```
            +--------------------+
            |       CSE 521      |
            | PROJECT 1: THREADS |
            |   DESIGN DOCUMENT  |
            +--------------------+
```

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Ved Harish Valsangkar <vedharis@buffalo.edu>
Gursimran Singh <gursimr2@buffalo.edu>
Harshal Ganesh Jagtap <harshalg@buffalo.edu>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

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.

Adding a new variable to the thread struct to hold the timestamp at which
time the thread must be taken out of blocked_list.

```
struct thread
{
    ...
    int64_t sleep_wt;                   /* Wake time stamp for looping through
blocked list. */

    ...
};
```

A new list created to hold the blocked threads thread.c

```
static struct list blocked_list;
```

---- ALGORITHMS ----

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

By our current algorithm, thread_block() should be called which will put
the current process in the blocked queue. It shall set the variable
sleep_wt to the expected tick stamp when the process should be woken
up. The blocked_list will always be in a sorted state.

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

Since, the blocked_list is in sorted state we need check the whether the

```
    head element needs to be unblocked. If so, we unblock all process whose
    timer has elapsed. If not, no time is spent on further processing.

---- SYNCHRONIZATION ----

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

As per our current machanism, the process of putting a thread into the
    blocked_list will be completed before thread_block() calls schedule().
    Hence, there won't be any new process in the running state before
    thread is put into blocked_list. That means that no two threads will
    try to access blocked_list simultaneously. Hence, race conditions are
    avoided.

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


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

To get rid of the busy waiting loop, we need to put the running thread
    into blocked_list instead of ready_list. By blocking the running
    thread instead of putting it into ready_list, we are ensuring that
    the thread doesn't get scheduled before its waiting time completion
    by putting it into the blocked_list.

We have attempted a simplistic approach towards this problem and have
    tried to include the concept of buffering. As yet we have not been
    able to complete the buffering section which will be required to
    tackle all future synchonization problems.

              PRIORITY SCHEDULING
              ===================

---- DATA STRUCTURES ----

>> B1: 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.

Adding a new variable to the thread struct to hold the timestamp at which
    time the thread must be taken out of blocked_list.

    struct thread
    {
        ...
        int64_t sleep_wt;                   /* Wake time stamp for looping through
blocked list. */

        ...
    };

A new list created to hold the blocked threads thread.c

    static struct list blocked_list;

>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation.  (Alternately, submit a
>> .png file.)
```

```
---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?

The ready_list is already sorted in a descending order according to priority.
    Thus higher priority threads are closer to the head. Two threads with
    equal priority are treated on an FCFS basis. Thus ready_list priority
    requirement is satisfied. The blocked_list is sorted based on the time
    stamp at which the thread is expected to be woken up in an ascending
    order. Hence, all the necessary time stamps can be woken up if required
    at any given tick. The sorting also ensures no thread is woken up before
    their wait time is elapsed regardless of the priority.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation.  How is nested donation handled?

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread_set_priority() and explain
>> how your implementation avoids it.  Can you use a lock to avoid
>> this race?

---- RATIONALE ----

>> B7: Why did you choose this design?  In what ways is it superior to
>> another design you considered?

              ADVANCED SCHEDULER
              ==================

---- DATA STRUCTURES ----

>> C1: 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.

A hash map shall be used to store multiple queues, each consisting of threads
    with equal priorities. The total number of such queues would be 64
    corresponding to 64 priority levels.

    static struct hash priority_bucket;

The current_list shall store the current list of threads to be executed in a
    round robin manner. This list is updated in a loop and the first non-empty
    queue from the hash map with the highest priority is assigned to the
    currnet_list. All threads in this list are of equal priority.

    static struct list current_list;

---- ALGORITHMS ----

>> C2: Suppose threads A, B, and C have nice values 0, 1, and 2.  Each
>> has a recent_cpu value of 0.  Fill in the table below showing the
>> scheduling decision and the priority and recent_cpu values for each
>> thread after each given number of timer ticks:

timer  recent_cpu    priority   thread
```

```
ticks   A   B   C   A   B   C   to run
-----  --  --  --  --  --  --   ------
 0      0   0   0   63  61  59      A
 4      4   0   0   62  61  59      A
 8      8   0   0   61  61  59      A
12      12  0   0   60  61  59      B
16      12  4   0   60  60  59      B
20      12  8   0   60  59  59      A
24      16  8   0   59  59  59      A
28      20  8   0   58  59  59      B
32      20  12  0   58  58  59      C
36      20  12  4   58  58  58      C
```

>> C3: Did any ambiguities in the scheduler specification make values
>> in the table uncertain? If so, what rule did you use to resolve
>> them? Does this match the behavior of your scheduler?

Considerations:
    - If two processes including the running process end up having same priority,
      then the running process is given priority.
    - If two processes other than running process end up having same priority,
      then there is no specific information given to tackle such a clash. We propose
      using nice values as tie-breakers with lower nice values getting higher
priority.

>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?

---- RATIONALE ----

>> C5: Briefly critique your design, pointing out advantages and
>> disadvantages in your design choices. If you were to have extra
>> time to work on this part of the project, how might you choose to
>> refine or improve your design?

The idea behind a hash-map is that we have a fixed no of priorities and hence we
require
    a fixed no of queues to hold the threads with their corresponding priority. Hash
maps
    in theory could help us in reducing fetching time for these queues.

>> C6: The assignment explains arithmetic for fixed-point math in
>> detail, but it leaves it open to you to implement it. Why did you
>> decide to implement it the way you did? If you created an
>> abstraction layer for fixed-point math, that is, an abstract data
>> type and/or a set of functions or macros to manipulate fixed-point
>> numbers, why did you do so? If not, why not?

SURVEY QUESTIONS

Answering these questions is optional, but it will help us improve the
course in future quarters. Feel free to tell us anything you
want--these questions are just to spur your thoughts. You may also
choose to respond anonymously in the course evaluations at the end of
the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

```
>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems?  Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?
```