Thread 과제 보고서

운영체제 (ELE-3021 12781)

교수명 : 원유집

조교명 : 오준택

학생명 :

서창범 2014004648, 서문은지 2016025487

제출일자 : 2018 년 5 월 11 일

Thread

- Alarm System Call

1. 과제 목표

Alarm : 호출한 프로세스를 정해진 시간 후에 다시 시작하는 커널 내부 함수이다.

핀토스에서는 알람 기능이 Busy waiting을 이용하여 구현되어 있어, Thread가 불필요하게 CPU를 점유하면서 대기한다. 이에 CPU 자원이 낭비되고, 소모 전력이 불필요하게 낭비될 수 있다. 본 과제에서는 알람 기능을 sleep/wake up을 이용하여 다시 구현 한다.

2. 함수 수정 및 추가

함수 수정 및 추가를 하기 전에 함수 수정 및 추가를 하기 위한 필드 몇가지를 먼저 추가 해주어야 한다.

첫째, thread 구조체에 깨어나야 할 tick을 저장할 변수인 int64_t wakeup_tick; 필드를 추가

둘째, THREAD_BLOCKED 상태의 스레드를 관리하기 위한 리스트 자료 구조체인 sleep_list를 추가 (thread.c/ thread_init(void) 함수내에서 sleep_list를 초기화 해주어야 한다.)

셋째, sleep_list에서 대기 중인 스레드들의 wakeup_tick 값 중 최소값을 저장하기 위한 변수인 int64_t next_tick_to_awake = INT64_MAX; 추가

(1) timer_sleep()

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

  /*
  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield (); //양보해줌
  */

  /* 새로 구현한 thread를 sleep queue에 삽입하는 함수를 호출*/
  ticks = start + ticks;
  thread_sleep(ticks);
}
```

busy waiting으로 구현되어 있는 부분을 주석 처리하고, thread_sleep(ticks); 를 호출하게 한다. 호출 시, sleep_list에 thread를 추가하고, thread_block을 해주게 된다.

(2) thread_sleep()

```
248 void thread_sleep (int64_t ticks) {
249   struct thread *cur = thread_current ();
250   /* list 삽입이 일어나는 동안 race condition이 발생하지 않도록
251      interrupt를 disable해준다. */
252   enum intr_level old_level = intr_disable ();
253   /* idle thread는 block상태가 되지않게 예외처리를 해준다.
254      그 이외의 thread들에 대해서는 sleep을 수행하기 위해 block시킨다.*/
255   if (cur != idle_thread)  {
256     cur->wakeup_tick = ticks;
257     list_push_back (&sleep_list, &cur->elem);
258     update_next_tick_to_awake (ticks);
259     thread_block ();
260   }
261   intr_set_level (old_level);
262 }
263
```

timer interrupt()로 tick을 체크하여 시간이 다 된 thread만 sleep_list에서 제거하고, ready_list에 추가하여 불필요하게 CPU를 점유하지 않는 방식으로 sleep을 구현하였다.

timer ticks는 OS부팅 이후 timer_interrupt가 호출될 때마다 카운트하는 값이다. timer tick 1번은 0.01초이다. (timer.h의 TIMER_FREQ 100 기준)

또한, thread_sleep() 수행 중에 race condition 발생을 방지하기 위해 intr_level old_level = intr_disable();로 인터럽트를 비활성화 해주었다. Idle thread의 경우에는 block 상태가 되는 것을 방지하기 위해 sleep이 되지 않게 예외처리를 한다.

sleep_list에 깨워야 할 ticks 정보를 담은 thread 구조체의 elem을 삽입하여 sleep list를 관리한다. 해당 리스트는 삽입 시 정렬하지 않으며, thread를 깨우기 위해 thread_awake() 에 의해 전체 순회가 된다.

(3) timer_interrupt ()

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

  if(ticks >= get_next_tick_to_awake())
  {
    thread_awake(ticks);
  }
}
```

매 tick 마다 sleep_list에서 깨어날 thread가 있는지 확인하여, 있다면 thread_awake(ticks)를 호출하도록 하였다.

