운영 체제 과제 보고서

project #6~9

과목명: 운영 체제

담당 교수: 원유집 교수님

담당 조교: 오준택 조교님

제출 일자: 2018/ 05/ 11

2014004411 김시완

2016025714 이동윤

Github address:

https://github.com/gimsiwan/Operating_system.git

목차

1. Alarm System Call

과제 목표:

알람 기능을 sleep/wake up 을 이용하여 구현

과제 설명:

호출한 프로세스를 정해진 시간 후에 다시 시작하는 커널 내부의 함수를 구현한다. 기존 Busy waiting 으로 구현 되어 있던 함수를 지우고, sleep/wake up 기능을 통해 보다 효율적으로 관리를 하게 만든다

a. thread 구조체/ 전역 변수 선언/ 초기화

```
····int64_t·wakeup_tick;·····//깨어나야할tick을저장
```

Sleep 상태에 들어간 각 스레드들마다 깨워줘야 하는 시간을 저장하기 위해 thread 구조체에 wakeup_tick 변수를 선언한다.

```
static·struct·list·sleep_list;·····//자고·있는·스레드를·저장하고·있는·리스트

static·int64_t·next_tick_to_awake·=INT64_MAX;·····//sleep_list에서·대기중인·스레드들의·wakeup_tick값·중·최소값
```

```
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);·····//sleep·list·초기화
```

Sleep 상태에 빠진 스레드들을 저장하고 있는 sleep_list 와 sleep_list 에 들어 있는 스레드들 중 가장 최소값의 tick 값을 저장하기 위한 next_tick_to_awake 변수를 전역으로 선언해주고 초기화 시켜준다.

b. timer_sleep 함수

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

  ASSERT (intr_get_level () == INTR_ON);
  thread_sleep(start + ticks);      //해당 스레드를 지정한 시간까지 block
}
```

스레드를 sleep 상태로 바꿔주는 함수로서 기존 while 문을 통해 busy_waiting 을 유발하는 코드를 지워주고, thread_sleep 함수를 호출하여 sleep/wake_up 형식으로 바꿔준다.

c. thread_sleep 함수

```
void thread_sleep(int64_t ticks){
    struct thread *cur;
    enum intr_level old_level;

    old_level = intr_disable();
    cur = thread_current();
    if(cur != idle_thread){     //현재 스레드가 idle 스레드가 아닐 경우만 thread의 상태 변경
        cur->wakeup_tick = ticks;       //재워 줄 시간 저장
        list_push_back(&sleep_list, &cur->elem);        //sleep list에 해당 스레드 저장
        update_next_tick_to_awake(cur->wakeup_tick);        //해당 tick값이 최소값이 될 수있으므로 해당 함수 호출
        thread_block();
    }
    intr_set_level(old_level);
}
```

해당 함수를 호출한 스레드가 idle 스레드가 아닌 경우 해당 스레드 구조체에 재워 줄 시간을 저장하고 sleep_list 에 넣어준다. 해당 tick 값이 현재 sleep_list 에 들어있는 스레드들의 list 값들과 비교해 보았을 때 최소값이 될 수 있으므로, update_next_tick_to_awake 함수를 호출하여 next_tick_to_awake 전역 변수를 업데이트 시켜준다. 또한, 마지막으로 해당 스레드를 block 상태로 바꾸어준다.

d. timer_interrupt 함수

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

  if(get_next_tick_to_awake() <= ticks){     //깨워야 하는 스레드의 최소 tick값이 현재 tick값보다 작으면 해당 스레드들을 깨워준다.
    thread_awake(ticks);
  }
}
```

thread_awake 함수를 호출하는 interrupt 로서 현재 tick 값보다 sleep_list 안에 들어있는 스레드들의 최소 tick 값이 더 작은 경우 thread_awake 함수를 호출하여 sleep_list 를 정리 시켜준다.

