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

---- GROUP ----

Anand <anand17218@iiitd.ac.in>
Kyzyl Monteiro <kyzyl17296@iiitd.ac.in>
Aman Rehman <aman17278@iiitd.ac.in>

---- PRELIMINARIES ----

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

We learnt how to use list.c functions and list manipulation from internet

---- TEST CASES ----

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

All 18/18 tests cases are passing successfully.

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

```
struct list sleeping_threads;
```

A List in which we maintain a list of sleeping threads sorted according to their wakeup times.

```
-In timer_sleep (int64_t ticks)
        enum intr_level old_level;
                Here we store the previous interrupt state, and later restore it
        struct thread *cur;
```

```
                Stores the current running thread on which sleep is being invoked
-In timer_interrupt (struct intr_frame *args UNUSED)
        struct list_elem *front;
                To access the front element of the sleeping threads list
        struct thread *entry;
                Stores the front thread which is later unblocked
```

In thread.c & thread.h

```
int64_t sleepingtime;
        Maintains the sleeping time/ticks of each thread
bool sleeptime_comparator(struct list_elem *a, struct list_elem *b, void *aux);
        A comparator that compares the sleeping time of two thread : used for
sorting list of sleeping threads
```

---- ALGORITHMS ----

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

In the ***timer_sleep***,initially we check if the argument is valid ,that is, ticks is greater than 0, then the interrupts are disabled to prevent racing condition, and the current interrupt status is stored. Then the current thread's sleeptime is updated to the sum of the argument that is passed and the number of timer ticks since the OS booted.The thread is inserted in the sleeping threads sorted list according to its sleeptime in increasing order (using sleeptime_comparator).The thread is then blocked and the interrupt is restored to the old state which was stored.
The ***timer interrupt*** handler like before increases the ticks and calls the thread_tick() which keeps track of thread statistics and triggers the scheduler when a time slice expires. Additionally now it checks whether the thread with the least sleeptime has to be unblocked that is essentially check if the threads sleeptime is greater than the ticks passed since the OS has booted. If the sleeptime of the first thread of the sorted sleeping thread list is greater, then that thread is unblocked and removed from the sleeping thread list, and the next thread's sleeptime is checked for the same condition as earlier and if true they are unblocked and removed from the list till the condition is false for a thread.

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

To minimize the time spent in the timer interrupt handler the list of the sleeping threads is kept sorted, according to their sleeptime, in increasing order (by making a sleeptime_comaparator) which essentially means that the first element of the list has the minimum sleeping time and should be the first thread that has to be awakened, therefore in the timer interrupt, the full list won't have to be traversed and only the first element will have to be removed and unblocked.

---- SYNCHRONIZATION ----

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

Race Conditions are avoided by disabling interrupts and restoring the interrupt_level until the current task of blocking the current thread and adding it to the sleeping list is completed.

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

When a thread calls timer_sleep(), interrupts are disabled, so timer interrupts will be disabled as well. This prevents racing conditions.

---- RATIONALE ----

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

Our final iteration, this design was the best as compared to the earlier designs we thought of, as initially our plan was of storing the current thread's priority and replacing it with 0 and when it has to be awakened it was restored but this was faulty as this was only correct if one thread was asleep at a time, which is not the case. So, we thought of maintaining a list of the sleeping threads which was traversed at every timer interrupt call, but we saw that time spent in timer_interrupt could be minimized by keeping a sorted list of sleeping threads.

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

```
bool priority_comparator(struct list_elem *a, struct list_elem *b, void *aux)
```

A comparator that compares the priority of two thread : used for sorting lists according to the priority of threads

```
int basepriority;                   //thread's most recent priority
struct thread *locker_thread;       //lock I am waiting for is acquired by this thread
struct list donation_list;          //list of all donated priorities (to me)
struct lock *waiting_on_lock;       //lock on which I am waiting
struct list_elem donorelem;         //list element for donations_list
```

-In thread.c

(i) In **thread_create (const char *name, int priority,thread_func *function, void *aux)**
Added this snippet:

```
enum intr_level old_level = intr_disable ();
 struct thread *runningThread=thread_current();
 if(runningThread!=idle_thread){
      if(t->priority >runningThread->priority)
      thread_yield();
 }
 intr_set_level (old_level);
```

This snippet disables interrupt and checks if the priority of newly created thread's priority, it calls thread_yield() immediately before waiting for the current thread's ticks to finish so that the thread with maximum priority runs first.

