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

---- GROUP ----

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

Group 19
李鹤洋 lihy3@shanghaitech.edu.cn
韩嘉恒 hanjh1@shanghaitech.edu.cn

Contributions:
Discuss together, and coding part:
Mission 1/2 mainly by Han Jiaheng,
Mission 3 mainly by Li Heyang.

---- 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.

```
struct thread
  {
```

```
    /* Original members... */
    /* Used to determine whether a thread is sleeping. True if
thread is sleeping. */
    bool sleep;
    /* Time(the timer ticks) to awake, meaningful only if
sleep,  otherwise meaningless. */
    int64_t  time_awake;
    /* Original members... */
  };
```

---- ALGORITHMS ----

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

Before modifying, `timer_sleep()` will check whether current thread have slept enough time, if yes, then exit `timer_sleep()`, otherwise yield() and let kernel schedule again. However, sleeping threads may be put into run repeatly and cause busy waiting.

After modifying, `timer_sleep()` will block the thread to let it sleep and then the thread will be in waiting queue and will not be schedule again. This avoids busy waiting. However, the thread needs to be awake after its sleeping time and this need to be check by kernel every time tick. So in the timer interrupt handler, the Boolean variable `sleep` and `time_awake` of each threads will be check to determine whether it should be unblock.

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

Instead of creating another list to store all sleeping threads, we choose to use `thread_foreach()` to iterate through all list and use Boolean variable `sleep` to determine whether a thread is sleeping. This is a tradeoff between overhead of iterating through waking threads and overhead of extra list operations.

---- SYNCHRONIZATION ----

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

```
  /* Some preparations before going to sleep... */
  /* Then go to sleep. */
```

```
  enum intr_level old_level = intr_disable ();
  sleep_thread->sleep = true;
  thread_block();
  intr_set_level (old_level);
```

This is part of the code in timer_sleep(). We don't really care about multiple threads call timer_sleep() simultaneously, as long as the update of Boolean variable sleep and thread_block() is done together, i.e. a thread blocked by sleep always has sleep == true. This is done by intr_disable () and intr_set_level (old_level).

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

As mentioned in A4, this is done by intr_disable () and intr_set_level (old_level).

---- RATIONALE ----

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

This is a light weighted implementation to avoid busy waiting. There are others designs such as we can use a list, or even a priority queue (ascending in waking time, so each time we don't need to iterate through threads that actually should not be waken.) to keep track of sleeping threads. However, those designs need extra list operations and the fewer thread iterations we want to achieve, the more list operation overhead we will have, and it is expensive. So we choose to use this easy and light weighted design.

PRIORITY SCHEDULING

---- DATA STRUCTURES ----

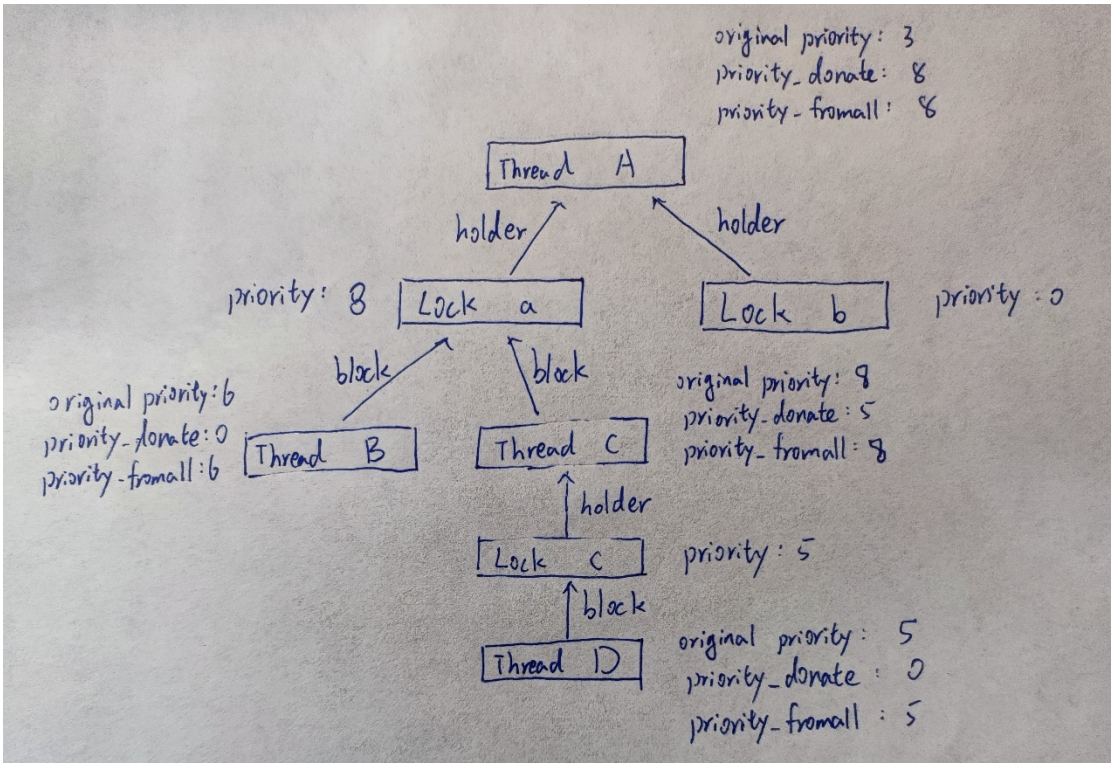
>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

```
struct thread
  {
    /* Original members... */
```

```
    /* Extra priority from donation, if no donation, then 0. */
    int priority_donate;
    /* Current priority after donation, which is always equals
to Max(priority,priority_fromall) */
    int priority_fromall;
    /* List of locks, which store all locks held by the thread,
empty if the thread holds no lock */
    struct list locks;
    /* The lock that blocks current thread, if current thread
not block by a lock, equals to NULL */
    struct lock *block;
    /* Original members... */
  };

struct lock
  {
    /* Original members... */
    /* The thread that currently hold the lock, original member
for debugging, also used in priority donation. */
    struct thread *holder;
    /* Priority of the lock. Equals to max priority of all the
threads that blocked by this lock. */
    int priority;
    /* List element, for list locks in Struct thread. */
    struct list_elem elem;
  };
```

