OS lab2 report - Team 16

I. Design document questions

Part.1 Alarm Clock

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.

Ans:

int64_t blocked_t: This is a newly added member in the struct **thread**, used to keep track of how many timer ticks the thread should remain blocked.

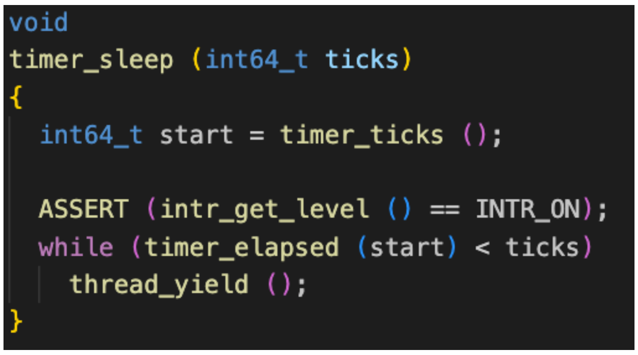
static struct list sleep_list: This is a static list added in thread.c that keeps track of threads currently sleeping, i.e., those with blocked_t > 0.

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

Ans:

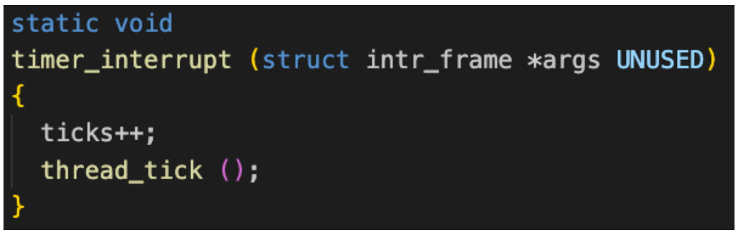
Initially (busy waiting)

In the original implementation of **timer_sleep()**, the system relies on a busy-waiting mechanism. This approach blocks the entire system if the number of elapsed ticks is less than the number of ticks that need to be waited.



```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

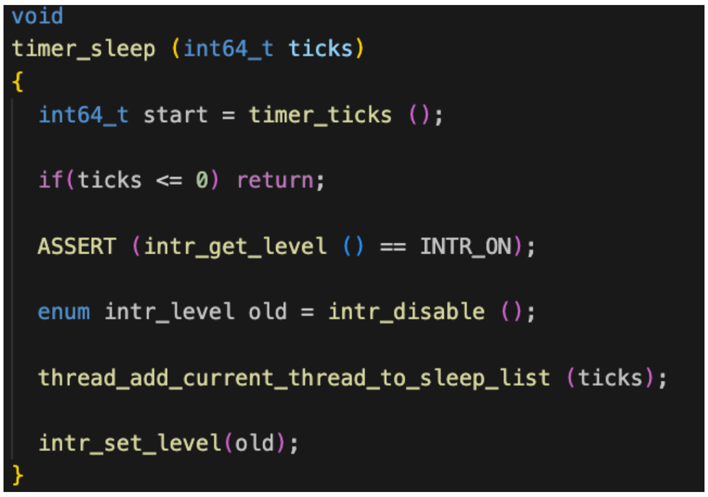
Within **timer_interrupt()**, the timer interrupt handler simply increases the global tick count to keep track of time progression.



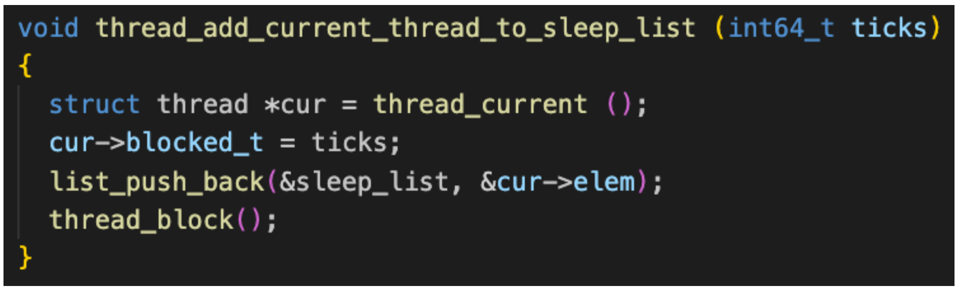
```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
}
```

After modified (avoid busy waiting)

In **timer_sleep()**, we begin by checking whether the number of ticks to sleep is greater than zero. If so, the current thread adds itself to the sleep list and blocks, giving up the CPU. During this process, interrupts are disabled to avoid race conditions.

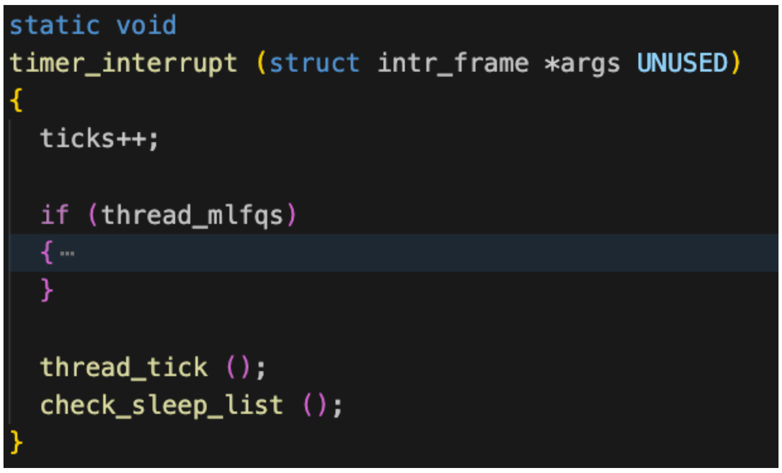


