

Lab1 Final Report



Member #1

Name: Suwon Yoon

Student ID: 20210527



Member #2

Name: Jiwon Lee

Student ID: 20210706

1. Alarm Clock

1.1. Solution

- busy waiting으로 구현된 기준의 방법에서는 ready_list에 sleep 하는 thread를 저장해두었다.
- sleep_list를 thread.c에 만든다.
- thread_init 함수에 sleep_list를 초기화하는 코드를 추가한다.
- sleep_list는 struct thread의 elem 변수로 thread를 관리한다.
- 언제 일어나는지 정보를 저장해야하니까 struct thread에 int64_t wakeup_tick 만든다. 이 변수에 언제 일어날지 저장할 것임.
- timer_sleep함수에서 thread_sleep을 호출해서 thread를 재울 것이다.
- thread_sleep 함수를 만들 것이다. → thread.h, thread.c에 만든다.
- thread_sleep 함수

```
void
thread_sleep (int64_t wakeup_tick)
{
    // TCB에 일어날 시간 저장
    // sleep_list에 추가
    // block

    ASSERT (!intr_context ());
    enum intr_level old_level;
    struct thread *cur;

    old_level = intr_disable ();
    cur = thread_current ();

    cur->wakeup_tick = wakeup_tick;
    list_insert_ordered (&sleep_list, &cur->elem, compare_wakeup_tick, NULL);

    thread_block();
    intr_set_level(old_level);
}
```

- interrupt handling 중에는 thread를 sleep 하지 않는다는 것을 보장 → ASSERT
- sleep 중에 다른 thread 실행되면 안되니까 interrupt를 꺼준다.
- 우리 구현에서는 일어날 tick(wakeup_tick)을 기준으로 정렬하여 sleep thread를 관리할 것임. 따라서 list_insert_ordered 함수 사용. list_insert_ordered 함수를 보면 비교를 위한 함수가 필요함을 알 수 있다. 그 함수를 cmp_wakeup_tick 함수로 만든다.(왜냐하면 c언어에서 람다함수 사용 불가)
 - compare_wakeup_tick 구현
 - 작은게 앞으로 오도록.

```

static bool // list_insert_ordered에 넣을 때 사용되는 wakeup_tick 비교용 함수
compare_wakeup_tick (struct list_elem *e1, struct list_elem *e2, void *aux)
{
    return list_entry(e1, struct thread, elem)->wakeup_tick > list_entry(e2, struct thread, elem)->wakeup_tick;
}

```

- 이제 `thread_block` 함수 호출한다.
 - `thread`의 `status`는 `THREAD_BLOCKED`가 될 것임.
 - `thread`는 `sleep_list`에서 관리됨.
- 이제 `thread_sleep`함수를 다 만들었으니 깨우는 것을 해보자.
 - `unblock`해야지. 그려려면 `thread_wakeup` 함수를 만든다.
- `timer_init` 함수를 보면 `timer_interrupt`를 핸들러로 등록한다. 따라서 이 함수에서 `ticks`를 증가시키는데, 이때마다 `thread_wakeup`함수를 호출하여 깨울 `thread`가 있으면 깨운다.

```

/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
    thread_wakeup (ticks);
}

```

- `thread_wakeup`

```

void
thread_wakeup (int64_t current_tick)
{
    // sleep list의 첫번째 원소의 wakeup_tick이 현재 tick과 같은지 확인
    // 같으면 깨우기
    // sleep_list에서 제거
    // unblock
    // 빈 경우에는 바로 리턴
    // 일어날 시간이 동일할 수 있으니 iteration 필요

    if (list_empty(&sleep_list))
        return;

    struct list_elem *first_element_in_sleep_list = list_begin(&sleep_list);
    struct thread* t= list_entry(first_element_in_sleep_list, struct thread, elem);
    ASSERT( t != NULL )

    while (t->wakeup_tick <= current_tick)
    {
        list_pop_front(&sleep_list);
        thread_unblock(t);

        if (list_empty(&sleep_list))
            return;

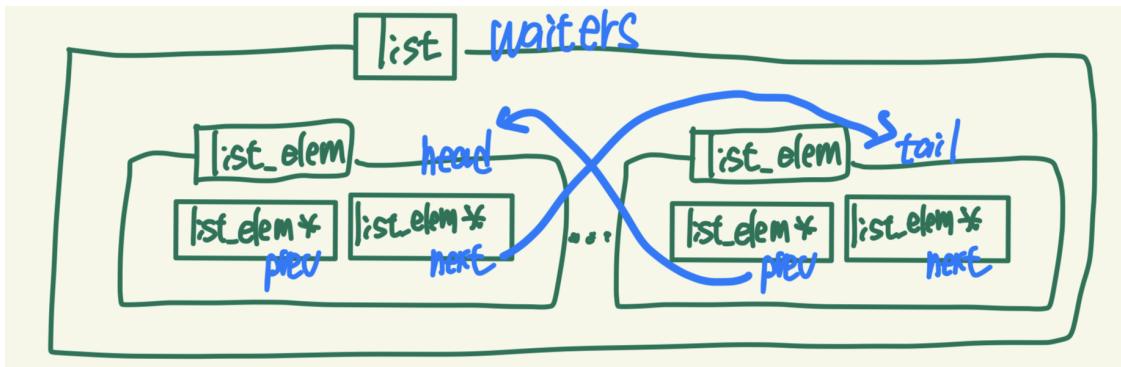
        first_element_in_sleep_list = list_begin(&sleep_list);
        t = list_entry(first_element_in_sleep_list, struct thread, elem);
        ASSERT(t != NULL)
    }
}

