Project 1: Threads

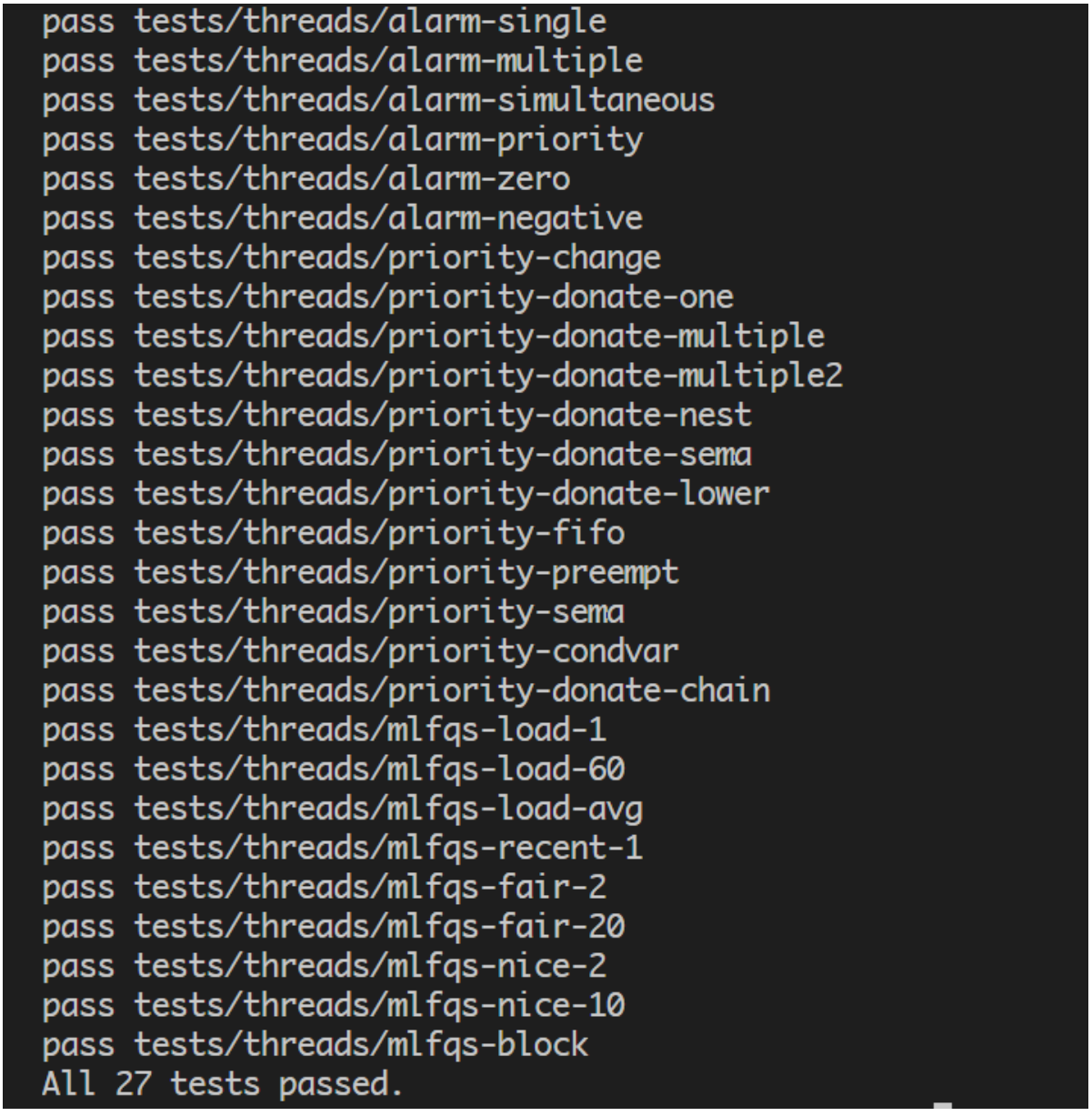
Preliminaries

> Fill in your name and email address.

