+--------------------+
| CS 330 |
| PROJECT 1: THREADS |
| DESIGN DOCUMENT |
+--------------------+

---- GROUP ----

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

FirstName LastName <email@domain.example>
신정윤 <jugipalace@kaist.ac.kr> 50%
김건우 <kakaloto85@kaist.ac.kr> 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.

>> 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.

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.

1. `int64_t wakeup_ticks;` 우리는 alarm clock을 위해 struct thread에서 int64_t wakeup_ticks라는 인자를 추가했다. wakeup_ticks는 thread가 wake up해야할 순간의 ticks 값을 저장해놓은 인자다.
2. `static struct list sleep_list;` 또한, sleep_list라는 list를 만들어 timer_sleep에 의해 wait 하고 있어야 할 thread들의 list을 만들었다. 이는 이후, thread_check_sleep_list에서 thread들을 확인할 때 쓰인다.
3. 현재 sleep_list의 wakeup_ticks중 가장 작은 값을 저장하였다. 이후 timer_interrupt의 효율성을 위한 변수이다.
4.
```
static int64_t min_sleep_ticks;
```

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),
>> including the effects of the timer interrupt handler.

1. timer_sleep(int64_t ticks) 는 ticks 를 받아서 ticks만큼 thread가 wait하게 하는 것인데, busy-waiting을 해결하기 위해 우리는 while 문에서 yield하는 것이 아니라, thread 에 깨어나야할 시간을 wakeup_ticks 로 저장해놓고 thread_sleep 함수를 실행하여, 현재 tick 값과 비교하여 깨울 thread를 찾을 것이다. 이때 min_sleep_ticks에 현재 wakeup_ticks중 최소값을 저장해두었다.

```
void
timer_sleep (int64_t ticks)
{
 int64_t start = timer_ticks ();
 int64_t wakeup_ticks=start+ticks;
 ASSERT (intr_get_level () == INTR_ON);
 enum intr_level old_level;

 old_level = intr_disable();
 thread_sleep(wakeup_ticks);
 if(wakeup_ticks<min_sleep_ticks){
   min_sleep_ticks=wakeup_ticks;
 }

 intr_set_level(old_level);

 /*gw: 이게 아니라 sleep을 위한 함수가 필요할듯*/
 // while (timer_elapsed (start) < ticks)
 //   thread_yield ();
}
```

2. thread_sleep (int64_t wakeuptick) 에서는 wakeuptick을 받아서, current thread의 일어날 시간을 wakeuptick으로 저장해놓고, 별도로 만들어 준 sleep_list에 wait할 thread들을 넣고, block해주는 함수이다. 이때 list_insert_ordered 와 wakeup_ticks_compare를 이용하여 wakeup_ticks 기준 오름차순으로 정렬했다.

```
void
thread_sleep (int64_t wakeuptick)
{
 struct thread *cur = thread_current ();
 cur->wakeup_ticks = wakeuptick;
 list_insert_ordered(&sleep_list, &cur->elem,wakeup_ticks_compare,0);
 thread_block();
}
```

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
```

```
{
 ticks++;
 thread_tick ();
 //현재 tick이 sleep_list의 최소 wakeup_ticks보다 적은 경우 thread_wakeup()을 실행
 if(ticks>=min_sleep_ticks)
   thread_wakeup();
}
```

timer interrupt handler의 시간을 단축하기 위해, 우리는 thread_wakeup() 함수를 만들었다. 이는, timer_ticks()를 실행된 시간에 받아, sleep_list에서 timer_ticks() 보다 작은 wakeup_ticks 값을 갖는 모든 thread를 깨우는 과정으로, 기존의 yield를 통해 status를 바꾸고 ready_list에 넣고 빼는 과정을 단순화한 방식이다. 또한 현재 ticks가 min_sleep_ticks보다 크거나 같은 경우에만 thread_wakeup()을 실행하도록 하여, 함수의 불필요한 실행을 줄였다. 아래는, 보다 자세한 함수 설명이다.

3. thread_wakeup(void) 에서는 현재 timer_tick() 값을 wakeuptick로 저장하고, thread_check_sleep_list에서는 sleep_list에서의 thread들의 wakup_ticks 값과 비교하여 깨울 tick들을 찾아낸다.