(4) thread_awake()

```
275 void thread_awake (int64_t ticks) {
276   struct list_elem *elem = list_begin (&sleep_list);
277   struct list_elem *cur_elem = NULL;
278   struct thread *pcb = NULL;
279   /* next_tick_to_awake 값이 새로 갱신될 수 있도록 충분히 큰 값으로 초기화 한다.
280      INT64_MAX 는 비교 연산 시 버그가 있어서 더 작은값으로 대체하였다. */
281   next_tick_to_awake = INT32_MAX;
282     /* sleep_list를 처음부터 끝까지 순회한다. */
283   while (elem != list_end (&sleep_list)) {
284     pcb = list_entry (elem, struct thread, elem);
285     cur_elem = elem;
286     elem = list_next (elem);
287     /* 현재 ticks보다 해당 pcb의 ticks가 작으면 해당 스레드를 깨운다. */
288     if (pcb->wakeup_tick <= ticks) {
289       list_remove (cur_elem);
290       thread_unblock (pcb);
291     } else {
292       /* 현재 ticks 보다 큰경우 next_tick_to_awake 를 갱신한다. */
293       update_next_tick_to_awake (pcb->wakeup_tick);
294     }
295   }
296 }
```

sleep_list의 모든 entry를 순회하며, 깨워야 할 tick이 현재 tick보다 크다면 sleep_list 에서 제거하고 unblock한다. 만약 작다면, update_next_tick_to_awake()를 호출하여 남은 list_elem중 wakeup_tick이 가장 작은 값으로 업데이트 해준다.

(5) update_next_tick_to_awake(), get_next_tick_to_awake()

```
void update_next_tick_to_awake(int64_t ticks)
{
  /* next_tick_to_awake가 깨워야할 스레드중 가장 작은 tick을 갖도록
  업데이트 한다*/
  if (ticks < next_tick_to_awake)
    next_tick_to_awake = ticks;
}
```

```
int64_t get_next_tick_to_awake(void)
{
  /* next_tick_to_awake을반환한다. */
  return next_tick_to_awake;
}
```

Sleep_list에서의 가장 작은 ticks 값을 저장하는 next_tick_to_awake를 다루는 함수들이다. 직접 값을 참조하지 않고 함수를 통해 접근한다.

3. 결과 비교

기존 상태에서 pintos -- -q run alarm-multiple을 실행시켰을 때는 idle ticks가 0이였으나, sleep/wake로 수정 해준 이후에는, 0이 나오지 않는다. 왜냐하면, idle ticks는 idle 스레드가 실행된 ticks를 말하는데, busy waiting일 때는 block상태로 가는게 없기 때문에(계속 다른 스레드가 실행하게 되므로) idle thread가 실행될 일이 없다. Sleep을 다시 구현하여 ready_list가 완전히 비어있는 경우가 발생해 idle thread가 실행되어 idle tick의 수가 count되었기 때문에 idle ticks가 0이 아니게 되었다.

과제 수행 전

```
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, 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: 928 ticks
Thread: 0 idle ticks, 931 kernel ticks, 0 user ticks
Console: 2950 characters output
Keyboard: 0 keys pressed
Powering off...
```

과제 수행 후

```
(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...
```

Thread

- Priority Scheduling

1. 과제 목표

현재 핀토스의 스케줄러는 라운드 로빈으로 구현되어 있다. 이를 우선순위를 고려하여 스케줄링 하도록 수정한다. 이번 과제에서 할 것은 **(1)**이다.

(1) Ready list에 새로 추가된 thread의 우선순위가 현재 CPU를 점유 중인 thread의 우선 순위 보다 높으면 기존 thread가 CPU를 양보하여 새로 추가된 스레드가 CPU를 점유할 수 있도록 한다. 또한, 새로 생성된 thread의 우선순위가 현재 실행 중인 thread의 우선순위 보다 높으면, CPU를 점유할 수 있도록 한다.

(2) 여러 thread가 lock, semaphore를 얻기 위해 기다릴 경우, 우선순위가 가장 높은 thread가 CPU를 점유 한다.

다음은 우리가 이번 과제를 통해 수정해야 할 내용들이다.

(1) 새로 생성된 thread의 우선순위가 현재 실행중인 thread의 우선순위보다 높으면, 새로운 thread가 실행 중인 thread를 선점하도록 한다. -> thread_create()

(2) 기존 핀토스에서는 ready_list에 새로운 thread가 삽입 되어도 선점이 발생 되지 않는다. ready_list에 thread가 삽입 될 때 정렬 되도록 해준다. (thread_unblock(), thread_yield() 수정)

(3) thread의 우선순위가 변경되었을 때, 변경된 스레드의 우선순위보다 ready_list의 thread 중 가장 높은 우선순위보다 낮으면 CPU를 양보(yield)하게 한다.

