```
+---------------------+
|        CSE 521      |
| PROJECT 1: THREADS  |
|   DESIGN DOCUMENT   |
+---------------------+
```

---- GROUP ----
>> Fill in the names and email addresses of your group members.
Rahul Reddy Talatala <rtalatal@buffalo.edu>
Charan Kumar Nara <cnara@buffalo.edu>
Sai Nikhil Somepalli <ssomepal@buffalo.edu>

---- 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.

References:
https://www.cs.princeton.edu/courses/archive/spr96/cs333/java/tutorial/java/threads/priority.html
https://inst.eecs.berkeley.edu/~cs162/sp20/static/sections/section5-sol.pdf

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)static struct list blocked_list;
This list is used to store all threads that are currently in a blocked or sleep state, waiting for a specific time to elapse before being woken up.

2) int64_t sleep_until;
This member variable in the struct thread represents the tick until which the thread can sleep. It is used to determine when a thread should wake up from sleep.

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

The present thread waits until a specific number of timer ticks have elapsed. This operation involves turning off interrupts determining the tick suspending the thread and subsequently reactivating interrupts.

- The purpose of this function is to check if the provided ticks parameter is greater than 0. If it is not, the function does nothing.
- The function ensures that interrupts are enabled when timer_sleep() is called by using the ASSERT macro.
- Interrupts are disabled, and the current interrupt level is stored.

- The current thread's sleep_until attribute is set to the specified number of ticks, indicating when the thread should wake up.
- The current thread is blocked, meaning it is added to the list of sleeping threads and will not be scheduled for execution until it is unblocked.
- Interrupts are re-enabled to the previous level.

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

To minimize `timer_interrupt` handler time, the global tick counter is efficiently updated, `thread_tick()` is called for thread management, MLFQS operations (if enabled) are performed efficiently, and threads are awakened using `thread_foreach()`. By optimizing these operations, the system responds promptly to timer interrupts, maintaining efficient performance.

Additionally, to reduce the duration within the timer handler, the timer_sleep() function briefly turns off interrupts while determining the end tick and sending the thread to sleep. This helps avoid context switches and decreases the workload, on the handler.

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

To prevent race conditions caused by threads invoking timer_sleep() at the time interrupts are temporarily disabled during crucial code sections where shared data structures are being accessed or altered. This safeguards, against threads attempting to access and modify the data concurrently.

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

To prevent issues, with race conditions that may arise if a timer interrupt happens while calling timer_sleep() it is important to implement synchronization methods like disabling interrupts during parts of the code. This helps maintain the systems stability and prevents any conflicts, between the timer handler and the timer_sleep() function.

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

The reason, for selecting this design is its efficiency in managing thread sleep. Wake operations according to timer ticks. By keeping a list, of blocked threads and utilizing the sleep_until variable to monitor sleep durations the system can precisely decide when threads need to wake up without squandering CPU resources. This methodology surpasses designs by reducing the duration spent in the timer handler and preventing race conditions through appropriate synchronization methods.

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. int64_t sleep_until; => Variable to store the tick until which the thread can sleep.
2. int base_priority; => Variable to store the base priority of the thread
3. int priority; => Variable to store the current priority of the thread
4. struct list locks_holdingOn; => List of locks the thread is holding.
5. struct lock *lock_waitingFor; => The lock the thread is waiting for.
6. int nice; => Integer to store the nice value of a thread

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

The following variables are used to track priority donation:

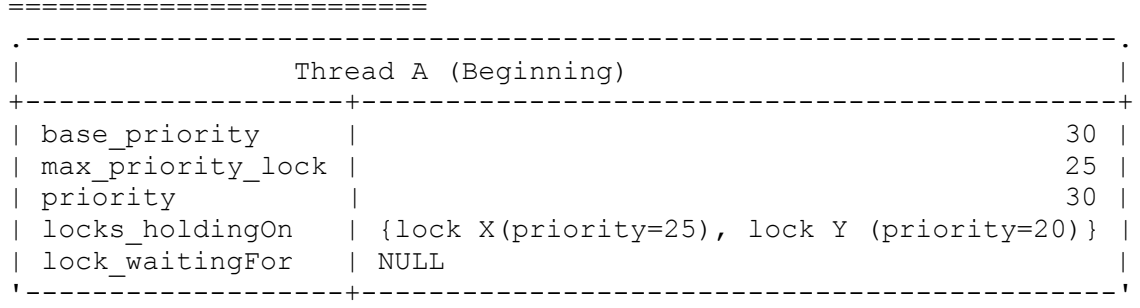
- priority: The current priority of the thread.
- base_priority: The priority the thread was initially assigned.
- priority: The current priority of the thread.
- locks_holdingOn: A list of locks currently held by the thread, sorted by the maximum priority of the lock.
- lock_waitingFor: A pointer to the lock that the thread is waiting for, if any.