```
void
thread_wakeup (void)
{
  int64_t wakeuptick = timer_ticks();
  thread_check_sleep_list(wakeuptick);
}
```

4. thread_check_sleep_list(int64_t wakeuptick) 에서 우리는, 어떤 int64_t 값을 받으면, (그 값은 thread_wakeup에 의해 현재의 timer_ticks() 값이다.) 이 값을 sleep_list에 있는 각 thread의 wakeup_ticks 값과 비교하여 thread들을 sleep_list에서 지우고, unblock해주는 과정을 거친다. 이때 sleep_list는 wakeup_ticks순으로 정렬되어 있기 때문에, 인자로 받은 wakeuptick 값보다 sleep_list의 원소의 wakeup_ticks이 더 크면 함수가 종료된다.

```
void
thread_check_sleep_list(int64_t wakeuptick)
{
 struct list_elem * cnt = list_begin(&sleep_list);
 // int64_t min = &cnt -> wakeup_ticks;

 while (cnt != list_tail(&sleep_list)&&list_entry(cnt, struct thread,
elem)->wakeup_ticks<=wakeuptick)
   {
     struct thread *t = list_entry(cnt, struct thread, elem);
      cnt = list_remove(cnt);
      thread_unblock(t);
   }
```

```
}
```

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
기존의 busy waiting 방식의 경우, 반복적으로 yield와 schedule을 하면서 여러 개의 thread 가 ready_list에 들어갔다 나오면서, 들어가는 순서에 따라 결과에 영향을 미칠 수 있었다. 하지만 우리는 thread를 block된 상태로 유지하고, sleep_list 에 sort해서 wakeup_ticks값이 작은 순서로 넣어주고 깨워줄 때에도 sleep_list상 앞에서부터 깨워주기에 race condition을 막을 수 있다.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
기존의 busy waiting 방식의 경우, timer_sleep에서 interrupt를 허용하는 상태였지만, 고친 코드에서는
timer_sleep()에서 thread_sleep()을 실행하기 전에 interrupt를 잠시 disable해주고 thread_sleep()이 끝난 뒤 interrupt를 허용해 주었기 때문에, timer_sleep()을 call하던 도중에 timer interrupt가 일어나도 문제가 생기지 않는다.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?
우리는 timer_sleep을 할 때 thread_block만 하여 thread를 sleep하게 하면, 깨울 때 일일히 thread의 status를 확인해야하기에 비효율적이라고 느꼈다. 또한, block이 timer_sleep에서만 쓰이는 것이 아니기에, timer를 위해 sleep하는 thread들을 모아주는 list를 만들면, 모든 thread의 status를 확인하지 않고, sleep_list만 확인해주면 되어 효율적일 것이라고 생각했다. 또한, 처음에는 이 list에서 함수가 실행될 때마다, tick값을 list에 있는 모든 thread와 비교하려했다. 하지만, 이 방법보다는 sleep_list를 깨울 tick 값에 맞춰서 sort해주어서 sleep_list의 모든 element를 확인하지 않아도 되게 만들었고, 최소 tick값과 현재 tick 값을 비교하여 현재 tick이 최소값(min_sleep_ticks)보다 더 작을 경우는, 함수를 실행시키지 않아도 되게끔하여, 작동 시간을 더욱 줄였다.

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 thread에서 우리는 다음과 같은 member를 추가했다. (thread.h)

