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

---- GROUP ----

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

Ahmed Ibrahim Hafez
Ahmed ElSayed Mahmoud
Mohamed Elsayed Mohamed

---- 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.

- static struct list sleeping_threads
It is added to 'timer.c' global. It is a List containing the currenty blocked threads till the ticks they should wait ends.
- int64_t wake_up_tick
It is added to 'thread.h' inside the thread struct. It represents the time when this thread should wake up (unblock)

---- ALGORITHMS ----

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

- As for timer_sleep():
    First we disable the interrupts, then get the current thread, then set the wake_up_tick to the sum of the current tick and the ticks it should sleep which is given in the parameter of the function, then we insert this thread in the sleeping_threads list by the elem list_elem and this insertion is made to be in ascending order by list_insert_ordered method, then the thread is blocked and the interrupt is enabled again.
- As for timer interrupt handler:
    We increment the 'ticks' then call 'thread_tick()'.Finally, awake sleeping threads with 'wake_up_tick' equal to 'timer_ticks'. The ready threads are then unblocked, and put into 'ready_list'.

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

- This the insertion in the sleeping_threads list was in order then the insertion was taking O(n) time but in the timer_interrupt method it will only take O(1) as while looping on the list if we found a waking_up_tick bigger than the current tick then it means that the rest of the elements the list has higher waking_up_tick the the loop should stop, and this modification is efficient as timer_interrupt is called every one tick which means 100 time in one second while the timer_sleep is called one time for each thread before blocking it for several ticks.

---- SYNCHRONIZATION ----

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

- By disabling the interrupt before setting the wake_up_tick value and enabling it after the thread is added tho the sleeping_threads list and blocking it.

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

- Since timer_sleep() the interrupts was disabled in it then the timer_interrpt would’t be able to interrupt this thread till it finishes the timer_sleep() i.e. The race condition here will not cause any problems.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?
- This design is intuitive and easy to implement. Sorting the threads makes the 'time interrupt handler' more efficient and faster and insertion in the sleeping_threads list is in order which costs O(n) time but in the timer_interrupt method it will only take O(1).

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' inside 'struct thread{}':

```
        int original_priority;  /* priority to know the Original priority
before donation. */
        struct thread* thread_waiting_for; /* the thread that the current
thread waiting for */
        struct list donors;  /* Pointers to donor threads that donated me. */
```

To know for each thread the donors that donated me, thread that this thread is waiting for and the original(base) priority before any donation.

- In 'thread.h' add new struct 'donor_elem':

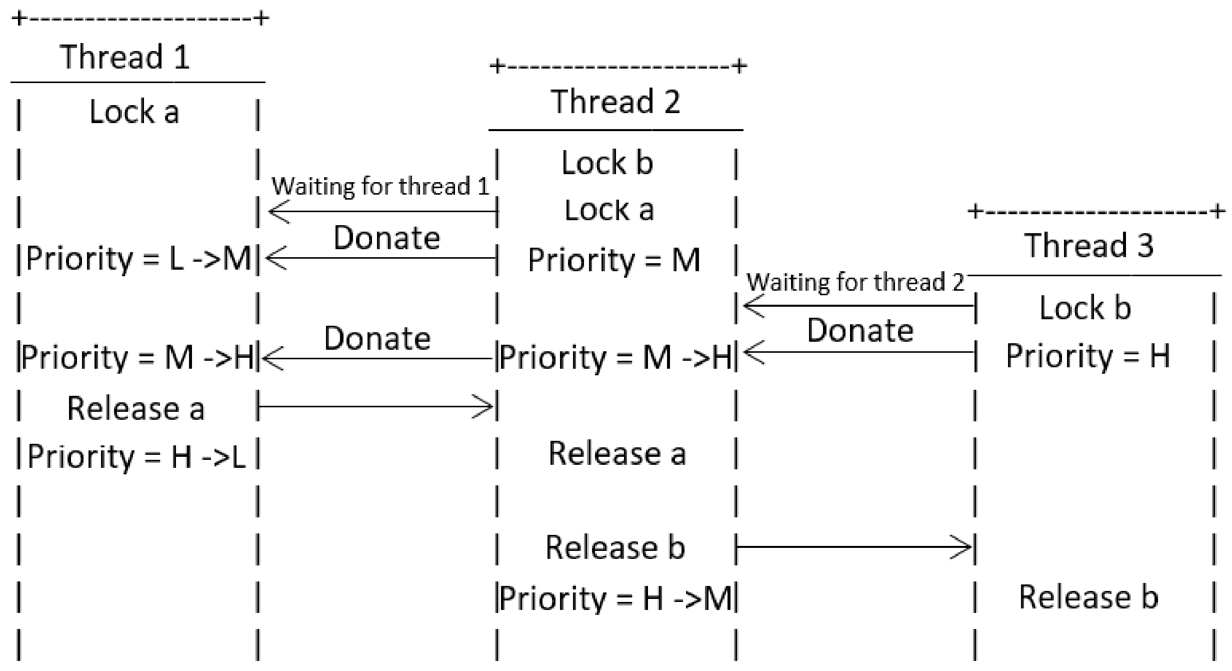
```
struct donor_elem
{
        struct thread* donor_thread;  /* The thread that donated to this
thread. */
        struct lock* lock_waiting_for;   /* The lock that donor_thread is
waiting for. */
        struct list_elem donor_list_elem;/* List elements. */
}
```
It is a struct element to be put in the donors list to know the thread that did donation and the lock that donor thread is waiting for.

- In 'thread.c':
We changed
```
        static struct list ready_list;
```
to
```
        static struct list ready_list[64];
```

It is an array of 64 ready queues to hold the 64 level of priority from 0 to 63. Each list represents the priority level from the lowest priority to the highest priority.

