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

---- GROUP ----

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

Yuxin Miao <miaoyx@shanghaitech.edu.cn>
Tianran Zhang <zhangtr@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.

Reference: (All of great help)
https://www.bbsmax.com/A/ke5jeDp7Jr/#timer_sleep
https://piazza.com/cuhk.edu.hk/fall2018/estr3102/resources
http://bits.usc.edu/cs350/assignments/project1.pdf
etc.

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.

------ thread.h -------
```
int  block_ticks  /* Record how many ticks the caller needs to sleep. */
```

ALGORITHMS

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

Previouly, if a thread calls timer_sleep() somewhere during its occupation of CPU, the timer_sleep() uses thread_yield() during the length of ticks the caller requring to sleep. However during this process, no other threads can occupy the cpu since they will all be yield in the time of "ticks".

After modification, each time a thread calls timer_sleep(), use the thread-property "block-ticks" to take down how many ticks is requires to sleep and block the caller, yield the cpu.
Then in timer_interrupt(), every tick passes by, check through those threads that are blocked with a block_ticks and minusing 1, since the ticks passed. Wake those whose block_ticks turn back into zero, in other words, have slept long enough. And add them back to ready_list to wait to be scheduled later.

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

In conclusion, to minimize the amount of time spent in the timer_interrupt() requires to minimize the calculations done in timer_interrupt.

By far, the most time consuming calculation is iterating through all threads to check and minus the block_ticks.
Thus, construct a block_list or sleep_list only to store the threads that are blocked via caller timer_sleep, thus we can only iterating through such a shorter list rather that longest all_list to wake up threads that have slept long enough.
However, we failed to debug this design thus going back to the previous all_ list design.
---- SYNCHRONIZATION ----

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

Use the intr commands to disable the interrupt at the beginning and re-enable it after. This operation is therefore atomic and no other threads can sleep while the current thread is trying to.

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

As mentioned above, by disabling interrupts and protects the variables and functions accessing in the criticle section can we avoid the imterrupt during a call to timer_sleep().

---- RATIONALE ----

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

Sincerely speaking, we really want to implement the "block_list" method in order to decrease the time spent in the timer_interrupt. This will void errors like cannot tick and calibrate correctly due to the heavy load of calculations.

On the contrary, the current implementation also has a benifit of easy implementing. Because it requires no need to construct more structures and can be implemented just by adding some new properties like above.

PRIORITY SCHEDULING

---- DATA STRUCTURES ----

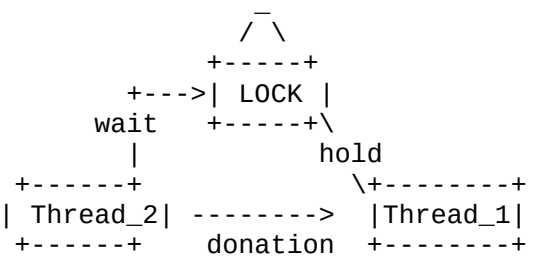
>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

```
------ thread.h -------
int block_ticks;                    /* Ticks to sleep. */
int old_priority;                   /* Priority that is signed at first. */
struct list owning_list;            /* The list of owning locks. */
struct lock *waiting_lock;          /* The locking I am waiting for. */

------ synch.h -------
struct list_elem elem;      /* List elem to add in the holding list. */
int lock_priority;          /* Max Priority of the lock */
```

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

QUESTION 1 data structure that tracks priority donation:
(ASCII Art Picture from Internet)



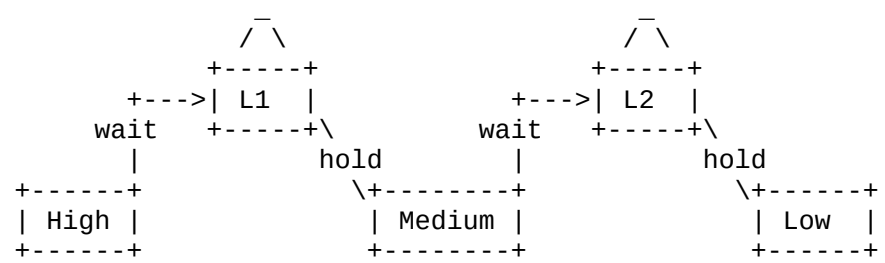
During a priority donation, first there this a THREAD_1 a that has already acquired the LOCK. The OWING_LIST of THREAD_1 right now stores the LOCK.

