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

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.

/* Added wake_up_time to track when a sleeping thread should wake up */
struct thread {
    int64_t wake_up_time; // Stores the tick at which the thread should wake up.
};

struct list_elem sleep_elem; // Used to link the thread into the sleep list.

static struct list sleep_list; // A list of sleeping threads maintained in sorted order by wake-up time.

---- ALGORITHMS ----

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

timer_sleep() retrieves the current tick and calculates the wake-up time.
The thread is added to a sleeping list sorted by wake-up time. The thread is blocked until the timer interrupt wakes it up.
The timer interrupt checks sleeping threads and unblocks those whose wake-up time has passed.

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

We sorted the sleeping threads list, so only the current element will need checking.
Threads are then only unblocked if their wake-up time has arrived, thus avoiding unnecessary, time-wasting loops.
Lastly, the list is updated efficiently to ensure minimal operations within the interrupt handler, thus minimizing the amount of time spent in the timer interrupt handler.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer_sleep() simultaneously?

We disabled interrupts during critical updates to ensure atomic operation.
This way, the list operations only happen when interrupt is disabled, thus avoiding race conditions.

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

When timer_sleep() is executing, interrupts are disabled during its critical sections. Which means that even if a timer interrupt is generated, the interrupt handler will not preempt the ongoing update of the sleep list. As a result, the sleep list modifications are performed atomically, which ensures that the timer interrupt handler sees a consistent state and race conditions are avoided.

---- RATIONALE ----

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

Based on the recitation, we got the idea to store sleeping threads in a list to keep track.
This implementation ensures efficient wake-up checks. Disabling interrupts helped us in preventing race conditions.
Another design we considered (but did not try implementing) was iterating over all threads, but this would not be very efficient.
Thus, we think our strategy was safe and reasonable 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.

```
// In 'struct thread' (thread.h):
int priority;                // Effective scheduling priority (after donations).
int real_priority;           // Base priority (without donations).

int arrival_order; /* For FIFO ordering of threads with equal priority */

struct list_elem donation_elem; // List element for linking into the donations list.

struct list donations;         // List of received priority donations.
struct lock *waiting_lock;     // Lock the thread is blocked on (if any).
struct thread *donation_target; // Thread to propagate donations to (for nested donations).
```

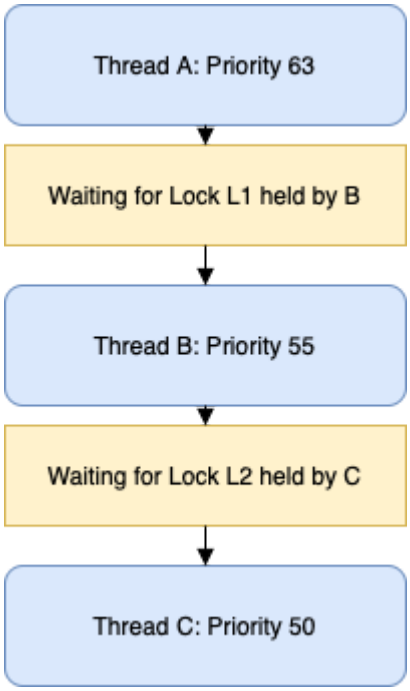
```
// Donation record (thread.h):
struct donation {
   struct list_elem elem;       // List element for donations list.
   struct thread *donor;        // Thread donating priority.
   struct lock *lock;           // Associated lock causing donation.
};

// In (thread.c):
static int next_arrival_order; // Counter used to assign FIFO order for threads with equal priority
```

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

```
// Donation record (thread.h):
struct donation {
   struct list_elem elem;       // List element for donations list.
   struct thread *donor;        // Thread donating priority.
   struct lock *lock;           // Associated lock causing donation.
};
```

Each thread that can receive donations maintains a list (named donations) of these donation records. When a thread for example Thread A needs a lock that is held by another thread Thread B, Thread A creates a donation record and adds it to Thread B's donations list. If Thread B, in turn, is waiting on another lock held by Thread C, it propagates the donation via its donation_target pointer, this ensures that the high priority flows through the chain of dependencies.



Here, Thread A (with priority 63) is waiting for Lock L1, which is held by Thread B. Thread B receives a donation record from A and boosts its effective priority. Thread B is then waiting for Lock L2, held by Thread C, so the donation (or the boosted priority) is propagated to Thread C. This chain of donation ensures that Thread C's effective priority increases and enables it to complete its work and release the locks, thus it will prevent priority inversion.

---- ALGORITHMS ----

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

We maintain the waiters in a priority-sorted list (e.g., in sema->waiters) using a comparator that orders threads by effective priority (and by arrival order to break ties).

This will ensure that the highest-priority thread is positioned at the front of the list. When a resource becomes available, we unblock the front thread. This will ensure that the highest-priority waiting thread is always served first.

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

1. If lock is held (lock->holder != NULL):
    a. Set current thread's waiting_lock and donation_target.
    b. Call donate_priority():
       Creates a struct donation record.
       Adds it to the holder's donations list.
       Recursively updates priorities up the chain via donation_target.
2. Blocks on sema_down().
3. On unblock: Clears waiting_lock, sets itself as holder.

Nested Handling:
Donations propagate recursively via donation_target until reaching a thread with no pending locks.

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

1. Removes lock from holder's locks_held list.
2. For non-mlfqs mode calls remove_donations() to clear donations associated with the lock.
3. Resets holder's priority to real_priority and updates effective priority via update_priority().
4. Calls sema_up(), unblocking the highest-priority waiter (if any).
5. If the holder no longer holds any locks, its donation_target is cleared.