e. thread_awake 함수

```
void thread_awake(int64_t ticks){

    struct list_elem *elem;
    next_tick_to_awake = INT64_MAX;

    for(elem = list_begin(&sleep_list); elem != list_end(&sleep_list);){
        struct thread *t = list_entry(elem, struct thread, elem);

        if(t->wakeup_tick <= ticks){
            elem = list_remove(&t->elem);
            thread_unblock(t);          //sleep list를 순회하면서 해당 스레드 tick값이 현재 tick보다 크다면 리스트에서 제거하고 unblock
        }else{
            update_next_tick_to_awake(ticks);    //현재 대기중인 스레드들의 wakeup_tick 변수 중 가장 작은 값을
            next_tick_to_awake 전역 변수에 저장
            elem = list_next(elem);
        }
    }
}
```

sleep_list 에 들어 있는 스레드를 깨우는 함수로서 인자로 주어진 tick 값보다 작은 wakeup_tick 을 가지고 있는 스레드들을 sleep_list 에서 제거하고 unblock 시켜준다. 또한, 지워준 스레드가 최소 tick 값을 지니고 있을 수 있으므로, sleep_list 에서 지워지지 않은 스레드들을 가지고 최소 tick 값을 변경시켜준다.

f. get_next_tick_to_awake / update_next_tick_to_awake 함수

```
int64_t get_next_tick_to_awake(void){
    return next_tick_to_awake;          //sleep중인 스레드들 중 깨워줘야 하는 시간 리턴
}

void update_next_tick_to_awake(int64_t ticks){
    if(next_tick_to_awake>ticks)
        next_tick_to_awake = ticks;
    /*만약 해당 tick값이 현재 깨워줘야 하는 스레드들의 tick값의 최소값보다 작으면 해당 tick 값으로 업데이트*/
}
```

get_next_tick_to_awake 함수는 전역 변수 next_tick_to_awake 를 반환하는 함수이다. update_next_tick_to_awake 함수는 인자로 주어진 tick 값이 현재 최소 tick 값인 next_tick_to_awake 보다 작을 경우 해당 값으로 전역 변수를 변경해 주는 함수이다.

실행 결과

```
(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: 943 ticks
Thread: 550 idle ticks, 396 kernel ticks, 0 user ticks
Console: 2952 characters output
Keyboard: 0 keys pressed
Powering off...
================================================================================
Bochs is exiting with the following message:
[UNMP ] Shutdown port: shutdown requested
================================================================================
```

실행 결과를 확인하기 위해 'pintos -- -q run alarm-multiple' 명령어를 호출해주었고, 이 결과 idle tick 값이 busy_waiting 인 경우와 비교하여 증가하였음을 확인 할 수 있었다.

2. Priority Scheduling

과제 목표:

Round Robin 으로 구현 되어 있는 스케줄러를 우선 순위 스케줄링으로 변경 해준다.

과제 설명:

Ready list 에 새로 추가된 스레드의 우선 순위가 현재 CPU 를 점유중인 스레드의 우선 순위보다 높은 경우, 기존 스레드를 밀어내고 CPU 를 점유하도록 한다.

a. thread_create 함수

```
··list_push_back·(&thread_current()->child_list,·&t->child_elem);
··t->fdt·=·(struct·file**)calloc(1,sizeof(struct·file*)*MAX_FILE);··//file·size(MAX_FILE)·is·256
··t->next_fd·=·2;·/*·0·is·stdin,·1·is·stdout·*/

/*initiallize·the·thread·structure·*/

··/*·Add·to·run·queue.·*/
··thread_unblock·(t);

··test_max_priority();
/*생성된·스레드의·우선·순위가·현재·실행중인·스레드의·우선·순위보다·더·클·수·있으므로·test_max_priority·함수를·호출하고,·우선·
순위가·더·높은·경우·cpu를·양보해준다.*/

··return·tid;
}
```

Thread_create 함수를 통해 새로운 스레드를 생성해주었을 경우, 새로 생성한 스레드의 우선 순위가 현재 실행 중인 스레드의 우선 순위보다 높은 경우 test_max_priority 함수를 통해 CPU 를 양보해준다.

b. thread_unblock 함수

```
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_insert_ordered(&ready_list, &t->elem, &cmp_priority, NULL);·····//스레드를 unblock 해주는 경우 우선 순위를 고려하여
  리스트에 넣어준다.
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

스레드를 unblock 해주는 경우 기존에는 list_push_back 함수를 통해 우선순위에 관련 없이 넣어주었으나 해당 함수를 지워 주고, list_insert_ordered 함수를 통해 우선 순위를 고려하여 ready_list 에 넣어준다.

c. thread_yield 함수