When a thread acquires a lock, it may need to donate its priority to another thread holding a lock that it needs. This process can result in nested priority donation chains, where a thread donates its priority to another thread, which in turn donates its priority to yet another thread, and so on.

Let’s understand the priority donation with the help of the following example:

Thread A has priority 30 and holds Lock X and Lock Y.
Thread B has priority 20 and holds Lock Z.
Thread C has priority 35 and is waiting for Lock Z.

Step 1: Initial State



```
=========================
.-------------------------------------------------------------------.
|                Thread B (Beginning)                               |
+------------------+------------------------------------------------+
| base_priority    |                                             20 |
| max_priority_lock|                                             35 |
| priority         |                                             20 |
| locks_holdingOn  | {lock Z(priority=35)}                          |
| lock_waitingFor  | NULL                                           |
'------------------+------------------------------------------------'

=========================
.-------------------------------------------------------------------.
|                Thread C (Beginning)                               |
+------------------+------------------------------------------------+
| base_priority    |                                             35 |
| max_priority_lock|                                           NULL |
| priority         |                                             35 |
| locks_holdingOn  | NULL                                           |
| lock_waitingFor  | {lock Z(priority=35)}                          |
'------------------+------------------------------------------------'
```

Step 2: Thread C tries to acquire Lock Z

Now Thread C which is a higher priority is waiting on Lock Z which is held by Thread B. So Thread B receives a priority donation from Thread C, raising its priority to 35.

```
=========================
.-------------------------------------------------------------------.
|                Thread A                                           |
+------------------+------------------------------------------------+
| base_priority    |                                             30 |
| max_priority_lock|                                             25 |
| priority         |                                             30 |
| locks_holdingOn  | {lock X(priority=25), lock Y (priority=20)}    |
| lock_waitingFor  | NULL                                           |
'------------------+------------------------------------------------'

=========================
.-------------------------------------------------------------------.
|                Thread B                                           |
+------------------+------------------------------------------------+
| base_priority    |                                             20 |
| max_priority_lock|                                             35 |
| priority         |                                             35 |
| locks_holdingOn  | {lock Z(priority=35)}                          |
| lock_waitingFor  | NULL                                           |
'------------------+------------------------------------------------'

=========================
.-------------------------------------------------------------------.
|                Thread C                                           |
+------------------+------------------------------------------------+
| base_priority    |                                             35 |
| max_priority_lock|                                           NULL |
| priority         |                                             35 |
| locks_holdingOn  | NULL                                           |
```

lock_waitingFor	{lock Z(priority=35)}

Step 3: Thread A priority adjustment

As Thread B has Lock Z, it inherits the priority of Thread C, which is 35. Because Thread A holds Lock Y, which requires a higher priority than its current priority of 30, Thread A's priority is boosted to 35 to prevent priority inversion.

Thread A	
base_priority	30
max_priority_lock	25
priority	35
locks_holdingOn	{lock X(priority=25), lock Y (priority=20)}
lock_waitingFor	NULL

Thread B	
base_priority	20
max_priority_lock	35
priority	35
locks_holdingOn	{lock Z(priority=35)}
lock_waitingFor	NULL

Thread C	
base_priority	35
max_priority_lock	NULL
priority	35
locks_holdingOn	NULL
lock_waitingFor	{lock Z(priority=35)}

Step 4: Nested Priority Donation Chain

Thread A indirectly affected Thread C by receiving a priority donation and releasing Lock Y's priority, which enabled Thread B to acquire Lock Z. This scenario creates a nested priority donation chain where the priority donation of one thread indirectly influences the priority of another thread through shared locks.

Thread A	
base_priority	30
max_priority_lock	25

priority	35
locks_holdingOn	{lock X(priority=25), lock Y (priority=20)}
lock_waitingFor	NULL

==========

Thread B	
base_priority	20
max_priority_lock	35
priority	35
locks_holdingOn	{lock Z(priority=35)}
lock_waitingFor	NULL

==========

Thread C	
base_priority	35
max_priority_lock	NULL
priority	35
locks_holdingOn	NULL
lock_waitingFor	{lock Z(priority=35)}

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

When a lock, semaphore or condition variable is needed the top priority thread, in the queue gets to go. This ensures that when a resource is ready the scheduler picks the thread waiting for it.

