```
+---------------------+
|       CSEx61        |
| PROJECT 1: THREADS  |
| DESIGN DOCUMENT     |
+---------------------+
```

---- GROUP ----

```
Mohamed Tarek Aiad          <es-mohamedaiad2000@alexu.edu.eg>
Mohamed momen salama         <es-mohamedsalama236@alexu.edu.eg>
Ahmed Abdallah Aboeleid    <es-Ahmed.Abdelsatar2024@alexu.edu.eg>
Mohamed Mostafa Goher        <es-MohamedIbrahim24@alexu.edu.eg>
Michael samir azmy           <es-michaelsamir2024@alexu.edu.eg>
```

---- 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.

* the great requirement explanation of Eng/ khaled eltahan
https://www.youtube.com/watch?v=RLx_0nnEjaM
* CSCI 350 Pintos Guide pdf
https://tinyurl.com/yc3rcfvp
* Dr.noha adly lectures from alexandria university

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.

```
->A: /*An ascending order list by sleep_ticks used to store threads
that                          are put to sleep*/
     struct list blocked_threads;

  /* add a new variable to store the time after which the thread will
wake up */
     int64_t wake_up_time;
```

---- ALGORITHMS ----

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

-> A:In a call to timer_sleep()

1. The current thread's waking up time is set to the number of ticks that the thread will sleep plus the current ticks.
2. Interrupts are disabled.
3. The thread is inserted to the blocked_threads list.
4. Block the thread.
5. Reset interrupts level to its old one.

So, in timer interrupt handler,

1. Check the blocked_threads list to see if there is any thread that needs to be woken up.
2. Remove it from the blocked_threads list,
3. Unblock the thread

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

->A: An ordered list(blocked_threads) which is sorted and inserted by waking up time, so that we check the list from the beginning and stop whenever the waking up time is larger than the current ticks, which guarantees the later threads in the sleep list don't need to be checked.

---- SYNCHRONIZATION ----

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

->A: All the operations (as block thread, insert thread to blocked list) is done when the interrupt is disabled.

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

->A: We disabled the interrupt.

---- RATIONALE ----

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

->A: To have a blocked_threads list to store the sleeping threads are the straight forward thought, so it was easy to implement and safe. The other design was to use semaphores but we avoid it as it was hard to implement.

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.

->A: IN struct of thread:

```
     1) /* Stored original thread priority. */
           int original_priority;
     2) /*to check if the thread is donated from higher threads or
        not*/
           bool donated;
     3) /* List of locks hold by thread */
           Struct list locks;
     4) /* the lock that the thread waiting to hold */
           Struct lock *waiting_for
   IN struct of lock:
```

1) /* List element for locks hold by threads. */
    Struct list_elem elem;

>> **B2:** Explain the data structure used to track priority donation. "Use ASCII art to diagram a nested donation".

->A: **"ART"**:

```
1)  | low | // a thread with low priority holds a lock
2)  | med | -> | low |  // a thread with medium priority wants
to acquire that lock
3)  | med | -> | med |  // that thread donates its priority to
the lock holder thread
4)  | high | -> | med | -> | med |  // a high priority thread
    wants a lock that medium has
5)  | high | -> | high | -> | med |  // high priority thread
donates it's priority to the next thread in the donation chain
6)  | high | -> | high | -> | high |  // the newly-high thread
donates it's new priority to the first thread(nested donation)
```

"We assume that a thread with priority "x" holds a lock, and after some time, there is another thread with priority "y" which is greater than priority x wants to acquire that lock, so the second thread will donate on the first thread and the priority of the first thread will be changed to priority y. And if there is a third thread with priority higher than priority "Y" it will donate on both the first and the second thread and their priorities will be equal to the third thread."

"Data Structure Explanation":

As we mentioned in the above question, we added original_priority,donated, locks, waiting_for a lock, and added elem to lock struct, to help track the priority donation.

Every time a lock is acquired by a thread, the lock will be inserted into the thread's locks field, which is an descending ordered list sorted by priority field in the lock. Correspondingly, when a lock is released, it's removed from its holder's locks list. And this is also where the elem inside lock struct plays a role.

In a single donation, when the lock is being acquired, the lock holder's priority is checked, if it's lower than the one who is acquiring lock, donation happens.In our implementation, thread's original_priority will change with priority except donation, so we can assume original_priority already preserves the current priority. Then the donee-thread's priority is set donor-thread's priority;

donated is set to true, if it’s not true already. The lock’s priority_lock is set to be the donor's priority, to keep track of the highest priority in the lock’s waiter list. And the donor-thread’s lock_blocked_by is set to be this lock.

Then it’s checked whether the donne-thread is blocked by another lock, which is needed for nested donation. If yes, another donation case will happen in the same procedure above except the new donor is the current donee, the new donee is the lock holder whose lock blocks the current donee. The nested case will keep checking recursively by function donate priority until there is no donee thread that is blocked by some other thread.

When a lock is released, the lock will be removed from the holder thread’s locks list and then comes the checking of whether multiple donations happened to this thread before. If the locks list is empty, no locks are held, it means that no multiple donation happened, the thread should return its priority to the original one using original_priority. Otherwise, get the first lock from the locks list, if the donated variable is false (which means no donation happened), the thread returns its priority to the original priority too. If the donated value is true (which means a donation has happened),then the holder’s priority should be reset to it. Since locks is a descending order list sorted by the priority_lock field, we can guarantee that the first lock in the list has the highest priority among all the waiters of all locks, which is the priority the holder should have.

