Design document (Project 1: Threads)

Team 3

- ChaYoung You chacha@kaist.ac.kr (contribution1)
- HyunJin Jung jhjnav7@kaist.ac.kr (contribution2)

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.

- `devices/timer.c` :
  - `static struct list thread_list` : List of threads currently sleeping.

Algorithms

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

When `timer_sleep()` is called, it cycles below instructions until `ticks` time elapsed:

1. Disable interrupts
2. Insert current thread to the `thread_list`
3. Block current thread
4. Restore interrupts level

The timer interrupt handler will unblock every thread in `thread_list` then this thread will check `timer_elapsed(start) < ticks` again.

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

It doesn't have anything complex like sorting or check the condition things. It just pop every

elements in `thread_list` and then call `thread_unblock()`.

Synchronization

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

We disable interrupts before insertion to `thread_list` and blocking a thread. Then we reset interrupt level to previous one, so we can prevent race conditions. Threads are administrated by `thread_list`. And we put current thread to `thread_list` and block it. Then we can wait until `thread_unblock()` call.

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

We disable interrupts before insertion to `thread_list` and blocking a thread. Then we reset interrupt level to previous one, so we can prevent race conditions.

Rationale

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

`thread_list` acts like a queue, because `timer_interrupt()` wakes threads up first in first out. Actually it really doesn't matter if we use `thread_list` like a stack, since we unblock all threads at the moment.

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 lock`
  - `struct list_elem elem` : Make `struct lock` can be stored in `struct list`.
- `struct thread`
  - `struct list locks` : List of locks the thread holds on.
  - `struct lock *acquiring_lock` : When a thread want to acquire a lock but it is already

acquired by other thread, then the thread make `acquiring_lock` to point to the lock. Then priority donation is started.
    - `int donation_level` : This is for the nested priority donation. Every `donation_level` starts from 0, increasing by 1. If this goes to 0, then the priority of thread should be equal to `base_priority` .
    - `int base_priority` : When a thread receive a priority donation, it stores original priority to `base_priority` . After returning every prioriy donation, its priority resets to `base_priority` .

> B2: Explain the data structure used to track priority donation. Draw a diagram in a case of nested donation.

When we start priority donation, we first set `acquiring_lock` . We recursively donate priority to other threads by increasing `donation_level` by 1 and check `acquiring_lock` of lock holder thread. Donated thread should set `base_priority` as its original priority. After we acquire a lock, we add it to `locks` to track priority donation. At lock release, traverse `locks` to determine next priority to set.

Algorithms

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

We manage waiters of semaphore by `list_insert_ordered()` by comparing priority of threads. But priority can change while it is blocked, so we sort waiters one more time in `sema_up` . Lock is handled by semaphore, so we can ensure that the highest priority thread wakes up first. For the condition variable, we insert waiter by comparing each waiter by its highest priority of semaphore.

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

If the holder of the lock is not `NULL` , we call `lock_donate()` and this compares priority to determine whether we have to donate. We set `base_priority` of holder if it's needed. Then set its priority to it of current thread, and increase `donation_level` by 1. Then check if there exists `acquiring_lock` of holder. If exists, then we have to call `lock_donate()` with `holder->acquiring_lock` . This handles nested donation.

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

When `lock_release()` is called, `lock_donate_return()` is called before `sema_up()`. We check `holder->locks` is empty. If not, we see the first element of it, and set priority as same as priority of the first thread of its semaphore waiters. In `lock_acquire()`, we insert lock in inserted order by comparing the first thread of semaphore, so we can ensure that the first element of `holder->locks` has thread with highest priority.

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?

If `thread_set_priority()` is called, it can mess up the process of priority donation. So we check `curr->donation_level`. If it's positive number, we only change `curr->base_priority` so donated priority is preserved.

Rationale

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

For waiters of semaphore, we usually use `list_insert_ordered()`, so we can just `list_pop_front()` to get the thread with highest priority. This takes O(n) time to insert, and reduce the amount of sorting time like in `lock_acquire()`, `lock_donate_return()` and `cond_signal()`. May we don't need to call `list_sort()` in `sema_up()` every time, but we couldn't find better solution for this time.

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 two problems in it, too easy or too hard? Did it take too long or too little time?

It was hard to understand overall concepts about this project.

> 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?

If there is more simple introduction documents about whole project, then it will be easy to start project.

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

If TAs give more tips to students (from there experience or commonly known facts, etc.), It will be more effective when we do those projects.

> Any other comments?