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

---- GROUP ----

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

Ahmed Gaber <ahmedgaber20@gmail.com>
Monica Salama <monicasalama6@gmail.com>
Mostafa Gabriel <gabriel.mostafa@gmail.com>

---- PRELIMINARIES ----

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

- Don't have any.

>> 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.

- Nothing.

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.

1. In the thread struct, we added those two variables:
    a. struct list_elem waitelem /* List element, stored in the wait_list. */
    b. int64_t sleep_endtick /* Used if the thread is sleep, the thread should awake after this tick */
2. We added a new list to hold all the waiting (or sleeping threads):
    a. struct list wait_list /* Holds all the currently sleeping threads. */
3. We changed the function thread_tick() to take the current tick created by the timer interrupt handler.

---- ALGORITHMS ----

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

1. When calling timer_sleep(), we first disable the interrupts and pass the sleeping period for thread_sleep_for_ticks().
2. thread_sleep_for_ticks() function sets the sleep_endtick for the running thread to the given value, and pushes the thread to the wait_list. After that, the function thread_block() is being called to actually put the thread to sleep and switch to another thread.
3. At each thread_tick(), we call the function thread_wakeup(), which loop over the wait_list and calls thread_unblock() for any thread reaches its sleeping period.

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

- We only used the wait_list to store all the waiting threads, and we need to check it at every tick to find if there is any thread needs to be awake. This operation can take only few millisecond to perform.

---- SYNCHRONIZATION ----

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

- The function timer_sleep() disables interrupts before executing the critical code. That means no race condition will happen between multiple threads.

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

- As mentioned in A4, no race condition can ever happen thanks to interrupts disabling.

---- RATIONALE ----

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

- This is almost the simplest design one can think of. Looping over the waiting threads at each tick may cause a tiny delay in the program, but as we mentioned before, it's only a matter of milliseconds. We thought we may use a hash to map directly to the threads needs to be awake.

PRIORITY SCHEDULING
==========

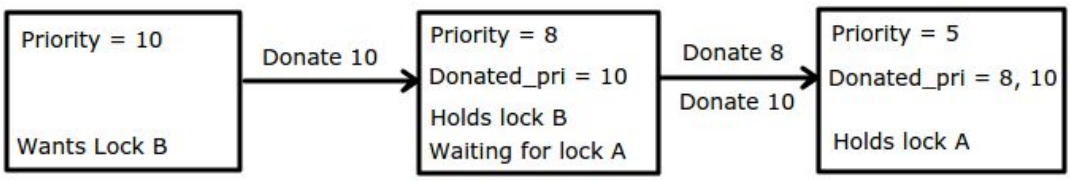
---- DATA STRUCTURES ----

>> 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.

1. Struct Donation: which contains the donated_priority and the lock that causes this donation.
2. Struct Thread: added list of donors to hold donations to that thread (list of struct donation).

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

- For each thread there's a list of donors where each element contains the donated priority and the lock that causes this donation



---- ALGORITHMS ----

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

- For each of these, there's a list of waiting threads so looping on them to get the one with the highest priority and wake it up.

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

- It first tries to acquire the lock, in success nothing happens.. if the lock is acquired by another thread (lower-priority thread) the current thread donates its priority to that

thread and if that thread is blocked on another lock the priority will be given to the holder of that lock and so on until reaching an unblocked thread.

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

- With a higher-priority thread waiting for the lock, that means the current thread has a donated priority and here comes two cases:

1. It has that priority because of that lock: so we'll remove that priority from its donors thus lowering its priority and unblocking the higher-priority thread.
2. It has that priority because of another lock: so nothing will happen to the donors list nor its priority so the higher-priority thread kept blocked until the current thread loses the donated priority by releasing another lock.

---- 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 could happen while set priority is updating the new priority , and the interrupt handler is writing to the same variable.
- Our Implementation avoids it by disabling interrupts while the thread's priority variable is being updated.
- No , a lock can not be used to avoid this race , interrupt handler are not supposed to acquire locks.

---- RATIONALE ----

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

- Simple, easy to implement

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.

1. Changes in <thread.h>
   - We added two more variables to the thread struct:
   1. int nice; /* Thread niceness, or how nice this thread is to other threads. */
   2. int recent_cpu; /* time spent by this thread on the CPU */
   - We declared two global variables:
   1. #define NICE_MAX 20 /* The maximum nice value a thread can have. */
   2. #define NICE_MIN -20 /* The minimum nice value a thread can have. */

2. Changes in <thread.c>
   - We declared two variables:
   1. int load_avg; /* System's load average -- used in mlfqs */
   2. #define TIMER_FREQ 100 /* Timer frequency -- used in mlfqs */

---- 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:

- As used in our program, in the program's initialization, let:
  load_avg = 0
  TIMER_FREQ = 100
  TIME_SLICE = 4

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

>> 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?

   - Yes. For example, in the case when two or more thread have the same priority, our scheduling implementation will choose thee first one appears in the ready list.

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

- The code runs inside the interrupt context runs slower than the code outside. Within the scheduler, most of the code needs to run inside the interrupt context, so it can be a little bit slower.

---- 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?

- This design is a simple, and to-the-point design, which do the work needs to be done. One disadvantage of this design is the use of lists which needs more time to perform some operations. If I have extra time, I would different data structures like hash maps and priority queues.

>> 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?

- As Pintos has it's own implementation for the fixed-point math, we wanted to make it available and reusable by all other Pintos functions, so, we implemented in it's own files so that any other file can include it and directly use it's functions.