```
                    +--------------------+
                    |        CS 140      |
                    | PROJECT 1: THREADS |
                    |   DESIGN DOCUMENT  |
                    +--------------------+
```

---- GROUP ----

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

Amira Mohamed <amirafathi3695@gmail.com>
Gehad Fathy   <gehafathy99@gmail.com>

 >

---- 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.

- thread.h
  - Struct thread
    int64_t wakeup_time:
      the time the thread should wake up at, it’s equal start time when calling timer_sleep plus ticks that the thread should sleep for.
        Wakeup = timer_ticks() + ticks
    struct list_elem sleeping elem:
      the element that is inserted in the sleeping list representing the sleeping thread.
- timer.h
  - struct list sleepingList:
      a list contains all the sleeping threads.

---- ALGORITHMS ----

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

timer_sleep
1- calculate *wakeup_time* for the current_thread (timer_ticks+ticks)
2 disable interrupts
3- push current_thread in the *sleepingList*
4- block current_thread
5- enable interrupts

timer_interrupt
1- increment ticks
2- call thread_tick()
3- loop through *sleepingList*
    if wakeup_time is less than or equal ticks
        remove the sleeping elem of this thread from the list
        unblock it

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?
inserting threads sleeping elements in increasing order according to their wake up times, so the loop will continue unblock threads with wakeup_time less than or equal ticks until finding a thread with wakeup_time greater than ticks.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
The interrupts are disabled at the beginning of timer_sleep to make adding sleeping elements to the sleeping list safe.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
Timer interrupts are called every tick and elements are removed from sleeping list, so interrupts are disabled before adding elements not to put these two events (adding and removing elements) in race conditions.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?
1- returning from *timer_sleep* if ticks is less than or equal zero so no need to push an element in the sleeping list and block the thread.
2- keeping the sleeping list sorted or not, if it’s kept unsorted the time is minimized in the *timer_sleep* as pushing the element will cost O(1) but the time in *timer_interrupt* will be always O(n), so we choose to minimize the time for timer_interrupt because the system call it every tick.

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.

New Struct members :

- In thread.h :

  - struct thread

    1- struct thread* donated_to

    it refers to thread which this thread dontes its priority to. it is only one thread as when the the thread donates its priority it blocks. We need it because when thread priority changed, the threads which take its thread should change also.

    2- struct list_elem donatedelem

    it refers to the threads which donate its priority to this thread . We need to store them as when thread releases lock we set its priority to:

    its base priority in case of no donors .

    else we set it to the largest priority of its donors .

3- struct list_elem waitedelem

it refers to the element which stored in the list of lock waiters .

- In sync.h

- struct lock

1- struct list waiters_on

it refers to which threads wait this lock

we use it when lock is released to know which waiter donate to the holder to remove it .

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

locks : A , B
threads : L , M , H

lock A :
L holds
M acquires

lock B :
M holds
H acquire

M donors list --- H .
L donors list ---> M .

H donated_to ---> M
M donated_to ---> L

A waiters_on list ---> M
B waiters_on list ---> H

L current priority = M
M current priority = H

- ALGORITHMS ----

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

We insert in all lists in order using list_insert_ordered(&list,less_func , NULL) and then return the max(front) where less_func is the function which sort accordin to priority
but in cond_signal we use less_sema_func which sort semaphore waiters according to priority

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

Sequence for donation :

1. insert the current thread in the waiting for this lock .
2. Insert the current thread in the lock holder donors list
3. set the current thread donated_to to the ock holder .
4. Set the lock holder priority to the current thread priority.
5. Handling the nested donation :
   If the holder donated to other thread set its priority also .

Pseudo code :

```
    if (lock->holder  != NULL){
          list_insert_ordered (lock->holder->donors, thread_current  )
          thread_current ()->donated_to = lock->holder .
          lock->holder->priority = thread_current->priority.
          insert_in_ready it in ready list again with the new priority .
          Temp =  holder.
          while (Temp donate to another thread){
                temp -> donated_to->priority =thread_current ()->priority;
                insert_in_ready it in ready list again with the new priority;
                temp  = temp  - > donated_to
          }
    }
    lock->holder = thread_current ()
```

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

1. Remove all waiters on this lock
2. If there is a waiter which has same priority of one of the holder donors then this the donor which the holder take its priority now so remove it from the donors .
3. Set the current thread priority
    if there is no donors : priority = its base.
    else priority = max priority of donors.
4. If there is thread in the ready list have priority > current thread new priority ----> current thread yields .
5. Set the holder to NULL .

