---- TEAM ----

>> Team name.

Team 19

>> Fill in the names, email addresses and contributions of your team members.

강현우(20150031) <kanglib@kaist.ac.kr> (70)

조영걸(20150717) <rangewing@kasit.ac.kr> (30)

>> Specify how many tokens your team will use.

0

---- PRELIMINARIES ----

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

NONE

>> 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.

NONE

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.

src/devices/timer.c

```
/* List of waiting threads. */
static struct list thread_list;
```

- static struct list thread_list: a list of waiting threads when it is sleeping by timer_sleep(). It is used to check the remaining ticks to unblock.

src/threads/thread.c

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    /* Owned by devices/timer.c. */
    int64_t waiting_ticks;              /* Remaining ticks for waiting. */
    struct list_elem timer_elem;        /* List element of thread_list. */

    /* Owned by synch.c. */
    struct donation donation;
    struct list donation_list;
    int deferred_priority;

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

- int64_t waiting_ticks: the remaining waiting ticks of the thread. The thread will be unblocked after these ticks.

- struct list_elem timer_elem: the list_elem for thread_list, which is explained above.

The other added members will be explained in B1, since they are for synchronization.

---- ALGORITHMS ----

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

When we call timer_sleep(), then the timer should be waited for the entered ticks, without busy waiting. Therefore, we has implemented this function by using the list that stores the waiting threads. When timer_sleep() is called, it disables the interrupt first. Next, it sets *waiting_ticks* of the thread to the entered ticks and push it to *thread_list*. After that, it blocks the thread and enables the interrupt.

For every timer interrupt, the handler iterates *thread_list* and decreases *waiting_ticks* of each threads in it. When some *waiting_ticks* become zero, it unblocks the thread and removes it from the list. In this way, waiting threads can be unblocked after *waiting_ticks*.

```
/* Suspends execution for approximately TICKS timer ticks. */
void
timer_sleep (int64_t ticks)
{
  /* Do not use ASSERT, or the test crashes. */
  if (ticks > 0) {
    enum intr_level old_level = intr_disable();
    struct thread *current = thread_current();
    current->waiting_ticks = ticks;
    list_push_back(&thread_list, &current->timer_elem);
    thread_block();
    intr_set_level(old_level);
  }
}
```

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  struct list_elem *e;

  for (e = list_begin(&thread_list); e != list_end(&thread_list);
      e = list_next(e)) {
    struct thread *t = list_entry(e, struct thread, timer_elem);
    if (--t->waiting_ticks == 0) {
      thread_unblock(t);

      /* list_remove(e) works just fine but is not ideal. */
      e = list_remove(e)->prev;
    }
  }

  ticks++;
  thread_tick ();
}
```

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

The timer interrupt handler looks for the waiting list (*thread_list*), and it decreases *waiting_ticks* and unblocks the thread if the tick becomes zero for each threads in the list. The time complexity of these steps is O(n), which is linear. But it is faster than using complex data structures because of its simplicity.

---- SYNCHRONIZATION ----

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

timer_sleep() adds the thread that called it to the waiting list (*thread_list*) individually, and block it. Race conditions are avoided by turning the interrupt off.

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

The timer interrupt handler checks for every entry of the waiting list and updates the ticks or unblocks some threads. If the interrupt has occurred just before adding to the waiting list, then the waiting ticks of the other waiting threads are decreased, but it does not affect the current sleeping thread. If it has done just before blocking the thread, it also does not affect but its sleeping ticks will be decreased by one.
Therefore, the race conditions are avoided. There can be some tick differences, but it does not matter because it does not need accurate waiting ticks.

---- RATIONALE ----

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

This design is faster and more efficient than calculating the remaining ticks every time. Most of all, its simplicity makes the errors reduced.

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.

src/threads/thread.h

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    /* Owned by devices/timer.c. */
    int64_t waiting_ticks;              /* Remaining ticks for waiting. */
    struct list_elem timer_elem;        /* List element of thread_list. */

    /* Owned by synch.c. */
    struct donation donation;
    struct list donation_list;
    int deferred_priority;

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

- struct donation donation: a structure for priority donation. It is explained below about this structure.
- struct list donation_list: a list that stores the information of the donor threads, which has been donated their priority to the thread.
- int deferred_prioirty: temporary priority storage when lowering it in the priority donation.