```
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,&cmp_priority,NULL);·····//현재 스레드를 yield함수를 통해 ready 상태로
    변경해주는 경우 우선순위를 고려하여 ready list에 넣어준다.
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

마찬가지로, thread_yield 함수를 통해 스레드를 ready_list 에 넣어주는 경우 list_insert_ordered 함수를 통해 우선 순위를 고려하여 넣어준다.

d. thread_set_priority 함수

```
void
thread_set_priority (int new_priority)
{
    thread_current ()->priority = new_priority;
    thread_current()->init_priority = new_priority;
    refresh_priority();
    test_max_priority();        //해당 함수를 호출하여 우선순위가 변경되었으므로 test_max_priority 함수를 호출하여 CPU를 양보해
    주어야 하는지 판단한다.
}
```

해당 함수는 우선 순위를 변경 하는 함수로서, 해당 함수를 호출함으로서 우선 순위가 변경되었으므로 test_max_priority 함수를 통해 현재 실행 중인 스레드보다 우선 순위가 더 높아졌을 경우 CPU 를 양보한다.

e. test_max_priority 함수

```
void test_max_priority(void){

    if(!list_empty(&ready_list)){

        struct thread *t = list_entry(list_front(&ready_list), struct thread, elem);

        if(t->priority> thread_get_priority())
            thread_yield();
/*현재 실행 중인 스레드와 리스트에 존재하는 우선 순위가 가장 높은 스레드를 비교하여 스케줄링 해준다.*/
    }
}
```

리스트에 첫번째 존재하는 스레드(가장 우선 순위가 높은 스레드)와 현재 스레드의 우선 순위를 비교하여 만약 리스트에 존재하는 스레드의 우선 순위가 더 높다면 CPU를 양보해준다.

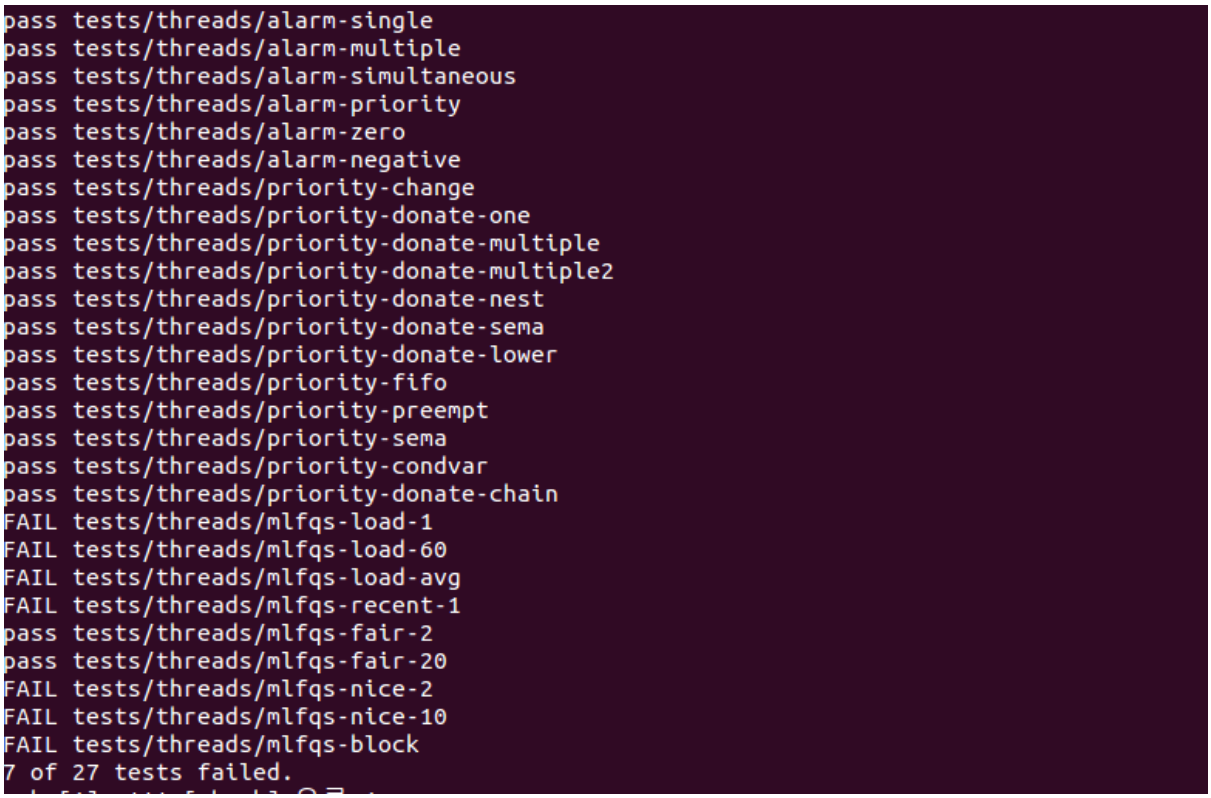
f. cmp_priority 함수

```
bool cmp_priority(const struct list_elem *a_, const struct list_elem *b_, void* aux UNUSED){

    return (list_entry(b_, struct thread, elem)->priority) < (list_entry(a_, struct thread, elem)->priority);
        //첫 번째 인자를 가지고 있는 스레드와, 두 번째 인자를 가지고 있는 스레드의 우선순위를 비교해준다.
}
```