```

1.2. Discussion

- `list`의 운영 과정을 명확히 파악하고 있지 못했다.

- `list_begin(&sleep_list)` 값이 NULL이면 리스트가 빈 것으로 생각하였으나, list.c 코드를 더 살펴보니 `list_begin(&sleep_list)` 값은 NULL 값을 줄 수 없는 구조임. 왜냐하면 리스트를 초기화하여 빈 상태라고 할 지라도



아래와 같은 그림처럼 초기화되기 때문에 head랑 tail은 존재한다. 따라서 리스트가 비었는지 확인하려면 `list_empty`을 사용해야함.

- `compare_wakeup_tick`
 - priority의 경우 큰 것이 먼저 오게 정렬해야하지만 `wakeup_tick`의 경우 작은 것을 먼저 오게 해야한다. 이 순서를 반대로 하여 디버깅하는데 시간을 오래 썼다.

1번 구현이 완료된 commit: 46638249021f651626a59f66484dc257da3d518c

2. Priority Scheduler

현재 thread가 스케줄링 될 때 `next_thread_to_run` 함수를 호출한다. 이 함수에서는 `ready_list`의 맨 앞 원소를 리턴하고 있다. 따라서 맨 앞 원소가 priority가 가장 높은 상태를 유지하도록 `ready_list`를 구현하면 이 함수의 수정 없이도 priority scheduler를 구현할 수 있다.

그렇다면 `ready_list`로 thread가 들어갈 때 priority에 따라 정렬되도록 하자.

- `thread_unblock`와 `thread_yield` 함수에서 `list_push_back`으로 thread를 넣지 말고 앞 timer 구현에서 했던 것처럼 `list_insert_ordered`와 `compare_priority` 함수를 사용하여 적절한 자리로 찾아들어가도록 하자.

`ready_list`로 thread가 들어가는 부분은 `thread_unblock` 함수 또는 `thread_yield` 함수가 호출되었을 때이다. 따라서 두 경우를 처리한다. 이로써 priority에 따라 thread가 스케줄링 되도록 하는 구현을 완료하였다.

이제 'preempt' 특성을 위한 구현을 해야한다. 지금 구현에서는 우선수위가 10인 thread A가 실행중일 때 `ready_list`에 우선순위가 20인 thread B가 존재한다고 하더라도 thread B는 실행되지 않는다. `ready_list`에 있는 thread의 priority는 현재 실행중인 Thread의 priority보다 낮은 것이 보장되어야 한다.

위 상황이 어떤 양상으로 발생되는지 알아보자.

1. thread가 생성되어 `ready_list`에 들어간 경우, 현재 실행 중인 thread보다 priority가 높을 수 있음.
2. `thread_set_priority` 함수를 실행시켜 현재 thread의 priority가 바뀔 수 있음.
3. `thread_unblock` 함수가 호출되어 `ready_list`에 thread가 추가되는 경우.

- discussion

첫 번째 구현: 1,2 → `thread_create` 함수와 `thread_set_priority` 함수 뒤에 '현재 thread와 priority비교하는 단계' 추가하여 현재 실행 중인 thread와 `ready_list`의 맨 앞 원소의 priority를 비교해야 한다.

두 번째 구현: 1번이 3번에 포함되는 것이라는 것을 깨닫고 나서 `thread_unblock` 함수 뒤에 `obey_priority` 함수를 넣었다.

마지막 구현: 근데 다시 생각해보니까 `thread_unblock` 함수는 thread를 스케줄링 시키는 것을 직접적으로 행하지 않는다는 특징이 있었다.(`thread_unblock`은 `ready_list`에 thread를 넣어주고 TCB 상태를 ready로 바꿔주는 역할을 함.)

`thread_block`에서 실질적인 스케줄링을 함 → 기존 구현에서 `thread_unblock`이 진짜 thread를 스케줄링 시키지 않는다는 특성을 고려하여 `sema_up` 함수가 구현되어 있음.)

