```
          +--------------------+
          |        CS 130      |
          | PROJECT 1: THREADS |
          |   DESIGN DOCUMENT  |
          +--------------------+
```

---- GROUP ----

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

Yi Qian <qianyi1@shanghaitech.edu.cn>

Guancheng Li <ligch@shanghaitech.edu.cn>

---- PRELIMINARIES ----

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

Yi Qian finished Task1 Alarm Clock and Task3 Advanced Scheduler and related part in design doc.

Guancheng Li finished Task2 Priority Scheduling and related part in design doc.

>> 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 "threads/thread.h" struct thread:

int64_t thread_sleepTime;

Purpose: Record for how long time the thread should continue to sleep.

---- ALGORITHMS ----

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

In a call to timer_sleep(), first ticks is checked. Then turn off interrupts, get current running thread and set the number of ticks it has to sleep over. Call thread_block() to stop the thread from running and turn interrupts back on.

In the timer interrupt handler, find every thread and check if it is sleeping. If it is, decrease thread_sleepTime by 1. If the sleep time is finished, put the thread back to ready state.

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

In part A, we are not quite considering this problem as we do not think decreasing a value for every thread takes much time. In part C, we try to reduce

the amount of time spent in the timer interrupt handler by only recalculate priority for the running thread as only the running thread will change its recent cpu time.

---- SYNCHRONIZATION ----

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

Interrupts are disabled.

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

Interrupts are disabled.

---- RATIONALE ----

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

This works.

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 "threads/thread.h" struct thread:

(1) int original_priority;
Purpose: To record thread's priority before being donated.

(2) struct list owned_locks;
Purpose: To maintain a list which records all the locks the threads is holding.

(3) struct lock* waiting_lock;
Purpose: It represents the lock current thread is waiting for.

In "threads/synch.h" struct lock:

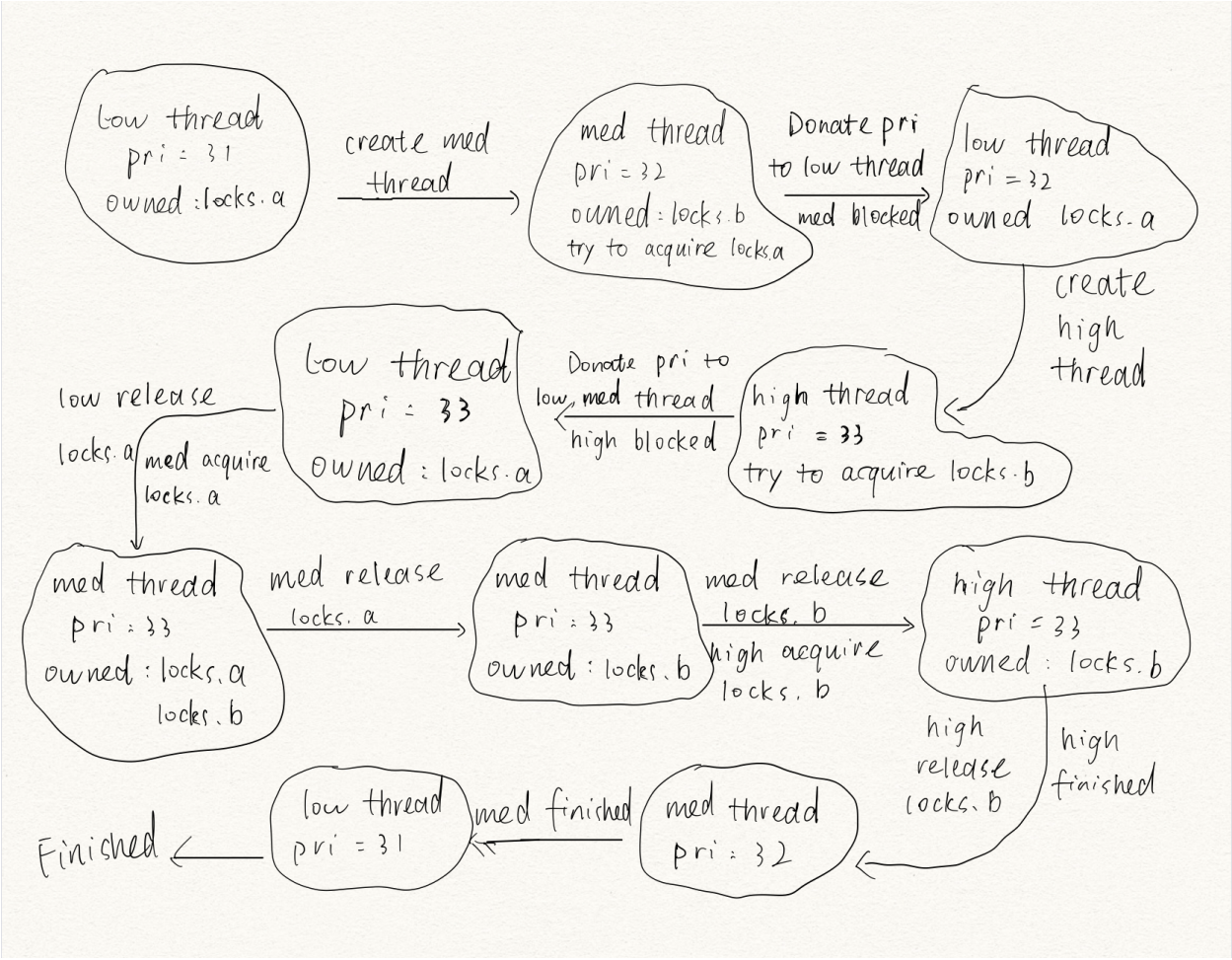
(1) int max_priority;
Purpose: It represents the lock waiter's or holder's max priority.

