+--------------------+

| CS 330 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

TEAM NAME : 31

Donghyun Ahn <segaukwa@kaist.ac.kr> 기여도 50%

Donghoon Ham <hdh8277@kaist.ac.kr> 기여도 50%

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, usage of tokens, or extra credit, please give them here.

This is TEAM 31, Project 1, 0 token use for this project

>> Please cite any offline or online sources you consulted while

>> preparing your submission, other than the Pintos documentation, course

>> text, lecture notes, and course staff.

interrupt disable에 관해 http://teraphonia.tistory.com/359를 읽고 이해했습니다.

ALARM CLOCK

==========

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

<src/thread/thread.h>

on struct thread :

int64_t allow_wakeuptick; /* For sleep : Tick that allows wakeup */

struct list_elem sleep_elem; /* list elem for sleeping threads */

<src/thread/thread.c>

/* sleep_list that contains sleep_elem*/

static struct list sleep_list;

/* detect next awake thread's tick and record it as global variable*/

static int64_t next_awakethread_tick = 0xFFFFFFFFFFFFFFFF;

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),

>> including the effects of the timer interrupt handler.

timer_sleep()을 call하면 깨어날 절대 시간을 input으로 하여 thread_sleep(int64_t)함수를 call하게 된다. int64_t 자리에는 sleep_due라는 깨어날 절대 시간을 넣어준다. 사실상 timer_sleep() 역할을 thread.c의 thread_sleep이 전부 한다. thread에 현재 스레드가 다시 깨어날 시간을 스레드 스트럭트 내의 int64_t 타입 allow_wakeuptick에 저장한다. 또한, A3에서 설명할 효율적인 timer interrupt handling을 위한 보조 global variable인 next_awakethread_tick의 값을 업데이트한다. 현재의 next_awakethread_tick이 sleep_due보다 큰 경우에 업데이트가 실행된다. 그 후 sleep_list에 현재 스레드를 추가한다. 이 과정에서 sleep_elem이 사용된다. 마지막으로, thread를 block한다. 나중에 다시 unblock할 경우 timer_interrupt부분의 thread_awake()에서 sleep_list의 모든 원소를 보며 현재 tick이 sleep_list에 들어간 thread->allow_waketick 보다 큰 경우 unblock시킬수 있게끔 설계하였다.

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

매 틱마다 timer interrupt handling이 일어나게 하는 대신, next_awakethread_tick을 이용하여 다음 틱은 언제 wakeup이 되는지를 미리 파악하여 그 사이 간격에서는 굳이 thread_awake()를 발동시키지 않았다. 어차피 깨어날 스레드가 없는 것을 알기 때문이다.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer_sleep() simultaneously?

timer_sleep() 을 할 때, thread_sleep() 함수에서 interrupt 를 disable 한 뒤 list operation 이 진행되기 때문에, multiple threads call 로 인한 race condition 을 피할 수 있습니다.

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer_sleep()?

A4 와 마찬가지로 interrupt 를 disable 시켰기 때문에 race condition 은 발생하지 않습니다.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to

>> another design you considered?

만약 한동안 깨울 thread 들이 존재하지 않음에도 timer_interrupt 에서 매 틱 마다 thread_awake 를 실행하여 모든 thread를 체크하는 것은 너무 비효율적이라고 생각했습니다. 그래서 저희는 가장 먼저 일어나야 할 thread의 tick 을 저장하여 그 전 tick 에서는 thread_awake 가 일어나지 않도록 했습니다. thread_sleep 과 thread_awake 를 하면서 next_awakethread_tick 을 update 하게 됩니다.

PRIORITY SCHEDULING

==========

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

```
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. */
    int64_t allow_wakeuptick;               /* For sleep : Tick that allows wakeup */
    int org_priority;                       /* Original Priority before donation */
    struct list_elem sleep_elem;            /* list elem for sleep */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;                  /* List element. */
    bool donated;                           /* flag for donation */
    struct list locks;                      /* list of locks */
    struct lock *blocked;                   /* the lock that blocks current thread*/
```

int priority : thread 의 priority

int org_priority : donation 때 원래 priority 저장

bool donated : donate 상태를 나타냄

struct list locks : thread 가 현재 가지고 있는 locks

struct lock *blocked : 현재 thread 가 acquire 한 lock(thread 가 block 되지 않았다면 NULL)

src/thread/synch.c

```
struct lock
  {
    struct thread *holder;          /* Thread holding lock*/
    struct semaphore semaphore; /* Binary semaphore controlling access. */
    struct list_elem holder_elem; /* list elem for locks list*/
```

```
    int lock_priority;
  };
```