특히 sema_up 함수에서 thread_unblock을 호출하는데, 여기서 thread_unblock이 thread를 스케줄링 시키는 것을 행하게 되면 semaphore 값이 아직 안 올라간 waiting thread를 스케줄링하게 되어 문제가 생길 수 있다. 그렇기에 thread_unblock에서는 obey_priority 함수를 호출하면 안된다는 결론을 내렸다.

결국 처음에 하려는대로 하게됨: thread_create 함수 끝나기 전에 obey_priority 함수 넣기로 함. 그리고 sema_up 함수에서 thread_unblock 함수 호출 이후에 obey_priority 함수 추가함. 이러한 구현을 하기 위해서는 아래의 기존 thread_unblock 함수와 기존 thread_block 함수의 기본적인 구현을 정확히 이해해야 한다.

```

224  /* Transitions a blocked thread T to the ready-to-run state.
225   This is an error if T is not blocked. (Use thread_yield() to
226   make the running thread ready.)
227
228   This function does not preempt the running thread. This can
229   be important: if the caller had disabled interrupts itself,
230   it may expect that it can atomically unblock a thread and
231   update other data. */
232 void
233 thread_unblock (struct thread *t)
234 {
235     enum intr_level old_level;
236
237     ASSERT (is_thread (t));
238
239     old_level = intr_disable ();
240     ASSERT (t->status == THREAD_BLOCKED);
241     list_push_back (&ready_list, &t->elem);
242     t->status = THREAD_READY;
243     intr_set_level (old_level);
244 }

```



```

208  /* Puts the current thread to sleep. It will not be scheduled
209   again until awoken by thread_unblock().
210
211   This function must be called with interrupts turned off. It
212   is usually a better idea to use one of the synchronization
213   primitives in synch.h. */
214 void
215 thread_block (void)
216 {
217     ASSERT (!intr_context ());
218     ASSERT (intr_get_level () == INTR_OFF);
219
220     thread_current ()->status = THREAD_BLOCKED;
221     schedule ();
222 }

```

- preempt를 가능하게 하는 함수의 이름은 obey_prioirty이다.

```

void
obey_priority()
{
    int highest_priority_in_ready_list;

    ASSERT(!intr_context())

    if (list_empty(&ready_list))
        return;

    highest_priority_in_ready_list = list_entry(list_begin(&ready_list), struct thread, elem)->priority;

    if (highest_priority_in_ready_list > thread_current()->priority)
        thread_yield();

}

```

한편, 락이나 세마포어, 컨디션 변수 각각은 waiters를 가지고 있다. 이 waiter도 우선순위에 따라 정렬되도록 해야한다. 동기화 관련 개념을 살펴볼 때의 흐름을 고려하면 세마포어, 락, 모니터 순으로 보는게 좋겠다 (이걸 unblock 함수의 내부에 뒷부분로 직에 추가하면 안되나? unblock 할 때마다 지금 도는거랑 비교하는거지 → 아니라는 것을 위에 1,2,3트하면서 깨달음)

- 세마포어 하면 락도 된다. 락은 세마포어를 활용하여 만들어진 것이 이유가 될 수 있다. - 앞에서와 같이 list_insert_ordered 함.

```

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, compare_priority, NULL);
        thread_block ();
    }
    sema->value--;
    intr_set_level (old_level);
}

```

한 가지 구현하지 않은 부분이 있는데

모니터에서의 priority scheduling은 세마포어와 유사하게 구현된다.

condition을 사용할 때, cond_wait 함수에서 세마포어를 initialize한다. 즉 호출을 할 때 세마포어를 만드는 것이다. 그리고 이렇게 생성된 세마포어는 cond_wait 함수가 종료되면 함수 종료와 함께 사라진다. 즉, condition variable이라고 특별한 것이 아니라 세마포어와 비슷한데, 더 추상화레벨이 높은 세마포어로 해석할 수 있겠다.

따라서 세마포어와 동일한 논리로 정렬해준다.

이제 donation 관련 문제를 해결해야 한다.

donation을 위해서는 lock에 대한 정보를 다루기 위해 TCB에 새로운 변수를 추가할 필요가 있다.

thread의 입장에서...

- 1- 자신이 어떤 thread에게 priority를 donate 하였는지

→ struct thread *

- discussion

처음에는 “자신이 어떤 thread에게 priority를 donate 하였는지”에 대한 정보를 따로 기록할 필요가 있다고 생각했다.
하지만 이는 이후의 구현에서 필요하지 않아 제거하였다.

- 2- 자신이 어떤 thread들로부터 priority를 donate 받았는지

→ struct list **donated_from_these_threads** (multiple donation이 있으니 하나로 안된다. - 하나의 thread는 여러 lock 을 가질 수 있다.)

→ struct list_elem **donor_elem** (donated_from_these_threads의 elem으로 사용될 수 있으려면 필요)

- ready와 block은 공존할 수 있는 상태가 아니므로(exclusive하므로) elem 원소 하나로 사용하였는데 donor의 status는 어떠한지 보장할 수 없음. → 이후에 lock_release 함수에서 더 이상 donate를 할 필요 없을 때, donated_from_these_threads에서 제거하는 용도로 사용한다.