2. 함수 수정 및 추가

(1) thread_create()

thread_unblock(t); 를 한 후에, test_max_priority(); 를 해주면 된다. 이 함수를 통해, 생성된 스레드의 우선순위가 현재 실행중인 스레드의 우선순위 보다 높으면 CPU를 양보한다.

```
394    t->next_fd = 2;
395
396    //커널에서 관리하는 모든 프로세스 리스트 구조체에 새로 생성된 PCB를 삽입함.
397    list_push_back(&thread_current()->child_list, &t->child_elem);
398    /* Add to run queue. */
399    thread_unblock (t);
400    test_max_priority ();
401    return tid;
402  }
```

(2) test_max_priority();

```
void test_max_priority (void)
{
  // ready_list를 가져올때 ready_list가 비여있지 않은지 확인해야 한다.
  if ( !list_empty(&ready_list) ) {
    struct thread *t = list_entry(ready_list.head.next, struct thread , elem);
    if( thread_get_priority() < t->priority) {
      thread_yield();
    }
  }

}
```

스레드가 새로 생성되어서 ready_list에 추가되거나((1)의 thread_create함수 경우를 말한다) , 현재 실행 중인 스레드의 우선순위가 재조정된 이후에, ready_list의 첫번째 스레드가 CPU 점유 중인 스레드보다 우선순위가 높은 상황이 발생할 수 있다. 따라서 스레드를 새로 생성하는 함수인 thread_create() 함수와 현재 스레드의 우선순위를 재조정하는 thread_set_priority 함수의 내부에 test_max_priority()를 추가한다. ready_list에서 우선순위가 가장 높은 스레드와 현재 스레드의 우선순위를 비교하여 스케줄링하게 된다.

(3) thread_unblock()

스레드가 unblock 될 때, cmp_priority함수를 이용하여 우선순위 순으로 정렬 되어 ready_list에 삽입 되도록 수정하였다.

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

  ASSERT (is_thread (t));

  /* 스레드가 unblock 될 때 우선순위 순으로 정렬되어
  ready_list에 삽입되도록 수정*/

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  //list_push_back (&ready_list, &t->elem);
  list_insert_ordered(&ready_list, &t->elem, cmp_priority, 0);
  t->status = THREAD_READY;
  intr_set_level (old_level);

}
```

(4) thread_yield()

현재 thread가 CPU를 양보하여 ready_list에 삽입 될 때, cmp_priority( )함수를 이용하여 우선순위 순서로 정렬되어 삽입 하도록 수정하였다.

```
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);
    list_insert_ordered(&ready_list, &cur->elem, cmp_priority, 0);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

(5) cmp_priority( )

list_insert_ordered( )함수에서 인자로 사용하기 위한 함수이다. 각 elem을 멤버로 하는 thread 구조체를 참조하여 priority를 비교한다. 첫번째 인자에 해당하는 priority가 더 크면 true를 반환한다.

```
bool cmp_priority (const struct list_elem* a_, const struct list_elem* b_,
void* aux UNUSED)
{
  return list_entry(a_, struct thread, elem)->priority > list_entry(b_, struct
thread, elem)->priority ;

}
```

(6) thread_set_priority( )

스레드의 우선순위가 변경되었을 때 우선순위에 따라 CPU 선점이 발생하도록 하는 함수이다.

```
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
  test_max_priority();
}
```

Thread

- Priority Scheduling and Synchronization

1. 과제 목표

현재 핀토스의 스케줄러는 라운드 로빈으로 구현되어 있다. 이를 우선순위를 고려하여 스케줄링 하도록 수정한다. 지난 과제에서 한 것은 (1)이고, 이번 과제에서는 (2)를 수행한다.

(1) Ready list에 새로 추가된 thread의 우선순위가 현재 CPU를 점유중인 thread의 우선 순위보다 높으면 기존 thread를 밀어내고 CPU를 점유할 수 있도록 한다. 또한, 새로 생성된 thread의 우선순위가 현재 실행중인 thread의 우선순위 보다 높으면, CPU를 점유할 수 있도록 한다.

