```
+--------------------------------------+
|                CS 140                |
|   PROJECT 1: THREADS                 |
|   DESIGN DOCUMENT                    |
+--------------------------------------+
```

---- GROUP 10----

Robin Babu Padamadan <robinb2009@gmail.com> Roll No.:17CS10045
Shivam Kumar Jha <shivam.cs.iit.kgp@gmail.com> Roll No.: 17CS30033

---- PRELIMINARIES ----

In the test given in the build directory (make test command runs it), the simulation was not powering off properly (and hence giving timeout) because in the latest versions of qemu the shutdown port has been changed. Thus we had to add a new line in src/devices/shutdown.c.
“ outw (0xB004, 0x2000); “

Reference: https://stackoverflow.com/a/45276093

ALARM CLOCK
===========

---- DATA STRUCTURES ----

A) In /devices/timer.c
    a) static struct list sleeping_threads;
       List of processes in THREAD_BLOCKED state, that is, processes that are required to be sleeping and waiting to be unblocked once their wake up time arrives.

B) In /threads/thread.h:
    a) uint64_t sleep_time :
       This variable stores current time + sleep time (essentially wake up time) when the thread needs to be sent to the ready queue.

    b) struct list_elem sleep_elem;
       List element for all sleeping threads list. This enables easy and relative access of list items (threads).

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),
>> including the effects of the timer interrupt handler.
Initially we have added a safety condition to return without execution if a timer_sleep() is called with a negative time value or 0. We have verified if the interrupt is on or not and then disabled the interrupt handler before entering the section we wish to behave as atomic.
Then we find out the current thread using the ‘thread_current()’ function and update the time when the thread has to wake up in the variable ‘sleep_time’. Then we add the thread to the ordered list ‘sleeping_threads’ using ‘list_insert_ordered()’ function which usese ‘lesser_sleep()’ function for comparative purpose. Then we block the thread which is the equivalent of putting it into the waiting state and finally, we enable interrupts again. Thus the timer interrupt handler cannot affect the timer_sleep() call.

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?
We have arranged the threads in the list of sleeping threads in increasing order of the time when they wake up (stored in the variable sleep_time in threads.h). The iteration that occurs to go through the lists is in the same ascending order of the wake-up times. Thus if the thread at the start of the list’s wake up time hasn’t arrived, then neither will have any of the other thread’s wake-up times, thus breaking out of the loop and thus the function in constant time. However, if the thread at the start of the list’s wake up time has arrived then we set it’s sleep_time to 0 and come out of the function unless other threads have their wake-up time set at the exact same time the occurrence of which is very low. Thus it can be expected that the amount of time spent in the interrupt handler is close constant time, however, in a worst-case scenario, it can be linear.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
In our implementation of the timer_sleep() function, we have disabled all interrupts after verifying whether the interrupts are on or not. Because of this all statements that occur after the interrupts are disabled and until the interrupt is enabled (which happens at the end of the function), no context switching can occur between threads, as no interrupt can be called during the execution of this function. Thus essentially that block of code acts as an atomic section and ALWAYS executes without any interruption in between that could lead to race conditions between multiple threads.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
Similarly as above since all interrupts are disabled for the main portion of the executing code in the function timer_sleep(), it behaves as an atomic element in the code, thus cannot be

interrupted by any means. Thus neither can timer interrupts occur during the access of the thread lists within this portion of the code, thus timer interrupts cannot cause race conditions during a call to timer_sleep(). Context switching can occur inside the function timer_sleep(), but not in the critical section bounded by the disabling and enabling of interrupts.

---- RATIONALE ----

>> A6: Why did you choose this design?  In what ways is it superior to
>> another design you considered?
We chose this design as it is optimal in choosing threads that need to be woken up at any given instant of time. This is ensured by sorting the wake-up times (stored as sleep_time in threads.h) in ascending order. Thus we only need to wake up the threads for which the wake-up time has occurred, which will be the threads with the lowest wake-up time, which will occur at the start of the thread, thus not needing to iterate through the entirety of the sleeping list every time we need to find out which threads to wake up.
Another design we considered is implementing a normal non-sorted list which would require us to sacrifice time complexity during the timer_interrupt call to wake up threads. However, the insertion during the sleeping of thread will be faster. We felt that it was more important for the waking up of threads to occur on time so CPU time can be given to thread execution faster, whereas sacrificing time while sleeping shouldn’t be as much of a problem. Thus our design was superior.