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

---- GROUP ----

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

Xiao Tang <xiaotang@ccs.neu.edu> <xiaotang>
Zhibo Liu <zhiboliu@ccs.neu.edu> <zhiboliu>
Yang Song <songy23@ccs.neu.edu> <songy23>

---- 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.

In thread.h: add wake_time to thread struct and add a new thread status.

int64_t wake_time: It's a new struct member in struct thread. This is to remember when we should wake up the thread.
THREAD_SLEEPING: we add another thread status THREAD_SLEEPING, which is waiting to be waked up.

---- ALGORITHMS ----

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

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  ASSERT (intr_get_level () == INTR_ON);
  if (ticks <= 0)
    return;

  enum intr_level old_level = intr_disable ();
  thread_sleep(ticks);
  intr_set_level (old_level);
}
```

When we call timer_sleep(), we pass a parameter TICKS timer ticks. If the parameter ticks is smaller then 0, we do nothing. Otherwise, we call the intr_disable () to

disable the interrupt and store the original interrupt state, then we call thread_sleep() to make the thread sleep for ticks time. When the ticks elapsed we call intr_set_level to restore the original interrupt state.

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

We plan to only check the threads in the sleeping_list to minimize the time with the following function.

```
while (e != list_end (&sleeping_list)) {
  struct thread *t = list_entry (e, struct thread, allelem);

  if (cur_ticks >= t->wake_time) {
    list_insert_ordered (&ready_list, &t->elem,
      (list_less_func *) &thread_compare_priority, NULL);
    t->status = THREAD_READY;
    temp = e;
    e = list_next(e);
    list_remove(temp);
  } else {
    e = list_next(e);
  }
}
```

We know only checking the threads in the sleeping_list will save the time the most. But there are exceptions when we test codes. So we have to switch to check all threads with no time saved.

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  /*add function to check each thread*/
  thread_foreach (sleeping_thread_check, NULL);
}
```

Our method is add a function thread_foreach (sleeping_thread_check, NULL) in the timer interrupt handler. We use function sleeping_thread_check to check each thread if their wake_time has elapsed. Well, the best solution is just to check the sleeping thread rather than all threads, we have tried but fail, we will try in the rest time

---- SYNCHRONIZATION ----

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

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  ASSERT (intr_get_level () == INTR_ON);
  if (ticks <= 0)
    return;
```

```
  enum intr_level old_level = intr_disable ();
  thread_sleep(ticks);
  intr_set_level (old_level);
}
```

We call intr_disable () to disable the timer interrupt and store the original interrupt state. So other threads cannot call timer_sleep() until the thread which got the lock finishes sleeping. Then use function intr_set_level() to restore the original interrupt state.

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

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  ASSERT (intr_get_level () == INTR_ON);
  if (ticks <= 0)
    return;

  enum intr_level old_level = intr_disable ();
  thread_sleep(ticks);
  intr_set_level (old_level);
}
```

In order to avoid the timer interrupt, we use intr_disable () before we call thread_sleep(). So the interrupt has no effect on the thread which got the lock. After this thread finished function thread_sleep(), we recover the original timer interrupt state.

---- RATIONALE ----

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

This design solves the concurrent exception well. And the code is simple. We use lock to implement the synchronization. We use sleep function to solve the busy waiting so that it saves the CPU source. Another design we have tried is using block list to put the sleeping threads, but it's not a good way to mix the blocked state with sleeping state. So we add another Thread Status, THREAD_SLEEPING.

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 synch.h:
Change struct lock. Add two variable: an integer priority and a list_elem elem. Priority represents the max priority among all the holder and waiters of this lock, and it is used for updating priority. Elem is for some thread who holds multiple locks and needs to store those locks in a lock list.