- 3- 어떤 lock을 기다리고 있는지 → struct lock * **waiting_lock**

- 4- donate를 받은 경우 끝나고 다시 돌아갈 수 있도록 기존의 priority 저장해두는 변수 → int **original_priority**

총 3가지 정보를 기록해야 한다.(**init_thread**에 추가한 변수 초기화 로직 추가, 관련 변화에 대한 추가 처리는 이후에 할 것)

이를 위한 멤버 변수를 struct thread에 추가하자.

어떤 함수를 수정하고 어떤 기능을 더 만들어야 하는지 알아보기 위해 priority donation이 필요한 상황을 먼저 생각해보자.

priority가 10인 thread A는 lock K를 acquire하고 있다.

priority가 20인 thread B의 실행 과정 중 lock K를 얻어야 하는 상황이 있다.

priority가 15인 thread C는 lock K와 무관하다.

지금까지의 구현에서는 thread B가 실행되다가 lock K를 기다려야 하는 시점에서 우선순위가 20보다 낮은 thread C가 먼저 실행된 다음에서야 thread A가 실행된다. 따라서 priority가 20인 thread B가 priority가 15인 thread C를 기다리는 꼴이 되어버린다.

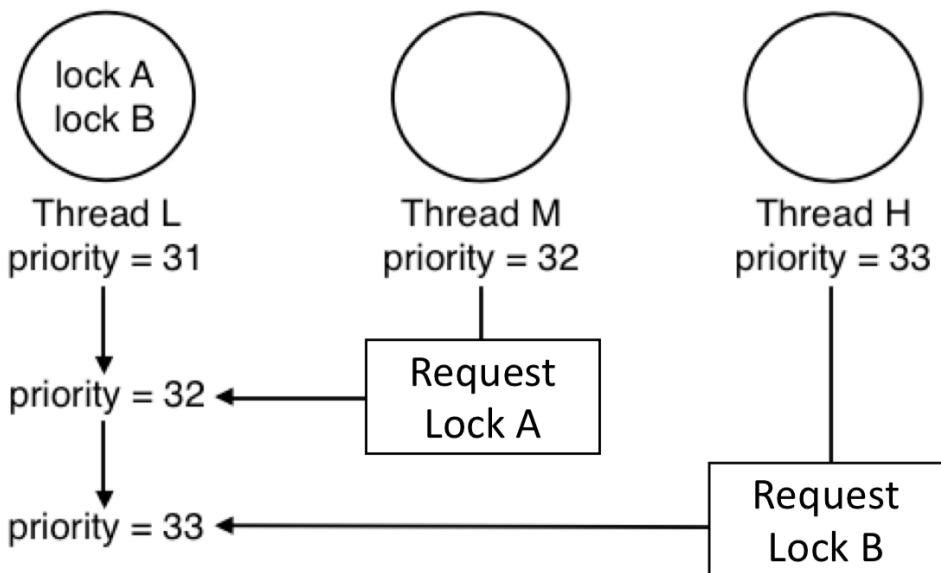
이는 'priority가 높은 thread를 먼저 실행시키자'라는 목적을 가진 priority scheduling에 반하는 동작이다. 이러한 문제를 해결하기 위해 thread B의 우선순위를 thread A에게 잠시 'donate'하여 thread A가 thread C보다 먼저 실행되게 하여 궁극적으로는 thread B가 thread C보다 먼저 실행될 수 있게 할 것이다. 물론 thread C 입장에서는 자신보다 priority가 낮은 thread A가 먼저 실행되는 것이 마음에 들지 않을 수는 있으나, thread A는 단지 thread B(thread C보다 우선순위가 높다.)의 실행을 위한 최소한의 부분(thread A가 lock K를 release할 때)까지만 실행되기 때문에 thread C 입장에서도 억울할 부분은 없다.

위 상황이, 기존에 구현된 코드의 어떤 부분에서 '시작'되는지 생각해보면 lock_acquire 함수라는 것을 알 수 있다. 왜냐하면 thread B가 lock K를 얻어야 하는 상황이 문제의 시작이기 때문이다. 만약 어떤 thread도 lock K를 acquire하고 있지 않은 상태라면 lock_acquire 함수는 기존과 동일하게 작동하면 된다.

하지만 다른 thread(thread A)가 lock K를 acquire하고 있는 상태라고 하면 lock_acquire를 호출한 thread(thread B)는 기존의 구현처럼 수동적으로 기다리기만 해서는 안된다.(단지 sema_down 함수만을 호출하는 것이 현재 구현) 대신에 lock을 acquire하고 있는 thread를 실행시켜야 한다. 이를 위해 thread B는 자신의 priority를 thread A에 잠시 빌려줄 필요가 있다.