```
int past_priority;                  /* jy past_priority for donation */
```

```
struct lock* lock_of_holder;
struct list lock_list;
```

1. int past_priority 는 scheduling 과정에서 thread의 priority가 바뀔 수 있으니, 생성 당시의 초기값을 저장한다.
2. struct lock* lock_of_holder는 해당 thread를 기다리게끔 하는 acquired_lock이 있으면, 이 lock을 나타내준다. 없으면, NULL이다. 만일 thread L이 lock A를 소유하고 있고 thread H가 락 A를 요청할 때 thread H의 lock_of_holder에 lock A의 포인터를 넣어준다.
3. struct list lock_list는 해당 thread가 holder로 있는 lock들의 list이다.

struct lock에서 우리는, 다음과 같은 member를 추가했다. (synch.h)

```
int lock_max;
struct list_elem lock_elem;
```

1. int lock_max 는 각 lock에 acquire한 thread 들의 priority 중 최대 priority 값을 해당 lock의 lock_max에 저장한다.
2. struct list_elem lock_elem은 lock들을 list에 넣어줄 수 있게 하기 위해 만들었다. thread들의 lock_list에 넣을 때 각 lock의 lock_elem 들을 넣어주게 된다.

>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation. (Alternately, submit a
>> .png file.)
우리는, lock_of_holder를 사용하여 cur thread가 acquire하는(현재 다른 thread가 소유하고 있는) lock을 알 수 있도록 만들었다. lock_acquire를 할 때 priority_donate라는 함수를 이용하여, 재귀적으로 lock_of_holder를 확인하여 해당 lock_of_holder의 holder의 priority를 확인해서 nested donation일 때 priority donation를 해결했다.

thread L(31) thread M(32) thread H(33)
Lock A(소유) ← Lock A(요청)
lock_of_holder에 lock A의포인터 추가
Lock B(소유) ← Lock B(요청)
lock_of_holder에 lock B의포인터 추가

위와 같은 예시에서, thread M이 Lock A를 요청할때 thread M의 lock_of_holder에 lock A의 포인터를 추가하고, priority donate를 실행하여 thread L의 priority를 32로 바꿔준다. 그 다음 thread H가 lock B를 요청할 때 thread H의 lock_of_holder에 lock B의 포인터를 추가하고, priority donate를 실행하여 thread M의 priority를 33으로 바꿔준다. 이 때 thread M에도 lock_of_holder가 존재하므로(lock A의 포인터) 재귀적으로 priority donate를 실행하여 thread L의 priority를 33으로 바꿔준다.
---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?

```
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)
   {
     /*새로 넣은 코드 */
     list_insert_ordered(&sema->waiters, &thread_current()-> elem,
list_elem_compare,0);
     thread_block ();
   }

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

```
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, list_elem_compare,0);
   thread_unblock (list_entry (list_pop_front (&sema->waiters),
                               struct thread, elem));

 }
 sema->value++;
   thread_yield();

 intr_set_level (old_level);

}
```

모든 structure에서 sema_down과 sema_up을 사용하여 waiters에 넣는 방식을 취하기에, sema_down과 sema_up에 대해서 설명을 하겠다.
sema_down을 할 때, 우리는 list_elem_compare라는 함수를 만들어, thread의 내림차순 정렬을 위한 함수를 만들었다. 이 함수를 이용하여, list_insert_ordered로 priority가 높은 순서로 thread들을 waiters 안에 넣어줬다. 따라서, sema_up에서 pop_front를 할때, 가장 priority가 높은 순서대로 나가게 되었다. 또한, unblock 하기 전에 sort를 한 번 더 하여,waiters에 있는 thread 중 몇몇이 다른 thread의 priority donation에 의해 priority가 바뀌었을 경우를 대비하여 재정렬해준다.