---- 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?

---- RATIONALE ----

Potential Race:
When thread_set_priority() is called when a donation is occurring, there can be a race between updating the base priority and donation mechanism recalculating effective priority

Our Implementation:

1. Disable Interrupts: We disable interrupts during priority updates which ensure that critical sections are executed atomically.
2. Two-Phase Update: First updates base priority (real_priority), and then recomputes effective priority via update_priority()
3. Immediate Preemption Check: Ensures higher-priority threads run immediately

Why it avoids the problem:
The critical section is <10 instructions and is too short for deadlock risk. Our solution maintains consistency between base/effective priorities and handles concurrent donation scenarios safely

Why Not Use a Lock?

1. Using a lock might lead to deadlock if priority donation is involved
2. Disabling interrupts is safer for this brief operation
3. The critical section is very short (just a few assignments)

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

Superior Aspects of our Design:

1. Our solution provides Nested Donation Handling:
    a. The donation_target chain ensures correct propagation through multiple locks
    b. Avoids priority inversion in complex locking scenarios

2. Our solution has efficient Priority Updates:
    a. Only recalculates priorities when necessary (donations/releases)
    b. Maintains both base and effective priorities separately

3. Clean Separation:
    a. Donation records are tied to specific locks
    b. Easy cleanup when locks are released

Alternatives that we considered:

1. Full priority inheritance. Doing this would permanently boost the holder's priority and it will be more complex to revert after lock release
2. No Nested Donations: This is simpler but couldn't handle multi-lock scenarios. This will also fail in some cases.

Our design has the right balance between correctness and performance and is handling all edge cases. The use of donation records and recursive propagation matches exactly how nested priority donation should work in a real OS.

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.

struct thread additions/changes:

- int nice: Thread's nice value (-20 to 20) affecting priority calculation
- fixed_point_t recent_cpu: Thread's recent CPU usage for MLFQS
- int64_t wake_up_time: Time when a sleeping thread should wake up
- struct list_elem sleep_elem: For maintaining sleep list

Global variables:

- fixed_point_t load_avg: System load average (0 initially)
- static int next_arrival_order: Counter for FIFO ordering of same-priority threads
- static struct list sleep_list: List of sleeping threads ordered by wake 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			priority			Thread to run
	A	B	C	A	B	C	
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**	**B**
20	**12**	**8**	**0**	**60**	**59**	**59**	**A**
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**	**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?

The specification doesn't define when exactly priority recalculations happen. We resolved this by:

1. Updating priorities every 4 ticks (TIMER_FREQ%4 == 0)
2. Updating recent_cpu every second (timer_ticks() % TIMER_FREQ == 0)
   This matches our implementation's behavior.

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

- Interrupt context: Minimal work (just incrementing ticks and checking sleep list)
- Outside interrupt: Heavy computations (priority/recent_cpu updates)
  This division prevents long interrupt handlers while ensuring timely scheduling decisions.

---- 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:

1. Sleep List Efficiency: We have implemented ordered insertion in timer_sleep(), which maintains the sleep list in ascending wake-up time using list_insert_ordered().
2. Scheduler Separation: We separated MLFQS and priority scheduling by using a thread_mlfqs flag.

- Priority Scheduler: Uses a combination of real_priority and donation records to handle priority donations and prevent inversion.
- MLFQS: Uses recent_cpu and nice values to adjust priorities dynamically based on CPU usage.

Benefit: No branch mispredictions in hot paths (each mode uses dedicated logic).

3. Load Distribution: We spread out heavy computations to avoid spikes:
    - recent_cpu updates: Every tick (running thread only)
    - Priority recalculations: Every 4 ticks
    - Decay: Every 100 ticks

Disadvantages:

1. Fixed-point math is functional but slow. Right now, we use generic macros like fix_div(), which compile to bulky code. We could:
    - Use x86's native 32-bit multiply/divide instructions via inline assembly
    - Replace divisions with multiplications by precomputed reciprocals (e.g., fix_div(x,4) → fix_mul(x, 0x4000))
2. The sleep list uses a simple linked list, so inserting threads takes O(n) time. We can try switching to a binary heap would cut down the insertion complexity to O(log n).

Improvements if given more time:

1. Optimize fixed_point inline operations: Exploring inline assembly or precomputed reciprocals to speed up fixed point arithmetic
2. Improving Sleep list efficiency: We can replace linked list with binary heap to reduce insertion time and improve performance when load is high.
3. Adaptive time slices: Let TIME_SLICE shrink when the system is busy (high load_avg) to improve responsiveness, like Linux's dynamic tick feature.
4. Cache donations: Right now we recompute the max donation every time. If we store it and invalidate it only when donations change, we can save cycles.
5. Prefetch sleep list nodes: Allocate them in a contiguous memory pool to reduce cache misses during insertion.

>> 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 implemented fixed-point math as a set of macros because:

1. Performance: The scheduler executes arithmetic in timer interrupts (every 1ms). Macros eliminate:
    - Function call overhead (~5-10 cycles per call)
    - Stack manipulation
    - Register save/restore

2. Precision Control
   We adopted Q17.14 format (17 integer, 14 fractional bits) because:
   - Range: ±131,072, which covers all MLFQS values
   - Precision: 4 decimal digits, which is sufficient for recent_cpu decay
   - Efficiency: Fits in 32-bit int with minimal overflow

3. Safety: The fixed_point_t wrapper prevents mixing regular and fixed-point numbers by accident.
4. Clarity: Macros provide additional clarity as the formulas look almost like normal math.