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

---- **GROUP 9** ----

Aditya Singh <aditya390402@gmail.com>
Itish Agarwal <itishagarwal2000@gmail.com>

---- **PRELIMINARIES** ----

>> References
https://web.stanford.edu/class/cs140/projects/pintos/pintos.html
https://practice.geeksforgeeks.org/problems/busy-waiting
https://www.ccs.neu.edu/home/skotthe/classes/cs5600/fall/2015/notes/pintos-project1.pdf
http://www.pvpsiddhartha.ac.in/dep_it/lecture%20notes/OS/unit3.pdf

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.

We added 1 new struct member in thread.h and 1 global variable in timer.c .

In thread.h,

int64_t time_wkup;

This new property stores the tick at which the thread should be woken up, if it is sleeping. Basically, it is the wakeup time for a particular thread.

In timer.c,

static struct list sleep_list;

A list of sleeping/blocked threads that contain the processes that are currently not ready and have to wait. These processes are added in timer_sleep() and removed in timer_interrupt() in which they are unblocked and forwarded to the ready list.

---- **ALGORITHMS** ----

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

In timer_sleep(), we add the number of ticks to the global ticks and pass it as an argument to the timer_sleep_implement() function (our code addition). It begins with checking if the number of ticks are less than or equal to 0. If it is, we exit the function. If not, then the thread is inserted into a sleeping list/queue which is sorted in ascending order, meaning that the first thread of the list will be woken up first. After insertion, the thread is then blocked. To ensure that the ticks may be reliably calculated and the thread can be blocked, interrupts are disabled for this process.

In timer_interrupt() (the interrupt handler), we increment the global number of ticks and call our timer_wkup() function (our code addition). The timer_wkup() function traverses through the sleep_list and checks if the threads sleep time is less than the global ticks. If so, it removes the threads from the list and unblocks them. If not, it breaks the loop since the sleep_list is in ascending order, meaning that the next threads are not ready to be woken up as well.

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

We have minimized the amount of time spent in the timer interrupt handler by keeping the sleeping list sorted. If the current thread of the sleep_list does not need to be woken up, the loop breaks since the list is sorted. Otherwise, we move on to the next thread. This prevents the handler from iterating the entire sleep_list at each interrupt, making it more efficient.

---- **SYNCHRONIZATION** ----

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

Race conditions are avoided when multiple threads are called by disabling the interrupt (which are disabled in timer sleep). Basically, interrupts are turned off when we add a new thread to the sleep list (i.e. when we block a thread), not allowing another thread to be put to sleep at the same time. This results in only one thread being in the function at a time so that no other thread can take the current thread off the CPU.

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

When a call to thread_block() is made, interrupts are turned off, so that there is no way for a timer interrupt to occur until the end of the function when the interrupts are re-enabled.

---- **RATIONALE** ----

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

This approach was clearly more efficient than busy waiting as in busy waiting CPU cycles are wasted when actually they can be utilized by some other process. Now here rather than engaging in busy waiting the

process can block itself. The block operation places the process into a waiting queue(or sleep list) and the state of the process (or thread is changed to WAITING.
Next, because we are storing the blocked threads in a sorted list, we only need to traverse over threads with WAITING state and not all the threads. So we are iterating through the sleep list and checking if a particular thread should be put into the ready_list. One major advantage is that because the list is sorted, we can break the moment we find a thread which needs to wait longer than present. Clearly, threads residing after this in the list will have even greater waiting times. Hence sorting the list makes it efficient, and we can use in-built heaps for maintaining the sorted property.