```c
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  if(ticks <= 0) return;

  ASSERT (intr_get_level () == INTR_ON);

  enum intr_level old = intr_disable ();

  thread_add_current_thread_to_sleep_list (ticks);

  intr_set_level(old);
}
```

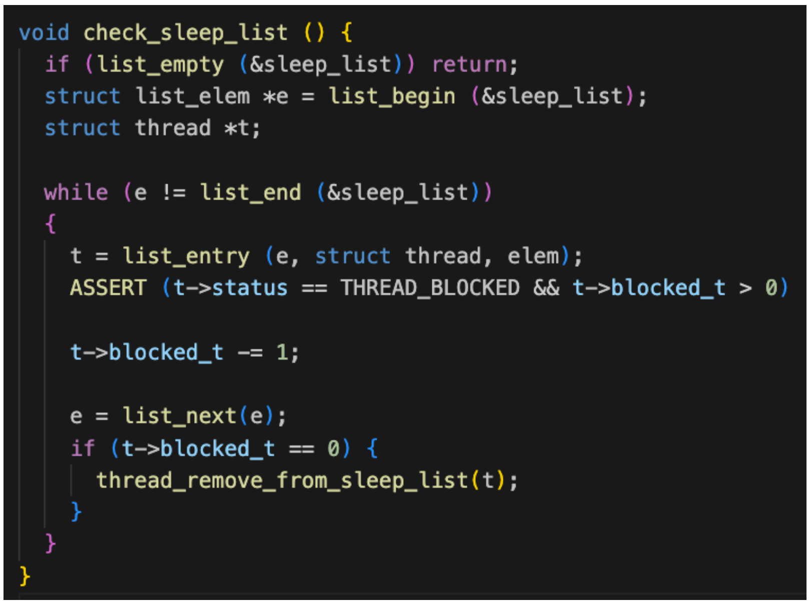


```c
void thread_add_current_thread_to_sleep_list (int64_t ticks)
{
  struct thread *cur = thread_current ();
  cur->blocked_t = ticks;
  list_push_back(&sleep_list, &cur->elem);
  thread_block();
}
```

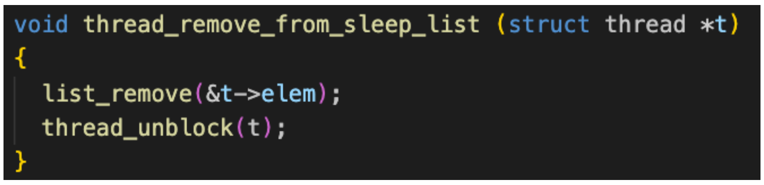
The timer interrupt handler, which is triggered regularly based on TIMER_FREQ, increments the global tick count with each tick. During each interrupt, it examines the sleep list and decrements the remaining sleep time for each thread by one tick. If a thread's **blocked_t** reaches zero — indicating it's time to wake up — it is removed from the sleep list and unblocked. This mechanism allows threads to be woken up efficiently without busy waiting.