(2) 여러 thread가 lock, semaphore, condition variable를 얻기 위해 기다릴 경우, 우선순위가 가장 높은 thread가 CPU를 점유한다. 현재 핀토스는 semaphore를 대기하고 있는 스레드들의 list인 watiers가 FIFO로 구현이 되어있다.

2. 함수 수정

(1) sema_down( )

semaphore를 얻기 위해 기다릴 경우, waiters 리스트에 우선순위대로 삽입되도록 수정한다. sema->value == 0 일 경우, 더이상 semaphore를 얻을 수 없으므로 waiters에 삽입 후, block을 하게 된다.

```
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);
      list_insert_ordered (&sema->waiters, &thread_current ()->elem, cmp_priority, NULL);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

(2) sema_up( )

thread가 waiters list에 있는 동안 우선순위가 변경 되었을 경우를 고려하여 unblock을 하기 전에 waiter list의 우선위를 정렬해주어야 한다. 그리고, waiters의 리스트에서 가장 우선 순위가 높은 스레드를 pop하여 unblock 해준다. 마지막엔 test_max_priority()를 통해 priority preemption을 해주어야 한다. 만약, waiters 리스트가 비어있다면 대기하고 있는 스레드가 없다는 뜻이므로, 이 과정 없이 바로 sema-> value++;를 할 수 있도록 예외처리한다.

```
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);

    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));

  }

  sema->value++;
  test_max_priority ();
  intr_set_level (old_level);
}
```

(3) cmp_sem_priority( )

첫번째 인자의 우선순위가 두번째 인자의 우선순위보다 높으면 1을 반환하고, 낮으면 0을 반환해주는 함수이다. cmp_priority()함수와 다르게, semaphore_elem에서는 바로 priority에 접근을 할 수 없으므로 semaphore_elem으로부터 스레드 구조체 (pcb)를 찾아야 한다. 찾은 후에 priority를 비교하여 return해주면 된다.

```
bool cmp_sem_priority (const struct list_elem *a, const struct list_elem *b, void *a
ux UNUSED) {
  struct semaphore_elem *sa = list_entry (a, struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry (b, struct semaphore_elem, elem);
  struct thread *thread_a = (struct thread*)pg_round_down (sa);
  struct thread *thread_b = (struct thread*)pg_round_down (sb);

  return thread_a->priority > thread_b->priority;
}
```

(4) cond_wait( )

기존의 list_push_back()함수를 주석처리하고, list_insert_ordered함수로 수정한다. 그리고 인자인 &waiter.elem으로 우선순위를 비교해주기 위해서 cmp_sem_priority() 함수를 사용하여 condition variable의 waiter list에 우선순위 순서로 삽입되도록 한다.

```
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);
  list_insert_ordered (&cond->waiters, &waiter.elem, cmp_sem_priority, NULL);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

(5) cond_signal( )

sema_up을 하기 전에(대기 중에 우선순위가 변경되었을 가능성을 생각해야 하므로) condition variable의 waiters list를 우선순위로 재정렬 해주어야 한다.

```
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);
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
}
```

