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.

A list store all the sleeping threads according to ascending order of the wake up time.

/\* sleeping\_queue \*/

struct list sleep\_queue

Added to struct thread:

/\* member for implementing Alarm Clock \*/

struct semaphore t\_sema; /\* semaphore for tracking sleeping thread \*/

struct list\_elem t\_elem; /\* list element for waiting queue \*/

int64\_t wakeup\_time; /\* Record for the thread sleeping time \*/

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer interrupt handler.

timer\_sleep() find the current running thread and put it into a sleep\_queue. And then it set up a semaphore for the thread and change the status to block. The timer interrupt handler can interrupt in any above process except sema\_down().

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

>> the timer interrupt handler?

The sleeping queue is designed in a way that a thread has less wakeup\_time will be in the front of the queue. So instead of finding out the earliest threads to be wakeup among all the sleeping thread, the interrupt will only need to check the front of the sleep\_queue.

---- SYNCHRONIZATION ----

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

>> timer\_sleep() simultaneously?

Because we disable interrupt before we insert, delete the thread in sleep\_queue, it make sure it won't have multiple threads process in sleep\_queue at the same time.

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

>> during a call to timer\_sleep()?

Since timer\_sleep() will disable interrupt before we inserting the thread into the sleep\_queue, timer interrupt won't happen during insertion and it will avoid race conditions.

---- RATIONALE ----

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

>> another design you considered?

The design has the advantage of simplicity and time-saving. Encapsulating most of the synchronization logic into a method to increase the code readability.

The other design I considered was using lock instead. But because lock can only be released by itself. Whenever a timer interrupt happens, we need to block the current running thread and switch to the sleep thread to release its lock. The thread switching operation will take longer to process and waste unnecessary time.