첫 번째 인자를 가지고 있는 스레드의 우선 순위가 더 높은 경우 1 을 반환, 두 번째 인자를 가지고 있는 스레드의 우선 순위가 더 높은 경우 0 을 반환해준다. list_insert_ordered 함수에서 우선 순위를 비교해주기 위해 사용된다.

실행 결과



make check 결과 alarm-priority 와 priority-fifo, priority-preempt 테스트 결과가 pass 로 바뀌었음을 알 수 있었다.

3. Priority Scheduling and Synchronization

과제 목표:

여러 스레드가 lock, semaphore, condition variable 을 얻기 위해 기다릴 경우 우선 순위가 가장 높은 thread 가 CPU 를 점유 하도록 구현

과제 설명:

lock, semaphore, condition variable 을 여러 스레드에서 요청한 경우 기존 과제에서 구현한 priority 를 기준으로 가장 높은 priority 를 가진 스레드에게 반환해준다.

a. sema_down 함수

```
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_insert_ordered (&sema->waiters, &thread_current ()->elem, &cmp_priority, NULL);
      /*더 이상 semaphore를 잡지 못하고 기다리는 경우 semaphore의 waiter 리스트에 우선순위를 기준으로 스레드를 넣어준다.*/
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

sema 의 value 값이 0 즉, 더 이상 semaphore 를 잡지 못하는 경우 전에는 list_push_back 함수를 통해 우선 순위와 상관없이 넣어주었지만, 이를 변경 하여 list_insert_ordered 함수를 통해 우선 순위를 기준으로 semaphore 의 waiters 리스트에 넣어준다.

b. sema_up 함수

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

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)){
    list_sort(&sema->waiters, &cmp_priority, NULL);
  /*스레드가 waiters list에 있는 동안 우선 순위가 변경 되었을
  경우를 고려하여 waiters list 를 우선 순위로 정렬한다.*/
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  }
  sema->value++;
  test_max_priority();    //sema up을 통해 unblock된 스레드의 우선 순위가 현재 실행중인 스레드보다 높을 경우 CPU를 양보해준다.
  intr_set_level (old_level);
}
```

스레드가 waiters list 에 존재하는 동안 스레드들의 우선 순위가 변경 되었을 수도 있으므로 list_sort 함수를 통해 waiter list 를 재정렬 시켜준다. 또한, unblock 시켜준 스레드의 우선 순위가 현재 스레드의 우선 순위보다 더 높은 경우 test_max_priority 함수를 통해 CPU 를 해당 스레드로 양보 시켜준다.