Thread

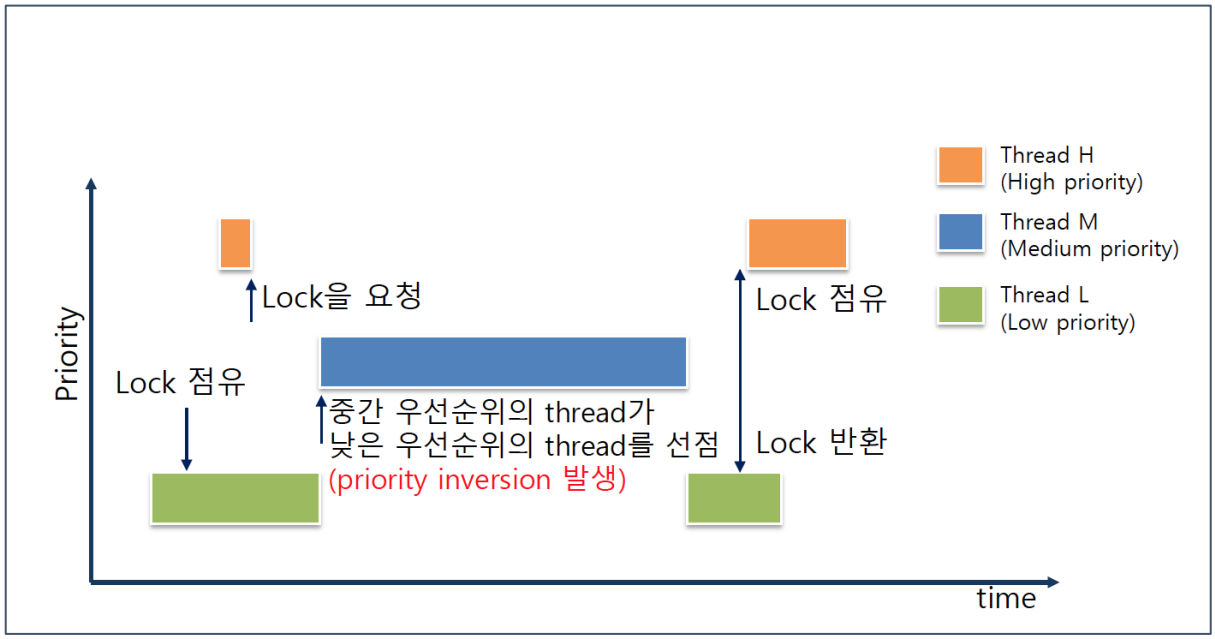
- Priority Inversion Problem

1. 과제 목표

(1) Priority donation 구현 (2) Multiple donation 구현 (3) Nested donation 구현

2. 과제 설명

위의 과제까지는 락을 대기 중인 스레드 중에 우선순위가 높은 스레드가 락을 먼저 획득 할 수 있도록 해준 것이었다. 하지만, 아직 특별한 이슈가 남아있다.



(1) Priority donation

스레드 L이 락을 점유하여 실행하고 있는 중에, 스레드 L보다 높은 우선순위를 가진 스레드 H가 락을 요청하여도 Lock은 가로챌 수 없으므로 스레드 L은 wiater list에 들어가 대기를 하게 된다.

그런데, 스레드 L보다 우선순위가 높은 Thread M이 CPU를 점유 하게 되면, 스레드 M보다 더 높은 스레드 H가 M을 기다리는 상황이 발생한다. 이 문제를 우선순위가 역전 되었다고 표현한다.

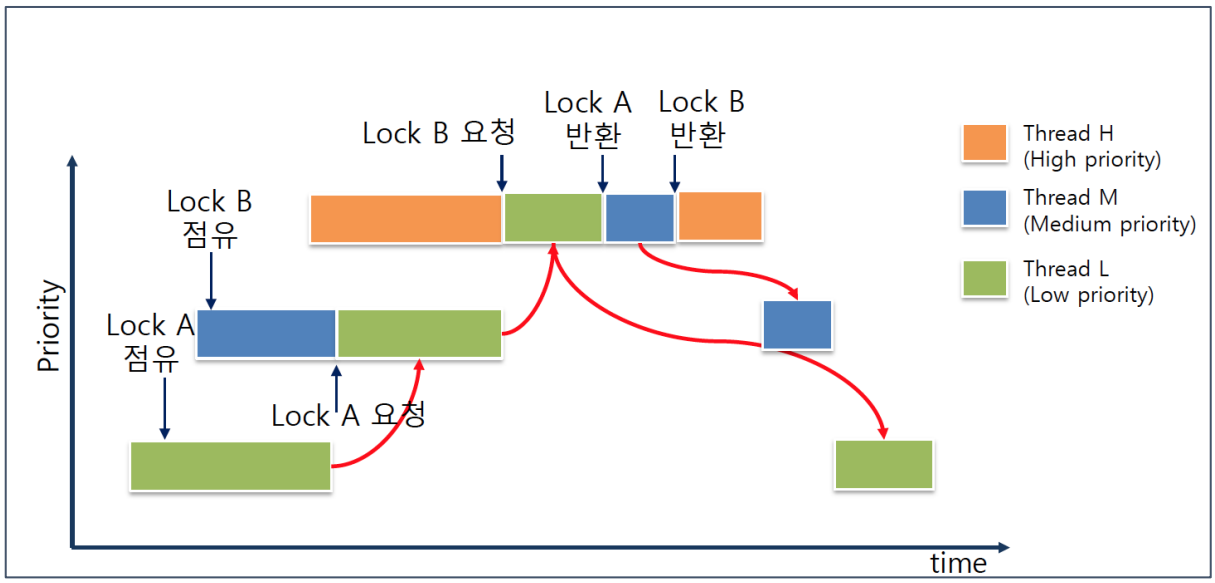
이것의 해결방법은 스레드 H가 락을 요청하고 대기하러 가기 전에, 자신의 우선순위를 donation 해주어 스레드 L이 빠른 우선순위(H의 우선순위)로 CPU를 점유하며 할 일을 하게 된다. 그럼 스레드 M이 CPU를 선점하지 못하고 우선순위 역전 현상을 막을 수 있다.