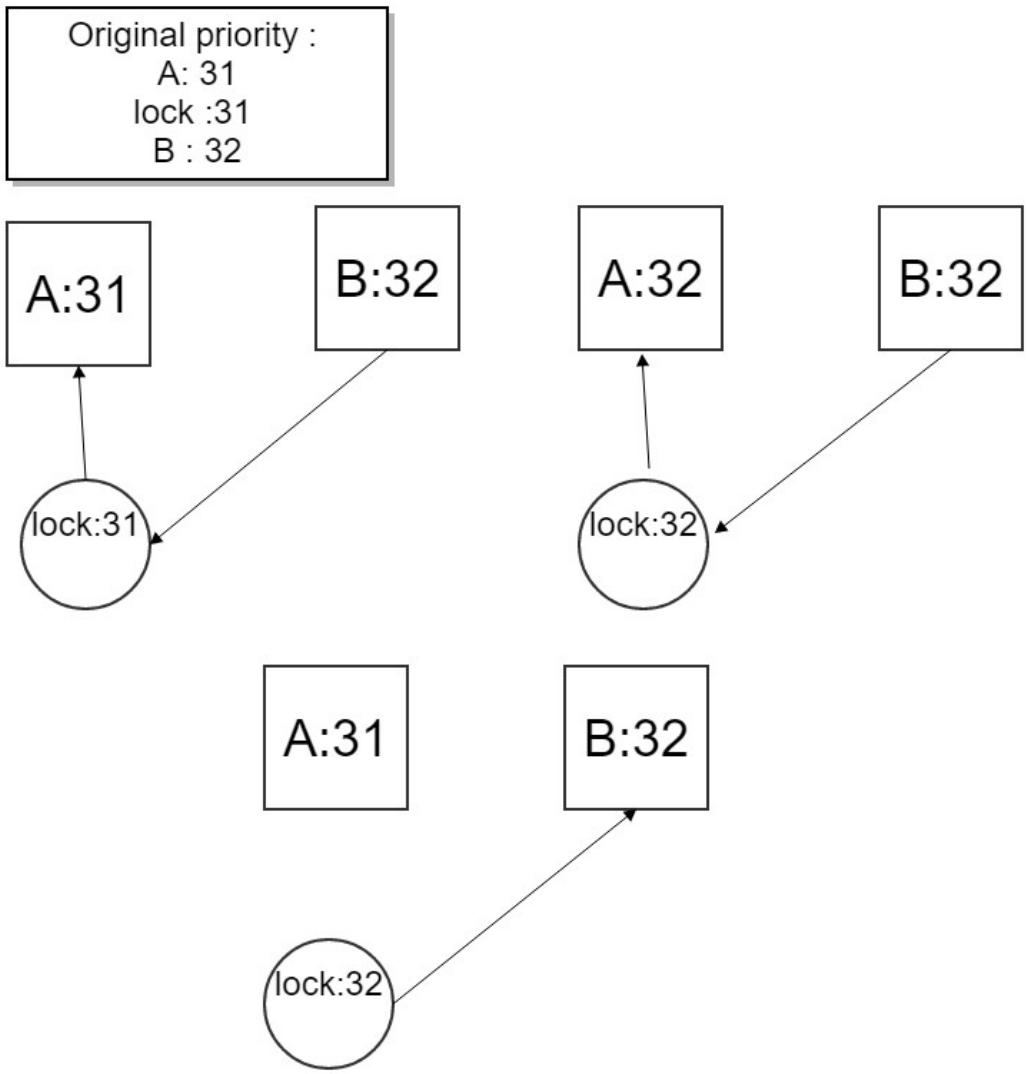
```
int priority;         /* Max priority among the holder and waiters of this lock */
```

```
struct list_elem elem;      /* List element. */
```