c. cmp_sem_priority 함수

```
bool cmp_sem_priority(const struct list_elem *a, const struct list_elem *b, void* aux UNUSED){
  struct semaphore_elem *sa = list_entry(a,struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry(b,struct semaphore_elem, elem);
    //해당 elem을 가지고 있는 세마포어 호출
  if(list_empty(&sa->semaphore.waiters))
    return false;
  //첫번째 세마 포어가 비어있는 경우 두번째 세마포어의 우선 순위가 더 높으므로 false 호출
  if(list_empty(&sb->semaphore.waiters))
    return true;
  //두번째 세마 포어가 비어있는 경우 첫번째 세마포어의 우선 순위가 더 높으므로 true 호출

  list_sort(&sa->semaphore.waiters, &cmp_priority, NULL);
  list_sort(&sb->semaphore.waiters, &cmp_priority, NULL);
  //혹시 세마포어 waiters 리스트에 존재하는 스레드의 우선순위가 변경 되었을 수 있으므로 sort해준다.
  struct thread *ta = list_entry(list_front(&sa->semaphore.waiters),struct thread, elem);
  struct thread *tb = list_entry(list_front(&sb->semaphore.waiters),struct thread, elem);
  //세마포어의 waiter 리스트의 첫번째 스레드들 호출
  return (ta->priority > tb->priority);    //스레드의 우선순위 비교
}
```

기존 cmp_priority 함수로는 semaphore waiters 리스트에서 스레드를 가져와 우선 순위를 가져오는 것이 불가능 하였기 때문에, 해당 함수를 구현해주었다. 첫 번째 인자를 지니고 있는 스레드의 우선 순위가 더 높은 경우 1 을 리턴 해주고,

두번째 인자를 지니고 있는 스레드의 우선 순위가 더 높은 경우 0 을 리턴 해준다. 또한, 두개의 세마포어 waiters 리스트가 비어 있는 경우를 예외적으로 따로 처리해 준다.

d. cond_wait 함수

```
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_insert_ordered (&cond->waiters, &waiter.elem,&cmp_sem_priority,NULL);
  //condition variable의 waiters list에 우선순위 순서로 삽입되도록 수정
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

condition variable 을 통해 lock 을 잡아주는 경우 waiters 리스트에 스레드를 넣어주는데 있어 기존 list_push_back 함수를 통해 우선 순위와 상관없이 넣어준 것에 반해 list_insert_ordered 함수를 통해 높은 우선순위 순서로 넣어준다.

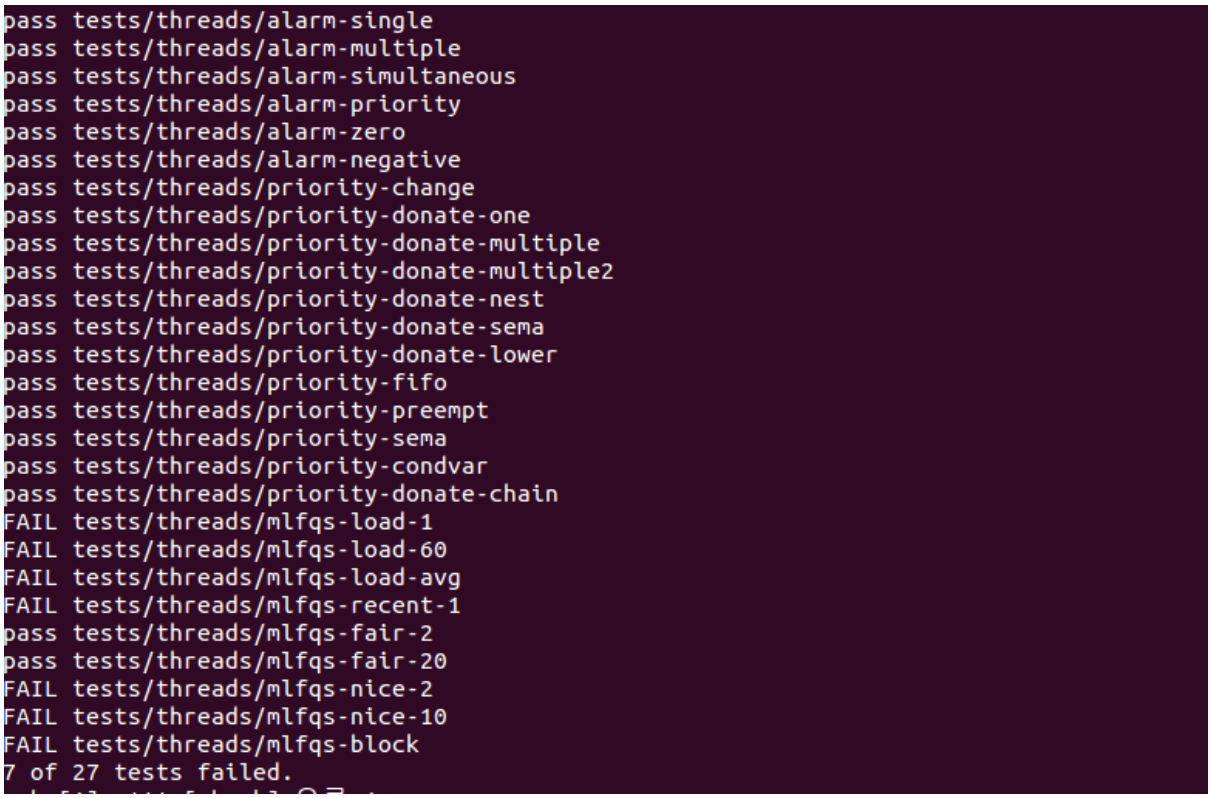
e. cond_signal 함수

