Project 1: Threads

Preliminaries

> Fill in your name and email address.

Jiaqi Si 2000012957@stu.pku.edu.cn

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

Does the TA have any materials to learn or any ways to search for some new operating system design ideas, like capacity-based security I learned in the ginkgo book(the OS book by Haibo Chen).

By the way, is TA really fat? It doesn't look like that.

But I am really fat, do you find it? QAQ

> 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
  {
    ...
    int64_t wakeup_time; /**< wake up time */
    ...
  };
```

record a wake-up time for a sleeping thread

```
static struct list sleep_list;
```

record sleeping threads with **THREAD_BLOCKED** state sorted by wake-up time from early to late.

ALGORITHMS

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

In timer_sleep:

1.calculate the wakeup time

2.insert the thread into sleep_list according its wakeup time **(thread.c/thread_insert_sleep)**

3.block the thread and then switch to another thread **(thread.c/thread_block)**

In the timer interrupt handler:

1.wake up the threads from sleeping (*too cruel* !!!)whose wake-up time is earlier than current ticks. **(thread.c/thread_wakeup)**

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

the threads in **sleep_list** are sorted

so if we find it is not the time for the front thread in **sleep_list** to wake up, the checking loop can terminate.

SYNCHRONIZATION

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

turn off the interrupt

because we must visit sleep_list in **time_handler**

The more specific reasons are provided in A5.

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

turn off the interrupt

this is the only way we can modify the thing used by some functions related to thread schedule

It is a **clear** way to keep some functions related to timer interrupts free.

This methods are used widely by pintos code base **itself**.

For example,

- thread.c/init_thread
- thread.c/thread_yield
- thread.c/thread_unblock
- synch.c/sema_down

And all the operations of **ready_list** are protected by interrupt off.

Some vital functions must run under interrupt disabled context, e.g. **thread.c/schedule**

In this design, the **sleep_list** has same status as **ready_list**. It is **hard** to discard intr_disable() function

If TA still want to give me penalty because of too many interrupt-offs, I have no words.

If you really do so, please introduce some **materials and codes** about **efficient and elegant kernel process scheduler design without turning off interrupts** to me.

RATIONALE

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

In the very first version, the definition of **sleep_list** is placed in timer.c

But it brings inconvenience to threads management.

Finally the **sleep_list** is set in threads.c with **ready_list** and **all_list** as a static variable.

All the operations on it are wrapped as interfaces in **thread.h** exposed to other files.

The related functions include

- **thread.c/thread_insert_sleep** (insert a thread to sleep list ordered by wakeup_time)
- **thread.c/wakeup_earlier** (auxiliary function for sleeping threads comparison)
- **thread.c/thread_wakeup** (called by timer_handler, wake up the threads should wake up before or at the current tick)

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.

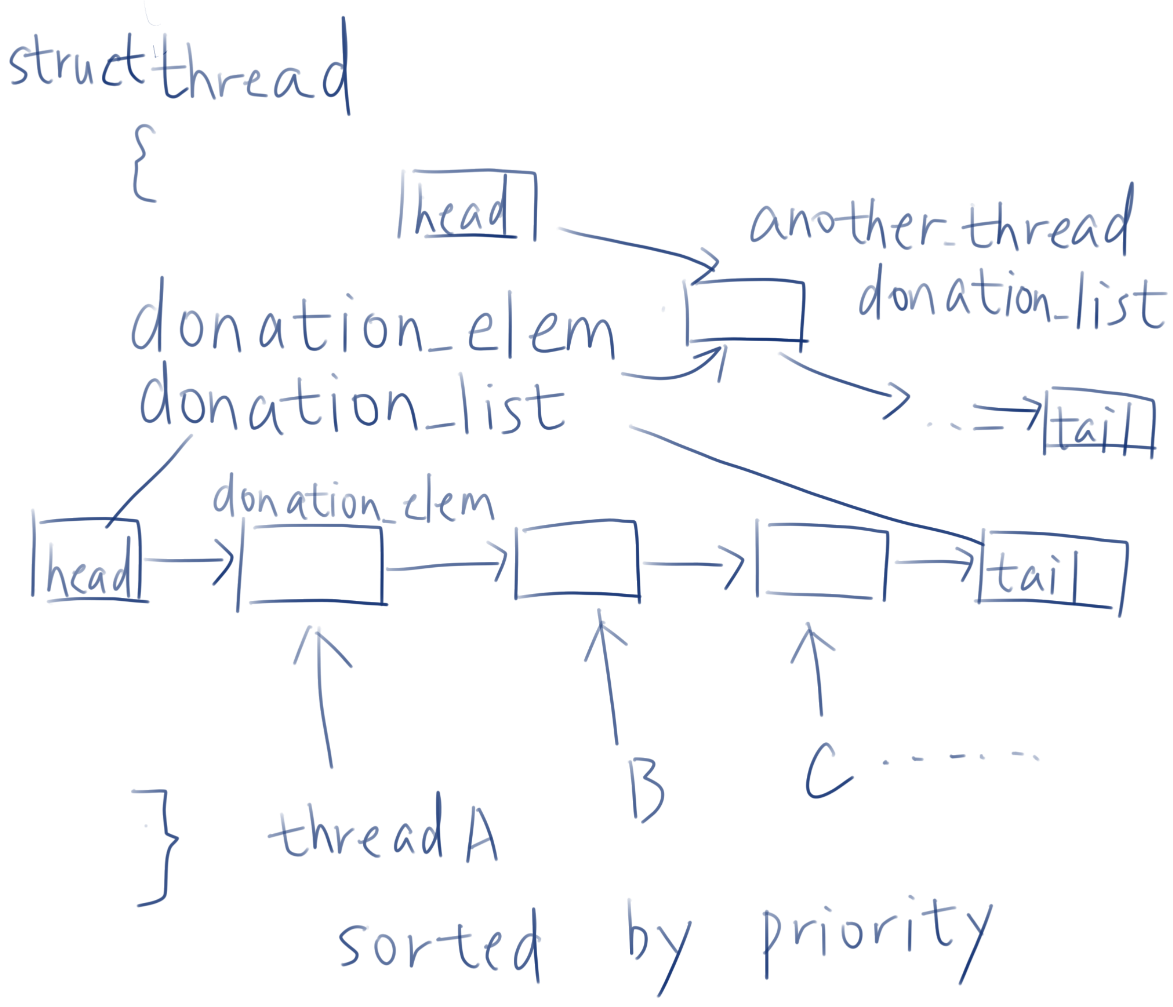
we give the each struct member's utility by comments