(ii) In **thread_unblock (struct thread *t)**
Replaced simple list push back with this:

```
list_insert_ordered (&ready_list, &t->elem,(list_less_func *)
&priority_comparator,NULL);
```

To insert the thread in sorted order of priorities in the ready list so that the threads with higher priorities get the context before lower priority threads after they are unblocked

(iii) In **thread_yield (void)**

```
list_insert_ordered (&ready_list, &cur->elem,(list_less_func *) &priority_comparator,
NULL);
```

removed simple list push_back, now the yielded thread is inserted in sorted order according to priority

(iv) In **thread_set_priority (int new_priority)**
Added this snippet:

```
enum intr_level old_level;
old_level=intr_disable();
thread_current()->basepriority = new_priority;
if(list_empty(&thread_current()->donation_list) ||
   new_priority > thread_current()->priority)
  {
       thread_current()->priority = new_priority;
  }

  if(!list_empty(&ready_list))
  {
       struct thread *front = list_entry (list_front(&ready_list), struct thread,
elem);
       if(front->priority > thread_current ()->priority)
       thread_yield();
  }

  intr_set_level(old_level);
```

Disables the interrupts to avoid racing conditions , sets the most recent priority of thread to the argument passed. If the donor's list is empty or the new priority passed in argument is greater than current priority, then sets thread's current priority as new priority. If the new priority is less than thread's current priority, don't change thread's priority, instead change its most recent priority variable.
Finally, if the front element of the ready_list has larger priority than current thread's priority, yield to cpu.

-In synch.h

```
bool conditional_var_comparator(struct list_elem *a,struct list_elem *b,void *aux);
```

For the purpose of sorting waiting list of conditional variable

-In synch.c

(i) In **sema_down (struct semaphore *sema)**

```
list_insert_ordered (&sema->waiters, &thread_current ()->elem,(list_less_func *)
&priority_comparator,NULL);
```

Insert in sema waiters list, sorted according to priority so that highest priority thread is awakened first

(ii) In **sema_up (struct semaphore *sema)**

```
struct list_elem *father_element;
if (!list_empty (&sema->waiters)){
    list_sort(&sema->waiters,(list_less_func *) &priority_comparator,NULL);
father_element=list_pop_front(&sema->waiters);
thread_unblock(list_entry(father_element,struct thread,elem));
}
```

Sorts sema->waiters list to assure that they are in descending order so that thread with highest priority is unblocked first

(iii) In **lock_acquire (struct lock *lock)**

Added this snippet:

```
enum intr_level old_level;
old_level= intr_disable ();
if(lock->holder==NULL)
       thread_current()->locker_thread = NULL;
else
{
       thread_current()->locker_thread= lock->holder;
       list_push_front(&lock->holder->donation_list,&thread_current()->donorelem);
       struct thread *cur_thread= thread_current();
       cur_thread->waiting_on_lock= lock;
       while(cur_thread->locker_thread!=NULL)
       {
              if(cur_thread->priority >cur_thread->locker_thread->priority)
              {
                     cur_thread->locker_thread->priority=cur_thread->priority;
                     cur_thread= cur_thread->locker_thread;
              }
       }
  }
```

(iv) In **lock_release (struct lock *lock)**

Added this snippet:

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

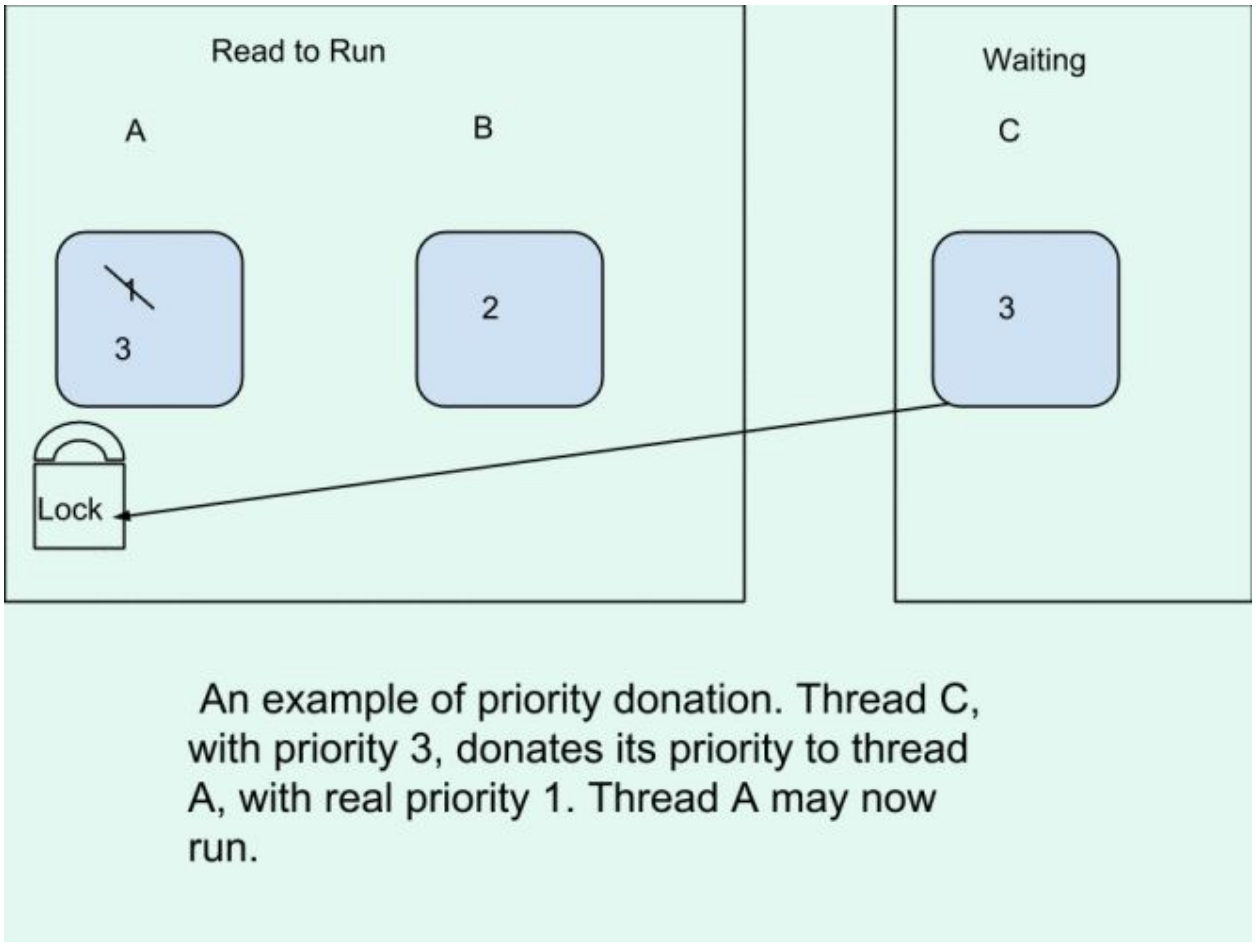
```
if(!list_empty(&thread_current()->donation_list)){
struct list_elem *iter;
for (iter = list_begin(&thread_current()->donation_list);
iter!=list_end(&thread_current()->donation_list);iter= list_next(iter))
{
  if(list_entry (iter, struct thread, donorelem)->waiting_on_lock ==lock)
  {
    list_remove(iter);
    list_entry(iter, struct thread, donorelem)->waiting_on_lock = NULL;
  }
  else continue;
}

struct thread *max_donor=
list_entry(list_begin(&thread_current()->donation_list),struct thread, donorelem);

for (iter = list_begin(&thread_current()->donation_list);
iter!=list_end(&thread_current()->donation_list);iter= list_next(iter))
{
  if(list_entry (iter, struct thread, donorelem)->priority > max_donor->priority)
   max_donor=list_entry(iter,struct thread,donorelem);
}

if(thread_current()->basepriority < max_donor->priority)
{
 thread_current()->priority = max_donor->priority;
 thread_yield();
}
else{
     thread_set_priority(thread_current()->basepriority);
    }

   }
else
  {
       thread_set_priority(thread_current()->basepriority);
  }

  intr_set_level (old_level);
```