Then a THREAD_2 comes and trys to acquire the LOCK, failed. THREAD_2 then checks that its priority (P2) is actually higher than that (P1) of the owner of the LOCK, thus THREAD_2 passes P2 to P1, so as the priority of the lock and THREAD_1 uses a ORIGINAL_PRIORITY to store the old P1. Futher, THREAD_2 updates its waiting_lock (LOCK).

When THREAD_1 releasing the lock, it fisrt either goes through the OWING_LIST if it's not empty and choose to highest_priority's lock 's priority to be THREAD_1's next priority, or it just reuses its ORIGINAL_PRIORITY.

Then unblock the highest priority thread in the waiting list of LOCK, THREAD_2 in this case and push it back to the ready_list to wait to be scheduled.

QUESTION 2 Nested Donation (ASCII Art Picture from Internet)



Stage 1: Low Priority Thread (LPT) holds L2 first.

Stage 2: Medium Priority Thread (MPT) comes and tries to acquires L2 fails, MPT then donates its priority to LPT and L2, itself then waiting for LPT to release L2.

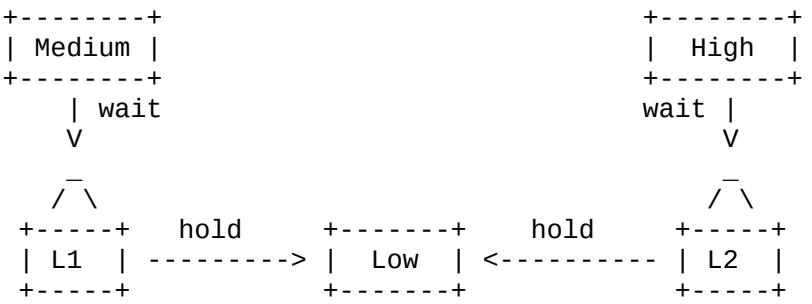
MPT also atempts to acquires L1. Since no one is holding L1, MPT successfully acquires L1.

Stage 3: High Priority Thread (HPT) now comes and tries to acquires L1, only to discover that L1 is held by MPT, so HPT donates its priority to MPT.

But HPT also spots that MPT is waiting for L2 and only after MPT acquires L2 successfully can HPT acquires L1. So through the lock that MPT is waiting for, HPT also donates its priority to the 2nd order holder of the lock, LPT in this case.

So right now even LPT has been donated ones, it still holds the priority of HPT after the nested donation.

PLUS Explanation: Multiple Donation (ASCII Art Picture from Internet)



Stage 1: The Low Priority Thread (LPT) holds both Lock1 and Lock2.

Stage 2: Medium Priority Thread comes the second and tries to acquires L1.
It fails so MPT donates its priority to LPT

Stage 3: Hight Priority Thread comes the third to acquire L2, also fails.
Futher HPT also spots that it has a higher priority than that
of the current holder of L2, no matter if that holder has been
donated or not.

So HPT donates its priority to LPT. So now LPT has a priority of
HPT, higher than that of the MPT.

Stage 4: When LPT either release L1 or L2, it goes through the OWNING_LIST
as mentioned above and changes it's priority to that one, if the one
is higher that the current priority, whether the current one is
donated or not.

So if LPT releases L2 first, its priority goes to the one of L1,
which is the MPT's priority.

Or if LPT releases L1 first, its priority remains the same, because
the fisrt lock in the remaining OWING_LIST has a priority not higher
than the current one.

---- ALGORITHMS ----

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

To make the waiting list a priority-sorted list, with two methods.

Method 1: Insert with order sorted by priority

Method 2: Sort the list by priority before pop the front to unblock.

We prefer the Method 2, because the priority doantion could happen
when the thread is in the waiting list, and thus, it's new priority
doesn't fit the position sorted by the priority when it was inserted.

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

The steps discussed are the steps taken in our version of implementation.

Step 1: When thread called lock_acquire(), it first checks if the lock
has a holder already. if not, down the semaphore of the lock,
add the lock to its owning_list and become the current holder of
the lock.

Step 2: If the lock has a holder already. Then make the lock become current

thread's waiting lock. Check the priority of the lock (or the holder)
thread's waiting lock. Check the priority of the lock (or the holder)
If it's greater than that of the current thread, added to the waiting
list and being blocked.