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

- We used 'struct list donors;' inside 'struct thread' in 'thread.h'.
It is a list of donor element struct which holds the the thread that make donation and the lock that the thread is waiting for.



---- ALGORITHMS ----

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

- By modifying the method next_thread_to_run to return the first element in the ready_list array that lies in the highest level by looping on it from 63 to 0 once it a list that is not empty it returns the front thread.
In Semaphore: when the value reaches 0 the thread is added to a waiting list then the thread is blocked. And when sema_up is called then if the waiting list is not empty then get the thread with the max priority then this thread from this list which will have the highest priority.
In Lock: It goes like the semaphore as it only calls sema_down when lock is acquired and donates its value to the lock holder if it has higher priority and put it in the ready_list array in its new level and it will be added in the lock holder donors list and calls sema_up when lock is released and remove all the donors from the donors list who are waiting for the same lock and if the list of donors is not empty then it will set its priority with the highest priority in that list.
In Condition Variables: when cond_wait is called the current thread is added to the waiting list of this condition variable and make sema_down to its semaphore, So when cond_signal is called and it found threads waiting for this condition variable it will remove the thread with highest priority from the waiting list and sema_up to this thread to be active again.

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

- When lock_acquire() is called if the lock has no holder this thread will be set as its holder but if there was already a holder, then a donor_elem is created having the lock this thread is waiting for and it set itself as the donator for the holder, and it also sets the thread it is waiting for to the holder thread, then it is pushed back to the list of donors of the lock holder, then it enters in a recursive function where it check at first if the current thread priority greater than holder priority then donate to the holder and remove holder from the ready_list and put it in the list according to its priority and if the holder is waiting for another thread then repeat this function, At the end it call sema_down to lock semaphore.

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

- When the lock is released then we will loop on all threads in donors list and remove those who are waiting on the same lock and remove current thread from all this thread_waiting_for and then set the priority of the current thread with maximum priority between the original_priority and the remaining threads in donors list. Then it remove this thread from lock holder and call sema_up to the lock semaphore.

---- 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?

- The potential race will happen as multiple threads will want to change their priority by thread_set_priority or by donation while there is a thread changing its priority by thread_set_priority also due to this change in priority the next thread to run might change so it had to be handled. So to prevent this race we needed to disable he interrupt during thread_set_priority and after the priority is set the interrupt is enabled again and the scheduler will choose the thread with the highest priority to run.

---- RATIONALE ----

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

- we need to make multi-level and nested donation, thread can hold multiple locks and for every lock we want the highest priority thread of all other threads which acquired the lock to donate its priority to the holder thread but this will cost a linear search every time, thread acquires lock to find this lock in the list of locks that holder thread have but we add any thread acquired lock to this list which now have more than one donor for the same lock and when lock released we delete all of them that cost one linear search when lock is released, and this better than linear search to find the specific lock with every thread acquires this 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.

- In 'thread.h' inside 'struct thread':

```
	fixed_t recent_cpu; /* To measure how much CPU time each process has
received recently. Fixed point number */
	int nice; /* An integer to determine how 'nice' the thread should be
to the other threads. */
```

- In 'thread.c' global:

```
	fixed_t load_avg; /* To estimate the average number of threads ready
to run over the past minute. Fixed point number. */
```

---- 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	recent_cpu			priority			thread
ticks	A	B	C	A	B	C	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?

- The specification does not mention the order in which 'load_avg' and threads' recent_cpu and 'priority' should be calculated. Also, 'load_avg' and 'recent_cpu' are real numbers but their precision is not given.

- We resolved the first issue based on the dependence of these variables. First, we increment current thread's 'recent_cpu'. Then we calculate load_avg as it depends only on the number of ready threads. Then, we calculate the new 'recent_cpu' (depends on 'load_avg') and the new priority (depends on 'recent_cpu').

- We resolved the second issue as for the real numbers, we implemented a fixed-point arithmetic with 14 fractional bits. This allows accurate representation of real. There is no need for big numbers as 'load_avg' uses only the number of ready threads for its calculation (which will not exceed a few thousands in the worst case). Also, 'recent_cpu' stays quite small because of small value of 'load_avg' and 'nice'.

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

- The code running outside the interrupt context runs faster than the code running inside the interrupt context. So, we minimized the amount of code that was inside interrupt context. This includes all the necessary calculations of 'recent_cpu', 'priority' and 'load_avg'. The rest of scheduling code is outside of the interrupt context to maintain a higher 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:
    I feel that this design is good in terms of simplicity and readability. The implementation is split well between a set of functions that can be clearly understood. The most changes are I 'thread_tick()'. All the functions

are also grouped together in one file. Macros provide a simple and efficient implementation of the fixed-point.h operations that are required.

- Disadvantages:
    Using if statements to check the mlfqs flag which indicates whether the basic or advanced scheduler is used.
If I was to have extra time, I would separate the two schedulers by using static functions that could be called without the knowledge of the active scheduler. The two schedulers could be separated into files. It would be something like a base abstract class and two subclasses in OOP. It would be bigger if there are more than two schedulers.

```
>> 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?
```

- To implement the fixed-point operations, we created a set of macros for all of the arithmetic operations given in the documentation. This makes the code more readable as the operations have names in the code rather than just looking at the implementation of each and having to find out what each one does. If an error is made in the implementation of an operation, then it only needs to be modified in one place.

- This can be done by using a function for each calculation rather than a macro, but this will cause overhead at runtime for the function call. The overhead of using macros is at the compile time when the pre-processor runs and replace the macros with the defined values.