```c
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;

  if (thread_mlfqs)
  {…
  }

  thread_tick ();
  check_sleep_list ();
}
```



```
void check_sleep_list () {
  if (list_empty (&sleep_list)) return;
  struct list_elem *e = list_begin (&sleep_list);
  struct thread *t;

  while (e != list_end (&sleep_list))
  {
    t = list_entry (e, struct thread, elem);
    ASSERT (t->status == THREAD_BLOCKED && t->blocked_t > 0)

    t->blocked_t -= 1;

    e = list_next(e);
    if (t->blocked_t == 0) {
      thread_remove_from_sleep_list(t);
    }
  }
}
```



```
void thread_remove_from_sleep_list (struct thread *t)
{
  list_remove(&t->elem);
  thread_unblock(t);
}
```

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

Ans:

To reduce the overhead of the timer interrupt handler, sleeping threads are tracked in a dedicated **sleep list**. On each timer interrupt, the handler only examines threads in this list. If a thread's scheduled wakeup time has arrived, it is unblocked and removed from the list. By focusing solely on sleeping threads, the handler achieves runtime proportional to the number of threads currently in the sleep list, rather than all threads in the system.

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

Ans:

Race conditions are avoided when multiple threads call timer_sleep() simultaneously by disabling interrupts during the critical section where each thread inserts itself into the sleep list and blocks. By temporarily turning off interrupts, the thread ensures that no other thread or interrupt handler can interleave operations that might corrupt the sleep list or cause inconsistencies. After completing the insertion and blocking, the original interrupt level is restored. This approach guarantees atomicity without requiring explicit locking, which is crucial because the timer interrupt handler also accesses the sleep list and cannot acquire locks.

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

Ans:

Race conditions during a call to timer_sleep() are avoided by disabling interrupts before accessing shared data structures such as the sleep list. When interrupts are disabled, any incoming timer interrupt is deferred until interrupts are re-enabled. This guarantees that no timer interrupt can preempt the insertion of the thread into the sleep list, ensuring atomicity and preventing any inconsistencies between the thread sleep operations and the timer interrupt handler's traversal of the sleep list.

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

Ans:

We adopted this design for its efficiency. Initially, we considered checking all threads on every timer interrupt. However, that approach scales poorly, as its overhead grows proportionally with the total number of threads, resulting in degraded performance. In contrast, our current design only examines threads in the sleep list, meaning performance is influenced solely by the number of sleeping threads. As a result, an increase in the total number of threads does not impact our design.

Part.2 Priority Scheduling

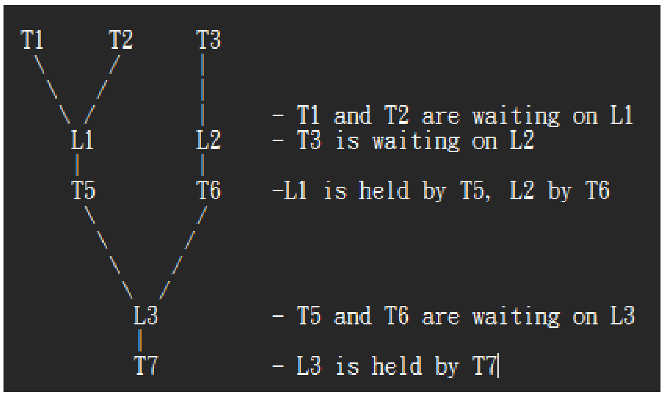
B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

Ans:

- **struct lock:**
    - **struct list_elem elem:** Used to insert the lock into a list of locks for priority donation.
    - **int max_priority:** Stores the highest priority among threads waiting on the lock for priority donation.
- **struct thread:**
    - **int original_priority:** Remembers the thread’s base priority before any donations occurred.

- **struct list locks:** List of locks the thread currently holds; used in donation logic.
- **struct lock *waiting_lock:** Lock the thread is currently waiting on; used to trace donation chains.

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

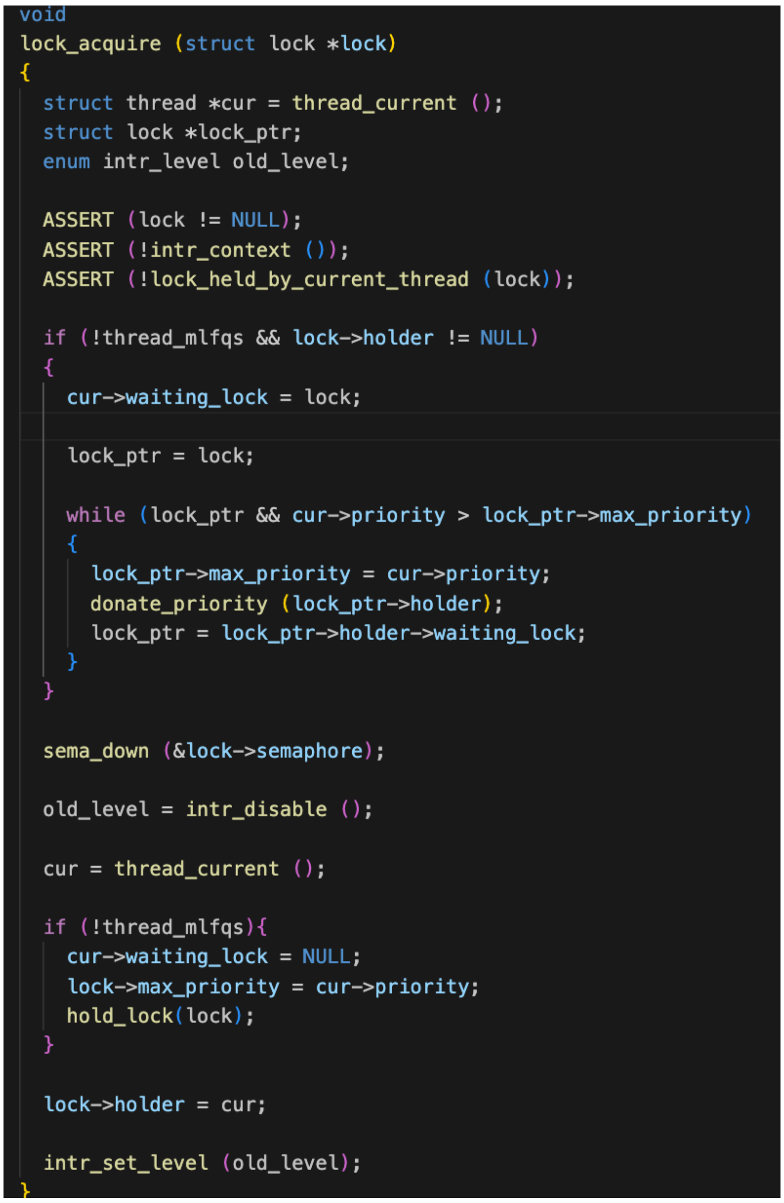


