-핀토스 프로젝트

과 목 명 : 운영체제 1분반
담당 교수 : 손성훈 교수님
학 과 : 경영학과, 컴퓨터과학과
학 번 : 201210569, 201311073
성 명 : 김성환, 이규한
제출 일자 : 2017.12.05

목차

Ⅰ. Timer_sleep()함수

1. 문제인식 (Timer_sleep 함수의 waiting 문제)

(timer_sleep 함수)

```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

timer_sleep() 함수는 호출한 스레드의 수행을 호출 시점부터 ticks 만큼 지연하는데 사용된다.

이 방식은 busy-waiting 방식이라고 하며, time_sleep() 을 호출한 스레드는 while 문을 실행하는데 호출한 시점부터 현재까지의 시간(timer_elapsed(start) 리턴 값)이 ticks보다 작다면 thread_yield(); 코드를 실행하게 되어 현재 가지고 있는 CPU를 반납하게 된다. 또한 동시에 이 스레드는 Ready Queue로 이동하게 된다. 여기서 문제가 되는 것은 스레드가 CPU를 빼앗기고 Ready Queue에 있는데, 빼앗긴 동안에도 계속 timer_sleep() 함수 내의 while문을 반복하여 실행하고 있는 중이다. 다시 말해서 CPU를 일부 점유하고 있는 것 이다. 이는 앞뒤가 맞지 않는 상황이기 때문에 Alarm을 개선할 필요성이 있다고 생각되었다. timer_sleep() 의 동작과정은 아래의 그림과 같다.

2. 해결방안과 과정

개선 방안을 오랜 시간 생각한 끝에 나온 알고리즘은 생각보다 어렵지 않았다. 기존에 timer_sleep()이 while문을 통해 thread_yield()를 반복했던 것 대신 timer_sleep() 함수 호출 시 thread 들을 block 시키고, 그 호출 시각에 ticks(수행 지연 시간)를 더한 시각을 경과하게 된다면 unblock시켜서 Ready queue로 이동하도록 한다. 이 기능을 수행하기 위해서는 sleep queue(waiting queue의 역할)가 필수 불가결하다. queue 의 역할은 block된 스레드들이 대기하는 리스트로써 사용된다. 우선 적으로 위에서 말한 sleep 하고 있는 thread를 깨울 (sleep list -> Ready queue 이동) 시킬 알람 시각을 구해야 한다. 또한 이보다 스레드마다 언제 깨울지를 저장하는 멤버변수를 TCB에 정의해야 하는데 이러한 TCB는 thread.h에 구조체

형태로 정의되어 있다. 이 변수를 good_morning 이라고 고정 시키도록 하겠다.

3. 소스코드

(thread.h)

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    int64_t good_morning;

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);

  thread_sleep(start + ticks);

  /*while (timer_elapsed (start) < ticks)
    thread_yield (); */
}
```

위에서 알람 시각은 timer_sleep 함수 호출 당시의 시각이 저장된 start와 수행 지연 시간 ticks를 더한 값으로 표현 가능하다. 그리고 이 스레드를 깨울 알람 시각을 인자로 받고, 현재 running 스레드를 sleep_list에 넣어야 하는데, sleep_list를 선언하고 초기화하는 과정이 먼저 이루어 져야 한다. sleep_list의 초기화는 thread_init 함수에서 list_init 함수를 통해 부팅과 동시에 진행하도록 한다.

다음으로 sleep_list로 넣음과 동시에 그 스레드를 block 시키는 thread_sleep 함수를 정의하여 추가해야 한다. 이 함수의 호출은 timer_sleep 함수에서 진행된다. thread_sleep 함수의 정의는 다음과 같다. 코드는 아래와 같다

(thread.c)

```
/* List of processes in THREAD_READY state, that is, processes
   that are ready to run but not actually running. */
static struct list ready_list;

static struct list sleep_list;
```

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);
  list_init (&sleep_list);

  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread ();
  init_thread (initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid ();
}
```

(thread.c)

```
void thread_sleep(int64_t our_ticks)
{
        enum intr_level old_level;
        struct thread *cur =  thread_current();
        old level = intr_disable();
        ASSERT(cur != idle_thread);
        cur->good_morning = our_ticks;
        list_insert_ordered(&sleep_list, &cur->elem,thread_less_sort, NULL);
        therad_block();
        intr_set_level(old_level);
}
```

위의 그림 ①번에서 enum intr_level old_level 에서는 현재 인터럽트 레벨을 담은 변수를 선언하고 intr_disable()을 통해서 interrupt를 disable 시키면서 인터럽트 레벨을 저장한다.
인터럽트를 사용하는 이유은 ②번 thread_block 함수 실행이 끝나면 ③번 intr_set_level을 통해서 원래의 인터럽트 레벨로 돌아온다. ④번의 thread__current()의 리턴 값 즉, 현재 running 중인 스레드의 시작 주소를 cur 구조체 포인터에 담는다. 이후 ⑤번에서 cur이 idle_thread는 아닌지 검사할 필요가 있으며, ⑥번 에서는 cur의 TCB에 멤버인 wake_up에 매개변수로 받은 real_ticks(알람 시각)을 입력한다.

(list.h)