Step 3: If the priority of current thread is greater. Then donate the
priority to the lock and its current holder.

Step 4 (nested donation): Check if the current holder has a waiting lock. If
so, donate the lock holder's current priority to the waiting lock and
its holder. Always check until the current lock holder doesn't wait
for another lock.

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

Step 1: Disable all interrupts.

Step 2: Remove the lock from the owing_list of the holder.

Step 3: From the owing_list of the holder, find the one lock with the highest
priority, make current thread's priority that one. If the owing_list
is empty just set the priority of the current thread to its original
one.

Step 4: Up the semaphore of the lock.

Step 5: Sort the waiting list of the lock and pop the front one. Unblock it
and add it to the ready_list. Yield the CPU and re-schedule because
a preempt race for the lock is allowed.

Step 6: Re-enable interrupts.

---- 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?

Thread_set_priority() can be called by any thread to set the priority of
a certain thread. Even though we directly assign the priority of threads
in our implementation, we can also call the thread_set_priority() during
donation or other priority setting behaviors.

A potential race in thread_set_priority() is that,assuming a thread A is
trying to set its property and a thread B with a higher priority cut in
the middle of the function thread_set_priority(). Thread B wants to take
a lock held by thread A and thus donates its priority to thread A. By
either directly set its priority or calling the function.

The priority of thread A is now being raced and thus when thread B returns
CPU back to thread A, A should have set a totally different priority
calculated before thread B kicks in.

Using a lock is not helpful at all, because it doesn't prevent another thread
taking over the CPU and cut into the middle of the function. Futher, racing
for a lock is one of the key causes here for the data race.

So the only way to prevent data race is to disable interrupts so that the function thread_set_priority() becomes atomic and cannot be cut into half.

---- RATIONALE ----

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

We choose this design because we have to always keep track of a lock_holders doner and donee if exists and reconstruct their relationship every time an action is taken towards the lock.

At fisrt we didn't attempted to modify the struct of the lock, rather, we decided to add two stacks to the the thread. One for recording the doner and the priority of each doner donated, the other for recording the donee and the priority donated to the donee. However, it becomes so hard to main the structure when situations become more and more complex.

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.

```
-------- thread.h ---------
int nice;                           /* The NICE value. */
fixed_point recent_cpu;             /* The recent usage length of cpu. */

-------- thread.c ---------
static  fixed_point load_avg;      /* The load_avg of working load. */
```

-------- fixed_point.h --------
This is the file uses Macro function to implement a fixed-point data structure based on int.

---- 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:

priority = PRI_MAX - (recent_cpu / 4) - (nice * 2).

recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice.

load_avg = (59/60)*load_avg + (1/60)*ready_threads.

"Round-Robin": i = (i + 1) mod n

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	B*
28	16	12	0	59	58	59	C*

32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	B*

>> 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?

First of all, the chart didn't show out the ready_list and didn't give any infomation about the initial load_avg, which inditaes the ready threads to calculate load_avg and futher the recent_cpu of each thread.

I then uses the actual runnning time to be the recent_cpu and assume the ready_list always has three threads.

In our implementation, we always calculate load_avg first, then steps by step to get the priority of the threads.

Secondly, when there are multiple threads holding the same priority, it's hard to decided which thread to run next. Thus, as described in the project description, I used the Round Robin schdeule to decide which to run next.

In my implementation, it's similar to Round-Robin. Every time a new priority is calculated, re-insert it into the ready_list. During the insertion, the newly calculated-thread would always be inserted after the threads with the same priority, thus, created the Round-Robin like schedulor.

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

The cost of calculation inside the interrupt context will definetely affect the performance or even the answers to the tests.While doing calculations inside the timer_interrupt, the time still ticks while it is not as recorded and thus will cause a synchronization problem.

---- 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?

In order to simplify the design, we didn't reperately implement the 64 queues and the Round Robin scheduling when facing the same priority in the ready_list.
Rather, we used two threads' properties (recent_cpu and nice) to take down the new changes and still use a single ready list, only by the feature of list_insert_ordered to implement the Round Robin Scheduling as mentioned as above.

If there were more time, I would like to implement the actual Round Robin Scheduling and shorten the extra codes that are not necessary.

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.

Yes, I have suggestions, but I have no time to complete the answers, I

will make an appointment with Pro. Xie or TA to give suggestions either
by eamil or via talk.

Thank you for reading.

>> 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?