struct thread *holder : lock 을 가지고 있는 thread

struct list_elem holder_elem : thread 가 가지고 있는 lock 들을 list 에 넣기 위한 list_elem

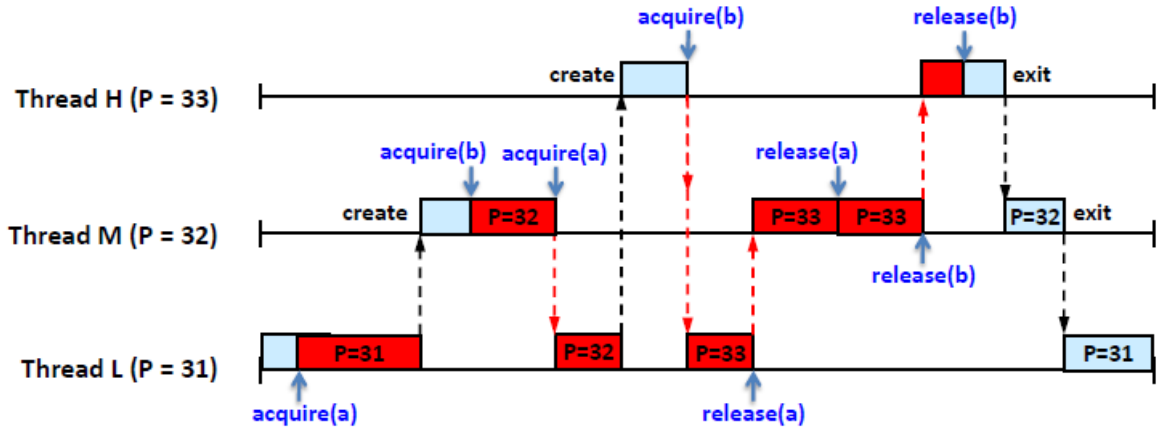
int lock_priority : 여러 개의 lock 을 가지고 있을 때, lock 들의 우선순위를 비교하기 위한 lock_priority

```
struct semaphore_elem
  {
    struct list_elem elem;              /* List element. */
    struct semaphore semaphore;         /* This semaphore. */
    int priority;                       /* Priority for semaphore */
  };
```

int priority : 기존에는 semaphore_elem에 thread의 priority를 따질 수 있는 요소가 없었다.

>> B2: Explain the data structure used to track priority donation.

>> Use ASCII art to diagram a nested donation. (Alternately, submit apng file.)



<그림 1>

Priority donation 이 발생하는 경우는 높은 priority 의 thread 가 lock 을 요청했을 때, 낮은 priority 의 thread 가 이미 그 lock 을 점유 중이라 제대로 된 우선순위 스케쥴링이 불가능할 경우 낮은 priority 의 thread 에 priority donation 이 일어난다.

Single priority donation 의 경우, lock_acquire 함수가 실행될 때 만일 lock 의 holder 가 존재한다면 holder->priority 와 thread_current()->priority 를 비교해 holder->priority 가 더 작을 때 priority donation 을 해준다.

Nested donation 의 경우 조금 더 복잡한데, <그림 1>을 예시로 설명하자면, 우선 L thread 가 가장 먼저 lock a 를 acquire 해 가지고 있는 상태이고, 그 뒤로 M thread 가 lock b를 acquire 한 뒤 lock a 또한 acquire 한다. 하지만 lock a 는 L 에 의해 점유된 상태 이므로 L 에 donation 이 발생해 priority 가 32 로 바뀐다. 이 때 가장 priority 가 높은 Thread H가 lock b 를 요청하면, 현재 b->holder 인 Thread M 이 lock a 에 의해 block 되어 있는 상태이므로, priority donation 이 발생해 M 과 L 의 priority 가 33으로 바뀐다. 이후 L 에서 a 가 release 되면 block 상태이던 M thread 가 unblock 되고 M 은 a 와 b 를 release 하게 된다. b 가 M으로부터 release 된 후에야 비로소 H가 lock b 를 가지게 되고 H로 thread 가 yield 된다. b 를 release 한 후 H가 EXIT 하면 M, L 이 차례로 EXIT 하게 된다.

더 긴 nested priority donation 의 경우도 recursive 한 implementation 을 통해 문제 없이 구현할 수 있었다.

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

lock, semaphore, condition variable에 대해 cmp 함수를 만들어 list에 넣을때 정렬해서 들어가도록 세팅을 해 놓았다. 예를 들어 lock의 경우 lock_acquire, lock_release에서 priority가 바뀌는 상황이 있을 때마다 정렬 삽입 (list_insert_ordered)을 사용하였고 원소를 빼낼 때에도 sort를 한 후 뽑아내었다. 그렇게 하면 highest priority를 가진 lock을 안전하게 얻어낼 수 있다. Semaphore, condition variable 역시 마찬가지다.