```
172 void list_insert_ordered (struct list *, struct list_elem *,
173                           list_less_func *, void *aux);
```

⑦번의 list_insert_ordered 함수의 형태를 먼저 살펴보면, 4개의 인자를 요구하고 있음이 확실하다.
sleep_list의 주소를 첫 번째 인자로 전달하여 그곳에 현재 스레드의 포인터를 연결하는데, 이때의 정렬 기준은 세 번째 인자로 주어진 함수 포인터이다. 그리고 네 번째 인자는 현재는 무의미 하므로 NULL로 전달하였다. ⑦에서 세 번째 인자로 thread_less_sort라는 정렬 기준이 되는 함수를 전달하였는데 그 내용은 다음과 같

다. 기존 생각으로는 함수 만들어서 전달하면 될 줄 알았는데 미리 typedef로 정의되어 있는 typedef bool list_less_func(const struct list_elem *a, const struct list_elem *b, void *aux) 형태로 구현해야 된다는 사실을 알아내는데 상당한 시간이 소요되었다.

<thread.c>

```
bool theread_less_sort(const struct list_elem *A,const struct list_elem *B)
{
        struct thread *A_thread = list_entry(A, struct thread, elem);   ← 1
        struct thread *B_thread = list_entry(B, struct thread, elem);   ← 2
        return A_thread->good_morning < B_thread->good_morning;
}
```

①②번의 list_entry 함수는 첫 번째 인자로 받은 스레드의 TCB 시작주소를 리턴하고 있다. 이 정렬이 핵심이라고 할 수 있는데, 교수님께서 강의시간에 스레드를 block 시켜 sleep_list에 매달 때 정렬을 하여 리스트의 맨 앞부분만 비교하도록 했으면 좋겠다고 하셨다. 여기서 비교란 나중에 thread_wake 함수를 이용하여 sleep_list에 있는 스레드를 현재 ticks와 비교하여 ready queue로 보내야 할 때의 비교를 뜻한다. 아무리 생각해도 리스트의 데이터들을 모두 비교하는 것 보다 훨씬 효율적이기 때문에 정렬을 구현해 보았다. 이는 오름차순으로 정렬을 하면 맨 앞 스레드보다 뒤에 있는 스레드들의 wake_up은 맨 앞 스레드의 wake_up 값보다 항상 크기 때문이다.

thread_wake 함수는 ticks가 증가할 때마다 호출 되어 sleep_list 내의 스레드들의 wake_up 값과 비교를 하여 ticks이 wake_up 값보다 크거나 같아진다면 일어날 시간이 된 것이므로 thread_unblock 함수를 이용하여 ready queue의 맨 뒤로 이동시켜주는 함수이다. 이를 구현한 함수 정의는 다음과 같다.

<thread.c>

①에서 sleep_list에 무언가 스레드가 존재할 때 동안 계속 해당 코드들을 반복하게끔 반복문을 배치한다. ② cur에는 sleep_list의 맨 앞에 존재하는 스레드의 TCB 시작 주소를 저장하고, if문에서는 이 스레드의 wake_up 값과 비교하여 알람 시각이 되었는지 확인 후 list_pop_front를 통해 맨 앞 스레드를 sleep_list로부터 꺼낸다. 그 다음, thread_unblock에 현재 스레드 TCB의 시작 주소를 전달하여 ready queue의 맨 뒤에 매달아준다. if문의 조건을 만족하지 않는다면 그냥 반복문을 탈출하고 다

음 timer 인터럽트 발생 시 다시 thread_wake 함수 호출을 기다린다. 다음 코드는

```
void thread_good(int64_t ticks)
{
        struct therad *cur;

        while(!list_empty(&sleep_list))
        {
         cur = list_entry(list_front(&sleep_list), struct thread, elem);
         if(ticks >= cur->good_morning)
         {
          list_pop_front(&sleep_list);
          thread_unblock(cur);
         }
         else
         {
           break;
         }
        }
}
```

① ②

timer_interrupt 함수에서 매 ticks가 증가할 때 마다 thread_wake 함수가 호출되게 끔 추가한 코드이다. 마지막으로 추가한 함수들은 헤더 파일에 prototype을 선언을 해야 컴파일이 가능하다.

<timer.c>

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  thread_good(ticks);
}
```

<thread.h>

```
void thread_sleep(int64_t our_ticks);
void thread_good(int64_t ticks);
bool thread_less_sort(const struct list_elem *A, const struct list_elem *B, void *aux);
```

4. 결과

- alarm-multiple

```
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 870 ticks
Thread: 550 idle ticks, 322 kernel ticks, 0 user ticks
Console: 2952 characters output
Keyboard: 0 keys pressed
Powering off...
================================================================================
```

위의 빨간 네모를 보면, 기존에는 스레드가 sleep 상태임에도 CPU를 점유하고 있기 때문에 idle thread를 점유하지 않는 것과 달리 개선 후에는 위 그림과 같이 CPU가 쉬는 동안 idle thread에 할당 되어 있는 것을 확인할 수 있다. 이는 busy-waiting을 완벽히 개선했다고 볼 수 있다.

