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

---- GROUP ----

Aya Gamal es-Aya.MOhammed2022@alexu.edu.eg
Khadija Assem es-Khadija.Saad2022@alexu.edu.eg
Linh Ahmed es-LinaAhmed2022@alexu.edu.eg
Norhan Magdi es-norhanmagdi2022@alexu.edu.eg

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.

Global variables declared :

static struct list time_sleep_list

This global variable maintains an ordered list for sleeping threads

Fields were added to the thread struct :

int64_t wake_up_time

The wake_up_time field stores the current wake_up_time value of the thread

---- ALGORITHMS ----

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

In timer_sleep we set the value of the current thread wake_up_time ,then insert the thread into time_sleep_list and block the thread

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

The time_sleep_list list of sleeping threads is **ordered**, with threads with the soonest wake-up times at the front.

The time interrupt handler only has to look at the head of the list, check to see if the wake-up time for that thread has passed, and if it has: wake up that thread, and traverse the list continuing to wake up threads **until we see a thread whose wakeup time has not passed**. At most, the interrupt handler examines one thread which does not need to be woken.

---- SYNCHRONIZATION ----

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

Wherever we insert into, or delete from time_sleep_list , we disable interrupts, so reads/writes are atomic and uncorrupted

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

Interrupts are disabled in the critical sections of timer_sleep, so the timer interrupt can not occur.
So no races between the timer interrupt and the timer_sleep can happen.

---- RATIONALE ----

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

We used this design because it is easy in implementation and easy and simple to understand.

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.

Changed structs :

In thread.h Added to the Struct threads:

```
int original_priority;          /* original Priority of the thread before thread donation */
struct list locks_list;         /* List of locks the threads holds*/
struct lock *blocking_lock;     /* The for which is blocked for  */
int number;                     /* The order of creation of thread, used for FIFO */
```

In synch.h Added to the Struct lock:

```
struct list_elem lockelem;      /* List element for list of locks holded by the  thread. */
int priority;                   /* Lock's priority calculated by the max priority of
threads waiting for this lock, used for multiple donation */
```

Added to Struct semaphore:

```
struct lock *lock;              /* The lock requested semaphore */
```

In synch.c Struct `semaphore_elem`:

```
int priority;          /* Priority of thread waiting for condition */
```

Added global variables (in threads.c):

```
static int num;        /* to keep track of Number of threads created, used for FIFO*/
```

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

Used for priority donation:

In Struct threads

Original_priority, priority, locks_list, blocking_lock

In Struct lock

lockelem, priority

In Struct semaphore

Lock

When a thread requests to acquire a lock

if no thread holds the lock then the thread will be the lock holder and the lock will be added to the locks_list
If a thread holds the lock then the thread will be added to the semaphore's waiters and the lock will be the lock_waiting
So, Set the lock in semaphore struct (sema->lock) equals to lock
In sema_down if the current thread is a waiter then set the sema->lock to the current thread lock_waiting
If the waiter priority is larger than lock priority then go to donate priority which loops on while the current thread (waiter) priority is larger from lock priority and level > 8
lock priority = current thread (waiter) priority, update the lock_holder priority by checking if the maximum priority from the locks_list is larger than thread's original priority then current thread's priority = the maximum priority from the locks_list else current thread's priority is set to the original, and reorder the readylist as priorities changed, then iterate to the next lock holder waiting lock
After returning from sema_down current_thread will be the holder so make current_thread waiting lock = NULL and lock priority = current_thread priority, and then add lock to the lock list as the current thread is the holder, finally set the current thread as the lock->holder

When a thread releases the lock

if current thread lockslist is not empty then remove the lock, then update the current thread priority by checking if the locks_list is empty then current thread's priority is set to the original priority if locks_list is not empty then if the maximum priority from the locks_list is larger than thread's original priority then current thread's priority = the maximum priority from the locks_list else current thread's priority is set to the original priority.

Diagraming a nested donation

Low-priority main thread L acquires lock A. Medium-priority thread M then acquires lock B then blocks on acquiring lock A. High-priority thread H then blocks on acquiring lock B. Thus, thread H donates its priority to M, which in turn donates it to thread L.