(v) In **cond_signal (struct condition *cond, struct lock *lock UNUSED)**

```
list_sort (&cond->waiters,(list_less_func *) &conditional_var_comparator, NULL);
```

cond->waiters list is sorted according to priority of first element of semaphore->waiters

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



Source: Internet.

For tracking priority donation, we maintained a list of priority donors to a thread which is sorted according to priorities. While the lock acquirer does not releases lock and if the current thread's priority is greater than lock acquirer's priority, the current thread donates it's priority to the acquirer of the lock. Basically, the thread which is receiving the priority, is checked. If it is in turn waiting
on a lock (cur_thread->locker_thread != NULL), then the thread on which it is waiting is given the priority. This process is continued in a loop until a thread which is not waiting on any lock is reached. So, we maintain a list to track priority donations.

---- ALGORITHMS ----

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

The list of conditional variable waiters are sorted according to their priorities while signalling (in the cond_signal function) and also, in the sema_up and sema_down functions, sema->waiters are sorted according to their priorities. This ensures that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first.

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

If a thread tries to acquire a lock which is not holded by any other thread, the thread simply acquires the lock. Otherwise: the thread is pushed to the donators list of the thread acquiring lock. While the lock acquirer does not releases lock and if the current thread's priority is greater than lock acquirer's priority, the current thread donates it's priority to the acquirer of the lock. Basically, the thread which is receiving the priority, is checked. If it is in turn waiting on a lock (cur_thread->locker_thread != NULL), then the thread on which it is waiting is given the priority. This process is continued in a loop until a thread which is not waiting on any lock is reached

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

While releasing a particular lock, it is checked if there are any waiting threads in the thread's donor's list. If the list is non-empty, maximum priority thread from the list is removed and unblocked

---- 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?

Potential race condition:
When setting/updating a thread's priority, timer interrupt can schedule a new thread so that the thread whose priority was intended to change, doesn't change.

For synchronization, as the codes are of small enough length and time spent during the interrupts are disabled will be less, we have used interrupt disabling.

We have disabled interrupts in thread_set_priority(). This is because we have to read / write to the current thread's priority, which is updated every 4 ticks in the interrupt handler. Because of this, we cannot use locks since the interrupt handler cannot acquire locks.

---- RATIONALE ----

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

Due to several reasons. First of all, minimize the code length. Also, I was able to use the list implementation provided by list.c and due to it sorting and manipulating lists became much easier. This design was also better since we had to disable interrupts for short durations only. To acquire locks, we maintained a list of waiters sorted according to priorities.

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?

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

>> 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?

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

>> Any other comments?