- alarm-single

```
Executing 'alarm-single':
(alarm-single) begin
(alarm-single) Creating 5 threads to sleep 1 times each.
(alarm-single) Thread 0 sleeps 10 ticks each time,
(alarm-single) thread 1 sleeps 20 ticks each time, and so on.
(alarm-single) If successful, product of iteration count and
(alarm-single) sleep duration will appear in nondescending order.
(alarm-single) thread 0: duration=10, iteration=1, product=10
(alarm-single) thread 1: duration=20, iteration=1, product=20
(alarm-single) thread 2: duration=30, iteration=1, product=30
(alarm-single) thread 3: duration=40, iteration=1, product=40
(alarm-single) thread 4: duration=50, iteration=1, product=50
(alarm-single) end
Execution of 'alarm-single' complete.
```

- alarm-simultaneous

```
Executing 'alarm-simultaneous':
(alarm-simultaneous) begin
(alarm-simultaneous) Creating 3 threads to sleep 5 times each.
(alarm-simultaneous) Each thread sleeps 10 ticks each time.
(alarm-simultaneous) Within an iteration, all threads should wake up on the same tick.
(alarm-simultaneous) iteration 0, thread 0: woke up after 10 ticks
(alarm-simultaneous) iteration 0, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 0, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 1, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 1, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 1, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 2, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 2, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 2, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 3, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 3, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 3, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 4, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 4, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 4, thread 2: woke up 0 ticks later
(alarm-simultaneous) end
Execution of 'alarm-simultaneous' complete.
```

- alarm-priority

```
Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
```

- alarm-zero

```
Executing 'alarm-zero':
(alarm-zero) begin
(alarm-zero) PASS
(alarm-zero) end
Execution of 'alarm-zero' complete.
```

- alarm-negative

```
Executing 'alarm-negative':
(alarm-negative) begin
(alarm-negative) PASS
(alarm-negative) end
Execution of 'alarm-negative' complete.
```

alarm-priority 테스트는 아직 priority 스케줄링 구현이 되어 있지 않아서 정상동작 하지 않았지만

나머지 함수들은 모두 timer_sleep 함수를 사용해서 pintos 커널뿐만 아니라 timer_sleep 함수가 수정된 후에도 이상 없이 정상동작 하므로 결과는 성공적이라 할 수 있다.

Ⅱ. Priority Scheduling

1. 문제인식

지금 우리가 사용하는 pintos는 Round-robin 스케줄링 방식으로 되어있다. 매 타이 머 인터럽트 발생 때마다 TIME_SLICE만큼 경과하면 Ready queue의 맨 마지막에 매달고 있다. 또한 최초 스레드 생성 시, thread_unblock() 함수가 호출될 때 Ready queue의 맨 마지막에 매달고 있다. 이렇게 항상 Ready queue의 마지막에 스레드를 매다는 것은 효율적이지 못하다. 왜냐하면 중요한 처리를 해야 하는 스레드가 있다면 항상 자기의 차례를 기다려야 하므로 Round-robin 방식은 적절하지 않기 때문이다.

2.해결방안과 과정

위에서 말한 것처럼 마지막에 스레드를 매다는 것은 비효율적이다. 그렇게 때문에 중요한 작업을 먼저 처리하는 것은 중요하다. 이를 위해서는 다음과 같은 특징들을 구현하여 해결해야한다.

첫째, Ready queue에 마지막에 삽입하던 방식을 TCB내에 있는 priority에 따라 리스트를 정렬해 주는 것이 필요하다. 이것이 priority 스케줄링 방식이다.

또한 과제의 목표에 맞게 Preemptive dynamic priority scheduling을 구현해야 한다.

둘째, preemption이 필요하다. 이것은 스레드가 Ready queue로 들어올 때, 이것이 running 스레드의 우선순위보다 높을 경우, CPU를 빼앗도록 하는 것이다. 이를 구현하기 위하여 스레드가 Ready queue로 들어오기 위해 실행되는 thread_unblock 함수를 수정해야한다. 마찬가지로 thread_yield 함수도 수정해야한다.

셋째, 역동성을 보장하기 위하여 스레드가 자신의 priority 값을 얻거나 변경할 수 있게 하도록 thread_set_priority 함수를 새로이 구현한다.

넷째, semaphore, lock 등을 가지고 waiting queue에 존재할 때도 우선순위에 따라 정렬된 상태로 유지해야한다.