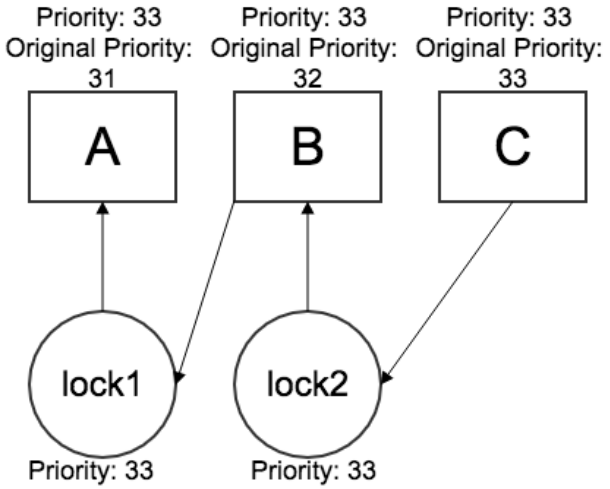
In thread.h:
Change struct thread, add an integer original_priority, a list struct locks and a point to lock lock_waiting. Original_priority is helpful when one thread's priority is donated by others and later restore to its original priority; locks is the list of locks that this thread holds; lock_waiting is the lock that this thread wants to acquire but currently held by others.

```
int original_priority;          /* Original priority (for donation purpose) */
struct list locks;              /* Locks that are currently held by this  thread */
struct lock *lock_waiting;      /* Lock that this thread is waiting for */
```

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



When B asks for A's lock, B's priority has been donated to the lock. Lock's priority becomes 32. So does A. when A releases the lock, A's priority goes back to original 31. And now B has the lock.



This diagram shows a condition when nested donation happens. Thread A has priority 31 and holds a lock1. Thread B has priority 32 and holds lock2 and it wants to acquire lock1. Thread C has priority 33 and wants to acquire lock2. When B tries to acquire lock1, it will be blocked and donates its priority (32) to A, and updates lock1's priority to 32. Then when C tries to acquire lock2 and is blocked, and C donates its priority (33) to B, and updates lock2's priority to 33. Then we will iteratively check if B waits for any other lock, and then updates the holder's priority that B is waiting for, which is A in this case. The graph shows the priority of the 3 threads and 2 locks after donation.

---- ALGORITHMS ----

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

We use the priority queue to fulfill the thread order in the list. Therefore, each time we need to wake up a thread, we will pick the thread with highest priority (which is the front element in the list).

Functions for compare priority:

```
/* Compare the priority of two input threads. If the priority of thread a is
higher, return true, otherwise return false. */
bool
thread_compare_priority (const struct list_elem *a,
                         const struct list_elem *b, void *aus UNUSED)
{
  return list_entry(a, struct thread, elem)->priority >
         list_entry(b, struct thread, elem)->priority;
}

/* Compare the priority of two waiters in a conditional variable. */
static bool
semaphore_compare_priority (const struct list_elem *a,
                            const struct list_elem *b, void *aus UNUSED)
{
```

```
  struct semaphore_elem *sema_a = list_entry(a, struct semaphore_elem, elem);
  struct semaphore_elem *sema_b = list_entry(b, struct semaphore_elem, elem);
  struct thread *waiter_a = list_entry (
                            list_front (&sema_a ->semaphore.waiters),
                            struct thread, elem);
  struct thread *waiter_b = list_entry (
                            list_front (&sema_b ->semaphore.waiters),
                            struct thread, elem);
  return waiter_a ->priority > waiter_b ->priority;
}
```

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

If the lock is not held by any thread (i.e lock->holder == NULL), we call sema_down on the semaphore of this lock, and then set the holder of this lock to current thread, set the priority of this lock to current thread's priority, set the lock_waiting of current thread to NULL, and then insert this lock to the lock list that current thread holds. The lock list is a priority queue, based on the priority of each lock. We are using priority queue here because if a thread holds multiple locks and are donated multiple times, when it releases the lock with highest priority, its priority should be set to the second highest priority in its lock list, and a priority queue can help us to find the next lock that triggers this donation. We add a new function lock_compare_priority here:

```
/* Compare the max priority of two locks. */
static bool
lock_compare_priority (const struct list_elem *a,
                       const struct list_elem *b, void *aus UNUSED)
{
  struct lock *lock_a = list_entry(a, struct lock, elem);
  struct lock *lock_b = list_entry(b, struct lock, elem);
  return lock_a ->priority > lock_b ->priority;
}
```

