```
+--------------------+
|      CS 390002     |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- GROUP 11----

Sagnik Roy sagnikr38@gmail.com 18CS10063
Debajyoti Kar debajyoti.apeejay@gmail.com 18CS10011

---- PRELIMINARIES ----

A new line
outw (0xB004, 0x2000) has been added in the file pintos/src/devices/shutdown.c in order to fix the timeout problem upon running qemu simulator.

Source:
https://stackoverflow.com/questions/39805784/timeout-in-tests-when-running-pintos/45276093#45276093

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

---- DATA STRUCTURES ----

1. **static struct list sleeping_threads** in devices/timer.c:
   List of threads whose wake up time is not yet over. They are stored in increasing order of their wake up times, are in BLOCKED state and will be unblocked once their wake up times expire.
2. **uint64_t sleep_time** in threads/thread.h:
   Stores the wake up time of threads, when the threads in BLOCKED state have to be unblocked and sent to the ready queue.
3. **struct list_elem sleep_elem** in threads/thread.h:
   To ensure easy access of threads.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.
In the reimplementation, whenever the timer_sleep() is executed, we first verify that interrupts are enabled and then disable them with intr_disable(). So, no interrupts can be called during its execution and the given block of code always runs. At the end, interrupts are again enabled using intr_enable().

>> A3: What steps are taken to minimize the amount of time spent in the timer interrupt handler?
The BLOCKED threads are stored in the sleeping_threads list, which is maintained as a priority queue (linked list implementation). The threads are arranged in increasing order of their wake up times and new threads are inserted maintaining this order with the help of the lesser_sleep() comparator function. So, if one of the threads in the list haven’t reached their wake time yet, all the threads after it would also not have woken up and can be ignored in the current iteration. When the wake time has arrived the thread’s wake up time is made 0, it is unblocked by thread_unblock() and removed from the sleeping_threads list. In the average case all this will be done in constant time, unless the threads have the same wake up time, in which case the execution time may be linear, but its occurrence would be very rare.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call timer_sleep() simultaneously?
Race conditions can occur only when 2 or more threads access shared data and try to change it at the same time. The shared data out here is the sleeping_threads list. However in our implementation of timer_sleep() all interrupts are disabled after an initial sanity check. So, no interrupt can occur during the execution of the function after they have been disabled. Since it executes without any interrupt, the possibility of race conditions dont arise.

>> A5: How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?
As mentioned earlier, the interrupts are disabled after after an initial sanity check. So the critical code fragment where race conditions could have occured are secure from external interrupts and no context switching can occur between threads till interrupts are again enabled by intr_enable() inside timer_sleep(). Note that context switching can occur inside timer_sleep() before the interrupts are disabled and after the interrupts are enabled but cannot occur during the execution of the critical code where the thread lists are accessed. So race conditions are successfully avoided.

---- RATIONALE ----

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

This algorithm is easy to implement. Race conditions are easily avoided by turning off interrupts at the beginning of timer_sleep() and then turning them back on at the end of timer_sleep(). Also the timer_interrupt() can also be easily implemented. Moreover, this design chooses to wake up the threads in an optimal fashion, that is only those threads are woken up whose wait time is over. Also, since the list is already sorted with respect to wake up times, choosing which thread to wake up is a process that can be completed efficiently in constant time.

Another variation could be in the type of data structures we use. Instead of a priority queue, we may have used a simple linked list to store the sleeping threads. However, here, insertion would take constant time at the cost of linear time during call to timer_interrupt(). However here it is the opposite. We concluded that incurring more time during insertion (when the threads are sleeping) is more sensible than during timer_interrupt(as then the threads can be transferred to the CPU faster for execution). So we feel our method would be more efficient.