```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters)){
    list_sort(&cond->waiters, &cmp_sem_priority, NULL);   //대기중에 우선 순위가 변경 되었을 가능성이 있으므로 리스트를
      sorting 해준다.
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
  }
}
```

해당 함수를 통해 waiters 리스트에서 스레드를 깨우는 경우 스레드의 우선 순위가 변경 되었을 수도 있으므로 sema_up 함수 호출을 통해 깨우기 전에 list_sort 함수를 호출하여 리스트를 우선 순위 순서로 재정렬 해준다.

실행 결과



make check 결과 priority-sema 와 priority-condvar 테스트 결과가 pass 로 바뀌었음을 알 수 있었다.

4. Priority Inversion Problem

과제 목표:

Lock 요청 대기 시 발생 할 수 있는 inversion problem 을 해결

과제 설명:

중간 우선 순위의 스레드가 낮은 우선순위의 스레드를 선점함으로 인해 우선 순위가 높은 스레드가 먼저 lock을 요청했음에도 불구하고 해당 스레드를 기다리게 되는 현상을 해결하기 위해 lock을 요청하는 함수와 해제 하는 함수를 수정. 이전 상태의 우선순위를 기억함으로써 multiple donation 문제를 해결하고, donation list를 구현함으로써 nested donation 문제를 해결한다.

a. thread 구조체

```
····int·init_priority;·····//우선순위를초기화하기위해초기값저장
····struct·lock·*wait_on_lock;·····//해당스레드가대기하고있는lock자료구조
····struct·list·donations;·····//lock·대기·스레드·리스트
····struct·list_elem·donation_elem;
```

스레드 구조체에 처음 상태의 priority 를 저장하기 위한 변수와, 해당 스레드가 대기하고 있는 lock 을 저장하기 위한 구조체와, lock 을 대기하는 스레드 리스트, 해당 스레드의 donations list elem 을 저장해준다.

b. init_thread 함수

```
···t->init_priority·=·priority;
···t->wait_on_lock·=·NULL;
···list_init(&t->donations);
·····//자료·구조·초기화
}
```

스레드 구조체에 추가한 변수와 구조체를 init_thread 함수에서 초기화 시켜준다.

c. lock_acquire 함수

```
void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  if(lock->holder){      //이미 락이 잡혀있는 경우
    thread_current()->wait_on_lock = lock;      //현재 잡으려는 락을 wait_on_lock 구조체에 저장
    list_insert_ordered(&lock->holder->donations,&thread_current()->donation_elem,&cmp_priority,NULL);
    //락을 잡고 있는 스레드의 donation list에 해당 스레드의 elem을 저장해준다.
    donate_priority();      //nested donation 해결을 위해 해당 함수 호출
  }