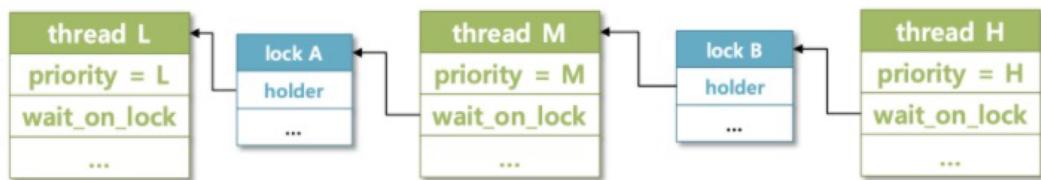
이 구현은 sema_down 이전에 해야겠다.

- **lock_acquire** 부분 수정
 - donate을 하기 위한 정보 저장
 - **thread A의 original_priority**에 **thread A의 priority** 저장
→ 대신에 init_thread 함수에서 thread를 생성할 때 같이 초기화하기. (가정: thread의 priority는 donation에 의해 겉보기로 바뀔 뿐이지 초기에 설정된 priority가 바뀔 일이 없기 때문에 이와 같이 구현하여도 된다 - mlfqs 구현에서는 예외.)
thread_set_priority 함수에 **original_priority**도 수정하도록 변경
 - **thread B의 waiting_lock**에 lock K 저장
 - **thread A의 donated_from_these_threads**에 **thread B**를 추가
 - priority가 크면 앞에 저장되도록 하기 위한 비교함수 필요: 이는 multiple donation을 위해 필요하다.
 - discussion
하나의 thread가 여러 lock을 가질 수 있다: multiple lock을 어떻게 얻지? 이 상황이 어떻게 생기나. - L이 있다가 M이 '생기고' M이 L에게 donate해서 32로 돌다가 H가 '생기고' H가 L에게 donate한다. 이때 H때문에 들던 L이 lock B를 release하면 31로 갈게 아니라 32로 가서 M이 시킨 할 일을 해야한다.)



- donate 진행

지금까지 다루고 있는 예시는 매우 간단한 예시라 donation을 해야하는 상황이 한 번만 존재한다. 하지만 thread B와 thread A의 관계가 반복적으로 나타날 수 있다.(nested donation) 따라서 재귀적으로 priority donation이 되도록 구현하는 것이 바람직하겠으나, 테스트 중 하나를 구현해 둔 priority-donate-chain.c 파일에 NESTING_DEPTH 가 8로 설정되어 있다. 따라서 반복문으로 8번까지만 iterate하는 방식을 사용하도록 하겠다.



- 우선순위가 가장 높은 현재 thread(current_thread 함수 사용)를 “시작”으로 donate를 시작한다.

- donate 과정(with discussion)

thread가 기다리고 있는 lock이 ‘있다면’

첫 번째 시도: 그 lock을 소유하고 있는 thread(thread A)의 priority를 original_priority(thread A의 original priority)에 저장하고(앞에서 하기했는데 여기서 또해주자. 반복을 하다보면 아닐 수 있으니까? → 생각해보니까 이렇게 하면 문제가 생긴다. 여러 Lock을 얻는 경우 original priority가 덮어써지는 문제 발생함. 따라서 여기서는 original_priority를 가만히 두자. original_priority는 lock_acquire에서 저장하고 thread_set_acquire 함수 말고는 건들지말자.)

마지막 구현: 현재 thread(thread B)의 priority를 thread가 기다리고 있는 lock에 양보한다.

- 이 과정에서 곧바로 priority를 양보한 thread를 실행시키는 것이 아니다. lock_acquire 함수 아래에 있는 sema_down 함수에서 thread_block 함수가 불릴 때 새로운 thread가 스케줄링 된다.
그런데, 위에서는 priority를 바로 덮어버리는 식으로 priority를 변경하였기 때문에 이 정보가 아직 각 lock → sema → waiters의 순서에 반영이 안되어있는 상태이다.

이 과정을 반복하다가,
thread가 기다리고 있는 lock이 ‘없다면’
해당 thread를 수행하면 된다. 더 이상의 nested 구조는 없다.

- 여기서 sema_down함수가 호출되어 thread_block함수가 호출되고, 실제로 thread가 스케줄링되는데 위의 iteration에서는 priority를 변경한 것 뿐이지 그 것이 실제 list(각 sema → waiters와 ready_list)에는 반영되지 않았다.

따라서 sema_up함수에서 thread_unblock을 하기 전에 정렬하는 로직을 추가한다.

- discussion

그렇다면 ready_list 정렬이 필요한가?

첫 번째 시도: 아니다. 왜냐하면 donation을 받아서 priority가 높아지는 thread들은 ready_list에 있을 수 없기 때문. ASSERT문으로 보장했다.

```
// nested donation에서 donate를 받는 Thread가 ready list에 없다는 것이 보장되어야 함  
// 왜냐하면 반복문에서 priority값을 변경하기 때문에, 이것이 보장되어야 아래 sema_down에서 스케줄링이  
// 올바르게 된다.
```

두 번째 시도: 그렇다. 그래서 sema_down 함수를 실행해서 ...