>> B4: Describe the sequence of events when a call to lock_acquire()

>> causes a priority donation. How is nested donation handled?

Chain 의 길이에 상관없이 nested priority donation 을 구현하기 위해, recursive 한 함수를 만들었다. Chain 의 일부분을 보게 되면, current thread, current thread 가 blocked 된 lock, lock 을 점유하고 있는 thread. 이렇게 세 부분으로 이루어져 있고, current thread 의 priority 보다 lock 을 점유중인 thread 의 priority 가 작다면, nested priority donation 이 일어나야 한다.

(current_thread, current_thread->blocked->holder) 의 관계를 recursive 하게 while 문 안에서 옮겨주면서, 최초에 lock 을 acquire 한 가장 priority 가 큰 thread 의 priority 를 donate 받게 만들었다. While 문은 current thread priority 보다 lock holder 의 priority 가 더 크거나 lock 의 holder 가 없을 경우 끝나게 된다.

그 뒤 sema_down 을 통해 임계 지점에 들어가도록 한다.

>> B5: Describe the sequence of events when lock_release() is called

>> on a lock that a higher-priority thread is waiting for.

Lock_release() 가 실행되면 우선 sema_up 을 통해 yield 가 일어나 lock 의 소유권이 다음 thread 로 넘어가게 되고, current thread 의 locks list 에서 대상 lock 을 remove 시키게 된다.

이 다음으로 중요한 포인트는 lock 이 release 된 thread 의 priority 를 재설정 해주는 일인데, 만약 locks list 가 empty 라면 더 이상 priority donor 가 존재하지 않으므로 미리 기록해 둔 org_priority 로 priority 를 바꿔주고, empty 가 아니라면 locks 들 중 가장 큰 lock_priority 를 가진 lock 의 priority 로 priority 를 바꿔주게 된다.

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread_set_priority() and explain

>> how your implementation avoids it. Can you use a lock to avoid

>> this race?

thread_set_priority() 때문에 발생할수 있는 race condition의 예를 들어보자면, priority donation이 일어나 thread의 priority가 바뀌는 중간에 thread가 스스로 priority를 바꾸고 싶어 하는 경우가 있을 것이다. 이때 이 스레드가 만일 다른 스레드에 priority를 donate하게 된다면, 두 일의 실행

순서에 따라 다른 스레드에 donate하게되는 priority가 바뀌게 될 것이다. 즉 race condition이다.

우리는 interrupt를 disable함으로서 위의 현상이 미연에 일어나는 것을 방지하였다. 우리의 코드에서는 lock을 사용하지 않았고 위의 상황을 해결하기 위해서만 해도 donor thread와 현재 스레드간의 race condition 발생을 막는 lock이 필요하다. 그런데 위의 상황 이외에도 race condition에 돌입할 상황이 많은 것으로 짐작되며, 그 경우를 모두 cover하면서도 deadlock에 걸리지 않을 로직을 짜야 한다는 것을 고려했을 때, 락을 통해 이 문제를 해결하기보다는 단순히 interrupt를 disable하는 것이 나을 것이라 생각된다. 따라서 본 코드에서는 interrupt를 disable하는 것만으로 race dondition 돌입을 방지하였다.

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to

>> another design you considered?

struct thread :

thread가 복수개의 lock을 소유할 수 있기에 홀딩하는 lock을 기록할 필요가 있었고(list locks), donated되었는지 안 되었는지에 따라 set_priority 방식을 달리해야 하였다. 그래서 해당 thread가 donation이 일어난 thread인지, 그렇지 않은지를 확인해 주는 bool 변수 donated를 만들었다. 또한 donation 과정에서 lock_acqire요청이 들어온 lock이 다른 스레드에 선점되고 그 스레드 역시 다른 락을 대기하고 있는 상태인 nested 상태에서의 priority donation을 위해서는 어떤 lock이 현재 thread를 가로막고 있는지를 기록해야 한다. 이를 위해 blocked를 사용하였다. 또한 priority donation 과정이 끝나고 원래 priority 로 회복을 위해 org_priority 역시 저장하여 사용하였다.

struct lock :

thread에서 locks라는 lock을 기록하는 list를 필요로 함에 따라, struct lock에 list_elem holder_elem을 추가하였다. 또한 한 thread가 여러 lock을 소유할때 어떤 lock의 release가 해제된 경우 priority 재조정을 위해 lock_priority라는 변수를 저장하였다. 이를 cmp_lock_priority라는 함수로서 크기 비교가 가능하게 하였다.

SURVEY QUESTIONS

==========

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?