2016 Spring CS330 Project 1: THREADS

---- TEAM ----

>> Team name.

Team 10

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

Tuan Manh Lai <laituan245@kaist.ac.kr> (Contribution1 = 50%)
Nham Van Le <levn@kaist.ac.kr> (Contribution2 = 50%)

>> Specify how many tokens your team will use.

0 Tokens

---- 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 timer.c, we added a global variable **static struct list sleeping_list**. This is the list of sleeping threads.
- In thread.h, we added three variables to the struct *thread*:
  - **int64_t wakeup_time**. If a thread is sleeping, this variable indicates the tick value when the thread should be woken up.
  - **struct list_elem sleeping_elem**. This variable is needed when we want to add (remove) the thread into (from) the list of sleeping threads (*sleeping_list*).
  - **struct semaphore sleeping_sema**. This semaphore is used to block the thread if the thread is currently sleeping.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.
When *timer_sleep()* is called, we will first calculate its *wakeup_time* (the tick value when the thread should be woken up). After that, we will put the current thread into the *sleeping_list*. Then we actually block the thread by calling *sema_down()* on the sleeping semaphore of the thread.

One of the tasks of the timer interrupt handler is to find and wake up sleeping threads that should be woken up. To find such threads, we need to iterate through the *sleeping_list* and look for the threads that have the *wakeup_time* value less than or equal to the current ticks.

```
for (e = list_begin (&sleeping_list); e != list_end (&sleeping_list); e = list_next (e)) {
   struct thread *t = list_entry (e, struct thread, sleeping_elem);
   if (t->wakeup_time <= ticks) {
      .... // Wake up the thread t and remove it from the sleeping_list
   }
   else {
      ....
   }
}
```

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

As mentioned before, we have *sleeping_list*, which is a list of sleeping threads in the system. In addition, for each sleeping thread, we also have a variable named *wakeup_time* that keeps track of when the thread should be woken up. So what we do to minimize the amount of time spent in the timer interrupt handler is to keep the *sleeping_list* sorted with respect to the values of *wakeup_time* variables. The interrupt handler does not have to iterate through the entire *sleeping_list* at every timer interrupt. As soon as the handler finds a thread with *wakeup_time* larger than the current *ticks*, the handler can just stop iterating through the list.

---- SYNCHRONIZATION ----

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

We simply disable interrupts when we are about to put some thread into the *sleeping_list*. It is because the list is prone to race condition: when a thread A finds its position in the *sleeping_list* (after iterating through the list) but hasn't finished putting it to the list, and another thread B preempts thread A and messes up the list order, then the position thread A has just found may no longer valid. Instead of disabling interrupts, other synchronization primitives such as lock or semaphore could also be used to protect the *sleeping_list*. However, we decide to just disable the interrupts.

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

```
122  intr_disable();
123  list_insert_ordered(&sleeping_list, &t->sleeping_elem, wakeup_time_less , NULL);
124  intr_enable();
125  sema_down(&t->sleeping_sema);
```

First, we disable interrupts during the execution of the timer interrupt handler. This is because only one external interrupt may be processed at a time. Neither internal nor external interrupt may nest within an external interrupt handler. This also ensures that the execution of the handler is never interfered by any execution of *timer_sleep()* of some thread.

In addition, when a thread is executing the function *timer_sleep()*, if the thread is about to access some critical global data that the handler may also use, then the thread will disable interrupts before accessing the data in order to avoid race conditions. For example, since both *timer_sleep()* and the timer interrupt handler access the *sleeping_list* at some point, before the

function *timer_sleep()* calls the function *list_insert_ordered()* at line 123 (timer.c), it disables interrupts at line 122 and enables it again at line 124.

By the reasoning above, it seems that we should also execute the line 125 while the interrupts are disabled, because it is possible that the handler may 'up' the *sleeping_sema* of the current thread before the *timer_sleep()* 'downs' the *sleeping_sema*. However, this doesn't matter. Logically, whether the order is 'up' then 'down' or 'down' then 'up', the intended behavior will still be observed.

---- RATIONALE ----

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

Initially, we just let the list *sleeping_list* to be unsorted. We thought that adding a new element to a sorted list will cost O(n) in general. Also, as discussed before, in order to avoid race conditions, we need to disable interrupts while adding a new element to the *sleeping_list*. This means that the whole system has to wait for the addition to be complete. So we used to believe that keeping the *sleeping_list* sorted would result in a lot of overhead and there would not be much improvement.

However, usually, timer interrupts occur much more frequently than calls to the *timer_sleep()* function. As mentioned in the Pintos document, timer interrupts occur every 4 ticks. And so we decide to keep the *sleeping_list* sorted. Because this will make the function *timer_sleep()* runs slower but it will make the timer interrupt handler runs faster (as discussed in A3). And since the timer interrupts occur more frequently, the overall performance of the system will be improved.