마지막 구현: 결론적으로는 필요 있다. 그 이유는 조금 복잡하다. 차근차근 따라가보자.

- 우선 current_thread는 READY상태이다.

- current_thread를 제외한 ‘wating_lock이 있는 thread’는 READY 상태일 수 없다. (lock을 기다리고 있으니까.)

다시 말하자면, ‘donate를 해주는 thread’(if(!))에서 break 당하지 않은 thread는 READY 상태가 아니라는 것이다.

‘wating_lock이 있는 thread’ == ‘donate를 해주는 thread’

- nest의 마지막thread인 경우 READY 상태일 수 있다. 이 thread의 priority를 마지막으로 바꾸고 iteration이 끝난다.

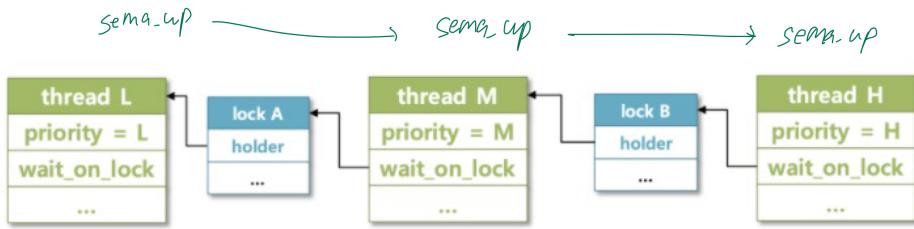
- 그 후에 sema_down을 하는데 이때 current thread가 block된다.

- ready_list에 있는 thread들 중 가장 우선순위가 높은 것은 가장 마지막에 donation을 받은 thread이고 얘를 위한 정렬을 해야한다. 마지막에 donation을 받은 thread가 ready_list에 없다는 것이 보장되면 정렬이 필요없지만 보장을 할 수 없기 때문에 정렬한다.

따라서 ready_list를 재정렬해야 한다.(하지만 테스트케이스는 이 기능 없이도 통과한다.)

donation과정에서 임의의 thread의 priority가 바뀌었을 수 있기 때문에 lock을 release하면 waiters를 ‘정렬하고’ 가장 priority가 높은 thread에 신호를 줘야한다. → 위에서 sema_up함수에서 thread_unblock을 하기 전에 정렬하는 로직을 추가했다.

- 추가내용



→ lock_unrelease 함수가 호출되면 priority를 donation한 반대 방향으로 신호를 쭉 전달하게 된다.
sema_up 함수에서 정렬을 하고 thread_unblock을 하니까 thread_unblock에서 '정렬되어있는 것이 보장된 ready_list'에 list_insert_ordered로 값을 넣는 것이다.

다시 lock_acquire를 호출하는 시점으로 돌아와서 생각해보면 thread B가 lock_acqurie 함수에 진입하였을 때, thread B보다 우선순위가 높은 thread가 존재할 수 있을까? 아니다. priority scheduling이라는 정책 아래에서 동작하고 있기 때문에 그럴 수 없다. 이는 lock을 가지고 있는 thread는 thread B보다 우선순위가 낮을 수 밖에 없다는 의미로 확장하여 해석된다.

lock_acuire 함수를 변경하여 lock을 acquire하는 과정이 바뀌었으므로, lock을 release하는 과정에도 변화가 필요하고 따라서 lock_release 함수 또한 변경해야한다.

그럼 어떻게 바꿀 것인가?

우선은

하나의 thread가 여러 lock을 가질 수 있기 때문에 어떤 lock을 release한다고 하더라도 아무런 검사 없이 original_priority로 돌아가서는 안된다.(multiple donation 상황이 생길 수 있으니까.)

곧바로 original_priority로 priority가 설정되는 경우는 lock_release 함수를 호출하는 thread(thread D라고하자)의 donated_from_these_threads가 비어있는 경우 뿐이다.

donated_from_these_threads에 원소가 있다면(priority가 큰 것이 먼저 오도록 정렬된 상태임. 왜냐하면 lock_acquire에서 정렬된 상태로 원소를 넣기 때문), 그 원소는 thread D에게 priority를 donate해준 thread(thread E라고 하자)이기 때문에 해당 원소가 lock을 기다리고 있는 상황을 해결해주기 위해 thread D의 우선순위를 thread E의 우선순위로 하여야 한다. 강조하자면, thread E는 donated_from_these_threads에 있는 원소들 중 priority가 가장 큰 thread이다.

한 가지 더 고려할 사항은, donation 관련 정보를 처리하는 것이다.

