) {
        if (time_after(jiffies - myprio_se->wait_time, time_to_raise_priority)) {
            temp_p = container_of(myprio_se, struct task_struct, myprio);
            if (temp_p->pid != p->pid) {
                printk(KERN_INFO "\t\t***[MYPRIO] priority raise: pid=%d, old prio=
                myprio_se->priority--;
                myprio_se->wait_time = jiffies;
                if (myprio_se->priority < curr_se->priority)
                    update_curr_myprio(rq);
            }
        }
    }
```

4. Aging 기법 상세 설명

- Task가 queue에서 대기한 시간 측정
  - OS가 task_tick_myprio 호출할 때 마다 대기중인 task들의 대기 시간 확인

  - (Jiffies - wait_time)으로 대기 시간 측정
    - Jiffies : linux 실행 시점부터 지금까지의 시간
    - Wait_time : linux 실행 시점부터 wait_time이 초기화된 시점까지의 시간
    - 따라서 (jiffies - wait_time)은 대기 시간으로 볼 수 있다

```
if (time_after(jiffies - myprio_se->wait_time, time_to_raise_priority)) {
```

- 우선순위가 Current task보다 높아진 경우
  - Update_curr_myprio 호출
    - Update_curr_myprio는 제일 높은 우선순위를 갖는 task를 queue 제일 앞으로 이동시킴
    - Pick_next_task_myprio는 queue 제일 앞의 task를 return하기 때문에 위의 로직 성립

```
if (temp_p->pid != p->pid) {
    printk(KERN_INFO "\t\t***[MYPRIO] priority rai
    myprio_se->priority--;
    myprio_se->wait_time = jiffies;
    if (myprio_se->priority < curr_se->priority)
        update_curr_myprio(rq);
}
```

5. 테스트 유저 프로그램 설명

- newclass_new.c
  - ./newclass_new p
  - 위의 명령어로 프로그램 실행
  - P 입력 시 myprio scheduler 동작

```
root@2020osclass:~/newclass# ls
Makefile  newclass  newclass2  newclass2.c  newclass3.c  newclass.c  newclass_new  newclass_new.c
root@2020osclass:~/newclass# ./newclass_new p
```

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

    ret = sched_setattr(my_pid, &attr, flags);
    if (ret != 0) {
        perror("sched_setattr");
        exit(1);
    }
}
```

- SCHED_MYPRIO
  - #define SCHED_MYPRIO 9로 설정

```c
#define SCHED_MYPRIO    9
```

- Child processor work

```c
/* child process work */
int j = 0;
for(j = 0; j < 20; j++) {
    int i = 0;
    int result = 0;
    for(i = 0; i < 200000000; i++)
    {
        result += 1;
    }
    printf("pid=%d:\tresult=%d\n", my_pid, result);
    //sleep(1);
}

exit(1);
```

  - Dequeue되지 않고 queue 내부에서 task들의 지속적인 priority 변화에 따른 current task 변화를 보기 위해 for문 안에서 sleep을 실행하지 않았습니다
  - 각각의 child process는 20번의 200000000을 더하는 작업 수행

- 나머지는 조교님께서 제공해 주신 유저 프로그램과 동일합니다

6. 테스트 유저 프로그램 결과 설명

```
root@2020osclass:~# dmesg --clear
root@2020osclass:~# ./newclass/newclass_new p
cpuset at [0th] cpu in parent process(pid=1869) is succeed
Child's PID = 1870
Child's PID = 1871
Child's PID = 1872
Child's PID = 1873
forking 4 tasks is completed
***[NEWCLASS] Select myprio scheduling class
***[NEWCLASS] Select myprio scheduling class
***[NEWCLASS] Select myprio scheduling class
***[NEWCLASS] Select myprio scheduling class
root@2020osclass:~# cpuset at [1st] cpu in child process(pid=1870) is succeed
cpuset at [1st] cpu in child process(pid=1871) is succeed
cpuset at [1st] cpu in child process(pid=1872) is succeed
cpuset at [1st] cpu in child process(pid=1873) is succeed
```

```
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1872:       result=200000000
```

```
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1873:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1871:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1872:       result=200000000
pid=1870:       result=200000000
pid=1870:       result=200000000
```

4개의 Child Process 생성 (1870, 1871, 1872, 1873)

각각 200000000을 더하는 연산을 20번씩 실행

총 80줄의 결과 출력

7. Aging VS No-Aging

- Aging을 적용한 Priority Scheduler
  - 우선순위가 낮은 task의 우선순위가 일정 시간이 지남에 따라 증가하게 됨으로 Starvation 상태가 되지 않습니다
  - 높은 우선순위의 task부터 실행되지만 낮은 우선순위의 task도 시간이 지남에 따라 우선순위가 높아지기 때문에 Priority Scheduler인 것을 감안하면 비교적 균등하게 기회를 가질 수 있습니다
  - EX) 크기가 낮을 수록 우선순위가 높은 정책에서의 예시
    - 2개의 프로세스 존재 : a(10), b(1000) 괄호 안은 우선순위
    - a가 무한정 실행되고 우선순위는 1초 대기마다 1씩 증가한다고 가정
    - 1) a가 CPU 독점
    - 2) 우선순위가 낮은 b는 CPU를 점유하지 못할 것 같지만 1초마다 우선순위가 1씩 증가하기 때문에 990초 후 CPU 점유 가능
    - 따라서 Starvation 발생하지 않음

- Aging을 적용하지 않은 Priority Scheduler
  - Aging을 적용하지 않은 경우 우선순위가 낮은 task는 계속 동작되지 않고 Starvation에 빠지는 현상이 발생합니다
  - Starvation에 빠진 task는 다른 task들이 모두 수행된 후에 자신의 일을 수행 할 수 있습니다
  - 하지만 지속적인 (계속 멈추지 않는) 작업 속에서 자신보다 우선순위가 높은 task들만 계속 enqueue된다면 해당 task는 영원히 일을 못하게 됩니다. 이 경우를 Starvation이라고 합니다
  - EX) 크기가 낮을 수록 우선순위가 높은 정책에서의 예시
    - 2개의 프로세스 존재 : a(10), b(1000) 괄호 안은 우선순위
    - a가 무한정 실행된다고 가정
    - 1) a가 CPU 독점
    - 2) 우선순위가 낮은 b는 CPU를 점유하지 못한다
    - 따라서 Starvation 발생
  - EX) 크기가 낮을 수록 우선순위가 높은 정책에서의 예시
    - 2개의 프로세스 존재 : a(10), b(1000) 괄호 안은 우선순위
    - A가 10초 동안 실행된다고 가정
    - (Aging 기법이 적용되지 않았지만 Starvation은 발생하지 않는 경우)
    - 1) a가 CPU 독점
    - 2) 우선순위가 낮은 b는 10초 후 a의 모든 작업이 끝난 뒤 부터 접근가능

- Starvation
  - task가 CPU를 할당 받지 못하고 계속 대기 상태에 머물게 되어 일을 하지 못하는 상황

8. OVA 파일 링크

- https://drive.google.com/file/d/1dYkHkCGAd912jsKAGbmpsfhxjL2SBwn-/view?usp=sharing