CS39002
Operating Systems Laboratory
Spring Semester 2020-2021

ASSIGNMENT 3: Improving Threads in PintOS
DESIGN DOCUMENT

GROUP 8		
Atharva Naik Roshan	18CS10067	atharvanaik2018@gmal.com
Radhika Patwari	18CS10062	rsrkpatwari1234@gmail.com

---- PRELIMINARIES ----

>> We have checked the following tests, for our modified code:

`alarm-multiple`	`make tests/threads/alarm-multiple.result`
`alarm-negative`	`make tests/threads/alarm-negative.result`
`alarm-simultaneous`	`make tests/threads/alarm-simultaneous.result`
`alarm-single`	`make tests/threads/alarm-single.result`
`alarm-zero`	`make tests/threads/alarm-zero.result`

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.

Answer:
a) Added a variable to the definition of thread in `$HOME/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. */

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

          /* Code added */
          int64_t abs_ticks;     /*amount of ticks that the thread has to sleep for*/
          /* Code ended */

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

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

->**abs_ticks** stores time from start of CPU, for which the thread should sleep, i.e. if **abs_ticks** = x, then, thread will remain in `THREAD_BLOCKED` state, at least till x ticks have elapsed from start of CPU. It moves to `THREAD_READY` state when more than x ticks have elapsed.

b) Added a list in `$HOME/pintos/src/devices/timer.c`

struct list sleeping_threads;

-> **sleeping_threads** is a doubly linked list that contains the sleeping threads

c) Added a comparator function in `$HOME/pintos/src/devices/timer.c`

```
bool compare_ticks(struct list_elem *a, struct list_elem *b)
{
  return list_entry(a, struct thread, elem)->abs_ticks < list_entry(b, struct thread,
elem)->abs_ticks;
}
```

-> **compare_ticks** is used to store the sleeping threads in the list in ascending order of their waking time

---- ALGORITHMS ----

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

Answer :
`timer_init()` initialises the sleeping_threads list
`timer_sleep()` performs the following steps :

- Calls the `thread_current()` to find the thread presently running on CPU
- Initialises start of timer as ticks (time elapsed since the 0S booted)
- Checks if interrupts are enabled
- Assigns the `abs_ticks` parameter of the currently running thread with the time till which it needs to sleep
- Disables the interrupt
- Store the currently running thread in `sleeping_threads` list using `compare_ticks` function
- Blocking this thread by calling `thread_block()`
- `thread_block()` sets the state of this thread as `THREAD_BLOCKED` and calls the scheduler to assign a next process to CPU and perform context switching
- Enables the interrupt

`timer_interrupt()` handles time interrupts

- When the `thread_ticks`(time elapsed since the thread is running on CPU) exceeds `TIME_SLICE`(time quantum set by Round Robin scheduler), an external interrupt is raised to trigger `thread_yield()` and perform context switching
- If an interrupt occurs, the currently running thread is set to `THREAD_READY` state and scheduler is called
- Looping over the `sleeping_threads` list and change the state from `THREAD_BLOCKED` to `THREAD_READY` for threads whose sleeping time is over

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

Answer :
`sleeping_threads` list maintains the threads in the ascending order of their waking time. If the waking time is reached, the thread is unblocked and pushed into the ready queue. On obtaining a process that violates the condition, the loop breaks.
Thereby maintaining the threads in terms of priority(here, waking up time) minimises the amount of time spent in `timer_interrupt()`.

---- SYNCHRONIZATION ----

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

Answer :
the `ASSERT` call in the code segment shown below waits until some other thread enables external interrupts.

```
ASSERT ( intr_get_level == INTR_ON )
```

This prevents race conditions due to multiple threads trying to access the main part of the, `timer_sleep()` function. (the part enclosed between the `intr_disable()` and `intr_enable()` function calls)

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

Answer :
All interrupts are disabled, with `intr_disable()` before entering the most critical part of the `timer_sleep()` function, to block calls to `timer_interrupt()`. After executing the critical section, interrupts are re-enabled, by calling `intr_enable()`.

```
intr_disable();
/*
Critical Section
*/
intr_enable();
```

---- RATIONALE ----

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

Answer :
Current design:
The design used currently is based on priority queue concept, where threads are stored in sorted order of their waking time. The thread with least waking time is of highest priority. We maintain an extra state of `THREAD_SLEEPING` to avoid busy wait. When a thread assigned to CPU wants to sleep, change its state to sleeping and schedule a new thread for CPU to ensure its utilization.When a sleeping thread has woken up,change its state back to `THREAD_READY`.
Also the complexity is less. O(log n) and O(1) for insertion and deletion of threads respectively in the `sleeping_thread l`ist.

Previous Design:
Idea:
- Storing the wait time interval in the sleeping thread struct as `wait_time`

Implementation:
- Adding a variable **int64_t wait_time;** to the definition of thread in `$HOME/pintos/src/threads/thread.h`

Drawbacks:
- Takes O(n) time complexity at every timer interrupt to loop over `sleeping_thread` and decrement wait_time for each thread by 1 [n is the number of threads in sleeping_thread list at any point in time]

Comparison:
- Current design stores the waking time and not waking interval
- No updates are required in the parameters of the sleeping threads
- Threads are stored in the double linked list in ascending order of waking time.We

loop over the threads,unblocking the threads till we reach a thread which is still sleeping. Thus at every timer interrupt, the worst case time complexity can be O(n) where n=# of threads in sleeping_thread when all threads in the list have woken up, however average case complexity is much less. Thus the complexity is much less than the previous design considered.