```
                  +-------------------+
                  |      CS 521       |
                  |   PROJECT 1       |
                  | DESIGN DOCUMENT   |
                  +-------------------+
```

---- GROUP ----

Zhe Zhang zzhang64@buffalo.edu
Jie Lyu
Sen Pan senpan@buffalo.edu

---- PRELIMINARIES ----
None.

```
                  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.
> Added to struct thread:

```
    int64_t time_to_wake;               /* At which time the thread should be waken ,meanless if thread i
    struct semaphore wake_sig;          /* sema down by itself, and up in timer interrupt */
```

New static Variabble

```
static struct list block_queue; /* block queue !*/
static struct lock block_mutex; /* only one thread have access to block queue at a time !*/
```

---- ALGORITHMS ----

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

1. When the thread call `timer_sleep` function, we use call

`push_thread_to_blockQ` function, instead of using busy waiting.

2. In `push_thread_to_blockQ` function, we first remove the thread from `ready_list`. Then assign the wakeup tick to `wake_sig` variable, and insert the thread into `block_queue` according to `wake_sig` in ascending order using the list function `list_insert_ordered`. Here we have to define a compare function `insert_by_target`.
3. call `sema_down(&th->wake_sig)`, note `wake_sig` is a local variable to struct thread.

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

1. Whenever `timer_interrupt` is called, it will call `thread_tick` which is in thread.c. We would modify this function.
2. When `thread_tick` is called, check `block_queue`:
    - If it's empty, do nothing.
    - If not, check the front element (which is a `elem` in thread struct), access the `time_to_wake` of this thread.
        - if it's smaller than current tick, then call `sema_down(&th->wakeup_sig`), and call `insert_ordered_list` back to the ready queue, according to `priority` in descending order.
        - else break
        - TODO: should we set a threshold for while loop in thread_tick, which check the head of `block_queue`?

---- SYNCHRONIZATION ----

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

1. since the only shared variable is `block_queue`, we would use a `block_mutex` to guarantee only one thread can insert a new element into block queue.

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

1. timer interrupt has higher priority and it can not be delayed.

2. timer interrupt can check if block_mutex has a holder, if true, then it won't do anything, else in the `thread_tick` function, it can delete the elements using a while loop. But we should set a threshold for the while loop.

---- RATIONALE ----

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

1. Timer interrupt function is the only way that won't introduce extra cost, the check the sleep queue, since the interrupt occurs every tick, and won't introduce much delay.
2. In this design we uses semaphore without using other calls like `thread_block` or `thread_unblock` since to call `thread_block` we must disable interrupt, while to call semaphore we won't necessary disable interrupt, if we have something like hardware supported instructions, though in pintos sema_down disables interrupts.
3. TODO: We can still improve data structure in a more compact way, since there is already a wait queue in the semaphore. Actually we can use a gloabal semaphore variable and use its waiting list as block_queue. And slightly modify sema_down, which is more memory saving and faster.
   ```
                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.

New compare function

```
/* return true if a's priority is greater than b's */
bool priority_comp(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED);

/* test if the newly created thread or unblocked or priority changed thread has higher priority than curr
    Compatible under interrupt context or kernel thread context.
*/
void try_preempt(void);
```

Added to struct thread:

```
    /* TODO not sure yet! */
    struct thread* next; /*donation from the thread */
```

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

TODO
---- ALGORITHMS ----

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

1. When a thread is waked up, it will insert into the ready queue according to the priority and call `sema_up(&th->wake_sig)`.
2. In sema_up, we would check whether the current thread has lower priority than the thread in the front of ready queue. If true, call `yield()` or `intr_yield_on_return()`.
3. When a thread expires a time slice, it will call `thread_yield()` first, which will insert current thread into ready queue. If the current thread has highest priority, it will goes to the head of the ready queue, and continue to be executed in the next time slice.

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

TODO

> B5: Describe the sequence of events when lock_release() is called

> > on a lock that a higher-priority thread is waiting for.

TODO
---- 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?

1. Condition Race:
    - Thread with lower priority never has a chance to set a priority until those threads with higher priority exit or are set to lower priority.
    - If a lower priority is assigned th current thread and the first thread in the ready queue has higher priority, then the thread with lower priority still takes the resources.
    - TODO
2. Solution:
    - If we set a lower priority to current thread, it should check the ready queue, and if the first thread has higher priority, then current thread yields.

---- RATIONALE ----

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

TODO

```
                  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.
> > Added to struct thread:

```
    int nice; /* nice number */
    MyFloat recent_cpu;
```

New struct _MyFloat and a set of function to the float

```
/* Implement float from scratch.
   ex: to define a 17.14 format float number
   set precision = 14
*/
typedef struct _MyFloat {
    int val;       /* integer and fraction part of a float */
    int precision; /* decimal fraction 2^(-precision)*/
}MyFloat;

/* init a with a initial integer value and choose a precision which is usually 14*/
void InitMyFloat(MyFloat* a,int integer, int precision);
/* copy the value in b to a */
void CopyMyFloat(MyFloat* a, const MyFloat *b);
/* float substraction */
MyFloat* MySubstraction(MyFloat* a, MyFloat* b);
/* float addition */
MyFloat* MyAdd(MyFloat* a, MyFloat* b);
/* float multiply */
MyFloat* MyMultiply(MyFloat* a, const MyFloat* b);
/* float divide */
MyFloat* MyDivide(MyFloat* a, const MyFloat* b);
/* float and int operations*/
MyFloat* MyMultiply_Int(MyFloat* a, int b);
MyFloat* MyDivide_Int(MyFloat* a, int b);
MyFloat* MyAdd_Int(MyFloat* a, int b);
MyFloat* MySub_Int(MyFloat* a, int b);

/* get the int value */
int MyFloat2Int(const MyFloat* a);
/* get the int value multiplying 100 */
int MyFloat2Int_100(const MyFloat* a);
```

New global variable:

```
MyFloat load_average; /* store load_average for all theads with float */
```

---- 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**	**recent_cpu**	**recent_cpu**	**priority**	**priority**	**priority**	**thread**
ticks	A	B	C	A	B	C	to run
0							
4							
8							
12							
16							
20							
28							
32							
36							

>> 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?

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

---- SYNCHRONIZATION ----

>> 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?

---- RATIONALE ----

> 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?

1. Advantages:
    - The priority related calculations are grouped together under timer_interupt() makes the code straightforward, simple and easy to understand.
2. Disadvantages:
    - TODO