먼저, timer.c에 있는 timer_interrupt 함수를 수정해야한다. 내용은 다음과 같다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  |
  thread_good(ticks);
}
```

앞의 'Ⅰ. Alarm clock의 개선'에서 진행했던 timer.c 의 tread_ticks 함수를 삭제했다. 이는 우선순위 스케줄링을 구현함에 있어서 timer interrupt가 발생해도 thread_tick 함수를 호출하지 않기 위함이다.

다음으로, thread_unblock 함수를 수정해야한다.

<thread.c>

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;
  struct therad *cur = thread_current();
  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_insert_ordered(&ready_list, &t->elem, thread_priority, NULL);   ①
  t->status = THREAD_READY;
  if(cur != idle_thread && cur -> priority < t->priority)   ②
   thread_yield();   ③
  intr_set_level (old_level);
}
```

기존의 list_push_back 함수처럼 ready queue의 맨 뒤에 넣는 것이 아니라 ①에서처럼 우선순위에 따라 정렬하여 넣도록 해야 한다. 세 번째 인자인 thread_priority_sort 함수는 우선순위에 따라 레디 큐를 정렬하기 위하여 구현한 함수이다.

<thread.c>

```
bool thread_priority_sort(const struct list_elem *A, const struct list_elem *B, void *aux)
{
  struct thread *A_thread = list_entry(A, struct thread, elem);
  struct thread *B_thread = list_entry(B, struct thread, elem);
  return A_thread->priority > B_thread->priority;
}
```

또한 thread_unblock 함수의 ②,③ 에서는 preemption을 구현하기 위한 코드인데, 이것은 현재 bool 실행중인 스레드와 unblock 되는 스레드의 우선순위를 비교하여 unblock되는 우선순위가 높을 경우 thread_yield 함수를 호출하여 preemption을 구현하고자 하였다. 그리고 ②에서 cur != idle_thread 라는 조건은 처음 pintos 부팅 시 idle 스레드가 생길 때를 고려하여 작성한 코드이다.

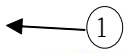
thread_yield 함수의 구현을 살펴보면, 다음과 같다.

<thread.c>

```
void
thread_yield(void)
{
 struct thread *cur = thread_current();
 enum intr_level old_level;

 ASSERT (!intr_context());

 old_level = intr_disable();
 if(cur != idle_thread)
   list_insert_ordered(&ready_list, &cur->elem, thread_priority, NULL);
 cur->status = THREAD_READY;
 schedule();
 intr_set_level (old_level);
}
```



마찬가지로 thread_yield 함수를 실행하게 되면 현재 실행중인 스레드를 ready queue에 넣게 되는데, ①처럼 이때도 우선순위에 따른 정렬이 필요하다. 또한, 우선순위가 바뀐다면 thread_yield 함수를 호출하여 preemption이 발생하도록 구현하기 위해 thread_set_priority 함수를 개선하였다.

<thread.c>

```
void
thread_priority_choice(int new_priority)
{
        thread_current ()->priority = new_priority;
        if(thread_current() != idle_thread)
         thread_yield();
}
```

다음으로 sema의 값이 0인 동안 block 되어 waiting queue에서 대기해야 하는데 이곳에서도 우선순위를 통한 정렬이 필요하다. 다음은 코드의 구현이다.

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()->elem, thread_priority, NULL);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

<synch.c>

```
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  list_push_back (&cond->waiters, &waiter.elem, thread_priority, NULL);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

마찬가지로 lock의 기능을 하는 cond_wait 함수 내의 코드도 수정하였는데, 이곳에서도 waiting queue의 스레드들을 우선순위대로 정렬하게끔 구현하였다.

4. 캡쳐 결과

- alarm-priority

```
Boot complete.
Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
```

Ⅰ에서의 alarm-priority 테스트 결과와 달리 우선순위대로 스레드들을 깨우고 있는 모습이므로 성공적인 테스트이다.

- priority-change

```
Boot complete.
Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 exiting.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
```

- priority-preempt

```
Boot complete.
Executing 'priority-preempt':
(priority-preempt) begin
(priority-preempt) Thread high-priority iteration 0
(priority-preempt) Thread high-priority iteration 1
(priority-preempt) Thread high-priority iteration 2
(priority-preempt) Thread high-priority iteration 3
(priority-preempt) Thread high-priority iteration 4
(priority-preempt) Thread high-priority done!
(priority-preempt) The high-priority thread should have already completed.
(priority-preempt) end
Execution of 'priority-preempt' complete.
```

이 테스트함수는 내부적으로 thread_yield 함수를 호출하는데, thread_yield 함수 내부에서 우선순위대로 ready queue를 정렬하였기 때문에 위와 같이 정상적으로 결과가 출력되었다.

- priority-fifo

```
Boot complete.
Executing 'priority-fifo':
(priority-fifo) begin
(priority-fifo) 16 threads will iterate 16 times in the same order each time.
(priority-fifo) If the order varies then there is a bug.
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) end
Execution of 'priority-fifo' complete.
```

이 테스트 함수는 main 함수보다 우선순위가 낮고 그 우선순위들이 모두 같은 스레드를 여러 개 생성하고, 그 스레드들은 thread_yield 함수를 16번 호출한다. 우선순위가 같을 경우 기존의 FIFO 형태로 나오도록 수정하였기 때문에 위와 같이 정상적으로 출력되었다.

- priority-sema

