```
+--------------------+
|       CSE231       |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- GROUP ----

Pranjal Kaura   <pranjal17079@iiitd.ac.in>
Samiya Caur   <samiya17094@iiitd.ac.in >
Nishtha Singhal <nishtha17302@iiitd.ac.in>

---- 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, and lecture notes.
http://csl.skku.edu/uploads/SWE3004S15/project1.pdf
http://courses.cms.caltech.edu/cs124/lectures/CS124Lec12.pdf
https://www.ida.liu.se/~TDDB68/labs/TDDB68___Lab_2.pdf

---- TEST CASES ----

>> Provide a list of failed test cases here. If none of them
>> are failing just mention, all tests are passing.

All tests pass.

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**,
static struct list timer_blocked_list; /* List of all the sleeping threads */
This is a list of all the sleeping threads in our directory, arranged in ascending order(in order of their sleeping time).

In **struct thread**,
int64_t sleep_time; /* Time at which thread is supposed to wake up. */
Indicates the tick value of the thread when it's done sleeping and ready to get active.
struct list_elem sleep_elem; /* List element for timer_blocked_list. */

---- ALGORITHMS ----

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

Firstly in timer_sleep through **timer_ticks()**, the current ticks value is stored in the **'start'** var. Then the sum of **start** and **ticks**(the time duration for which the thread sleeps) is saved in the current thread's **sleep_time** attribute. Further, interrupts are disabled and a sorted insert of **sleep_elem** of current thread into the struct list **timer_blocked_list** is made. For this insertion **'sleep_time_order'** has been used, which is a comparator function in **thread.c**. The current thread is then blocked.

In **timer_interrupt_handler**, firstly if **thread_block_list** is empty, the front end of the sorted list **timer_blocked_list** is returned. Further the **sleep_time** attribute of this thread is compared to the ticks var. If it's greater than ticks, break command has been added to exit while loop as there is no sleeping thread to wake up. However, if it's less than the current ticks then the thread is unblocked and removed(popped) from the list. These steps are repeated until the list is empty, or all list entries having **sleep_time** greater than ticks have been unblocked.

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

timer_blocked_list struct has been used to store all the sleeping threads, but they have been arranged in ascending order. Just taking the front element from of this list will give us the youngest thread. Thus time is saved as complete list iteration has been avoided.

---- SYNCHRONIZATION ----

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

Race condition will be avoided because interrupts have been disabled. Hence any other arbitrary thread will not be able to pass an interrupt to preempt the execution of the current running thread.

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

Due to the same interrupt disabling, done through the following command

```
enum intr_level old_level=intr_disable();
```

---- RATIONALE ----

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

The current design was chosen to increase efficiency. Thus an ordered array of sleeping threads has been maintained which results in faster selection of required thread in the **interrupt handler**. Although inserting into this list is a slight overhead, but overall the efficiency increases.

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,**

int original_priority; /* Original priority of the thread which it can switch back to after releasing locks*/

/* Members for implementing priority scheduler. */
struct lock *waiting_for_lock; /* The lock which the thread is waiting to be released*/
struct list waiting_threads; /* List of all threads that are waiting for the thread to release the lock which it is currently holding, sorted by descending order of priorities */
struct list_elem wait_for_lock_elem; /* List element for waiting_threads */

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

We have implemented priority donation through the function **donate_priority(struct thread *th)**. This function uses the struct lock and list.
A lock **waiting_for_lock** keeps the reference to the lock that the thread wants to acquire and is waiting to be released.
For a given thread, the list **waiting_threads** keeps track of all the threads that want the given thread to release the lock it is holding. These threads are sorted in a descending order of their priorities. These are all the threads that can possibly donate priority to the given thread.

We have taken care of nested donations by making the **donate_priority(struct thread *th)** function recursive. We check whether th is waiting for a lock or not. If it is waiting for a lock (say L), then we check whether L->holder has lower priority than th. If so, then L->holder->priority is set to th->priority. We then call donate_priority(L->holder).

I)

```
                H            M            L
                          +------+     +------+
                High --->| Med | ---> | Low |
                          +------+     +------+
```

```
| K E Y
| Threads
| Priority
| +-------+
| | LOCK |
| +-------+
```

A high-priority thread (say H) waits for a lock held by a medium-priority thread (say M), which in turns wait for a lock held by a low-priority thread (say L).
H has priority = High, M has priority = Med, and L has priority = Low.

II)

```
            H           M             L
                     +-------+     +------+
            High --->| High | ---> | Low |
                     +-------+     +------+
```

Through **donate_priority()**’s first call, H donates its priority to the M. So now, H and M both have priority = High.

III)