Using the data structure and algorithm above, priority donation, including the simplest donation, multiple donation, and nest donation, can be achieved.

---- ALGORITHMS ----

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

->A: Every time the waiters list is checked, we find the thread that has the highest priority in the list and make this thread wake up.

Whenever a lock or semaphore is released or upped, we perform a check of all the waiting threads and wake up the thread with the highest priority. This thread is then removed from that list of waiters. For a condition variable, the same thing happens with the list of semaphore elements, we find the semaphore element that has the thread with the highest priority waiting on it.

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

```
->A: Steps:
   1. Disable interrupts
   2. Donation
     2.1 IF lock_holder is NULL
     2.1.1  sema_down: if sema value is 0, put all threads acquiring
this
            lock into the sema's waiters list until sema value
becomes
            positive
     2.1.2  Set the current thread to this lock's holder
     2.2 ELSE compare lock_holder's (L) priority with current
thread's (C)
         priority:
     2.2.1  IF L's priority > C's priority
     2.2.1.1  Does sema_down until the sema value becomes positive
              which means lock is released
     2.2.1.2  Set the current thread to this lock's holder
     2.2.2  ELSE:
     2.2.2.1  [Donation] Set L's priority to C's priority
     2.2.2.2  Does sema_down, until the lock is released
     2.2.2.3  The current thread becomes this lock's holder
```

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

->A: When a lock is released, that thread returns back to its original priority. Then lock->holder pointer is set to null, and sema_up is called. Then we check to see if the current thread priority is less than the max priority on the ready_list, yielding if it is.

---- 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?

->A: We disable the interrupt to prevent it from happening, then we set the new priority and finally we enable the interrupt.

---- RATIONALE ----

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

->A: In order to avoid making struct thread and struct lock bigger, we just added a boolean variable “donated” to know if the priority is donated or not, also we added variable “original priority” to store the priority of the thread before donation. For example, we initialize original_priority equal to the priority when it is created. Whenever a donation happens, we set the bool variable donated equal true and the priority will equal the donated priority. Thus the code is more readable and easy to understand.
For struct lock, we added a new member in lock to get the highest priority in its waiters list. It also make the code understandable and readable.
For multiple donations, we use the function “donate_priority” that takes a lock and the thread wants to acquire that lock, it works recursively by checking for every lock holder if it waits for another then donating the priority if the first is larger than the second.This logic was easy to make and splitting them to two functions make the code simpler.

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.

```
->A: IN real.h:
        1) /* Struct real */
                struct  real {
                    int x;
                };
   IN struct thread:
        1) /* Thread nice value. */
                int nice;
        2) /* CPU ticks while thread using cpu. */
```

```
                 Struct real recent_cpu;
```

---- 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	1	2	63	61	59	A
4	4	1	2	62	61	59	A
8	7	2	4	61	61	58	B
12	6	6	6	61	59	58	A
16	9	6	7	60	59	57	A
20	12	6	8	60	59	57	A
24	15	6	9	59	59	57	B
28	14	10	10	59	58	57	A
32	16	10	11	58	58	56	B
36	15	14	12	59	57	56	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?

->A: recent_cpu here is ambiguous here. When we calculate recent_cpu,
we did not consider the time that CPU spends on the calculations
every 4 ticks, like load_avg, recent_cpu for all threads, priority
for all threads in all_list and modifying the ready_list order. When
the CPU does these calculations, the current thread needs to yield
not running. Thus, every 4 ticks, the real ticks that are added to
recent_cpu (recent_cpu is added 1 every ticks) and this is less than
4 ticks (not exactly 4). But we could not figure out how much time it
took. What we did was adding 4 ticks to recent_cpu every 4 ticks.

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

->A: If the CPU spends too much time on calculations for recent_cpu,
load_avg and priority, then it takes away most of the time that a

thread before enforced preemption. Then this thread can not get enough running time as expected and it will run longer. This will cause itself to be blamed for occupying more CPU time, and raise its load_avg, recent_cpu, and therefore lower its priority. This may disturb the scheduling decisions making. Thus, if the cost of scheduling inside the interrupt context goes up, it will lower 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?

->
A: Our design did not apply the 64 queues. We used only one queue -- the
ready_list that Pintos originally have. But as the same as what we did for the task 2, we push them on the ready_list normally but when get them we use the function get_max_periority to get the max priority thread. The time complexity is O(n). Every fourth tick, it is required to calculate priority for all the threads in the all_list.
If n becomes larger, thread switching may happen quite often. If we use 64 queues for the ready threads, we can put the 64 queues in an array with an index equal to its priority value. When the thread is first inserted, it only needs to index the queue by this thread’s priority. This will take only O(1) time. After priority calculation for all threads every fourth tick, it takes O(n) time to re-insert the
ready threads. But our implementation is better than this situation --ready_list is not ordered. Like pintos originally did, for every new unblocked thread, just push back to the ready_list. When it needs to find the next thread to run, it has to reorder the ready_list. Sorting takes O(n lgn) time. It needs to repeat this sorting whenever we need to call thread_next_to_run, and after calculating all threads’ priorities every fourth tick.

>> **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?

-> A: As we know recent_cpu and load_avg are real numbers (can’t be integers), but pintos disabled float numbers. Instead of using float numbers, we can use fixed-point numbers. So we use fixed-point numbers to represent recent_cpu and load_avg.We defined the struct real and the prototype of the functions in the file “real.h” and implement them in the file “real.c”.