```
Boot complete.
Executing 'priority-sema':
(priority-sema) begin
(priority-sema) Back in main thread.
(priority-sema) Thread priority 30 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 29 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 28 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 27 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 26 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 25 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 24 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 23 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 22 woke up.
(priority-sema) Back in main thread.
(priority-sema) end
Execution of 'priority-sema' complete.
```

이 테스트 함수는 main 함수보다 낮은 우선순위(31 미만)를 갖는 스레드들을 우선순위 값을 1씩 감소시키며 생성한다. 내부적으로 sema_down 함수를 호출했을 때 원래는 우선순위대로 정렬이 되어있지 않아 정상적인 결과가 나오지 않았지만, sema_down 함수를 우선순위대로 정렬하도록 구현하였으니 위와 같은 정상적인 결과가 출력되었다.

Ⅱ. Priority-scheduling-donation

1. 문제인식

기본적으로 pintos scheduler 방식은 Round robin 방식을 이용한다. thread time slice를 초과하면 Context switch가 발생하여 runnig thread가 ready_queue 가장 끝에 삽입된다. 또한, block 되있는 thread를 ready 상태로 바꾸어 ready_list의 가장 뒤에 넣어준다.

2. 해결방안과 과정

기존의 Round robin 방식에서 스레드가 생성될 때마다 Preempt schedule 방식을 통해 우선순위를 비교하여 running thread가 새로 생성된 스레드 보다 우선 순위가 낮으면 cpu를 반납하는 방식으로 바꾸는데 기존에는 단순히 list의 끝에 입하였지만 변경된 방식은 thread가 ready_queue에 삽입 될 때 우선순위대로 정렬 될 수 있도록(내림차순)으로 정렬하여 삽입하는 방식으로 변경하였다. 또한 동일한 lock을 사용함에 있어서 낮은 우선순위 thread가 높은 우선순위 thread가 waiting 상태로 빠졌을 때 먼저 실행되어 우선순위역전이 발생한다. 우선순위 역전에 대해서는 코드로 자세히 설명하겠다.

3. 소스코드

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    int real_p;
    struct list_elem allelem;           /* List element for all threads list. */
    struct list lock_a;
    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
    struct lock *lock_b;
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

struct list lock_a lock에 연결되어 있는 리스트를 의미한다.
struct lock * lock_b 위에서 이야기한 lock 리스트에 매달려 지는 아이들을 의미한다.

```
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
    struct list_elem elem;
    int max_a;
  };
```

int max_a = 매달려진 아이들의 우선순위를 의미

```
bool thread_priority_sort(const struct list_elem *A, const struct list_elem *B, void *aux)
{
        struct thread *A_thread = list_entry(A, struct thread, elem);
        struct thread *B_thread = list_entry(B, struct thread, elem);
        if(A->priority > B->priority)
         return true;
         else return false;
}
```

struct list_elem elem(lock리스트에서 맴버들 간의 위치를 가르키는 변수)
list_elem형 주소를 매개변수로 받아와서 list_entry를 통해 스레드의 시작주소를 받고 포인터 변수 a와 b를 통해 우선순위를 비교한뒤 우선순위 값에 따라 true or false를 반환

```
void
find_bigger_p(void)
{
        enum intr_level old_level = intr_disable();
        if(!list_empty(&ready_list) && thread_current () -> priority < list_entry(list_front
((&ready_list),struct thread, elem)->priority)
         thread_yield();
        intr_set_level(old_level);
}
```

```
  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;
  enum intr_level old_level;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();
  sf->ebp = 0;

  /* Add to run queue. */
  thread_unblock (t);
  find_bigger_p();
  return tid;
}
```

thread_create 함수는 thread 생성시 find_bigger_p함수를 호출. 그리고 find_bigger_p 함수는 ready list에 대기중인 thread가 있고 runnig thread의 ready list의 맨 앞에 위치한 thread와 우선순위를 비교하고 thread_yield를 호출 그리고 스케줄링한다.

```
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_push_back (&ready_list, &cur->elem,thread_priority_sort,NULL);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_push_back (&ready_list, &t->elem,thread_priority_sort,NULL);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

기존에는 ready_list에 순서대로 들어갔지만 thread_unblock함수와 thread_yield 함수를 수정하여 변경한 뒤 우선순위에 따라 정렬되어 list에 아이들중 맨 앞에 가장 높은 우선 순위를 가진 아이가 매달리게 구현하였다.

```
bool thread_priority_sort(const struct list_elem *A, const struct list_elem
*B, void *aux)
{
        struct thread *A_thread = list_entry(A, struct thread, elem);
        struct thread *B_thread = list_entry(B, struct thread, elem);
        return A->max_a > change->max_a;
}
```

priority_sort 함수는 list를 우선순위대로 정렬시 우선순위 비교를 위해 사용 된다.

```
void
thread_set_priority (int new_priority)
{
  enum intr_level old_level = intr_disable();
  struct thread *t = thread_current();
  int old_priority = t->priority

  t->readl_p = new_priority;
  if(new_priority < old_priority && list_empty (&t->lock_a))
        {
          t->priority = new_priority;
          find_bigger_p();
        }
  intr_set_level (old_level_;
}
thread_current ()->priority = new_priority;
  if(thread_current() != idle_thread)
        thread_yield();

}
```