```
T1      T2      T3
 \      /       |
  \    /        |
   \  /         |     - T1 and T2 are waiting on L1
    L1          L2    - T3 is waiting on L2
    |           |
    T5          T6    -L1 is held by T5, L2 by T6
     \          /
      \        /
       \      /
        \    /
          L3          - T5 and T6 are waiting on L3
          |
          T7          - L3 is held by T7
```

Figure. Data structure to track priority donation

In the figure, three threads (T1, T2, and T3) are involved in priority donation through a series of locks.
T1 and T2 are waiting on lock L1*(T1->waiting_lock = T2->waiting_lock = L1)*.
T3 is waiting on lock L2*(T3->waiting_lock = L2)*.
Lock L1 is currently held by thread T5*(L1 in T5->locks)*
Lock L2 is held by thread T6*(L2 in T6->Tlocks)*.
Both T5 and T6 are, in turn, waiting on lock L3*(T5->waiting_lock = T6->waiting_lock = L3)*, which is held by thread T7*(L3 in T7->locks)*.
Since T1, T2, and T3 have higher priorities and are blocked by locks held by T5 and T6, their priorities must be donated transitively through the lock holders. Priority donation flows from T1 and T2 through L1 to T5, and from T3 through L2 to T6, and then both T5 and T6 donate to T7 through L3.
This nested donation ensures that the thread at the bottom of the dependency chain (T7) inherits the highest priority, allowing it to run promptly and eventually release the locks, thus unblocking all dependent threads.

Below is our code implementation, in the while loop, the priority is recursively donated through waiting lock and lock holder. The function **donate_priority()** will then deal with the interior priority update of the lock holder, for instance, the sorting of its own locks.



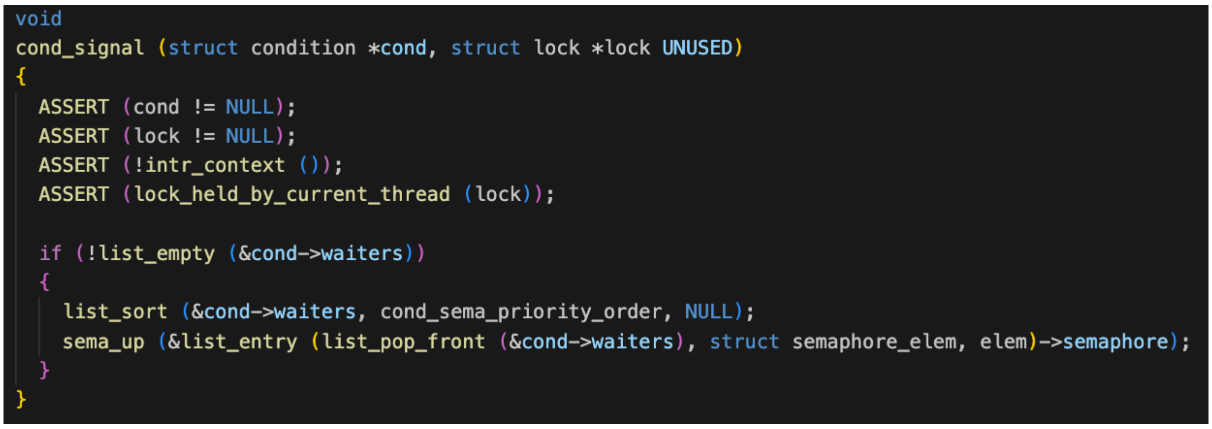
```c
void
lock_acquire (struct lock *lock)
{
  struct thread *cur = thread_current ();
  struct lock *lock_ptr;
  enum intr_level old_level;