Pseudo code :

```
for each e in lock ->waiters_on list {
      thread waiter = list_entry (e, struct thread, waitedelem).
      int p = waiter -> priority .
      Remove this waiter .
      for each  e2 in thread_current()->donors list {
            thread *t = list_entry (e2, struct thread, donatedelem);
```

```
                if (t->priority == p )

                        Remove this donor, t->donated_to =NULL

           } end

           if (no donors) :

                thread_current() -> priority = thread_current() -> base ;

            else :

                thread_current() -> priority = max prioity of donors

                yield ();

      } end

    lock->holder = NULL;

 } end lock_release
```

---- 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?

Race condition may happen when priority variable is updated the thread_tick updated it at the same time (because it updated it each 4 ticks).

To void it we need to make some synchronization technique so I choose to disable the interrupt in the beginning of the function and enable it at the end.

We can't use lock because thread tick is called in interrupt handler and interrupt can't acquire locks .

---- RATIONALE ----

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

We choose this design for many reasons :

1. I choose to add donor list in stuct thread to handle multiple donation as they can be more than one donated to it
2. We choose the donated_to to be onle one element because when thread donate to other thread it blocks so there will be no more than one element
3. We choose to add list waiters in struct lock because many threads can wait on the lock.

It's better than other designs We considered :

1. In one of them to add list of donated_to but it it's redundent overhead .

2. the other is to add waited lock in struct thread but the thread can acqure

more than on lock .

ADVANCED SCHEDULER

---- DATA STRUCTURES ----

>> C1: 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.

- thread.h
  - struct thread
    - int nice : represent how "nice" the thread should be to other threads threads.
    - int32_t recent_cpu : measure how much CPU time this thread has received "recently".
- thread.c
  - load_avg : represent the system load average, estimates the average number of threads ready to run over the past minute.

- new fixed-point.h file : define the macros that performs fixed point operations(MUL, DIV, ADD, SUB, CONVERT ..)

---- ALGORITHMS ----

>> C2: Suppose threads A, B, and C have nice values 0, 1, and 2. Each
>> has a recent_cpu value of 0. Fill in the table below showing the
>> scheduling decision and the priority and recent_cpu values for each
>> thread after each given number of timer ticks:

timer ticks	recent_cpu A	recent_cpu B	recent_cpu C	priority A	priority B	priority C	thread to run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

>> C3: Did any ambiguities in the scheduler specification make values
>> in the table uncertain? If so, what rule did you use to resolve
>> them? Does this match the behavior of your scheduler?

1- at the first choosing A to run despite the equality in recent cpu and priority but according to be the first one in the ready list, this behavior is the same as my scheduler.

2- when two or more threads are equal in the maximum priority, choose the one with minimum recent_cpu, the same as my scheduler.

>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?

Most of calculations of advanced scheduler are made every tick or slice or second within *thread_tick* and some calculations unfortunately require disabling interrupts to perform them safely like priority in *set_priority, lock_acquire, lock_release* and that weakens the performance.

---- RATIONALE ----

>> C5: Briefly critique your design, pointing out advantages and
>> disadvantages in your design choices. If you were to have extra
>> time to work on this part of the project, how might you choose to
>> refine or improve your design?

Advantages
- algorithm is simple in sorting according to the highest priorities secondly according to recent_cpu.
- using the same ready_list used in the priority scheduler so no overhead in the data structure.

Disadvantages
- insertion in order to keep the data structure sorted is slow (as requires to loop through the list every time O(n)).
- interrupt disabling make the system more slow to protect some calculations (like priority).
- the overhead of fixed-point calculations instead of floating point calculations.

HOW To Improve it
- using better data structures(like heaps or BST as insertion and retrieving in O(log(n))) or more efficient sorting algorithms.
- using more considerations in scheduling threads not only priority and recent_cpu to make the system more fair.
- make the system variables independent as possible to avoid interrupts disabling.

>> C6: The assignment explains arithmetic for fixed-point math in
>> detail, but it leaves it open to you to implement it. Why did you
>> decide to implement it the way you did? If you created an
>> abstraction layer for fixed-point math, that is, an abstract data
>> type and/or a set of functions or macros to manipulate fixed-point
>> numbers, why did you do so? If not, why not?

How it’s implemented :
- it is implemented in just a header file and methods explained in the documentation are implemented as macros: ADD, SUB, MUL, DIV, CONVERT
- By including this fixed-point header file and using these macros.

why we decide to implement it in this way :
- easy to use.
- easy to code.

The benefit of using abstraction layer :
- abstraction make the code more readable and maintainable.