```
struct thread
{
   /* Owned by thread.c. */
   ...
   int priority;              /**< Priority. */
   int donation_state;        /**< record whether current priority is donated, 1
is donated, 0 is not*/
   int origin_priority;       /**< original priority of one thread,
                                    changed only in thread_set_priority and
initialization*/
   struct lock *wait_lock;    /**< a thread only waits one lock at a time*/

   struct list donation_list; /**< a list for donation_elem */
   struct list_elem donation_elem; /**< record the threads waiting this thread's
lock */
   ...
};
```

```
/** One semaphore in a list. */
struct semaphore_elem
  {
    ...
    struct thread *t; /**< Record the thread using this semaphore to wait a
condition variable*/
  };
```

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



We choose the same way like **all_list** and **ready_list** to link donators together.

The order of donator may be changed during the nested donation even if it is blocked. We can adjust its position by refinding its place in the donation list.

ALGORITHMS

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

modify the list insertion of **synch.c/sema_down** and **synch.c/cond_wait**

In **synch.c/sema_down**

```
void
sema_down (struct semaphore *sema)
{
  ...
  while (sema->value == 0)
    {
      list_insert_ordered (&sema->waiters, &thread_current()->elem,
priority_higher, NULL);
      ...
    }
  ...
}
```

The thread with the highest thread is inserted to the first of the waiters list

In **synch.c/cond_wait**

```
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ...
  list_insert_ordered (&cond->waiters, &waiter.elem, waiter_priority_higher,
NULL);
  ...
}
```

because the element is in the different sturcture, we can not use the comparing function which is used above

we define a similar function in synch.c to realize the same functionality for comparing elements inserted into cond's waiters list

```
bool waiter_priority_higher
(const struct list_elem *a,const struct list_elem *b, void *aux UNUSED) {
  struct semaphore_elem *sa = list_entry (a, struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry (b, struct semaphore_elem, elem);
  return sa->t->priority > sb->t->priority;
}
```

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

```
void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  bool success;
  struct thread *cur = thread_current(), *target;
  enum intr_level oldlevel;
  success = sema_try_down(&lock->semaphore); /* add a trying for whether to
execute a donation*/

  if (success) {
    lock->holder = cur;
    return;
  }
  else {
    target = lock->holder;
    cur->wait_lock = lock;

    oldlevel = intr_disable();
    thread_acquire_donation(target, cur, 0); /*the process are abstracted as a
interface in thread.c*/
    intr_set_level(oldlevel);

    sema_down(&lock->semaphore);
    lock->holder = cur;
  }
  cur->wait_lock = NULL;
}
```

To judge whether it needs to donate its priority, we modify the single **sema_try_down** and if it is unsuccessful then running donation code and **sema_down** to wait.

The donation function is provided by thread.c