Tong WU 2200013212@stu.pku.edu.cn

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

My local test result:



> 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_sleep {
 // Keep track of threads' time to sleep
 struct thread *t; // The ptr to the sleeping thread
 int64_t wakeup_time;   // The time(tick) to wake up the thread
 struct list_elem elem; // To use it in a PintOS list
};

static struct list thread_sleep_list; // List of thread_sleep
```

ALGORITHMS

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

In my design, first we should check if the ticks to sleep is not positive and disable interruption to avoid race conditions. Then we add the thread calling `timer_sleep()` to the global list of sleeping threads, recodring the time to wake it up. Here we use `list_insert_ordered()` to ensure the list is in order with respect to the wake up time. Finally, we call `thread_block()` to keep the thread away from the ready queue.

When the timer interrupts, the timer interrupt handler is called. It will first increase the system tick by 1 then check the sleeping list from front to end whether it is time to wake up sleeping threads(i.e. removing them from sleeping list and adding to the ready queue). Also, to enable preemption, `intr_yield_on_return()` is called if at least one thread is waken up.

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

we use `list_insert_ordered()` to ensure the list is in order with respect to the wake up time. As a result, in `timer_interrupt()` we no longer need to sort the list or check every thread's wake up time. Instead, whenever we encouter the first thread that still needs sleeping, we can directly 'break' from the checking loop.

SYNCHRONIZATION

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

When timer_sleep() is called, interrupt is disabled before being added to the list of sleeping threads.

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

When `timer_sleep()` is called, interrupt is disabled before being added to the list of sleeping threads.

RATIONALE

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

At first I considered simply adding a new thread to the sleeping list when it calls `sleep()`, and check all threads' wake up time when handling interruption of the timer. However the current implementation is more efficient, considering that we can only check a few threads in every interruption.

Also, not every thread will can `sleep()` , so I maintain the list of sleeping threads in `timer.c`, rather than adding new member variables to `struct thread`.

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.

I add `priority` and `elem` in `struct lock`.

```
/** Lock. */
struct lock
  {
    ... // Previous variables
    int priority;               /**< Priority of the lock, equal to the highest priority of the threads that are waiting for the lock */
    struct list_elem elem;      /**< For lock list. */
  };
```

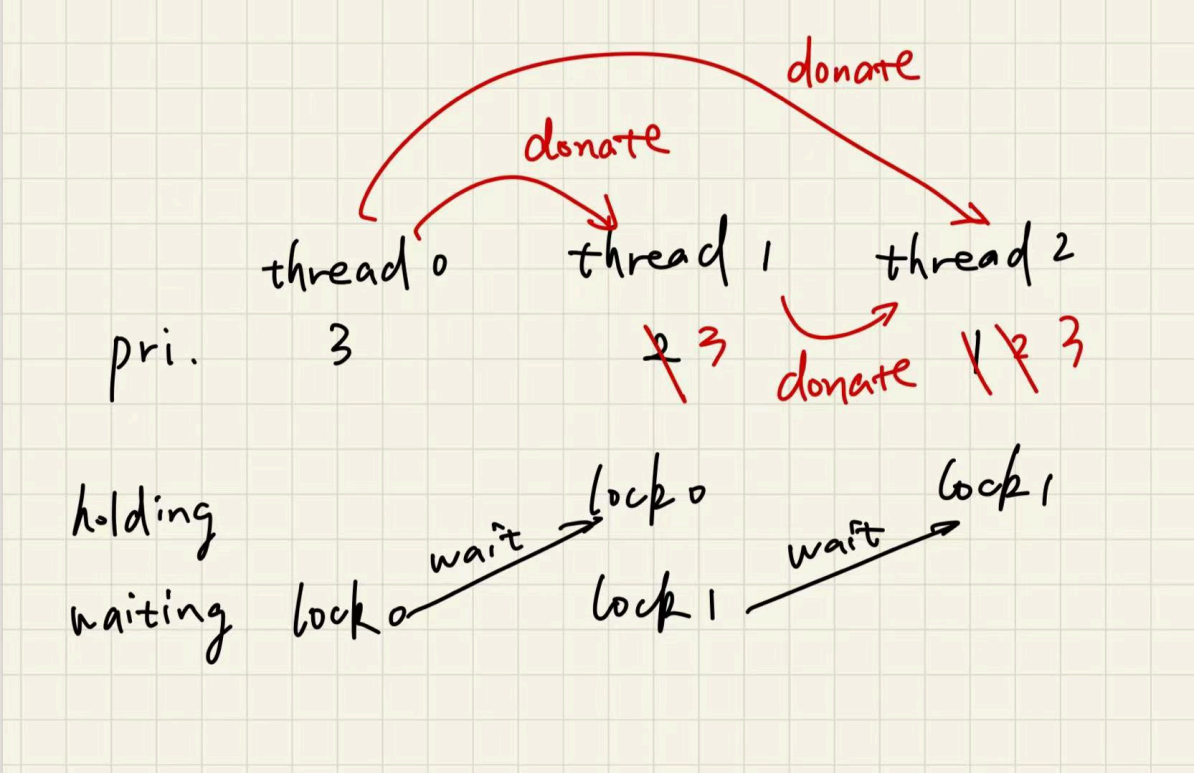
Also, I add these struct members to `struct thread`:

