```
+-------------------------------------------------------------+
|        CS39002                                               |
|        ASSIGNMENT 3: PintOS and Implement Alarm Clock        |
|        DESIGN DOCUMENT                                       |
+-------------------------------------------------------------+
```

---- GROUP 22 ----

(18CS30047) Somnath Jena <somnathjena2011@gmail.com>
(18CS10069) Siba Smarak Panigrahi <sibasmarak.p@gmail.com>

---- PRELIMINARIES ----

1. **Terminating qemu simulator**:
   After successful execution of pintos run alarm-multiple (or any similar run command) the qemu simulator did not terminate. We added an additional -q flag (**pintos -- -q run alarm-multiple**). It led to the termination of the qemu simulator after successful execution. Further, while executing the make commands, the inbuilt make file adds the above -q flag.
2. Additionally we had to change the **shutdown.c**, according to Timeout in tests when running pintos. This led to the prevention of the TIMEOUT after a certain (generally 61) seconds) issue on qemu simulator.

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.

1. **In pintos/src/threads/thread.h**

```
struct thread
 {
    /* Owned by thread.c. */
    tid_t tid;                    /* Thread identifier. */
    enum thread_status status;    /* Thread state. */
    char name[16];                /* Name (for debugging purposes). */
    uint8_t *stack;               /* Saved stack pointer. */
    int priority;                 /* Priority. */
    struct list_elem allelem;     /* List element for all threads list. */
```

```
    int64_t wake_tick;            /* tick when a sleeping thread will wake up */
    struct list_elem sleepelem;           /* list element for all sleeping threads list */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;                  /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;            /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;               /* Detects stack overflow. */
 };
```

MODIFICATIONS:

i. **struct list_elem sleepelem**: list element embedded into the struct thread which is part of the sleeping_threads list that maintains a list of all threads which are sleeping, i.e. in THREAD_BLOCKED state and waiting upto wake_tick ticks.

ii. **int64_t wake_tick**: this variable stores the tick at which the thread will wake if it is sleeping, i.e. sum of start time of sleep and ticks to sleep.

2. **In pintos/src/devices/timer.c**

   static struct list sleeping_threads;

   MODIFICATIONS:

   i. **static struct list sleeping_threads**: static variable of type struct list which consists of struct list_elem members embedded inside struct thread. It maintains the list of all threads currently sleeping and in THREAD_BLOCKED state.

---- ALGORITHMS ----

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

A sanity check is initially added in **timer_sleep()** to return without execution if input ticks parameter value is negative or 0. The next step is to verify if the interrupt is on or disabled. Interrupt handler is then disabled before adding the current thread in **struct list sleeping_threads**. This prevents any race conditions related to interrupt handling. We obtain the current thread using the **thread_current()** method and update the time when the thread has to wake up in the variable wake_tick. We add the thread to the ordered list **sleeping_threads** with the aid of **list_insert_ordered()** which uses **list_less_func** type function **list_less_sleep()** for comparison. We block the current thread, i.e. change the state from THREAD_READY to THREAD_BLOCKED and enable the interrupts. Thus the timer_sleep() call is not affected by any other interrupts.

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

Everytime the state of a thread changes from THREAD_READY to THREAD_BLOCKED, it is added to the list of waiting threads (**struct list sleeping_threads**). The list is an ordered list (i.e. all the threads in the list are sorted in increasing order by wake_tick value. This is a boost to the interrupt handler. Now, it has to iterate only on the list and need not sort the sleeping_threads list every time it is called. It ensures that if one of the threads is not ready to wake-up, then any other threads after this thread can not wake up also. This leads to amortized constant access of the list as seen from the viewpoint of the interrupt handler. Further, when **ticks** (time since OS boot time) becomes greater than **wait_tick** of a thread, it is woken up, The thread's wake_tick is set to 0, it is unblocked and removed from the **struct list sleeping_threads**.

---- SYNCHRONIZATION ----

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

The **timer_sleep()** function disables interrupt temporarily(after checking that interrupt is enabled initially) before adding the thread into the **struct list sleeping_threads** and enables after the thread is added to the **sleeping_threads**. This ensures that no other thread can preempt the current thread (since it cannot cause an interrupt) while the thread gets added to the list. This assures that there would not be any issue while adding the current list_elem to **sleeping_threads**.

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

Timer interrupts can never preempt in **timer_sleep()**. This is because **timer_sleep()** calls **intr_disable()** to obtain the **old_level** and then adds the current thread into the list sleeping_threads. After successful addition, the interrupt is again enabled with the help of **intr_set_level (old_level)**. This interrupt disabling avoids any such race conditions. Since no interrupt can occur when one thread is being added to **sleeping_threads**. No race condition can arise due to timer interrupt.

---- RATIONALE ----

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

We chose our design because of the following reasons:

1. The interrupt disable before addition of the current thread assures that there would not be any problems due to race conditions involved with **timer_interrupt()**
2. Another feature is to avoid decrementing the number of sleeping ticks (**wake_tick**) for every single sleeping thread, by simply storing the number of ticks it takes since OS boot time. This means a simple comparison of **wake_tick** and that of ticks(since OS boot time) is enough to determine whether the thread is to be woken up or not.
3. The **struct list sleeping_threads** is a sorted list. This leads to amortized constant time insertion and deletion of threads from the list in the **timer_interrupt()**. But in any case, if the state change from THREAD_READY to THREAD_BLOCKED occurs for a large number of threads, the insertion would be linear time.

Other approaches and why ours is superior:

1. **Addition of state THREAD_SLEEPING**: This is completely unnecessary as there is already a THREAD_BLOCKED state and our design also uses THREAD_BLOCKED state.
2. **Design additional function for wakeup event**: Again this is not necessary to keep certain memory spaces occupied for this function. Our function efficiently carries this in **timer_interrupt()** by unblocking and removing the particular thread from the **struct list sleeping_threads** and changing the **wait_tick** field of the thread to be 0.
3. **Maintaining the sleeping_threads as a non sorted list**: This seems lucrative because the insertion into the **struct list sleeping_threads** is faster but we have to sacrifice the time-complexity of the interrupt handler. But ultimately our design i.e. keeping a sorted list of sleeping threads is superior since the interrupt handler is called very frequently compared to **timer_sleep()**. Thus the interrupt_handler must be as optimized as possible.
4. **Decrementing the wake_ticks for every sleeping thread (i.e. wake_tick represent the remaining time to wake up rather than the ticks from OS boot time)**: This is a heavy computation process compared to our approach (especially when the number of sleeping threads is higher). Our approach simply stores the number of ticks it takes since OS boot time, and a single comparison with the **wait_tick** of the thread provides information regarding the event waking up a sleeping thread.