```
+--------------------+
|        CS 333      |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- GROUP ----

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

A. Ahmed Rizk<ahmed.rizk1419@outlook.com>
Mohamed Elsayed Helmy<mohamedhelmy51288@gmail.com>
Yahia ElShahawy <yahia.elshahawy@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.

A- we add new members to struct thread in thread.h file and they are:

- *int64_t waketime: this variable indicates the time at which the thread must wake after it was suspended for specific time.*

- *#define compare_effective_ordered 0*
  This constant passed as aux parameter to list_less function to sort the ready list According to their priority.

- *#define compare_wake_time 1*
  This constant passed as aux parameter to list_less function to sort the sleeping list According to their wake_time.

B- we add new global variables in thread.c file and they are:

- *static structs list sleeping_list: this is static list of the suspended threads in sorted Way according to the wake_time variable in the struct thread and it olds them until They wake and when they wake they moved to the ready list.*

- *. static struct list ready_list: this list holds threads which are ready to run and they are sorted according to the priorty of each thread.*

- *list_less_func *less: this is a pointer variable from the type list_less_func and it is Used to point to list_less function which compare between threads to sort them in The lists like ready and sleeping list.*

---- ALGORITHMS ----

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

*We call **thread_suspend** function and this function has one argument which is the time In which the thread must wake up. And this function add this thread to the **sleeping list** in the order of **wake _time** And at the end of this function we call the **schedule** function so that another thread run.*

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

*In the timer interrupt handler we call **check_suspend_queue** function and this function check if any thread of the suspended thread must wake or not ,so that we must search in the **sleeping list** for the threads which must wake but if we search for these*

*threads in every tick this considered waste of time so we add the threads in the sleeping list in sorted way according to the **wake time** ,so we does not need for search the sleeping list every time because we will check until the wake time of any thread greater than the current time ,and if we in the case which we loop all the list this means all threads must wake up and moved to **ready list.***

---- SYNCHRONIZATION ----

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

It's guaranteed that only one thread is running at a time which can call time_sleep(), so no other thread can call timer sleep at the same time.

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

*This avoided by disable the interrupt using **intr_disable** function in the function **thread Suspend** which called **in timer sleep function** and return it to its previous state using **intr_set_level(old_level)** where the old level is the previous state of the interrupt*
So that timer interrupt will not happen until the interrupt enabled.

---- RATIONALE ----

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

When we think about the design of this project we find this is the best because it simply adds 3 functions in thread.c file and add some variables.
And other designs we think about it mainly the same but they are different in calling some functions in other positions different from our design.

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 thread.h file we added members to struct thread:

- *int effective_priority: this variable holds the current priority of the thread because the priority of the thread may change because of donation or set priority function and if the priority does not change it holds the original priority of the thread.*

- *struct lock *waited_lock: this pointer variable to the lock for which the current thread Wait it to realize the lock.*

- *struct list holded_locks; this is the list of locks which this thread hold it so that when A thread release a lock the priority of the thread must be equal to the maximum priorty Of a lock of this list of locks.*

B-In sync.h we added some members to struct lock:

- *int max_waiter_priority: this variable indicates the maximum priority of the thread which waiting or hold the lock and this for donation purpose so we add it in the lock struct because the priority of the thread changed by the lock which holded by it.*

- *struct list_elem elem: this to manage to make a list of locks for the previous purposes.*

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

Priority donation is tracked by:
1-keeping track of all locks that a thread hold.
2-holding a reference to the lock which the thread waits for it.

Thread - - - - - - hold a list of - - - - - >locks<- - - - waiting for - - - - waiting thread
<- - - - - lock holder - - - - - -

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

First, *we donate its priority to the thread which hold the lock to release the lock Quickly.*

Second *we add waiting thread in the waiting thread in sorted way according to the Priority of these threads.*

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

First we check if we try to acquire the lock throw lock_try_acquire function
If it return false this mean this thread will blocked so will solve the donation problem.
So, we will do that:

- *set the waited lock pointer in thread struct point to the lock to which it waits.*
- *check if the priority of the waiting thread greater than the max_waiter_priority Then make max_waiter_priority equal to priority of the waiting thread.*
- *check if the priority of the waiting thread greater than the lock holder thread then we will update the priority of the lock holder thread.*
- *repeat 2 and 3 until waited lock pointer points to null or the condition in 3 equal false.*

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

---- 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?

No synchronization primitives is needed because there is one thread is running so there is no potential race only one thread can access this function, so no lock is needed.

---- RATIONALE ----

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

This the first design we reach to it and we implement it and we think it is the best way.

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-typedef int fixed_point**: differ normal integer from a fixed-point representation number to apply the operation corresponding to each type independently.*

***2-in thread.h:** struct thread --> added*

- *fixed_point recent_cpu: store the recent cpu number of each thread in fixed point type.*
- *int nice: store the nice number of each thread in fixed point type.*

***3- in thread.c** global -->added*

- *fixed_point load_avg: store the load average to be seen globally by all threads.*
- *fixed_point parent_recent_cpu: store the parent recent cpu number to set the child's recenet cpu by it at creation time.*
- *int parent_nice: store the parent nice number to set the child's nice by it at creation time*.

---- 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?

No, the given values were enough to calculate the priority of each thread after every four ticks, the priority depends on the recent cpu number and the nice which is constant in the whole of the program. And in the cases when the priorities of the threads are equal we choose the oldest thread with this value to run.
In this case we assume the load average in the start of the program equal zero to it would change every one second so after 100 ticks and the previous program was only 36 ticks so it won't affect on the recent cpu.

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

Inside the interrupt in timer interrupt we update:

- *the recent cpu for the running thread by increment it by one*
- *load average every one second (100 ticks) and all recent cpu to all threads.*
- *the priorities of each threads every 4 ticks.*

But to get the next thread to run call it in the scheduler outside interrupt to reduce the cost of scheduling.

---- 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:

- *fixed_point.h file can be modified easily adding more operations for that type.*

- *we can change the representation format numbers for example: current default format of fixed point is p:17, q:14 where q is fractional bits and p is the decimal bits of the signed 32-bit integer. we can change it by calling function init_fixed_pointint new_p, int new_q).*
- *the partitioning of the equations of variables(recent_cpu,load_avg,priority) makes it easy to read or modify.*

If we have more time we could modify:

- *We could have minimized the equations more to reduce the number of* operations.

>> 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?

Choose this implementation because specifying 14-bit for fraction makes it more accurate and the whole number takes the same size of signed 32-bit integer.
Fixed point as an abstract data type represented in fixed_point.h can be modified easily adding more operations for it.

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?