우선 thread_set_priority함수는 현재 가지고 있는 우선순위를 변경하는데 사용된다. 만약 새로 할당받은 priority 값이 기존의 old_priority보다 작고 lock_a 리스트가 비워져있다면 running thread의 우선순위를 바꿔줄 new_priority로 바꾸어 주고 find_bigger_p()함수를 호출하여 우선순위가 높은 순으로 스케줄링 해준다.

```
void donate_p(struct thread *t)
{
  enum intr_level old_level = intr_disable();
  over_p(t);

  if(t->status ==  THREAD_READY)
   {
        list_remove(&t->elem);
        list_insert_ordered(&ready_list, &t->elem, thread_priority_sort, NULL);
  intr_set_level(old_level);
}
```

```
void over_p(struct thread *t)
{
  enum intr_level old_level = intr_disable();

  int lock_p;
  int first_p = t->readl_p;

  if(!list_empty (&t->lock_a))
    {
        list-sort(&t->lock_l,thread_priority_sort,NULL);
        lock_p = list_entry (list_front (&t-?lock_a),struct
lock,elem)->max_a;
        if(first_p<lock_a)
          first_p = lock_a;
    }

  t->priority = first_p;
  intr_set_level (old_level);
}
```

함수의 주 기능은 over_p 함수를 호출하고 현재 실행중인 thread가 ready 상태이면 ready list에 우선순위 스케줄링하여 삽입한다. 그리고 lock list를 priority_sort 함수를 사용해서 우선순위대로 정렬해주는 함수이다.

```
bool thread_priority_sort(const struct list_elem *A, const struct
list_elem *B, void *aux)
{
        struct thread *A_thread = list_entry(A, struct thread, elem);
        struct thread *B_thread = list_entry(B, struct thread, elem);
        return A->max_a > B->max_a;
}
```

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()-
>elem,thread_priorty_sort,NULL);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

semephore가 0일 때 waiters 리스트를 우선순위 스케줄링하여 정렬한다.

```
void
lock_acquire (struct lock *lock)
{
  struct thread *t = thread_current();
  struct lock *l;
  enum intr_level old_level;
  int length =1;

  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  if (lock->holder !=Null)
   {
     t->lock_m = lock;
     l = lock;
     while(l && t->priority > l->max_a && length++ <10)
        {
          l->max_a = t->priority;
          donate_p (l->holder);
          l = l->holder -> lock_m;
        }
    }
  sema_down(&lock->semaphore)

  old_level = intr_disable();
  t = thread_current ();
  t->lock_m=NULL;
  lock->max_a = t->priority;

  ordered_lock_list(lock);
  lock->holder = t;
  intr_set_level (old_level);
}
```

lock이 holder가 존재하면 lock * a와 thread의 lock * lock_b 값을 저장한다. while 문은 9번 실행되고 thread의 priority가 lock의 max_a보다 큰 동안에 priority donation이 일어나게 된다. 그리고 thread의 lock_list를 NULL로 하고 lock의 max_a는 현재 thread의 priority로 바꾸어준다.

```
void ordered_lock_list(struct lock *lock)
{
  enum intr_level old_level = intr-disable();
  list_insert_ordered(&thread_curretn () -> lock_l,&lock-
>elem,thread_priority_sort,NULL);
  if(thread_current()->priority < lock->max_a)
   {
     thread-current()->priority = lock->max-a;
     find_bigger_p();
   }
  intr_set_level(old_level);
}
```

ordered_lock_list 함수가 호출되면 lock list에 우선순위 스케줄링을 하여 멤버를 삽입한다. 이때 lock의 max_a가 현재 thread의 우선순위보다 크면 현재 thread의 우선순위를 max_a로 바꾸어준다.

```
void
lock_release (struct lock *lock)
{
  enum intr_level old_level;

  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  old_level = intr_disable();
  remove_ordered(lock);

  lock->holder = NULL;
  sema_up (&lock->semaphore);

  intr_set_level(old_level);
}
```

lock_release함수는 매개변수로 온 lock을 remove_ordered함수를 통해 해제시킨다. 그리고 semaphore를의 값을 올려준다.

```
void remove_ordered (struct lock *lock)
{
  enum intr_level old_level = intr_disable();

  list_remove(&lock->elem);
  over_p (thread_current());
  intr_set_level(old_level);
}
```

lock_release함수가 호출되면 remove_ordered함수가 호출되어 매개변수로 전달된 lock을 연결되어 있는 리스트를 제거해주고 thread의 lock list를 갱신하여 정렬시켜준다.