- 해당 기능 구현

1. 'lock_release 함수를 호출하는 thread'에게 priority를 donation 해 주는 thread가 기다리고 있는 lock이

이 함수에서 release하는 lock이라면, 지우기

- 여러 thread들로부터 donate를 받을 수 있으므로 리스트의 모든 원소를 확인한다.

2. donated_from_these_threads에서 가장 우선순위가 큰 priority반기

- lock_acquire에서 정렬된 상태로 원소를 넣어뒀기 때문에 가장 앞에 있는 원소의 priority를 사용하면 된다.

priority donation 기능을 구현하는 초반에 thread_set_priority 함수에서 original_priority에 관한 고려를 해주었는데, 위 2번 기능 구현을 하며 donation되는 경우도 다룰 수 있도록 2번 기능을 함수(set_right_priority)로 만들어서 thread_set_priority에서 동일한 로직을 사용하도록 구현을 변경함.

thread_get_priority 함수는 현재 구현을 따른다면 자동적으로 priority donation 상황에서 높은 priority를 리턴한다.

3. Advanced Scheduler

3.1 Solution

우리는 가장 처음에는 MLFQS 계산에서 사용할 floating point 연산을 int 형식으로 바꿔서 진행하는 용도의 함수들을 만들었다. 이는 제공된 펫토스 스텐포드 문서에 있는 Fixed Point Real Arithmetic란을 보고 작성했으며, 이렇게 작성된 함수를 문서명과 동일한 fixed_point_real_arithmetic.h에 몰아놓았다.

```
int int_to_fixed(int x);
int fixed_to_int_round_down(int x);
int fixed_to_int_round(int x);
int fixed_add(int x, int y);
int fixed_subtract(int x, int y);
int fixed_int_add(int x, int y);
int fixed_int_subtract(int x, int y);
int fixed_multiply(int x, int y);
int fixed_int_multiply(int x, int y);
int fixed_divide(int x, int y);
int fixed_int_divide(int x, int y);
```

가장 기초가 될 연산함수들을 정의하고 난 다음에는 MLFQS를 구현하기 위해서 `struct thread`에 멤버 변수 `nice`, `recent_cpu`를 추가해주었다. 이 값은 thread마다 존재하는 값이므로 멤버변수로 추가하였다. 또한 전역변수로 `load_avg`, `ready_threads`를 추가해주었다. 이 값들은 MLFQS 스케줄링 과정에서 system wide하게 사용되는 값이기 때문에 전역변수로 선언하였다.

MLFQS를 위해 필요한 변수들이 다 갖춰진 다음에는 MLFQS의 핵심인 실시간으로 갱신되는 priority 알고리즘을 `thread_tick`마다 실행시켜주도록 설정해주었다.

이러한 기초 단계들이 전부 갖춰진 다음 실제로 변수값들을 조정하는 함수들을 만들었다.

가장 먼저 만든 함수는 thread의 priority를 계산하는 함수다.

$\text{priority} = \text{PRI_MAX} - (\text{recent_cpu}/4) - (\text{nice} * 2)$ 라는 수식을 사용해서 priority를 계산해주었다.

```
// priority = (recent_cpu/4) + PRI_MAX - (2*nice)
int calculate_priority(struct thread *t, void *aux){
    int recent_cpu = t->recent_cpu;
    int nice = t->nice;
    if (t == idle_thread)
        return;
    t->priority = fixed_to_int_round_down(fixed_int_addition(fixed_int_division(recent_cpu, -4), PRI_MAX - nice * 2));
```

그 다음에는 현재 `recent_cpu`를 계산해주는 함수와

$\text{recent_cpu} = (2 * \text{load_avg}) / (2 * \text{load_avg} + 1) * \text{recent_cpu} + \text{nice}$

```
// recent_cpu = (2*load_avg) / (2*load_avg+1) * recent_cpu + nice
int calculate_recent_cpu(struct thread *t, void *aux){
    int recent_cpu = t->recent_cpu;
    int nice = t->nice;
    if (t == idle_thread)
        return;
    t->recent_cpu = fixed_int_addition(fixed_multiplication(fixed_division(load_avg, 2), fixed_int_addition(fixed_int_multiplication(load_avg, 2), recent_cpu), nice));
```

`load_avg`를 계산해주는 함수도 만들었다.

$\text{load_avg} = (59/60) * \text{load_avg} + (1/60) * \text{ready_threads}$

```
// load_avg = (59/60) * load_avg + (1/60) * ready_threads
void calculate_load_avg(){
    int num_of_ready_threads;
    if (thread_current() == idle_thread)
        num_of_ready_threads = list_size(&ready_list);
    else
        num_of_ready_threads = list_size(&ready_list)+1;
    load_avg = fixed_addition(fixed_multiplication(fixed_division(int_to_fixed(59), int_to_fixed(60)), load_avg), fixed_int_multiplication(fixed_division(int_to_fixed(1), int_to_fixed(60)), num_of_ready_threads));
```

여기서 calculate_priority와 calculate_recent_cpu같은 경우에는 형태가 좀 특이한 것을 확인 할 수 있다. 사용이 되지 않는 변수 aux가 있기 때문인데, 이는 모든 threads들에게 특정한 함수를 다 실행시켜주는 역할을 맡고 있는 함수 foreach_threads를 사용하기 위해서 갖춰야할 형태이기에 이렇게 구성했다.

