+------------------------------ +
| CS 39002 |
| Assignment3: THREADS |
| DESIGN DOCUMENT |
+------------------------------ +

GROUP - 37

Hardik Aggarwal <hardik8464@gmail.com>
Sriyash Poddar <poddarsriyash@gmail.com>

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

---- DATA STRUCTURES ----

-> 1. **Added variable** int64_t sleep_time to the structure thread
File changed : $HOME/pintos/src/thread/thread.h
Purpose: It stores the time at which we need to wake the process up.
Calculated as sleep_time = startime + time to wake up

2. **Added the function** static bool custom_comp(struct list_elem *a, struct list_elem *b)
File changed : $HOME/pintos/src/devices/timer.c
Purpose: It is a custom comparator that compares two threads based on the time at which they will wake up.

3. **Added variable** blocked_threads of type struct list
File changed : $HOME/pintos/src/devices/timer.c
Purpose : It contains the list of all threads that are in a blocked state.

---- ALGORITHMS ----

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

added list_init(&blocked_threads) to the timer init function

timer_sleep():

- *Compute start time*
- *Ensure that interrupts are on*
- *Disable interrupts*
- *Get the current thread and set its thr-> sleep+time = start + ticks*
- *insert this into the blocked_threads list sorted in increasing order according to sleep_time attribute*

- *block this thread*
- *enable the interrupts again*

timer_interrupt(): (interrupt handler)

- *Increase the total ticks and call thread_tick() to tick the timer of threads*
- *Get the front element of the list and compare its sleep time with current time*
- *If the thread sleep_time is greater then break*
- *remove and unblock the thread*

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

The threads are kept in the blocked_threads list sorted according to their sleep_time (time to wake up again). We are checking if the wake_time of the first thread is greater than the current time. If it is then we just break out. This is ensured that no other thread needs to be checked due to the order of sorting. We are doing this check in a while loop because multiple threads can have the same wake up time. Hence this function checks the thread to wake up in **O(1) time** which can blow up to linear time complexity, probability of which is quite low.

---- SYNCHRONIZATION ----

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

We have disabled the interrupts just as we enter the function timer_sleep using **intr_disable()**. This disables the interrupt for any other thread while we are processing the current thread. This makes sure that there is no switching between the threads and hence prevents racing. We are ensuring that the interrupts have been enabled by other threads by the assert condition at the beginning.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
Same as above, when we disable the interrupts in the call to the function timer_sleep() which disables the interrupts. Then we process the threads before enabling the interrupts once again. Thus no interrupt can cause racing when we are inside this function.

---- RATIONALE ----

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

We chose this design mainly due to the following two reasons:

- **Very fast access to the front element** on every timer interrupt.
- **Handling of racing conditions** while context switches

The other design we considered was to insert it into a Binary search tree sorted according to their sleep times. This will decrease the insert time which can be linear in our case but will also increase the cost to search for a process with minimum time to wake up.

Our design is superior from this design because we are making use of the fact that interrupts handle much more frequently than the insertion of new processes. Hence it was important to have a quick access time than improving the insert time and compromising on the look up part.