```
/* Priority donation information.
   Used in thread structures. */
struct donation {
  struct lock *lock;                  /* Lock caused priority inversion. */
  struct thread *donor;               /* Donor thread. */
  struct thread *donee;               /* Donee thread. */
  int hp;                            /* Higher (donated) priority. */
  int lp;                            /* Lower (original) priority. */
  struct list_elem elem;              /* List element of donation_list. */
};
```

- This is a structure for priority donation. It stores the information of the donation, including that of the lock, associated threads, and priorities.

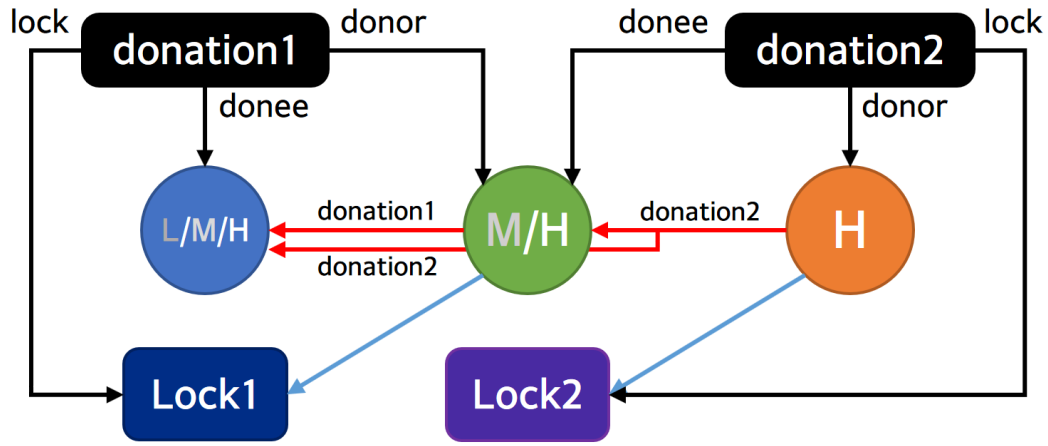
src/threads/synch.c

```
/* One semaphore in a list. */
struct semaphore_elem
  {
    struct list_elem elem;              /* List element. */
    struct semaphore semaphore;         /* This semaphore. */
    struct thread *holder;              /* Thread holding semaphore. */
  };
```

- struct thread *holder: this indicates which thread is holding the semaphore.

>> B2: Explain the data structure used to track priority donation. Draw a diagram in a case of nested donation.

We used a structure named donation to track priority donation. It stores the information of the donation, including that of the lock, associated threads, and priorities.




```
/* Priority donation information.
   Used in thread structures. */
struct donation {
  struct lock *lock;                  /* Lock caused priority inversion. */
  struct thread *donor;               /* Donor thread. */
  struct thread *donee;               /* Donee thread. */
  int hp;                            /* Higher (donated) priority. */
  int lp;                            /* Lower (original) priority. */
  struct list_elem elem;              /* List element of donation_list. */
};
```

---- ALGORITHMS ----

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

A lock, semaphore, and condition variable all use semaphore as its internal operation. Therefore, we can ensure that every synchronization can be modified with the edition of semaphore.

In sema_up(), it unblocks the thread that has the highest priority in the waiting lists using *thread_get_highest()* function, and yields if its priority is higher than that of the current thread.

The function finds the thread that has the highest priority in *ready_list* and pops it.

When a thread is yielded, it calls schedule(), which chooses the next thread to run and switches to it. The chosen thread here is ensured to have the highest priority, by *thread_get_highest()* function. Therefore, we can ensure the highest priority thread wakes up first.

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;
  struct thread *t = NULL;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty(&sema->waiters)) {
    t = thread_get_highest(&sema->waiters);
    thread_unblock(t);
  }
  sema->value++;
  intr_set_level (old_level);
  if (t && t->priority > thread_get_priority())
    thread_yield();
}
```

```
static void
schedule (void)
{
  struct thread *curr = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (curr->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (curr != next)     prev = switch_threads (curr, next);
  schedule_tail (prev);
}

static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))     return idle_thread;
  else     return thread_get_highest(&ready_list);
}
```