현재 running thread의 recent_cpu에 1을 더하는 함수도 만들어주었다. 여기서 recent_cpu는 fixed point고 1은 int니 이 덧셈을 수행할 수 있는 앞서 구현해 놓았던 함수를 사용했다.

```
// recent_cpu = recent_cpu + 1
void add_one_recent_cpu(){
    struct thread * current_thread = thread_current();
    if (current_thread == idle_thread)
        return;
    current_thread->recent_cpu = fixed_int_addition(current_thread->recent_cpu, 1);
}
```

여기서 또 다른 언급할 만한 우리의 구현 특징은 바로 idle thread에 대한 처리이다. idle thread의 경우에는 recent_cpu를 올리면 priority가 올라가기에 recent_cpu를 올릴 이유가 없다. Idle thread는 실행할게 없을때 실행되어야하기 때문에 그 어떤 thread들보다도 우선순위를 가지면 안된다고 생각했다. 그래서 idle thread는 애초에 실행이 되지 않는다고 priority가 올라가는 일이 없도록 예외처리를 해줬다.

마지막으로는 foreach_threads 함수를 사용하여 모든 threads들에 대해 calculate_priority와 calculate_recent_cpu를 실행시켜주는 함수를 구현하였다.

```
// updates all of the threads' recent_cpu value using calculate_recent_cpu
void calculate_all_threads_recent_cpu(){
    struct thread* all_list_thread;
    struct list_elem *iter;
    thread_foreach(calculate_recent_cpu, NULL);

}

// updates all of the threads' priority value using calculate_priority
void calculate_all_threads_priority(){
    struct list_elem *iter;
    struct thread* all_list_thread;
    thread_foreach(calculate_priority, NULL);

}
```

이렇게 함수들 구현이 완료된 다음에는 timer.c에 구현되어있는 timer interrupt handler에 의해 적절하게 호출이 되도록 설정해주었다. timer interrupt마다 현재 실행 중이던 threads의 recent cpu 값에 1을 더해주도록 했다. 그 다음 4 tick마다 모든 threads의 priority들이 갱신되도록 했다. 마지막으로 TIMER_FREQ ticks마다 load_avg와 모든 threads들의 recent_cpu 값을 갱신해주도록 구현했다. 여기서 현재 thread의 recent_cpu와 시스템의 load_avg 계산이 모든 threads들에 대한 recent_cpu 계산을 선행하고 그 다음에야 모든 threads들에 대한 priority 계산을 하는 이유는 priority 값 계산에 쓰이기 때문이다.

```

/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
    if (thread_mlfqs){

        add_one_recent_cpu();

        if (ticks%TIMER_FREQ == 0){
            calculate_load_avg();
            calculate_all_threads_recent_cpu();
        }

        if (ticks%4 == 0){
            calculate_all_threads_priority();
        }
    }
    thread_wakeup (ticks);
}

```

우리의 MLFQS 구현에서 마지막으로 짚고 넘어가야 할 점은 MLFQS일때 ready_list 큐 64개를 안쓰고 1개만 쓴 것이다.

우리는 큐를 64개를 만들어서 priority가 바뀔때마다 그 값에 맞는 큐로 갈 수 있도록 기존 큐에서 빼주고 새로운 큐에서 넣어주고 하는 구현보다는, 그냥 priority값 64개를 기준으로 가장 큰값부터 시작하도록 정렬하는 구현이 더 관리하기 쉽고 공간적으로 효율적이라고 생각했다. 어떻게 구현을 했든 kernel behavior가 MLFQS의 프로토콜과 동일하면 MLFQS를 성공적으로 구현한 것이기 때문에 문제가 되지 않을 것으로 생각했다.

우리의 구현은 현재 MLFQS의 조건인 in case of highest priority, use round robin scheduling이라는 조건과 가장 높은 priority인 thread들부터 실행해야한다는 조건을 list_sort를 써서 결국 하는 scheduling이 큐 64개를 쓰는 것과 동일할 수 밖에 없도록 구현하였다. priority값이 바뀌면 바로 ready_list를 priority 값을 기준으로 sort 하기 때문에 무조건 가장 priority가 높은 thread로부터 실행되는 것이 보장된다. sorting 함수는 merge sort를 사용하기 때문에 같은 priority끼리는 기준 순서가 유지된 상태로 정렬되기 때문에 명시적인 함수 사용 없이도 round robin scheduling을 수행하도록 구현했다고 할 수 있다.