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

---- GROUP ----

Zhe Zhang zzhang64@buffalo.edu
Jie Lyu jielv@buffalo.edu
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 is
    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. 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);
(updated)
bool donate_compare(const struct list_elem *a, const struct list_elem *b, void *aux);
bool comp_priority_cond(const struct list_elem *a, const struct list_elem *b, void *aux);
bool lock_compare(const struct list_elem *a, const struct list_elem *b, void *aux);

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

Added to struct thread:

```
(updated)
    int donated;      /* total amount the thread donated */
    bool priority_delayed;
    int priority_aysnc; /* set priority async */

    struct list donator_locks; // donator locks
    struct lock* donee; // maybe we have multiple donee, but here we use just one donee to pass test cases
```

Added to struct lock

```
(updated)
    struct list_elem lock_elem; /* Goes to th->donator_locks */
    int donation; /* the amount the first thread donated to the holder */
    int highest_priority; /* compare lock priority when inserting the lock to thhead->donator_locks*/
```

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

(updated)

```
before donation:
L(31)  M(32)  H(33)
 \    / \     /
  \  /   \   /
   \/     \ /
   a       b

after donation:
L(33)  M(32)  H(31)
 \    / \     /
  \  /   \   /
   \/     \ /
    a      b
   (2)    (2)

after L release:
L(31)  M(33)  H(32)
      / \     /
     /   \   /
    /     \ /
   a       b
   (0)    (1)

after M release:
L(31)  M(32)  H(33)
              /
             /
            /
   a       b
          (0)
```

- Donate forward:
    - `lock->holder->donee` (which is a lock)
    - `donee->holder` (which is a thread holding the lock)
- Return donation (or donate backward):
    - `list_entry(list_front(&th->donator_locks), struct lock, lock_elem)` (which goes from the current thread to the first donator lock)
    - `struct thread *first_thread = list_entry(list_front(&fisrt_lock->semap` (which goes from the current thread to donator thread)
- Multiple donator: `(struct thread* th)->donator_locks` which stores all donator locks. Note only the first lock will contribute to the donation.

---- 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?

(updated)

1. If current lock has no holder
    - take the lock and store dontated number in lock.
2. If current thread has higher priority than the holder of the lock
    - 2.1 If the holder has no other donator locks, current lock will be the donator lock. Update donee and donated and priority for current thread (call `Donate_Forward`).
    - 2.2 If the holder has donation from another lock, and the donation comes from the current lock, we check whether current thread will donate more than the first thread in `lock->semaphore->waiting_list`.
        - If current thread donate less than the first thread, then just insert the thread into waiting_list. Update coresponding values.
        - Recover the priority in the first thread of waiting list, update donation and insert the thread into head of waiting_list.
    - 2.3 If the holder has donation from current lock, then compete the donation between lock. Update coresponding values.
        - If current lock has higher donation, then preempt the

donation from the lock, and insert the current lock into the head of `donor_locks`.
        - Else repeate 2.2
3. If current thread has lower priority
    - we should have repeated 2.2 procedure. Since testing cases does not come across this part and we don't have extra time, just `continue`.

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

(updated)

1. If the current lock is donating to the current thread.
    - Call `Donate_Backward()`
    - 1.1 If there is another donator lock, let the lock donate. Update corresponding value.
    - 1.2 If no other lock, clean the value related to donation.
    - Update donator's priority, re-insert the thread into `ready_list`.
2. Else
    - Call `Donate_Backward()`
3. If there is new priority assigned to current thread, set the priority to the new value, else restore original priority value.
4. schedule

What happens in `Donate_Backward()`:

1. Our goal is to shift the priority of current thread to the first thread in the semaphore's waiting list, and use a recursive call the return donation backward.
2. Update lock's donation value.
3. update current thread's donation value.

---- 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?

(updated)

1. Condition Race:
   If the program is calculating the donation value according to its original priority p1, but without finally update the priority value and `donator_locks` when there is a context switch, another thread tries to set the priority to p2. This will result in mis-calculation for priority donation recovery, and the thread's priority will becomes p1 after releasing the lock. This means `thread_set_priority` takes no effect.
2. Solution:
   We can add a lock in thread structure to gurantee that when `thread_set_priority` and `lock_acquire` try to access priority, it is mutual exlusive.

---- RATIONALE ----

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

(updated)

- This design is quite straightforward, we try to keep every possible value that may be useful later, and we have to use a list to keep track of donator locks in order to pass multiple donation test. And use a pointer to store the lock the current thread donate to, in order to implement chain donation.
  We think there is some redundancy in the parameters we take, and we think if given more time, we can refine the solution in a more compact way.
- Another deficiency is that current implementation can only deal with only one lock contributed to the thread and there is some implementation issues still remain to solve as mentioned in B4.3.

```
                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	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59		58
32	16	12	4	59	58	58	A
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?

(updated)
Uncertaintities:

1. The specification does not point out the order of updating recent_cpu and priorities of each threads if both of them should be updated.
2. The specification also does not specify whether the current thread should yield, and whether the ready_list should be rearranged after updating each thread's

priority.

Our solution:

1. We update recent_cpu first and then update priorities. Since priorities depend on recent_cpu.
2. We compare the priority between thread_current() and the one in the head of ready_list, which means our implementation of BSD scheduler is also preemptive. However, we do not update rearrange ready_list because it's very low efficient which takes O(nlong) or worse, and in most case, only part of the order of threads will change. But it still obey the BSD scheduler scheme,
   and will introduce a little delay.

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

(updated)

- We do scheduling outside interrupt whenever:
  - call sema_up
  - create a new thread
  - release a lock
  - set a new nice number
- We do scheduling inside interrupt whenever:
  - update each thread's priority
  - wake up a sleeping thread

Since most of MLFQs priority calculation happens inside context switch, and whenever a scheduling happens inside interrupt is will disable interrupts, the timer interrupt will not be excuted at this point. Chances are that current_thread may loose recent_cpu and update_priority_all() may not be excuted, which will introduce some precision loss.

---- 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?

1. Advantages:

- The priority related calculations are grouped together under timer_interupt() makes the code straightforward, simple and easy to understand.

(updated)
2. Disadvantages

- Iterating each thread is time-consuming if the number of threads increases. And it willcompress the excution time for each thread.
- The excution time becomes uneven. For example, if ticks%4 == 0 we have to scan all the threads before exit timer interrupt context, otherwise exit timer interrupt immediatedly.

3. Possible improvement

- Is it plausible for us to update partial of the MLFQs? For example, we only update the first 4 level of MLFQs.
- Each timer tick, we only update one fourth of the threads in the ready_list.

---- 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?

(updated)
We create an abstraction for float and create a set of related API for floating calculation, i.e. `_MyFloat`. We use an integer to save the float value, and an integer to indicate precision.

Reason:
By providing a set of API for our calculation, it's very easy to understand implementation for other group members as well as for the coder.
Besides, it's very easy to debug if there are any problems, since we can debug each API independently.

Refinement:
Actually we can use the last 4 bits to define the precision of float, the remaining 28 bits as value, which we think is enough to pass the test case.

```
ex.

interger       decimal       precision
 ____________ _____________ ________
|0         14|15         28| 1 1 1 0|
 ____________ _____________ ________
```