(2) Multiple donation

예를 들어, 스레드 L, M, H/ 락 a, b가 있다. 우선순위가 가장 낮은 스레드 L이 락 a, b를 둘다 점유하고 있고 스레드 M이 락 a를 요청하게 되면 스레드 L은 스레드 M의 우선순위를 donation받은 상태일 것이다. 여기서 스레드 H가 락 b를 요청하면 스레드 L은 H의 우선순위를 donation받게 된다. 스레드 L이 락 b를 해지하면, 우선순위를 반납해야 하는데 이때 스레드 M의 우선순위가 되어야 한다.

즉, 자신이 donation받았던 우선순위들을 모두 기억해야 한다. 이를 위해 PCB에 donation list를 만들어서 이전 상태의 우선순위를 기록 해놓도록 한다.

(3) Nested donation



스레드 H가 lock B를 얻기 위해 대기 하고있고, 이때 스레드 M은 lock A를 얻기 위해 대기 하고 있다. 이 상황에서, 스레드 H가 대기하게 되면서 스레드 M한테 준 우선순위가 쓸모 없어지게 된다. (스레드 M은 lock A를 얻기 위해 이미 대기 하고있는 중이므로)

그래서, 스레드 H의 우선순위는 가장 마지막에 실행 중인 스레드까지 우선순위를 donation해주어야한다. 위 그림에서는 스레드 H의 우선순위가 스레드 M, L 에게 모두 donation 되어야한다.

3. 함수 수정

(1) init_thread( )

```
t->init_priority = priority;
t->wait_on_lock = NULL;
list_init (&t->donations);
```

priority donation 관련 자료구조를 초기화 하는 코드를 삽입 해주었다.

(2) lock_acquire( )

해당 lock을 이미 점유하고 있던 thread가 존재한다면 다음 작업을 수행한다.

- 현재 스레드의 wait_on_lock변수에 획득하기를 기다리는 lock의 주소를 저장
- 현재 스레드의 donation_elem을 lock을 점유한 thread의 donation_list에 추가한다. 이 과정은 lock을 점유하고 있는 thread가 이전 priority, 즉 donation 받았던 priority를 기억하기 위해 필요하다.
- nested priority donation을 수행하기 위해 donate_priority를 수행한다. 실질적으로 lock을 점유하고 있는 thread의 priority를 변경해주는 작업을 한다.

그리고, if문 밖에서 lock을 획득 한 후, lock holder를 갱신한다.

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

  if (lock->holder) {
    cur->wait_on_lock = lock;
    list_insert_ordered (&lock->holder->donations, &cur->donation_elem,
cmp_priority, NULL);
    donate_priority ();
  }

  sema_down (&lock->semaphore);

  cur->wait_on_lock = NULL;
  lock->holder = thread_current ();
}
```

(2) donate_priority( )

자신이 가지고 있는 priority를 자신이 acquire하려고 하는 lock을 갖고있는 thread에게 donate하고, nested되어 있는 경우 최대 8개 depth 까지 donation이 이뤄진다.

lock_acquire(), thread_set_priority()에 의해 호출되어진다.

```
void donate_priority (void) {
  int depth = 1;
  struct thread *thread = thread_current ();
  struct lock *lock = thread->wait_on_lock;
  while (depth < MAX_DEPTH && lock) {
    if (   lock->holder
        && lock->holder->priority < thread->priority ) {
      lock->holder->priority = thread->priority;
      thread = lock->holder;
      lock = thread->wait_on_lock;
      depth++;
    } else {
      break;
    }
  }
}
```

해당 스레드가 기다리고 있는 락이 있을 경우, 현재 스레드의 우선순위보다 현재 스레드가 기다리고 있는 락을 점유한 스레드의 우선순위가 작을 때, 락을 점유하고 있는 스레드의 우선순위가 높아질 수 있도록 현재 스레드의 우선순위를 donation한다.

반복문 내의 조건문을 만족하지 못하면, nested된 thread가 더이상 없거나, 있더라도 자신보다 우선순위가 더 높은 경우이므로 더 이상의 nested priority donation이 불필요하므로 nested priority donation을 수행하는 반복문을 탈출한다.

(3) lock_release( )

remove_with_lock()함수와 refresh_priority()함수를 추가해주었다.

이 함수를 추가함으로써 락을 해제할 때 donate 받은 priority를 반납하고(donation list에서 해당 락을 기다리면서 priority donation을 해준 스레드 elem을 제거한다), donate 받기 전 priority로 복원한다.