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

- A thread attempts to acquire a lock.
- When another thread already holds the lock the current thread shares its priority with the holder.
- If the holder is waiting for a lock it also becomes a donor initiating a process.

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.
- The lock has been unlocked.
- The holders priority is returned to its priority (before the donation).
- In case the unlocked lock was donated nested donation guarantees that the priority of the waiting thread is adjusted accordingly.

---- 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?

When examining the queue, for the priority there could be a race condition if changes are made to the priority donors list simultaneously. To prevent this issue it is essential to disable interrupts when accessing parts of the code. Additionally employing a lock can help mitigate this race condition.

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

The system uses the priority check feature to insert threads into the list in order of their priorities. This method organizes threads from highest to lowest priority making scheduling more effective. When inserting threads their priorities are compared to ensure that higher-priority threads are positioned near the top of the list enabling selection, by the scheduler. This strategy enhances system responsiveness without requiring data structures or algorithms.

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. fixed_point_t load_average; => a fixed point variable to store the load average
2. int nice; => Integer to store the nice value of a thread
3. fixed_point_t recent_cpu_usage; => Fixed point number to store the recent cpu usage

---- 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			priority			Thread to run	Notes
	A	B	C	A	B	C		
0	0	0	0	63	61	59	A	Initial state, all threads have the same priority
4	4	0	0	62	61	59	A	A continues running due to higher priority
8	8	0	0	61	61	59	A	A continues running due to higher priority
12	12	0	0	60	61	59	B	B's nice value (1)

								comes in, lowering its priority
16	12	4	0	60	60	59	B	B runs because its recent_cpu is lower than A's now
20	12	8	0	60	59	59	A	B's recent_cpu reaches A's, scheduling switches back
24	16	8	0	59	59	59	A	A keeps running due to equal priority with B
28	20	8	0	58	59	59	C	C's nice value (2) allows it to run as both A and B's recent_cpu are high
32	20	12	4	58	58	58	B	B gets control back after C's timeslice

>> 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, there were ambiguities in the scheduler specification that made the values in the table uncertain. These are the ambiguities:

1. Updating recent_cpu vs. priority:
There is an uncertainty in the order of operations when updating thread priorities based on recent CPU usage. If we update priorities before recalculating A's priority could lead to a situation where A appears less favorable (due to the incremented `recent_cpu`) even though it's still the running thread. This might cause unnecessary context switching. If we update priorities after recalculating A's priority could potentially give A an unfair advantage by not fully reflecting its recent CPU usage in the initial priority calculation for other threads.

So we have calculated priorities after updating `recent_cpu` for all threads. This ensures a more accurate reflection of recent CPU usage in determining thread favorability.

2. Scheduling with equal priorities:

There is an uncertainty in how the scheduler handles situations where multiple threads have the same priority because the scheduler uses "greater than" for priority comparison when selecting a thread. It's unclear if this includes "equal to" or not.

However, the PintOS scheduler does not yield the running thread if its priority becomes equal to another thread's priority. Thus, reducing unnecessary context switches.

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

By performing most thread priority updates outside interrupts, the scheduler avoided potentially expensive calculations within the critical interrupt context. This helps minimize interrupt latency and improve overall system responsiveness.
Only the selection of the next thread to run, which likely involves a small constant number of operations, happens in the interrupt context. This minimizes the time the scheduler spends within interrupts, allowing for faster handling of external events.

---- 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?

Our design had a clear separation of concerns like interrupt and non-interrupt context for performance. We've implemented prioritization based on recent CPU usage for fairness and responsiveness.

The major disadvantages of our design is that there is a potential overhead by frequently updating nice values. As of now we are only considering recent cpu usage for determining priority.Additionally, there is a lack of context on preemption handling.

If we had more time we could have implemented lazy updating of the nice values and implemented dynamic 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 did not create any additional helper functions for the fixed point calculations as the given functions were sufficient to carry out the MLFQS calculations

Contributions

1. Rahul Reddy Talatala (rtalatal@buffalo.edu) - All 3 Phases test cases and Design Doc
2. Charan Kumar Nara (cnara@buffalo.edu) - All 3 Phases test cases and Design Doc
3. Sai Nikhil Somepalli (ssomepal@buffalo.edu) - All 3 Phases test cases and Design Doc

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?