Step 1 : begin
==========

Thread L	Original_priority	31
	priority	31
	locks_list	{lock A (priority_lock = 0)}
	lock_waiting	--

Thread M	Original_priority	32
	priority	32
	locks_list	{lock B (priority_lock = 0)}
	lock_waiting	--

Thread H	Original_priority	33
	priority	33
	locks_list	{}
	lock_waiting	--

Step 2 : thread M acquires lock A
==========

Thread L	Original_priority	31
	priority	32
	locks_list	{lock A (priorities_lock= 32)}
	lock_waiting	--

Thread M	Original_priority	32
	priority	32
	locks_list	{lock B (priority_lock = 0)}
	lock_waiting	lock A

Thread H	Original_priority	33
	priority	33
	locks_list	{}

	lock_waiting	--

Step 3 : thread H acquires lock B

Thread L	Original_priority	31
	priority	32
	locks_list	{lock A (priorities_lock= 32)}
	lock_waiting	--

Thread M	Original_priority	32
	priority	33
	locks_list	{lock B (priority_lock = 33)}
	lock_waiting	lock A

Thread H	Original_priority	33
	priority	33
	locks_list	{}
	lock_waiting	lock B

Thread L	Original_priority	31
	priority	33
	locks_list	{lock A (priorities_lock= 32)}
	lock_waiting	--

Thread M	Original_priority	32
	priority	33
	locks_list	{lock B (priority_lock = 33)}
	lock_waiting	lock A

Thread H	Original_priority	33
	priority	33
	locks_list	{}

	lock_waiting	lock B

Step 4 : thread L releases lock A
==========

Thread L	Original_priority	31
	priority	31
	locks_list	{}
	lock_waiting	--

Thread M	Original_priority	32
	priority	33
	locks_list	{ lock B (priority_lock = 33) , lock A (priority_lock = 32)}
	lock_waiting	--

Thread H	Original_priority	33
	priority	33
	locks_list	{}
	lock_waiting	lock B

Step 5 : thread M releases lock B
==========

Thread L	Original_priority	31
	priority	31
	locks_list	{}
	lock_waiting	--

Thread M	Original_priority	32
	priority	32
	locks_list	{lock A (priority_lock = 32)}
	lock_waiting	--

Thread H	Original_priority	33
	priority	33

	locks_list	{lock B (priority_lock = 33)}	
	lock_waiting	--	

---- ALGORITHMS ----

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

When waking up a thread in sema_up , we use **list_max** to give the maximum priority thread waiter in sema->waiters to ensure that the highest priority thread wakes up first. Lock and condition variables use semaphore so they don't have to be modified.

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

when a thread calls lock_acquire()

If there is no thread holds the lock

1. Set the lock->semaphore.lock equals to the lock
2. Go to sema_down

If sema->value equals to zero

push the thread acquiring this lock into sema->waiters list, set the lock as the thread blocking lock ,update the lock's priority to the priority of maximum thread waiting for the lock (taking into consideration nested blocking by checking the lock) and block the thread until the sema value is larger than 0.

After returning from sema_down current_thread will be the holder so make current_thread waiting lock = NULL and lock priority = current_thread priority, and then add lock to the lock list as the current thread is the holder, finally set the current thread as the lock->holder

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

if current thread lockslist is not empty then remove the lock, then update the current thread priority by checking if the locks_list is empty then current thread's priority is set to the original priority if locks_list is not empty then if the maximum priority from the locks_list is larger than thread's original priority then current thread's priority = the maximum priority from the locks_list else current thread's priority is set to the original priority.
Then set lock->holder equals null
And go to sema_up which will increase the sema->value by 1 and that means this lock is going to be acquired by sema->waiters

---- 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 thread may want to change its priority. On the other hand, in priority donation lock->holder's priority may be changed if max priority from the locks_list is larger.
Handled by if The main thread attempts to lower its priority, should not take effect until the donation is released.

no , lock can not be used to avoid this race.

---- RATIONALE ----

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

We used this design because it is easy in implementation and easy and simple to understand

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.

Typedefs declared:

typedef fixed_point_t

This typedef provides a layer of abstraction for the fixed point type.

Global variables declared:

static fixed_point_t load_avg

This global variable maintains the load_avg as described in the assignment description

Fields were added to the thread struct:

int nice

The nice field stores the current nice value of the thread.

fixed_point_t recent_cpu

This stores a value that approximates how much the thread has been using the CPU recently

---- 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	A
12	12	0	0	60	61	59	B
16	12	4	0	60	60	59	A
20	16	4	0	59	60	59	B
24	16	8	0	59	59	59	A
28	20	8	0	58	59	59	B
32	20	12	0	58	58	59	C
36	20	12	4	58	58	58	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?

The specification did not specify

- whether recent_cpu is to be updated before or after updating the priorities.
  In both the table and implementation, we update recent_cpu before updating the priority.
- whether we should yield when the current thread's priority becomes equal to the priority of another thread

In both the table and implementation, we yield in case that there is another thread that is strictly greater

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

If too much time is consumed to calculate load_avg, recent_cpu, and priority then running thread would take much time than expected .that would increase its recent_cpu value and decreases its priority because of overhead .thus if the cost of scheduling inside interrupt context will decrease 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?

Given extra time we could maintain a variable to stores highest priority between all threads to find next thread to run faster instead of looping through all threads to find max priority

>> 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?

We used fixed point numbers to represent recent_cpu and load_avg to provide a layer of abstraction for fixed point.