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

Let "donor" be the thread that donates its priority, and "donee" be the thread that gets the priority of the donor. Then the donor will be the current thread, and the donee will be the holder of the lock.

First it stores higher priority (which is of the donor) and determines whether the priority donation is needed, by checking the existence of the holder of the lock, and the priorities of two threads. If needed, it initiates the donation structure and push back to the donation list of donee.

Next it searches the nested threads to handle the nested donation. While the donation has the lock, it finds the donation of its donee and donates the priority of the donor. By these steps, the nested donation can be handled.

After the semaphore's down, it restores the priority of the donee.

```
void
lock_acquire (struct lock *lock)

{
  struct thread *donor;
  struct thread *donee;
  struct donation *d;
  int hp;
  int lp;

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

  donor = thread_current();
  donee = lock->holder;
  d = &donor->donation;
  hp = thread_get_priority();

  if (donee && donee->priority < hp) {
    struct donation *cursor;

    lp = donee->priority;
    donee->priority = hp;

    d->lock = lock;
    d->donor = donor;
    d->donee = donee;
    d->hp = hp;
    d->lp = lp;
    list_push_front(&donee->donation_list, &d->elem);

    for (cursor = &donee->donation; cursor->lock;
        cursor = &cursor->donee->donation) {
      if (cursor->donee->priority >= hp)
        break;
      cursor->donee->priority = hp;
    }
  }
```

```
  sema_down (&lock->semaphore);
  lock->holder = thread_current ();

  if (d->lock) {
    list_remove(&d->elem);
    d->lock = NULL;
    if (list_empty(&d->donee->donation_list))
      d->donee->priority = d->donee->deferred_priority;
    else
      d->donee->priority = d->lp;
  }
}
```

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

When lock_release() is called in the situation, it calls sema_up() for the semaphore. Then it finds and unblocks the waiting threads that has the highest priority, using *thread_get_higest()* in the waiting list (*waiters*). After unblocking, it cause the thread to yield if its priority is lower than the unblocked thread.

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

  lock->holder = NULL;
  sema_up (&lock->semaphore);
  if (!list_empty(&thread_current()->donation_list))
    thread_yield();
}
```

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;
  struct thread *t = NULL;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty(&sema->waiters)) {
    t = thread_get_highest(&sema->waiters);
    thread_unblock(t);
  }
  sema->value++;
  intr_set_level (old_level);
  if (t && t->priority > thread_get_priority())
    thread_yield();
}
```

---- 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?

When setting the priority, the race condition can be occurred if some other threads or interrupt try to change priority while it is changing. We can avoid it by disabling interrupt while changing the priority.

We cannot use a lock because it might cause a page fault.

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  enum intr_level old_level;
  struct thread *t = thread_current();

  old_level = intr_disable();
  if (list_empty(&t->donation_list)) {
    int old_priority = t->priority;
    t->priority = new_priority;
    t->deferred_priority = new_priority;
    if (old_priority > new_priority)
      thread_yield();
  } else {
    t->deferred_priority = new_priority;
  }
  intr_set_level(old_level);
}
```

---- RATIONALE ----

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

Since the information of donation can be accessed by both threads, donor and donee, it can be the solution that uses global variables to store them. But this design is not good for entire system, because it is bad for locality, can make the list traversal long, as well as can cause a lot of errors.

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 two problems in it, too easy or too hard? Did it take too long or too little time?

These two assignments are hard. :(

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

A bit

>> 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?

NONE

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

NONE

>> Any other comments?

NONE