4. 캡쳐 결과

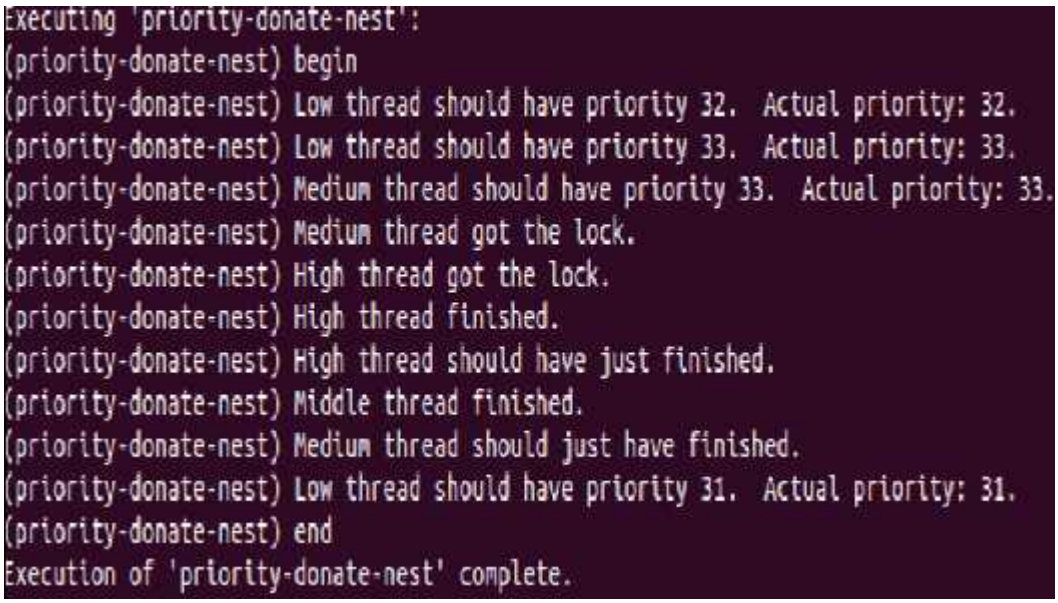
priority-donate-one

```
Executing 'priority-donate-one':
(priority-donate-one) begin
(priority-donate-one) This thread should have priority 32.  Actual priority: 32.
(priority-donate-one) This thread should have priority 33.  Actual priority: 33.
(priority-donate-one) acquire2: got the lock
(priority-donate-one) acquire2: done
(priority-donate-one) acquire1: got the lock
(priority-donate-one) acquire1: done
(priority-donate-one) acquire2, acquire1 must already have finished, in that order.
(priority-donate-one) This should be the last line before finishing this test.
(priority-donate-one) end
Execution of 'priority-donate-one' complete.
```

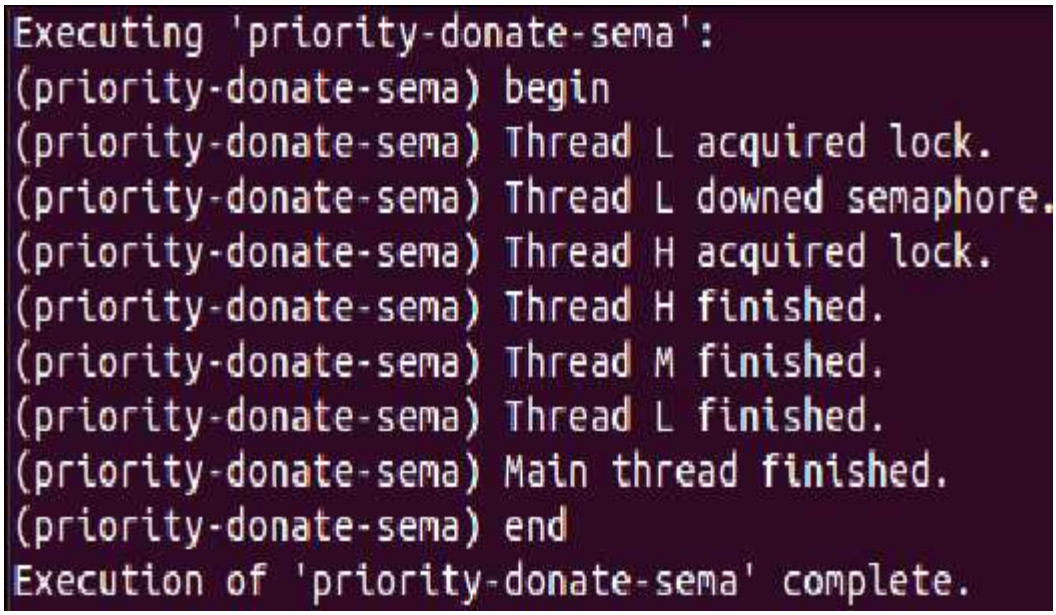
priority-donate-multiple

```
Executing 'priority-donate-multiple':
(priority-donate-multiple) begin
(priority-donate-multiple) Main thread should have priority 32.  Actual priority: 32.
(priority-donate-multiple) Main thread should have priority 33.  Actual priority: 33.
(priority-donate-multiple) Thread b acquired lock b.
(priority-donate-multiple) Thread b finished.
(priority-donate-multiple) Thread b should have just finished.
(priority-donate-multiple) Main thread should have priority 32.  Actual priority: 32.
(priority-donate-multiple) Thread a acquired lock a.
(priority-donate-multiple) Thread a finished.
(priority-donate-multiple) Thread a should have just finished.
(priority-donate-multiple) Main thread should have priority 31.  Actual priority: 31.
(priority-donate-multiple) end
Execution of 'priority-donate-multiple' complete.
```