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
  thread_current()->wait_on_lock = NULL;
}
```

lock 을 잡아주는 lock_acquire 함수에서 이미 lock 을 다른 스레드가 잡고 있는 경우 해당 스레드의 구조체에 해당 lock 값을 저장한다음 donation list 에 저장시켜준다. 또한 이 때 발생할 수 있는 nested donation(스레드가 대기하고 있는 lock 을 잡고 있는 스레드가 또 대기하고 있는 lock 이 존재 할 시 해당 lock 을 잡고 있는 스레드의 우선순위도 같이 변경)문제를 해결 하기 위해 구현한 donate_priority 함수를 호출 시켜준다.

d. donate_priority 함수

```
void donate_priority(void){
    struct lock *wait_lock = thread_current()->wait_on_lock;
    int i = 0;
    while(wait_lock && i<8){      //현재 스레드가 대기하고 있는 lock이 존재 할 시
                                  //nested depth는 8로 제한시켜주었다.
        if(wait_lock->holder->priority < thread_current()->priority){
            wait_lock->holder->priority = thread_current()->priority;
            wait_lock = wait_lock->holder->wait_on_lock;
            i++;
        }
/*대기하고 있는 lock을 잡고 있는 스레드의 우선순위가 현재 lock을 대기하고 있는 스레드의 우선순위보다 낮을시 우선순위를 변경
시켜준다.*/
        else return;
    }
}
```

해당 함수는 nested donation 문제를 해결해주기 위한 함수로서 nested depth 8 까지 대기하고 있는 lock 을 잡고 있는 스레드가 대기하는 lock 이 존재할 시, 그 lock 을 잡고 있는 스레드까지 우선순위를 변경시켜주는 함수이다.

e. lock_release 함수

```
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  remove_with_lock(lock);
  refresh_priority();

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

remove_with_lock 함수와 refresh_priority 함수를 호출하여 lock 해제 시 우선순위 문제와 donation list 정리를 해준다.

f. remove_with_lock 함수

```
void remove_with_lock(struct lock *lock){

    struct list_elem *elem;

    for(elem = list_begin(&(lock->holder->donations));elem != list_end(&(lock->holder->donations));){

        struct thread *t = list_entry(elem,struct thread, donation_elem);

        if(t->wait_on_lock == lock)
            elem = list_remove(&(t->donation_elem));
        else
            elem=list_next(elem);

    }          //donation list에서 해제한 lock을 대기하고 있는 스레드의 elem을 제거시켜준다.
}
```

해당 함수는 unlock 해주는 lock 을 대기하고 있는 스레드들의 elem 을 donation list 에서 제거해주는 함수이다.

g. refresh_priority 함수

```
void refresh_priority(void){

    list_sort(&(thread_current()->donations), &cmp_priority, NULL);
        //스레드의 우선순위가 변경 되었을 수 있으므로 list 내 우선순위를 재배열 하기 위해 sort함수 호출
    struct list_elem *elem = list_begin(&(thread_current()->donations));

    if(elem != list_end(&(thread_current()->donations))){        //리스트가 비어있지 않은 경우
        struct thread *t = list_entry(elem, struct thread, donation_elem);

        if(thread_current()->init_priority < t->priority)
            thread_current()->priority = t->priority;
        else thread_current()->priority = thread_current()->init_priority;
/*donation list의 우선순위가 가장 높은 스레드(현재 스레드가 잡고 있는 lock을 잡기 위해 대기중인 스레드)가 현재 스레드의 원래 우선
순위보다 높은 경우 우선 순위를 해당 스레드의 우선 순위로 변경 시켜 준다. */
    }
    else thread_current()->priority = thread_current()->init_priority;
}
```

