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

---- GROUP ----

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

Wang Xinyi <wangxy6@shanghaitech.edu.cn>
Zhu Yifan <zhuyf@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.

https://www.cnblogs.com/laiy/p/pintos_project1_thread.html

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.

Add **int64_t block_time** to the thread struct in thread.h

It counts how long a thread should be blocked. It decreases by one after every tick. When turning to zero, the thread should be awaken.

---- ALGORITHMS ----

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

When calling time_sleep(), first set the block_time. Then call thread_block(), set thread status to THREAD_BLOCKED. Putting the thread into the sleeping queue and calling schedule() to arrange another new thread to run from the ready_list. In every interrupt tick, check every thread’ s block_time to see if it is ready to call unblock() to put it back to ready_list.

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

Instead of checking every thread’ s block_time, we can construct a sleeping queue to store all the blocked thread and sort it from small to large. We only need to check the first element to see if it needs to be unblocked. This will save much time.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
Using thread_block() can avoid the race conditions.

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

Disable the interrupt so that interrupt will not affect the timer_sleep().

---- RATIONALE ----

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

Using block_time to count how long the thread need to be blocked is much better than the original way it provided to us. The original one is “busy waiting” and it occupies a lot of CPU resources. So this design can easily solve this problem.

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.

int original_priority

Save the earliset priority of the thread.

struct list hold_locks
Save all the locks that the thread holds.

struct lock *waiting
Save the lock that the thread is waiting for.

struct list_elem elem
The elem in the current list.

int max_priority
The largest priority of the current lock among all of its priority.

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

H, M ,L are threads.
1 and 2 are locks.
→ means the left one donate its priority to the right one.

```
  1   2
H → M → L
```

L’s priority now is H.
M’s priority now is H.
When lock 2 is released, the priority of L becomes L and priority of M is still H, then the thread M runs. After lock 1 is released, the priority of M becomes M, and then H runs, then M, then L.

---- ALGORITHMS ----

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

All of the threads which are ready status are in the list of ready_list, then picking the thread having the max priority and wake it up. Then the current thread has the highest priority.

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

The purpose of lock_acquire is to give the lock to the current thread. Using iteration to find the donating threads’ priority. Find the largest one of these priority and give this to the current lock, which means the lock has the max priority to ensure that the current thread can keep on running until the lock is released.

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

First traverse the list of locks that the current thread is holding. If it is found, then do the remove operation. Remove this lock from the list, and then update the priority of the current thread by finding the max priority in the current lock list which has already remove the donation of the removed lock. And let the holder of the lock be NULL.

---- 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?

When the thread_set_priority() is running, we can set another function to avoid other threads calling this thread_set_priority(). Therefore, there should always be one thread that is running in this function.

---- RATIONALE ----

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

In finding the max priority of the thread in ready_list, another design is to sort the list first by decreasing order, then pick out the first element, which must be the largest one.

But using this design, the threads that isn’ t used also be ordered, which is not necessary. So the max_picking design can avoid ordering these threads and raise the speed of the function.

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.

Add **int nice** and **int recent_cpu** in thread.c. Add **static int load_avg** in thread.h.

There three variables are used to store value of nice, recent_cpu and load_avg.

---- 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	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. If two threads have the same priority, which thread should be run next can be ambiguous.
If two threads have the same priority, we choose the one that is not recently running to run first. This matches the behavior of my scheduler.

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

In the timer_interrupt,recent_cpu of current thread increases by one per tick, priority updates per four ticks, load_avg changes every one TIMER_FREQ and recent_cpu of every thread changes every one TIMER_FREQ.
Outside the timer_interrupt, only nice updates.
So if there are a lot of threads in the ready list, it will take a lot of time to update recent_cpu in the timer_interrupt. In this way, the performance can be affected a lot.

---- 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?

Advantages: Using three functions to update priority, recent_cpu and load_avg separately so that the logic of the code can be very easy to understand.

Disadvantages: When there are a lot of threads in the ready list, the performance will be affected because there are a lot of tasks to do in timer_interrupt such as updating every thread’ s recent_cpu every one TIMER_FREQ.

Improvement: Find a good way to decrease the tasks that need to be done in timer_interrupt. Find better data structure to solve the problem.

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?