priority-donate-nest



priority-donate-sema



priority-donate-multiple2

```
Executing 'priority-donate-multiple2':
(priority-donate-multiple2) begin
(priority-donate-multiple2) Main thread should have priority 34.  Actual priority:
(priority-donate-multiple2) Main thread should have priority 36.  Actual priority:
(priority-donate-multiple2) Main thread should have priority 36.  Actual priority:
(priority-donate-multiple2) Thread b acquired lock b.
(priority-donate-multiple2) Thread b finished.
(priority-donate-multiple2) Thread a acquired lock a.
(priority-donate-multiple2) Thread a finished.
(priority-donate-multiple2) Thread c finished.
(priority-donate-multiple2) Threads b, a, c should have just finished, in that orde
(priority-donate-multiple2) Main thread should have priority 31.  Actual priority:
(priority-donate-multiple2) end
Execution of 'priority-donate-multiple2' complete.
```

priority-donate-chain

```
Executing 'priority-donate-chain':
(priority-donate-chain) begin
(priority-donate-chain) main got lock.
(priority-donate-chain) main should have priority 3.  Actual priority: 3.
(priority-donate-chain) main should have priority 6.  Actual priority: 6.
(priority-donate-chain) main should have priority 9.  Actual priority: 9.
(priority-donate-chain) main should have priority 12.  Actual priority: 12
(priority-donate-chain) main should have priority 15.  Actual priority: 15
(priority-donate-chain) main should have priority 18.  Actual priority: 18
(priority-donate-chain) main should have priority 21.  Actual priority: 21
(priority-donate-chain) thread 1 got lock
(priority-donate-chain) thread 1 should have priority 21. Actual priority:
(priority-donate-chain) thread 2 got lock
(priority-donate-chain) thread 2 should have priority 21. Actual priority:
(priority-donate-chain) thread 3 got lock
(priority-donate-chain) thread 3 should have priority 21. Actual priority:
(priority-donate-chain) thread 4 got lock
(priority-donate-chain) thread 4 should have priority 21. Actual priority:
(priority-donate-chain) thread 5 got lock
(priority-donate-chain) thread 5 should have priority 21. Actual priority:
(priority-donate-chain) thread 6 got lock
(priority-donate-chain) thread 6 should have priority 21. Actual priority:
(priority-donate-chain) thread 7 got lock
(priority-donate-chain) thread 7 should have priority 21. Actual priority:
```

Ⅲ. 후기

이규한 : 지금껏 운영체제가 어떻게 만들어 졌고 구성되어 있는지 아무 생각도 없었으나 이번 기회를 통해 컴퓨터가 어떻게 부팅되는지 또 어떻게 자원이 할당 되는지 정확히 이해할 수 있는 계기가 되었다. 또 리눅스에 대해서 고민할 수 있는 시간을 얻은 것이 너무도 좋았다. 결과가 계속해서 나오지 않았음에도 답을 해결하기 위해 생각하는 과정이 지금 생각해보니 다 의미있는 순간들이었다.
이번에 어려운 과제를 통해 지금 나의 실력을 객관적으로 느낄 수 있었으며 더 열심히 해야 할 명확한 동기부여를 마련할 수 있었다.

김성환 : 참 어려웠던 과제였습니다. 사실 운영체제라는 과목에 대해서 쉽게 생각하고 학기를 시작했었는데 생각보다는 복잡하던 것들을 수업을 듣고 직접 운영체제를 다뤄보면서 동작하는 원리와 구조들을 이해하게 되었습니다. 평소에 컴퓨터를 사용하면서 각각 다른 프로그램을 한꺼번에 실행시키다 보면 프로그램 간의 진행 속도가 엇갈리고 버벅거리던 것들을 단순히 느리다고 짜증만 냈던 것에서 이번에 그러한 상황을 공부하고 직접적인 원인을 해결해보니 상황에 대한 이해도 생겼습니다.
과제를 진행하면서 마치 알파고가 딥러닝을 하듯이 계속 새로운 상황에 부딪히며 인식하고 해결하기 위해 노력했던 것들이 의미있었던 과제였습니다.

티켓 :
Alarm Clock 개선 이규한4 김성환3
Priority Scheduling 이규한3 김성환4