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 thread.h, we added three new variables to the struct *thread* for this part:
  - **int assigned_priority**. This variable denotes the base priority of the thread, while the variable *priority* denotes the effective priority.
  - **struct lock *waiting_lock**. If the thread is waiting for some lock to be released, this pointer will point to the lock (it will point to NULL otherwise).

- **struct list acquired_locks**. This list keeps track of the locks that the thread has successfully acquired and is currently holding.

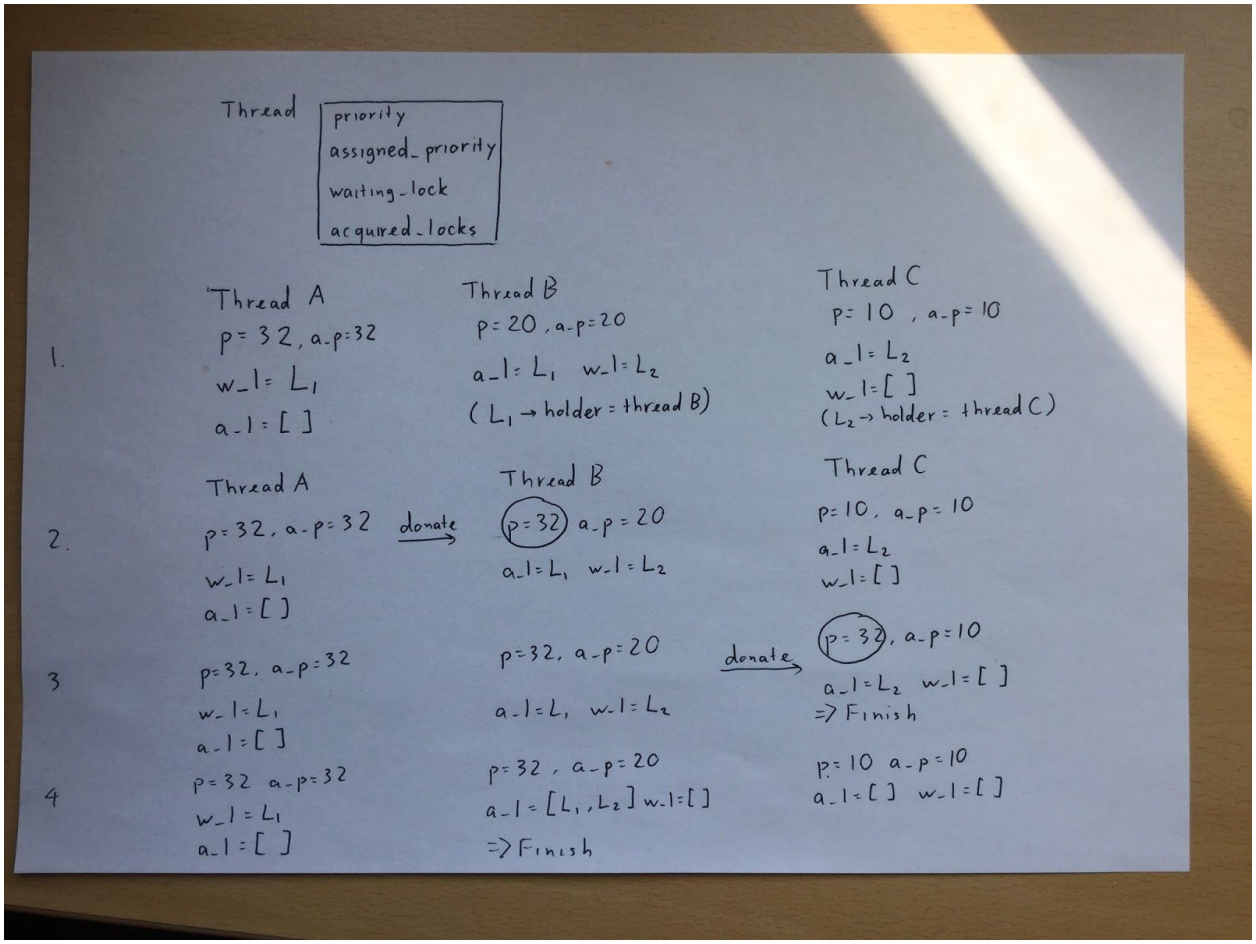
- In synch.h, we added a new variable **struct list_elem elem** to the struct *lock*. This variable is needed when we want to add (remove) the lock into (from) the list *acquired_locks* of some thread.