```
248 void
249 lock_release (struct lock *lock)
250 {
251   ASSERT (lock != NULL);
252   ASSERT (lock_held_by_current_thread (lock));
253
254   lock->holder = NULL;
255   remove_with_lock (lock);
256   refresh_priority ();
257   sema_up (&lock->semaphore);
258 }
259
```

(4) remove_with_lock( )

락을 해제하기 전에 해당 lock을 기다리기 위해 우선순위를 현재 스레드에게 donate한 스레드들의 donation_elem을 현재 스레드의 donations list에서 제거한다. 이는 donated priority가 락을 해제함으로써 반납되는 것을 의미한다.

```
void remove_with_lock (struct lock *lock) {
  struct list_elem *elem = NULL;
  struct list *lock_waiters = &lock->semaphore.waiters;
  struct thread *donator = NULL;
  for (elem = list_begin (lock_waiters)
      ; elem != list_end (lock_waiters); elem = list_next (elem)) {
    donator = list_entry (elem, struct thread, elem);
    list_remove (&donator->donation_elem);
  }
}
```

(5) refresh_priority( )

어떤 스레드가 우선순위가 새로 바뀌거나, lock을 반환하고 donated priority를 반납하면서 우선순위가 변경될 때, 변경된 우선순위와 다른 lock을 점유하면서 다른 스레드들이 해당 lock을 대기하면서 donate해준 priority들과 비교하여 가장 높은 priority로 적용하는 작업을 한다.

첫번째 if문에서 자신의 priority는 바뀌었지만 비교할 donated priority가 없다면, 새로 바뀐 priority를 적용하고 return 한다. init_priority에는 초기에 설정된 priority가 저장되어 있어서 donated priority를 반납 받고, 원래 priority로 돌아갈 수 있다.

donated priority 목록(donations)가 비어있지 않다면, donated priority가 있는 경우, 새로 변경된 priority와 donated priority들을 비교하여 최대값을 취한다. donations list는 우선순위에 따라 내림차순으로 정렬되어 있어 리스트 head가 가리키고 있는 element가 가장 큰 값이다.

```
void refresh_priority (void) {
  struct thread *cur = thread_current ();
  struct thread *donator = NULL;
  struct list_elem *elem = NULL;
  int max_priority = cur->init_priority;

  if (list_empty (&cur->donations)) {
    cur->priority = cur->init_priority;
    return;
  }

  elem = list_begin (&cur->donations);
  donator = list_entry (elem, struct thread, donation_elem);

  if (max_priority < donator->priority) {
    max_priority = donator->priority;
  }
  cur->priority = max_priority;
}
```

(6) thread_set_priority( )

Priority donation을 고려하여 thread_set_priority 함수를 수정하였다. refresh_priority()함수를 사용하여 우선순위 변경으로 인한 donated priority와 초기 priority를 고려하여 priority를 갱신하는 작업을 하고, donation_priority()함수와 test_max_priority() 함수를 사용하여 priority donation을 수행하고 priority를 고려한 스케줄링을 수행한다.

```
void
thread_set_priority (int new_priority)
{
  struct thread *cur = thread_current ();
  int old_priority = cur->priority;
  cur->init_priority = new_priority;
  refresh_priority ();
  if (old_priority < cur->priority) {
    donate_priority ();
  }
  test_max_priority ();
}
```

<Thread 최종 결과>

1. make check

```
ckdqja0225@ubuntu:~/pintos/project/src_changbeom/threads$ make check
cd build && make check
make[1]: Entering directory `/home/ckdqja0225/pintos/project/src_changbeom/threads/build'
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
7 of 27 tests failed.
```

2. bitbucket 주소

git clone

https://seomooneunji@bitbucket.org/pintos_ejcb/project.git

에서 **" src_changbeom 폴더 "** 입니다.