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

---- GROUP ----

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

[Team 17]
Paul Kim <paulkim3151@kaist.ac.kr>
Sangwoo Kim <woodmac@kaist.ac.kr>

---- 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.

thread.h

```
struct thread {
        ...
       int64_t time_wakeup; //Time to wake up For Sleeping Threads
       ...
}
```

thread.c

```
static struct list sleep_list;  // List of All threads Blocked by Time_sleep
static int64_t time_wakeup_min; // Minimum wakeup time in sleep_list
```

---- ALGORITHMS ----

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

timer interrupt handler always do next thing
1. call thread_wakeup_call(now), now is the current time of timer
2. in wakeup_call method, if there is any element in the list whose time_wakeup was reached by now, then call thread_wakeup for that thread.
2-a. thred_wakeup does essentially same thing with thread_unblock, except that this method removes thread from sleep_list. thread’s status is now ready, and it is in ready list.
3. time ticks (default job)

Things happen after calling timer_sleep()
1. time_wakeup of current thread is set.
2. thread_sleep() in thread.c is called
3. in thread_sleep(), status of current changed from THREAD_RUNNING to THREAD_BLOCKED
4. then current thread are push into sleep_list instead of ready list
5. And schedule() is called, so context switch occurs

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

Most wasted time in timer interrupt handler is time consumed for searching sleep_list, even if there is no element which need to be wake up for a while. So we reduced this time by blocking search if it is not time to search list. We set the nearest future time to look into the list, and if timer not passed enough searching is ignored.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?

Before calling thread_sleep() in timer_sleep(), we call intr_disable() which prevent interrupt. This avoid race conditions.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?

Same as previous answer. Disabling interrupt avoid race conditions.

---- RATIONALE ----

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

To avoid busy wait, the sleeping thread must not be pushed into ready_list. (otherwise, it will be scheduled) So we make a sleep_list in order to wake appropriate time to wake up.
If we make another data structure (such as thread etc.) to manage other sleeping thread, its overhead is too big. So we choose timer_interrup to wake sleeping threads.

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.

thread.h

```
struct thread {
         ...
        int priority_original;              /* Original Priority */
        int priorities[LIST_SIZE];          /* List of related thread's priorities */
        struct lock *lock;                  /* lock which thread is held */
        ...
}
```

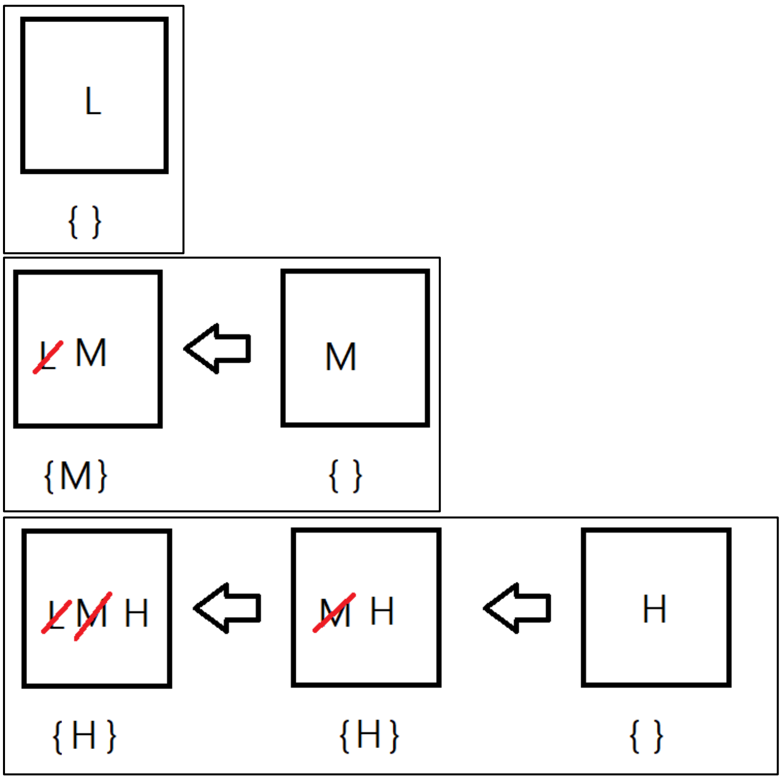
synch.h

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

>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation. (Alternately, submit a
>> .png file.)

Main data structure for priority donation is array called priorities. This array is initially empty, but when if there is any thread which has been blocked by a lock, holder of the lock add blocked thread's priority into its priorities list. Then, holder's priority is maximum among original priority and priorities list.

Following images briefly explains this data structure with simple nested donation case.



---- ALGORITHMS ----

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

1. In semaphore, we pop the maximum priority thread in sema_waiters when sema_up() is called.
2. In lock, sema_up() is wake up method, which is same as semaphore case. So we can ensure.
3. condition variable has own priority for its semaphore. Sorting according to priority make the thread who has highest priority awake first.

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

1. First insert current thread's priority into priorities list of lock's helder.
2. Then Broadcast the change of priority through the nested chain of locks.
3. set 'lock' variable of current thread into 'lock' which is argument of lock_acquire
4. sema_down, which is sleeping until lock_release is called
5. after lock_release is called, lock is disappeared so set 'lock' variable of current thread into NULL

Nested Donation is handled while updating priorities list and current priority of holder thread. If any change in priority happens, it recursively call every nested locked threads to change their priority. Priority is maximum priority among priorities list and thread's own original priority.

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

1. Remove priorities of threads belong to waiting list of lock one by one. Update nested donated priorities and reset threads lock variable into NULL one by one.
2. Then sema_up for semaphore of lock.

While calling update function, donated high priority of current thread, which is caused by locked thread is excluded, so current priority become lower.

---- 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?

There could be race condition if timer interrupt is called during priority donating process (thread_priority_update() fuction). For example, thread A -- which is suspended by lock L -- is resetting its priority. During this process, the timer interrupt is called and it changes running thread to holder of L. This could be race condition.
we avoid this problem by disabling interrupt during thread_priority_update() call.
In fact, disabling interrupt can be substituted to using a lock. We think this will affect the same.

---- RATIONALE ----

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

To construct multiple donation, a thread must remember priorities of threads which is locked into lock which was acquired by the thread. So we need to construct the whole list of potential priorities. There might be another design which rely on heritance relation of threads. But the way we implemented is better because by our method, we can access the state of whole system, so we can implement various functions which can intervene into system. And this method is also easy to debug.

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?
We do not want it to become harder. It’s burdenful enough.

>> 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 NOT trust code that you received. It may have logical errors. Try to understand how they really work.

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

CONTRIBUTION

All problem was solved while we are in face to face with each other. Both can propose his idea, while writing code was usually done by Paul, who is better at typewriting.

In more details, idea for timer_sleep was mainly proposed by both Sangwoo and Paul. While most of lock and priority donation algorithm was designed by Sangwoo, main modification on semaphore and condition variables was done by Paul.