lock 의 경우에는 lock_up과 lock_down으로 별도로 함수를 만들어주었다. 이는, lock_acquire와 lock_release 시, sema_up과 sema_down과 동일한 형태로 작동할 수 있도록 만든 함수들이다. 파라미터로 sema가 아닌, lock을 받게끔하기 위해 만들었고, lock_up, down 시 lock_max 값을 갱신해주는 기능이 추가적으로 있다.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?

```
void
lock_acquire (struct lock *lock)
{
  ...
 struct thread* tholder = lock->holder;
 int current_priority=thread_get_priority();
 struct thread* current_thread =thread_current();

 if(tholder!=NULL){
   enum intr_level old_level;
   old_level = intr_disable();

   current_thread->lock_of_holder = lock;
   priority_donate(current_thread);
   intr_set_level(old_level);
}
...
}
```

```
void priority_donate(struct thread* cur){
 struct lock* acquired_lock=cur->lock_of_holder;
 if(acquired_lock !=NULL){
   struct thread* holder=acquired_lock ->holder;
   if(holder !=NULL){
     if(holder->priority<cur->priority){
       holder->priority = cur->priority;
       priority_donate(holder);
     }
   }
 }
}
```

lock acquire를 실행하면

1. 해당 하는 Lock의 홀더를 찾는다
2. Lock의 홀더가 NULL이면 바로 donate할 필요가 없으므로 if 문을 나가고 아니라면 if 문 안의 함수들을 실행한다
3. 해당 Lock의 홀더가 NULL이 아니면 현재 thread(current_thread)의 lock_of_holder에 lock의 포인터를 넣고 priority_donate 함수에 인자로 current_thread를 넣어준다.

4. priority_donate함수에서는 인자로 들어온 cur(현쟈는 current_thread)의 lock_of_holder(acquired_lock)이 NULL인지 확인한다.
5. NULL이 아닐경우 해당 acquired_lock의 홀더가 NULL인지 확인한다
6. NULL이 아닐경우 holder의 priority보다 cur의 priority가 더 크면 holder의 priority를 cur의 priority로 갱신해주고 holder에 대해 priority_donate를 실행시켜준다.
7. 만일 nested condition의 경우 holder도 lock_of_holder를 가질 것이므로 B2에서처럼 priority_donate가 재귀적으로 실행되어 nested priority donation이 완성된다.

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.

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

 ASSERT (lock != NULL);
 ASSERT (lock_held_by_current_thread (lock));
 struct thread* tholder = lock->holder;
 old_level = intr_disable();
 lock->holder = NULL;

 list_remove(&(lock->lock_elem));
 if(!list_empty(&tholder->lock_list)){
      if(lock_list_max (&tholder->lock_list)>=(tholder->past_priority))
       tholder->priority=lock_list_max (&tholder->lock_list);
   }
 else{
         tholder->priority=tholder->past_priority;
 }
 intr_set_level(old_level);

 lock_up (lock);

}
```

```
void
lock_up (struct lock* lock)
{
 enum intr_level old_level;
 struct semaphore* sema=&lock->semaphore;
 ASSERT (sema != NULL);

 old_level = intr_disable ();
 if (!list_empty (&sema->waiters)) {
   list_sort (&sema->waiters, list_elem_compare,0);
```

```
   thread_unblock (list_entry (list_pop_front (&sema->waiters),
                               struct thread, elem));
        lock->lock_max = lock_max(lock);
 }
 sema->value++;
 if(check_readylist_prioirty()>thread_get_priority()){
   thread_yield();
 }
 intr_set_level (old_level);
}
```

```
int check_readylist_prioirty(void){
 return list_entry(list_begin(&ready_list),struct thread,elem)->priority;
}
```

1. release를 하기 전, current thread(lock의 홀더)의 lock_list 에는 Lock_release의 인자로 들어온 lock이 들어가 있기 때문에, 먼저 해당lock의 lock_elem을 lock_list에서 제거해준다.
2. 만약 lock_list가 비어있다면, multiple donation이 아니기 때문에 해당 thread(tholder)의 priority를 past_priority(priority 초기값)으로 바꿔준다.
3. lock_list가 비어있다면, multiple donation을 위해, lock_list에 있는 각 lock의 lock_max값중 가장 큰 값이 tholder의 past_priority보다 크다면, tholder의 priority를 해당 값으로 갱신해준다.
4. 그 뒤 lock_up 함수를 실행하여 해당 lock의 semaphore의 waiter리스트중 가장 priority가 큰 것을 unblock해준다(해당 thread를 ready_list에추가). 이때 waiter 리스트에 있는 thread들의 priority의 변동이 있을 수도 있으니 sort해준다(내림차순)
5. sema 의 값을 올려주고, 만일 readylist의 priority중 가장 큰 값이(맨 앞)이 현재 priority보다 크다면(ex)방금 ready_list에 추가한 thread가 현재 thread보다 priority가 높은 상황) thread_yield를 호출하여 schedule을 진행한다.
6. 그 뒤 스케쥴링에 따라 차례로 lock_up함수와 lock_release함수를 종료한다.

---- 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?