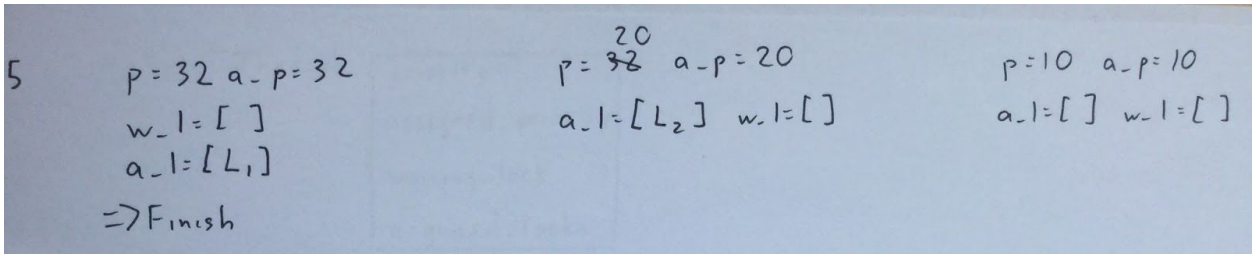
>> B2: Explain the data structure used to track priority donation. Draw a diagram in a case of nested donation.

In order to track priority donation, we need to add

- To the thread struct:
  - Int assigned_priority: to recover the priority of the thread after donation
  - List acquired_locks: to keep track of the locks that a thread has acquired successfully.
  - Lock waiting_lock: to keep track of the lock that a thread is currently waiting
- To the lock struct:
  - List_elem elem: to be able to push the lock into lists.

Diagram:





---- ALGORITHMS ----

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

Whenever some thread wants to 'up' some semaphore, we will first iterate through the list of all the threads waiting on the semaphore to find the thread with the highest priority. The thread is then removed from the list of waiters. It is also unblocked (i.e., its state will be changed from *THREAD_BLOCKED* state to *THREAD_READY* state). Note that if the thread's effective priority is larger than the current thread's effective priority, the *thread_yield()* function will be called at the end of the execution of the function *sema_up()*.

By implementing the function *sema_up()* this way, we also ensure that when a lock is released, the highest priority thread waiting for the lock (if any) will be woken up first. This is because a lock is just like a semaphore with an initial value of 1 and also the function *lock_release()* is implemented on top of the function *sema_up()*.

Finally, when some condition becomes true and the function *cond_signal()* is called, similar things will happen as in the case of upping a semaphore. For each condition variable, there is a list of semaphore elements associated with it. We first find the semaphore that has the thread with the highest priority waiting on it. We then remove the element from the semaphore list, and up it (this will indirectly wake up the highest priority thread).

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

1. If someone is holding the lock L:
    a. Set the waiting lock to L
    b. Donate priority
2. If the lock is available:
    a. Acquire the lock
    b. Set L->holder to the current thread
    c. Add the lock to the current thread's acquired_locks list
    d. Set the current thread's waiting_lock to NULL

*donation:

We recursively call the donate function with the parameters are the thread->waiting_lock->holder and the deeper level of donation.

```
if(t->priority > t->waiting_lock->holder->priority){
    t->waiting_lock->holder->priority = t->priority;
    donate(t->waiting_lock->holder, level+1);
  }
```

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

1. The pointer *holder* of the lock is updated to NULL
2. The current thread now removes the lock from its *acquired_locks* list.
3. The current thread also needs to recalculate its effective priority because its effective priority could have changed.
    a. The first step to recalculate the effective priority is to find the highest priority thread waiting for some lock that the current thread is holding. This is done by the function *get_max_donated_priority()*. Basically, the function consists of two nested loops. The outer loop is for iterating through the *acquired_locks* list of the current thread. And for each lock that is held by the current thread, the inner loop will iterate through the list of threads waiting for it to be released. Let *max_donated_priority* denotes the priority of the thread with the highest priority that we find in this step. Note that if there is no thread waiting for some lock to be released by the current thread, *max_donated_priority* will be set as -1.
    b. Let *assigned_priority* denotes the current thread's base priority
    c. Finally, the new effective priority of the current thread will be *max(max_donated_priority, assigned_priority)*

4. After the previous step, we then call the function *sema_up()* to actually release the lock. As discussed in B3, our implementation of the function *sema_up()* ensures that the highest priority thread waiting for the lock will be woken up first.

---- 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 would be with the *ready_list*. It could be changed during interrupt. Because the potential race condition could be caused by interrupt, we can't use locks since the interrupt handler cannot acquire locks.

To solve the problem, we turn off interrupts before working on the *ready_list*.

---- RATIONALE ----

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

Since both locks and condition variables are implemented on top of semaphores, we did consider the option of handling priority donation logic in the *sema_down()* and *sema_up()* functions. However, because we only need to implement priority donation for locks and need not to implement priority donation for other synchronization constructs, we decided to implement priority donation in the *lock_acquire()* and *lock_release()* functions. This would make our code somewhat more readable (by looking at our code in synch.c, it is immediately clear that we implement priority donation only for locks).

Also, initially, for each thread, we wanted to have a list of the locks that the thread is trying to acquire. However, we soon realized that such kind of list is not necessary since a thread cannot wait for more than one lock at the same time. That's why for each thread, we use only one pointer to track the lock that the thread wants to get (if there is any).