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



`struct lock *block` in `struct thread` points to the lock that block the thread.
`struct thread *holder` in `struct lock` points to the thread that hold the lock.

With these two, we can build a bottom to up tree with thread and lock as nodes and we can track priority donation with the tree. Also, each node has a list of its children, which is used in updating priority.

---- ALGORITHMS ----

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

All these three have a list of waiters, and each time a lock, semaphore, or condition variable should wake a thread, we can pick the thread with the highest priority in the waiter list to wake up.

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

First, we call `intr_disable ()`, the following action should not be interrupted. Next we check if current lock is free, if free, acquire it, if not, block the thread by the lock and ask the thread to donate priority for the lock(update priority) and yield and wait until being

waken and acquire the lock. After acquire the lock, set the holder of the lock to be current thread and put it into locks, and ask the lock to donate priority(update priority). Finally we set back the old interrupt level.

When updating priority of a thread or a lock, nested donation may happen. So we should first update the priority of the thread or lock itself by the locks it holds or the threads it blocks, then we need to update the blocker of a thread or the holder of a lock if it exist, which is block or holder.

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

First, we call intr_disable (), the following action should not be interrupted. Before we actually release the lock, we should update all information, otherwise when release the lock, if there is a higher-priority thread waiting for, the current thread will immediately yield and leave information not updated. To update information, we should set the lock to be hold by no one and remove it from current thread' s locks, because some of the priority of current thread may come from donation of the lock, so we need to update its priority. Finally, we release the lock and set back the old interrupt level.

---- 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?

A thread in thread_set_priority() may be interrupted before it finish updating all information, such as chain donation, and a new thread is scheduled. This can cause chaos. intr_disable () is used to ensure all information are updated properly before switch to other threads.

This can not be avoided by a lock because lock itself is involved in priority donation, the thread should not acquire a lock before its priority is set properly, otherwise the lock will get a wrong priority. Another problem is that if a thread is blocked by the thread_set_priority lock, it will donate priority to the thread whose priority is being modified and that does not make sense.

---- RATIONALE ----

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

The main advantage of this design is that it is O(1) when querying priority and it only does essential update from bottom to up.

Another design may be traverse the tree from top to bottom when query priority and when updating, it only update current node.

It can be observed that when the kernel runs, querying priority happens a lot, if each time querying priority needs traversal along the whole tree, it is very costly. Instead, updating priority happens less frequently. So we choose the first 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 `thread.h`, added to struct thread:

```
    int nice;                           /* Nice value of a thread. */
    fixed_point recent_cpu;             /* recent_cpu. */
```

In `thread.c`, new global variable:

```
fixed_point load_avg;
```

Add `fixedpoint.h` and `fixedpoint.c`, and new type fixed_point defined in `fixedpoint.h`:

```
typedef int fixed_point;
```

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

There are several ambiguities.
The question only provides with niceness and three threads, we don’t know time frequency to know how many ticks a second costs. So we can’t calculate load_avg and update recent_cpu as the formula goes. So I just update each thread’s recent_cpu using niceness, i.e. the timer ticks is 0, which means it should be start of a second. And then on, I add 1 to currently running thread’s recent_cpu each tick, check and update their priorities by formula. But in my implement, I achieve all the update that should realize by formula and principle. When choosing next thread to run, we notice that there are always multiple threads that have equal priority, and I choose next thread

by FIFO, i.e. first in first out principle. The thread got that priority earliest would be the next thread to run. And our scheduler matches this behavior, and this is called “round robin”.

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

Because we calculate load_avg, recent_cpu and priority very frequently, and for a system that has lots of threads, it costs a lot of resources to achieve the whole computing. Our scheduling is inside the interrupt context, it increases the cost of scheduling, hence decreasing the performance.

---- 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?

Advantage: Simply added several variables and functions to realize the whole design. We didn’t apply 64 queues in the way document has explained, however, we used only 1 queue to achieve this goal, which provides efficiency and readability. In addition, we relied a lot on the functions and data structures the original system has offered, maximizing our stability and flexibility.

Disadvantage: We turned off the interrupt during the process, which may affect the performance and cause declining the efficiency of system.

If we have extra time, we may implement 64 queues instead of only 1 queue, and use lock instead of turning off the interrupt.

>> 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 defined a new type called ‘fixed_point’ in a new header file ‘fixedpoint.h’, and implement all the arithmetic functions we need in ‘fixedpoint.c’. The reason I did in this way it that it provides

flexibility and readability, making us easier to modify if we want and convenient to read for readers. In a nutshell, it’s simple, and faster.

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?