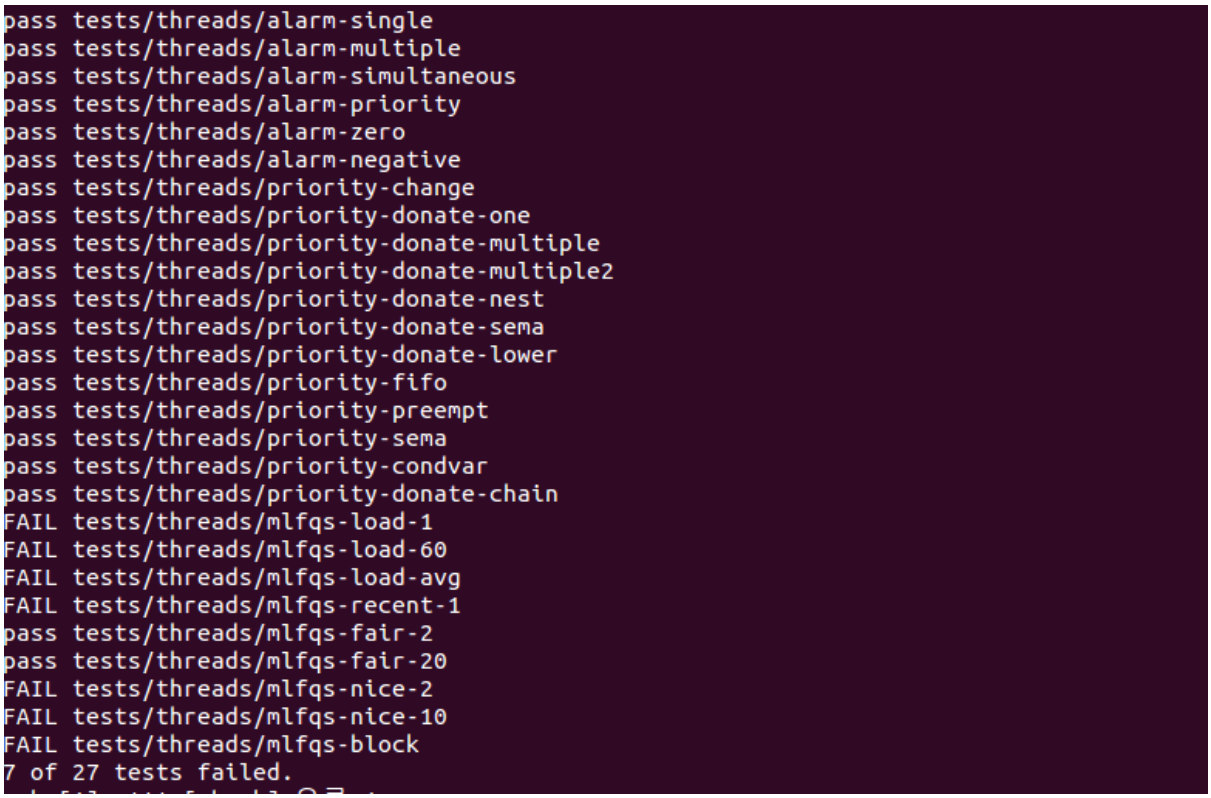
해당 함수는 lock 을 해제한 스레드의 우선순위를 재지정 해주는 함수로써 현재 스레드가 잡고 있는 lock 이 존재 할 시 해당 lock 을 대기 하고 있는 스레드들 중 가장 높은 우선순위를 가진 스레드와 현재 스레드의 원래 우선순위를 비교하여 더 높은 우선순위로 지정해주는 함수이다. 이 함수를 호출함으로써 multiple donation 문제를 해결 할 수 있다.

h. thread_set_priority 함수

```
void
thread_set_priority (int new_priority)
{
    thread_current ()->priority = new_priority;
    thread_current()->init_priority = new_priority;    //priority를 변경 할 경우 init priority도 같이 변경시켜준다.
    refresh_priority();    //변경한 우선순위가 제일 높은 우선순위 일 수 있으므로 refresh_priority를 호출시켜준다.
    test_max_priority();    // 해당 함수를 호출하여 우선순위가 변경되었으므로 test_max_priority 함수를 호출하여 CPU를 양보해
    주어야 하는지 판단한다.
}
```

스레드의 priority 를 변경시켜주는 경우 스레드의 init_priority 값도 같이 변경 시켜주고, 또한 변경 시켜준 우선 순위가 대기중인 스레드들의 우선 순위보다 더 높을 수 있으므로 refresh_priority 함수를 호출 시켜준다.

실행 결과



make check 결과 priority donate-* 테스트 결과가 pass 로 바뀌었음을 알 수 있었다.