```
+--------------------+
|        CS 140      |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- GROUP ----

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

wenhui qiao <qiaowh@shanghaitech.edu.cn>
zhaozheng shen <shenzhzh@shanghaitech.edu.cn>

---- 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.

In struct thread, we added the following member variables:
    struct list_elem wait_elem;   /* List element in the waiting list. */
    int64_t wait_value; /* Time need to wait before unblock. */
In thread.c, we added a static variable:
    static struct list waiting_list; /* List of processes in THREAD_WAITING state, that is, processes that are ready to run but not actually running. */

---- ALGORITHMS ----

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

The current thread need to sleep n ticks if we call timer_sleep(n). If n <= 0, then the thread doesn't need to sleep, and we can just return. Change the thread into blocked status and set the wait_value n, and add the thread into waiting list. Finally, after put the current thread into waiting list, we call schedule() to find the next thread to run.

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

The initial operation is to put the thread into ready queue, the thread will in running status and ready status alternately, which will waste a lot of cpu time. Instead of putting the thread into ready queue, the thread will not wake up and change into ready status untill its wait_value decreased to 0. To check whether the thread is ready to go into ready queue, which means its wait_value decreased to 0, we check at timer_interrupt function after every tick++, if any thread in the waiting queue is ready to go into ready queue, then unblock the blocked thread and put it into ready queue.

---- SYNCHRONIZATION ----

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

We firstly call intr_disable() to disable interrupt, and after finishing the sleep operation, we enable interrupt, which keeps atomic operations and avoid race conditions when multiple threads call timer_sleep() simultaneously.

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

We firstly call intr_disable() to disable interrupt, and after finishing the sleep operation, we enable interrupt, which keeps atomic operations and avoid race conditions when a timer interrupt occurs.

---- RATIONALE ----

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

This design is easy to implement, and convenient to modify.
We also consider to add another waiting_list to store the waiting threads in the wait_value order, and only compare the first item in the list each ticks++, but our design is more convenient to modify.

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.

In struct thread, we added the following member variables:

```
    struct list locks;            /* Locks that threads hold. */
    struct lock * lock_wait_for;  /* The lock thread is waiting for.
```

```
*/
    int lock_priority;            /* Max priority among locks list.
*/
    int original_priority;        /* Original priority regardless of
locks. */
```

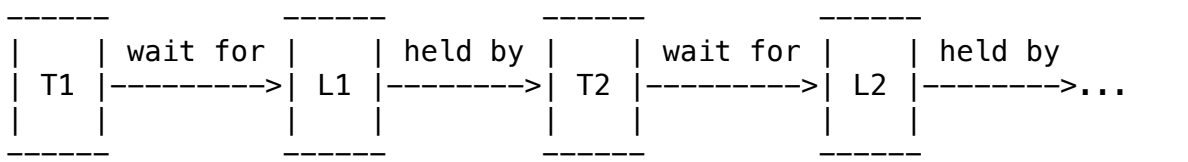
In this way, the original variable 'priority' maintain the maximun between
lock_priority and original_priority.
In struct lock, we added the following member variable:

```
        struct list_elem lock_elem;   /* Elem in thread.locks */
    int priority;                 /* Max priority among
sema.waiters. */
```

As the thread has a list of locks, the list_elem is necessary in the structure of lock. This priority is set to make it easy for comparison,
which improve its efficiency.

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

Notation: T - thread L - locks
Each threads has a pointer point to the lock it is waiting for, and each
locks has a pointer point to its holder, which is a thread.
In this way, it looks like a linke list:



---- ALGORITHMS ----

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

Every time before releasing a lock or sema or condition variable, the
program iterate through the list of threads waitig for it, to find the
threads with highest priority, and wake it up.

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

When a thread is waiting for a lock, if the priority of lock is less than the priority of thread, then higher the prioriy of lock to the priority of threads. Then consider the lock and its holder. If the holder's priority of locks is less than that of the lock, then higher
it.

The lock's holder maybe is waiting for another lock, so the process above keep running until the donation is not necessary(i.e. the thread
does not wait for any lock or prioriry ls not higher than the following
priority)
After donation, set the pointer lock_wait_for to NULL and add the lock
to its list, and update the owner's priority.

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

Set the holder of lock to NULL, and remove the lock from the current thread's lock list. Update the current threads priority. Then iterate
through the waiters of the lock, find the threads with max priority, and wake it up.

---- 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?

During the donation, the priority of a thread may be modified by another
thread. In the same time, it may use the thread_set_priority to set itself. Then there is a potential race.
We disable interupt to prevent it from happening. A lock may not work in
our implementation, as we did not provide interface to achieve this.

---- RATIONALE ----

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

This design is easy to come up with, easy to implement, and convenient
to maintain and modify.
We also consider to add another list of waiter to the structre of lock,
but it would make the struct much larger, which cause much more memory
space.

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.

In struct thread, we added the following member variables:
    int nice; /* nice value for priority. */
    int recent_cpu; /* recent_cpu value for priority. */
In struct thread.c, we added the following `typedef':
#define max(a, b) (((a)>(b))?(a):(b)) /* define max function */
#define min(a, b) (((a)<(b))?(a):(b)) /* define min function */
In struct thread.c, we added the following globel variable:
    int load_avg = 0; /* define load_avg for calculating priority*/

---- 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 ticks	recent_cpu A	recent_cpu B	recent_cpu C	priority A	priority B	priority C	thread 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	B
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?

Yes, we use LFU rule to resolve the ambiguities in the scheduler
specification make values in the table uncertain, which means when
ambiguities, we let the thread who has been waiting for the longest
time in the waiting list to run. This match the behavior of our
scheduler.

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

Inside: we recalculate recent_cpu and load_avg for each thread(not
just the running thread) every second, which cost cpu a lot of work

and affect performance.
Outside: outside interrupt will only affect nice value, which will affect priority.

---- 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?

This design is easy to implement, and easy to understand. But it is better to add macro and declare floating point arithmetic.
If we were to have extra time to work on this part of the project, we might choose to add macro and declare floating point arithmetic, which save a lot of time and make the struct more clearly.

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?

Part 2 took us a lot of time, but we understand how priority scheduling work more cleary.

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

Yes.

>> 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?

Plus, wenhui qiao implemented part 2, and zhaozheng shen implemented part 3, and we implemented part 1 together.