(2) struct list_elem elem;
Purpose: Consider lock as a element in owned_locks.

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

To track the priority donation, some date structures are added as mentioned above.

If a thread holds a lock, the lock is added to thread's owned_locks. If a thread
release a lock, the lock is removed from thread's owned_locks. That's the effects
of list owned_locks and elem.
When we check whether a thread will be donated, we will need the data structure
waiting_lock to check recursively.
And when we need to set back the thread's priority, we need the data structure
original_priority and lock's max_priority. That's the effects of lock's max priority
and thread's original priority.



---- ALGORITHMS ----

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

Firstly, in semaphore or condition variable, there is a list contains all the waiting threads. If a thread is waiting for a lock, it will be added in the waiting list.

Secondly, when we need to wake up a thread, we first order the waiting list by thread's priority in descending order. Then the waiting list

becomes a priority queue. And we unblock the front thread which has the highest priority.

So we can ensure that the highest priority thread will be waken up first.

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

First, check whether the argument is valid and disable the interrupt.

If lock_acquire() will cause a priority donation, it must be in some circumstances: The lock has a holder, the lock's holder isn't current thread and the lock's max_priority less than current thread's priority.

Then current thread will donate its priority to the lock's holder. And the lock will record the priority in max_priority.

But it is possible that the lock's holder is also waiting another lock and its priority is changed. So it maybe also need donate its priority to other threads again. So we will do what mentioned above recursively until the circumstances are not satisfied.

Then do sema_down() to acquire the lock as normal. But we need set the lock's
max_priority to the current thread's priority and add the lock to thread's owned_locks.

Finally, enable interrupt.

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

First, check whether the argument is valid and disable the interrupt.

Then remove the lock from thread's owned_locks. After that, check whether current thread has another lock. If had, we should compare the locks' max_priority
to current thread's original_priority and choose the highest priority. This event
is in order to check whether the current thread will be donated or set back to its original priority.

Then set the current thread's priority correctly.

Finally, do sema_up() to release the lock and enable the interrupt.

---- 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?

Because current thread may set its priority and another thread may donate its priority to current thread simultaneously. So we should

disable the interrupt first to avoid this race.

Then we change the original_priority directly, but only the new priority higher than current thread's priority or the current thread doesn't own any locks, its priority can be changed immediately. According to the test cases.

---- RATIONALE ----

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

I think there will be another design which is adding a list in struct lock which represents the lock's waiting threads instead of max_priority.

But in my opinion, it will has a higher complexity when release the lock and find whether the current thread is still being donated by other thread.

It's less convenient than current design.

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 "threads/thread.c", add global variable:
fixed_point load_avg;
Purpose: The value load_avg is used by all threads for calculating priority.

In "threads/thread.h":
fixed_point recent_cpu;
int nice;
Purpose: The nice value and recent cpu time for each thread.

---- 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	1	2	63	61	59	A
4	4	1	2	62	61	59	A
8	8	1	2	61	61	59	A
12	12	1	2	60	61	59	B
16	12	5	2	60	60	59	B
20	12	9	2	60	59	59	A
24	16	9	2	59	59	59	A
28	20	9	2	58	59	59	B
32	20	13	2	58	58	59	C
36	20	13	6	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?

The order of A, B, C runned is not given. If they have the same priority values, it is ambiguous to decide which to run.
Our scheduler uses FIFO to decide. In practice, there will be a order.

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

If too much time is consumed calculating priority inside the interrupt, the time for current thread to run is taken, but recent cpu time will increase anyway. This will cause the priority of important tasks to decrease faster than it should be.

---- 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?

Our design is using the ready list as the only queue for scheduling thread, instead of 64 queues. As making ready list a priority queue, it serves the same result as 64 queues. The advantage of our design is that it takes less space, and it is not significantly slow when the number of threads are not very large. However, when we have hundreds of threads running at the same time, using 64 queues may be more effecient.

If given extra time, we would like to implement other advanced schedule algorithms like round robin, as we are currently just letting the one with the highest priority run.

>> 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?

I implement the "fixed-point.h" and "fixed-point.c" according to the contents in project manual "B.4.4BSD Scheduler".

The reason for design fixed-point is that recent_cpu and load_avg need to be calculated in type of value but c99 doesn't support this type.

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?

It’s ok.

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

Sure. It helps a lot.

>> 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?

In order to complete the project, you should read the textbooks first.

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