```
void thread_acquire_donation(struct thread *dest, struct thread *src, int level)
{
  if(level >= NESTED_MAX) return;
  // the donation is added to the list whether it is larger than destination
priority
  list_insert_ordered(&dest->donation_list, &src->donation_elem,
priority_higher_donation, NULL);

  if (src->priority > dest->priority) {
    dest->priority = src->priority;
    dest->donation_state = 1;
    adjust_elem(dest);
    if(dest->wait_lock) {
      list_remove(&dest->donation_elem);
      thread_acquire_donation(dest->wait_lock->holder, dest, level + 1);
    }
  }
}
```

We used the level to avoid kernel running into endless loop due to dead lock. And it may cause some priority donation inconsistence.(A thread hold the old priority another thread had donated to it) But it is rare for 8-level donation occurence.

Whether the donation can increase destination thread or not, we save this donation in donation list. Because we may call **thread_set_priority()** to lower its own priority.

If a higher priority comes, the destination's priority is to be updated. The destination elem is in one of the **ready_list**, **sleep_list** or a **sema waiters**. We adjust its placement by reinsertion implemented by **thread.c/adjust_elem**.

If the destination is locked, we used a recursive way to update the lock holder.

The adjustment function is listed below

```
void adjust_elem(struct thread *t) {
  // this way can avoid judgment of which list we use
  struct list_elem *elem = list_prev(&t->elem);
  struct thread *at;
  list_remove(&t->elem);
  while(elem->prev != NULL) { // find a new place for this element
    at = list_entry(elem, struct thread, elem);
    if(at->priority >= t->priority) {
      break;
    }
    elem = list_prev(elem);
  }
  list_insert(elem->next, &t->elem);
}
```

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

```
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));
  enum intr_level oldlevel;

  oldlevel = intr_disable();
  thread_release_donation(lock); /* provided by thread.c */
  lock->holder = NULL;
  sema_up (&lock->semaphore);
  intr_set_level(oldlevel);
}
```

we just call the thread_release_donation to change current thread's priority first.

And the higher-priority is to release from prison by sema_up immediately. The thread_yield is called by current thread anyway due to a probably higher-priority thread is ready or the current thread's priority is decreased.

for the function **thread.c/thread_release_donation**

```
void thread_release_donation(struct lock* lock) {
  struct thread* cur = thread_current();
  struct list_elem* e;
  struct thread *t;
  // remove the donation
  for(e = list_begin(&cur->donation_list) ; e != list_end(&cur->donation_list) ;
e = list_next(e)) {
    t = list_entry(e, struct thread, donation_elem);
    if(t->wait_lock == lock) {
      list_remove(e);
    }
  }
  // update the priority
  if(list_empty(&cur->donation_list)) {
    cur->priority = cur->origin_priority;
    cur->donation_state = 0;
  }
  else {
    t = list_entry(list_begin(&cur->donation_list), struct thread,
donation_elem);
    if(t->priority > cur->origin_priority) {
      cur->priority = t->priority;
      cur->donation_state = 1;
    }
    else {
      cur->priority = cur->origin_priority;
      cur->donation_state = 0;
    }
  }
}
```

in this function, we traverse all the donation of this list and remove the donation due to this lock

Then we update the priority of the current thread.

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?

No, I can't do that due to the need of visiting the first element of **ready_list**, which is protected by interrupt off all the time.

It seems we can record the largest value globally and use a lock to protect it, but it will be a redundant object to my **elegant design**.

And if it is not the thread with the largest priority, it is to give up CPU immediately and turn off interrupt in **thread_yield**

Using a lock here seems meaningless

And here I repeat my opinion again. If TA has decided to reduce my scores due to this reason, please offer a **more elegant implementation** to me. I would be happy to see a succinct OS design with synchronization primitives, whether pintos or not.

RATIONALE

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

All the functions to manipulate thread is moved to **thread.c** and the interfaces are exposed in **thread.h**

The global list is static in **thread.c**

This is a really elegant design to divide ownership of different files. And it increases the readability a lot.

I adore my design idea and even think that it should be presented to others in OS class.(an ordinary but confident girl?)

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
struct thread
{
   ...
   /*MLFQ variable*/
   int niceness; /* Niceness between [-20,20] */
   fpreal_t recent_cpu; /* recently used cpu time */
   ...
};
```

```
// in thread.c
/** mlfq lists to record threads in this priority */
static struct list mlfq[PRI_MAX + 1];
/** caculate load average of the whole system */
static fpreal_t load_avg;
/** the number of threads in READY of RUNNING state */
static uint32_t ready_threads;
```

There are four new variables and **new multiple priority queues** we need to maintain.

niceness and **recent_cpu** belong to each thread

load_avg and **ready_threads** are global variables that describe the current state

ALGORITHMS

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

The update can be divided into two kinds.

The first class is that update the running thread (every four ticks, update **recent_cpu** and **priority**)

The second class is that update all the threads' **recent_cpu**

The first class is done in the **thread_tick** in interrupt context

The second class is done outside the interrupt context like thread_yield (the function we use is named as **thread_secondly_update**)

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?

The largest advantage of my design is that moving all the operation to global list in **thread.c**, which brings improvement to modularization.

But I have done bad work in multiple schedulers switch. Two schedulers are highly overlapping and distinguished by if state. It seems clumsy. What will happen if I want to append a new scheduler, such as **CFS**? It would be a disaster.

> 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 set the functions as inline function in a new created file **fpreal.h**.

It makes my code cleaner with self-explaining interfaces.