```
            H           M              L
                     +-------+     +-------+
            High --->| High | ---> | High |
                     +-------+     +-------+
```

As M is waiting on lock held by L, we call **donate_priority(M)**. Through **donate_priority()**’s second recursive call, M donates its priority to the L. So now, H, M and L, all have priority = High.
As L is not waiting for any lock, we do not call **donate_priority()** again.

---- ALGORITHMS ----

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

In **sema->waiters**, list elements are always inserted in sorted manner according to decreasing priority by using the comparator **priority_ordering**. Hence, thread with the highest priority is the first element of **sema->waiters**. In **sema_up**, the thread that wakes up is the top element of **sema->waiters** ie, the highest priority thread.
The lock uses semaphore, hence waking up of highest priority thread first is implicit.
For condition variables, we sorted **cond->waiters** with the help of the comparator **condition_priority_ordering.** We sorted the **semaphore_elem** in **cond->waiters** according to the descending priority of the highest priority threads in **semaphore->waiters** list of semaphore stored in **semaphore_elem->semaphore**.

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

lock_acquire() causes a priority donation when the lock that the current thread wishes to acquire is held by a thread which has a lower priority than the current one.
When a call to **lock_acquire** is made, the thread tries to acquire the desired lock through **lock_try_acquire**. **cur->waiting_for_lock** is set to the lock for which we called **lock_acquire**. If it is successful, it returns true; the current thread is made the holder of the lock and **cur->waiting_for_lock** is set to NULL i.e. the current thread is no more waiting for any lock.
If **lock_try_acquire** fails, it means that the lock is acquired by another thread and the function returns false. Now, first we disable the interrupts to avoid any race conditions. Then, we call **sema_down** to wait for the **sema->value** to be positive and return after decrementing it. While waiting for **sema->value** to become positive, **sema_down** calls **donate_priority** for the current thread. In **donate_priority(struct thread *th)** we check whether **thread th** is waiting for a lock or not. If it is, we check if the **struct thread *holder=lock->holder** has lower priority than th. If it does, then we will set priority of **holder** equal

to priority of **th**. We then, recursively call **donate_priority** for the **holder** to check for nested donations.

>> **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, the priority of the current thread is same as the priority of higher-priority thread (say A) due to priority donation. We will disable the interrupts to avoid race conditions.
If there are some threads waiting for the lock for which **lock_release()** is called and we iterate through these threads. Since A is waiting for this lock, it will be part of the **waiting_threads** list of current thread (say cur).
All the threads in **waiting_threads** which are waiting for the same lock for which **release_lock()** was called are removed from the cur's **waiting_threads** list as these threads are no more dependent on the current thread to acquire lock.
A is also removed from **waiting_threads**. Hence, the priority of cur must be changed from the higher priority donated by A.
We do this by calling **recalculate_priority()**.
In **recalculate_priority()** if **waiting_threads** is not empty and the priority of the first element of **waiting_threads** is greater than **cur->original_priority**, **cur->priority** is set to priority of first element of **waiting_threads**. Otherwise, it is set to **cur->original_priority**. Then we return to **lock_release()**.
The **lock->holder** is now set to null, i.e. no thread holds the lock anymore and the **sema->value** is incremented through **sema_up()** which also wakes up one of the threads waiting for the locks. Finally, the interrupts are enabled again.

---- 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?

Let's say we have a thread A holds a lock L1. We call **thread_set_priority** for thread A and while we are executing **recalculate_priority**, another thread B

having higher priority preempts. B is waiting for lock L1 and hence, donates its priority. Now thread A will run again however it will set the value of its priority as per the value calculated before the priority donation. This will lead to incorrect priority.
Race condition will be avoided because interrupts have been disabled. Hence any other arbitrary thread will not be able to pass an interrupt to preempt the execution of the current running thread.
A lock cannot be used to avoid race condition as competing for this lock can lead to further race conditions.

---- RATIONALE ----

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

Initially instead of using struct list **waiting_threads**, we were planning to track all threads waiting on a thread to release a lock by using **lock->semaphore->waiters**. In this case, we would have to maintain a list of locks held by a thread. And every time, we call **recalculate_priority** for a thread, we would have to iterate through the list of locks held by the thread and then find the highest priority which a thread has in **lock->semaphore->waiters** of each lock. This would be O(n) where n is the number of locks held by the thread.
Instead we made it more efficient by maintaining list of all threads waiting for a thread to release locks. These threads are maintained in struct list **waiting_threads** in the decreasing order of their priority. Hence, **recalculate_priority** becomes more efficient as we now need to check only the priority of the first element of **waiting_thread**.
Earlier during **lock_acquire** and **lock_release**, we had not disabled interrupts. However, this lead to race conditions and our tests were failing. After disabling interrupts in these functions, all our tests passed.
The design used by us is intuitive and easy to implement. We were able to implement our design by using simple linked lists and pointers.