```
/* Utility for priority donation */
    int old_priority;                   /**< Original priority before donation */
    struct list holding_locks;     /**< Locks held by the thread. */
    struct lock *waiting_lock;          /**< Lock that the thread is waiting for. */
    // Every thread can only acquire 1 lock, but holding >1 locks.
```

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

Every thread can only acquire 1 lock, but can hold >1 locks. As a result, I use a waiting_lock member of each thread to keep track of the lock they're waiting for, and a list `holding_locks` to keep track of the lock they're holding. Also, old_priority is recorder, so that when a thread no more receives priority donation, its priority can be set back as before the donation.

A potential nested donation:



ALGORITHMS

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

- Semaphore: When `sema_up()` is called, I use `list_max()` to acquire the waiting thread with highest priority and unblock it first.
- Lock: In PintOS, lock is a specialization of a semaphore with an value <=1 and only one holder, so all we need to do is ensure `sema_up()` is correctly implemented.
- Cond. var: The idea is similar to semaphore. When `condvar_signal()` is called, I compare different semaphores based on their waiter's highest priority and wake up the most prioritized waiter first.

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

Suppose thread 1, 2, 3 has priority 1, 2, 3. Thread 3 calls is waiting for a lock that thread 1 is holding. Thread 3 call lock_acquire(), but thread 2 has higher priority than thread 1. So thread 3 donates its priority to thread 1 and now thread has a priority of 3, which is higher than thread 2, so that it can successfully release the lock and let thread 3 run.

Nested donation is handled in `lock_acquire()`. When a thread is trying to acquire a lock, it will potentially recursively donate priority. It may donate to the holder of the lock it is waiting for, and to the holder of another lock that the holder of the lock it is waiting for is waiting for, and so on.

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

`lock_release()` first resets the relevent information, such as the lock's holder and the thread's `holding_locks`. Then it resets the current thread's priority to that before the lock is acquired by the current thread. Finally, `thread_yield()` is called to schedule the waiting thread with higher priority.

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?

Suppose a thread is lowering its priority, suddenly it's interrupted by the timer, and another thread is scheduled to run, which donates priority to the thread. However, the previous thread has decided to lower its priority, making the priority donation worthless.

I think that the code that Isubmitted did not avoid the potential race. I may disable the interruption during the function to avoid this race.

RATIONALE

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

Actually I only figure out the current design :(

I think one benefit of the current design is that the holder member of a lock and holding_locks and waiting_lock members of a thread enable us to conveniently recursively find the relationship between threads and locks, so that we can modify their information and priority without trouble.

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 `fixed-point.h`

```
typedef int fixed_point;
static int f = 16384; // 2^14
```

In `thread.c`

```
/** Global load_avg for mlfqs */
static fixed_point load_avg;
```

Adding new members to thread struct:

```
/* Values for mlqfs */
    int nice;                           /**< Nice value for mlfqs. */
    fixed_point recent_cpu;             /**< Recent CPU for mlfqs. */
```

ALGORITHMS

> C2: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

1. Updating `recent_cpu` every tick in the timer handler is executed only once and takes little time.

2. Updating `recent_cpu` and `load_avg` takes little time, too, because they are updated in this way once in a second.
3. Updating `priority` every 4 ticks is time-consuming, as we have to go through every thread.

RATIONALE

> C3: 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:
  - A simply implementation without multi-level queues is easy and straightforward.
- Disadvantages:
  - The implementation is not efficient enough, because I have to go through all threads whenever I decide the next thread to run.

I may improve by defining 64 queues with different priority level so that I can efficiently find the next thread with highest priority to run.

> C4: 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 use `typedef` to alias int32 as fixed_point so that it is efficient and explicit enough.

If I create an abstraction layer, it may be more explicit and provide better interface, whiling sacrificing some efficiency.