If the lock is held by other thread (i.e lock->holder != NULL), we first set lock_waiting of current thread to this lock, and here we need to deal with nested donation (e.g, thread A wants to acquire lock-1 held by thread B, while thread B is waiting for lock-2 held by thread C, and priority of A > B > C, then both B and C's priority should be promoted to A's priority). To do that we start from this lock, find the holder of this lock and if current thread has a higher priority, donate priority to the holder; and if the holder of this lock is waiting for other lock, find the holder of that lock and donate priority if current has a higher priority, and so on and so forth, iteratively promote the priority of all related threads and locks:

```
Changes made in lock_acquire:
if (lock->holder != NULL) {
    cur->lock_waiting = lock;
    // multi-level priority donate (priority-donate-nest)
    for (temp = lock;
         temp != NULL && cur->priority > temp->priority;
         temp = temp->holder->lock_waiting)
    {
      thread_donate_priority(temp->holder);
      temp->priority = cur->priority;
    }
  }
```

```
thread_donate_priority:
/* Donate priority of current thread to the given thread t */
void
thread_donate_priority (struct thread *t)
{
  enum intr_level old_level;
  struct thread *current_thread = thread_current ();

  old_level = intr_disable ();
  if (current_thread->priority > t->priority) {
    t->priority = current_thread->priority;
  }

  if (t->status == THREAD_READY) {
    // maintain max-heap property of the priority queue ready_list
    list_remove (&t->elem);
    list_insert_ordered (&ready_list, &t->elem,
                         thread_compare_priority, NULL);
  }

  intr_set_level (old_level);
}
```

Once this lock is released, current thread may go on and acquire the lock. The rest is the same as the first condition.

>> 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)
{
  enum intr_level old_level;
  struct thread *cur = thread_current ();

  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  old_level = intr_disable();
  lock->holder = NULL;
  list_remove(&lock->elem);  // remove current lock's elem, return the element that
follows elem

  // update the priority of current thread, depends on whether
  // there are other threads waiting for other locks that current
  // thread is holding.
  int updated_priority = cur->original_priority;
  if (!list_empty (&cur->locks)) {
    // locks are sorted in descending order based on their priority
    struct lock *lock_with_max_priority =
            list_entry (list_front (&cur->locks), struct lock, elem);
    if (lock_with_max_priority->priority > updated_priority) {
      updated_priority = lock_with_max_priority->priority;
    }
  }
  cur->priority = updated_priority;
  sema_up (&lock->semaphore);
```

```
  intr_set_level(old_level);
}
```

We first set updated_priority to current thread's priority. If there is a waiting thread whose priority is high than the current thread's priority, we set the updated_priority to waiting thread's higher priority. Now current thread's priority is the updated_priority, which is the higher one in the waiting list. As a result, the thread with a higher priority will get the lock next.

---- 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 potential race condition in thread_set_priority() is when multiple calls of thread_set_priority() to one thread at the same time, some changes of previous thread_set_priority() may be overriden by latter calls.

Yes we use a lock (intr_disable ()) in thread_set_priority to prevent such a race condition, so that each set priority operation could be granularity.

thread_set_priority:

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  old_level = intr_disable ();
  cur->original_priority = new_priority;
  if (list_empty (&cur->locks) || cur->priority < new_priority) {
    cur->priority = new_priority;
    thread_yield();
  }
  intr_set_level (old_level);
}
```

---- RATIONALE ----

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

In our design, we store the original priority, currently holding locks and waiting lock in thread struct, and store the max priority as well as list elem in lock struct. In this way, we track the priority of both threads and locks, so when nested donations happen, we are able to trace back to each thread involved in the nested condition. Also, when one thread holds several locks and are donated by multiple threads, and then release one lock, we can update the priority of that thread by looking at the lock list that it holds and check the priority of those locks. Moreover, we decide to keep the waiter list in semaphores and conditional variables as priority queues, so every time when we need to unblock the thread with the highest priority, we can just look at the element at the front of the list.

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?

The nested priority donation part is hard, and in order to understand its logic, we spend a lot of time reading and analyzing the test cases. The whole project 1 takes us around 50 hours.

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

The schedule function in thread.c gives us an insight on how CPU schedules threads based on their priority (flip around current thread and the topmost thead in the priority queue).

>> 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?

I think it might be helpful to first go through all the test cases in tests/thread. That helps us a lot to have a better understanding on the logic of priority donation.

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

>> Any other comments?