```
void
thread_set_priority (int new_priority)
{
 thread_current ()->priority = new_priority;
 thread_current ()->past_priority = new_priority;
 if(lock_list_max (&thread_current ()->lock_list)>new_priority)
   thread_current()->priority=lock_list_max (&thread_current ()->lock_list);
 if(!list_empty(&ready_list)&&thread_get_priority()<list_entry(list_begin
(&ready_list), struct thread, elem)->priority){
   thread_yield();
```

```
 }
}
```

발생할 수 있는 race에는, 우리가 현재 thread의 priority를 thread_set_priority() 를 이용해 일부러 낮출 때, 발생할 수 있다. 이때 우리의 implementation은 if문에서 ready_list에서 낮춰진 current의 priority보다 높은 priority의 thread가 있을 때, thread_yield()를 통해, 알맞은 thread가 schedule 될 수 있도록 하였다.

```
 if(!list_empty(&ready_list)&&thread_get_priority()<list_entry(list_begin
(&ready_list), struct thread, elem)->priority){
   thread_yield();
 }
```

하지만 만약 현재 thread가 lock을 가지고 있고, 그 lock을 요청하는 더 높은 priority의 thread가 waiting list에 있을 때에는 dead lock문제가 발생할 수 있기 때문에

```
 if(lock_list_max (&thread_current ()->lock_list)>new_priority)
   thread_current()->priority=lock_list_max (&thread_current ()->lock_list);
```

로 priority를 회복시켜 주었다.

---- RATIONALE ----

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

우리는 우선, waiters에 thread들을 넣을 때 내림차순으로 넣어서 순서를 정리하는 것이 우선이라 생각했고, 이를 통해 쉽게 waiters에서 high priority를 뺄 수 있게 하는 것이 중요하다고 생각했다. 그 후, nested donation에서 lock A의 holder가 lock B의 waiters 에 있고, 이런 상황이 반복될 경우를 생각했다. 이런 상황을 쉽게 파악하기 위해, 새로운 struct member lock_of_holder를 만들어서, thread 간 관계성을 파악하고, priority donation을 current에서 실행할 때, current thread와 nested donation으로 관련된 모든 thread의 priority를 바꿔주기 위해, 재귀함수의 형태로 구현했다.

multiple donation의 경우, 초기의 계획으로는 thread에 lock_acquire_list를 만들어서, holder가 있는 lock이 acquire될 때마다, holder의 lock_acquire_list에 해당 lock을 acquire하는 thread를 저장하였다. 하지만 이 경우, 같은 lock을 acquire하는 thread가 생길때마다 lock_acquire_list에 추가되어서, holder가 lock_release를 실행할 때 lock_acquire_list의 모든 thread를 확인해야 하는 문제가 생겼다.
따라서 기존의 계획에서, thread에 lock_list를 만들어서 holder가 lock을 acquire 할 때 해당 lock만을 저장하고 lock 구조체 안의 semaphore의 waiters list를 이용하여 해당 lock을 요청하는 thread의 priority의 max 값을 저장하도록 하였다. 이 방식을 통해 매 release마다, 현재 thread가 소유하고있는 lock을 요청하는 모든 thread를 확인하는 것이 아니라, max값만을 비교하도록 하여 효율성을 높였다.

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?