  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  if (!thread_mlfqs && lock->holder != NULL)
  {
    cur->waiting_lock = lock;

    lock_ptr = lock;

    while (lock_ptr && cur->priority > lock_ptr->max_priority)
    {
      lock_ptr->max_priority = cur->priority;
      donate_priority (lock_ptr->holder);
      lock_ptr = lock_ptr->holder->waiting_lock;
    }
  }

  sema_down (&lock->semaphore);

  old_level = intr_disable ();

  cur = thread_current ();

  if (!thread_mlfqs){
    cur->waiting_lock = NULL;
    lock->max_priority = cur->priority;
    hold_lock(lock);
  }

  lock->holder = cur;

  intr_set_level (old_level);
}
```

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

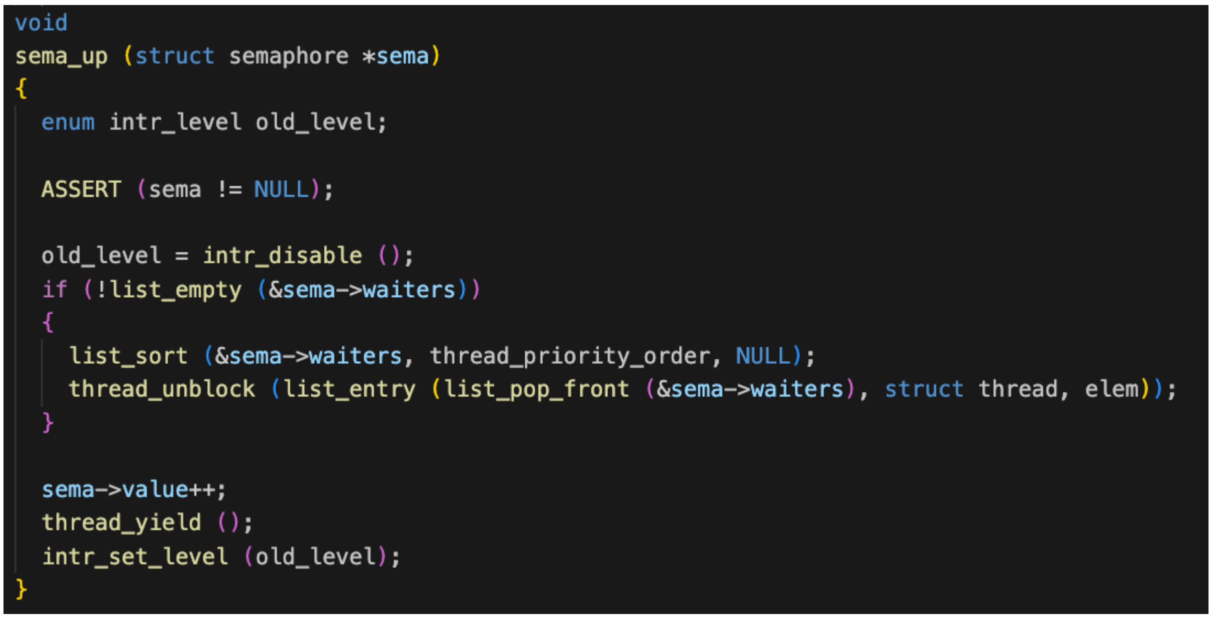
<u>Ans:</u>

To ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first, threads are inserted into the waiters list in priority order. When a thread needs to be added to a waiters list (for example, when calling sema_down, lock_acquire, or cond_wait), it is inserted into the list using list_insert_ordered, maintaining the list sorted by thread priority. Consequently, when the lock, semaphore, or condition variable becomes available, the highest priority thread is always at the front of the waiters list and will be unblocked first.

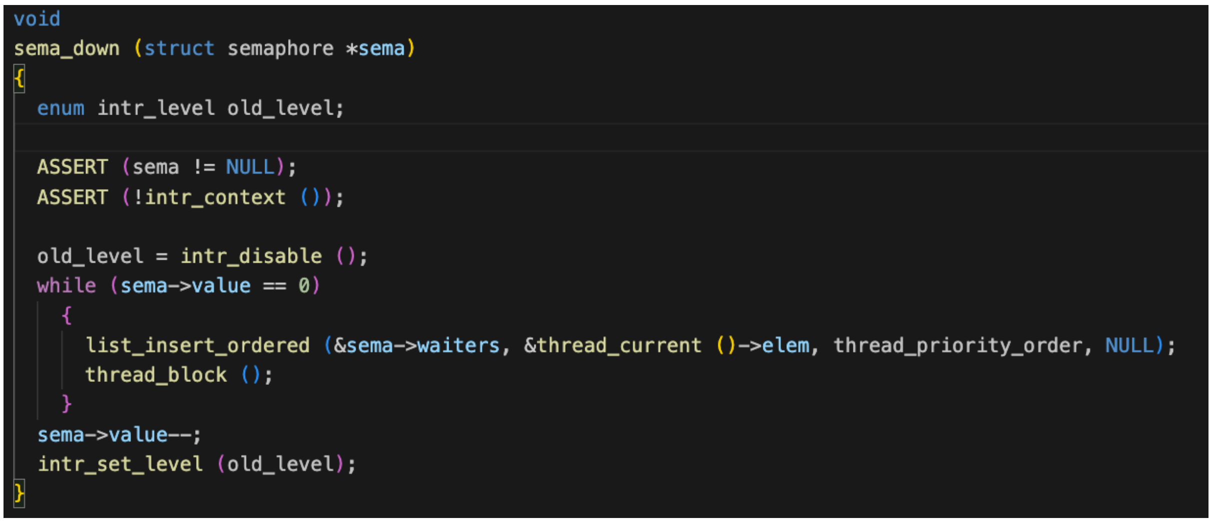
code implementation



```c
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters))
  {
    list_sort (&cond->waiters, cond_sema_priority_order, NULL);
    sema_up (&list_entry (list_pop_front (&cond->waiters), struct semaphore_elem, elem)->semaphore);
  }
}
```



```c
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
  {
    list_sort (&sema->waiters, thread_priority_order, NULL);
    thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
  }

  sema->value++;
  thread_yield ();
  intr_set_level (old_level);
}
```



```c
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_insert_ordered (&sema->waiters, &thread_current ()->elem, thread_priority_order, NULL);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

B4: Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?
Ans:
When a thread calls lock_acquire() and finds that the lock is already held by another thread, priority donation occurs.The calling thread first sets its **waiting_lock** field to point to the lock it is trying to acquire. It then compares its effective priority with the max_priority of the lock and donates its priority to the lock holder if its own priority is higher. If the lock holder is itself waiting on another lock, the donation propagates recursively along the **waiting_lock** chain. This ensures that all threads along the dependency chain inherit the highest priority.

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

(1) The releasing thread removes the lock from its list of held locks and updates its effective priority, since the donation associated with the released lock is no longer valid.
(2) The lock's holder field is then cleared.
(3) Sema_up() is called on the lock's underlying semaphore, which increments the semaphore value and unblocks the highest-priority thread waiting for the lock.
(4) The releasing thread yields the CPU if there is a higher-priority thread ready to run.

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?
Ans:
A potential race in thread_set_priority() occurs if another thread preempts and modifies the ready list while the current thread is in the process of changing its priority. This could result in inconsistent or corrupted scheduling behavior, where threads are inserted into the ready list with outdated priorities.

To avoid this race, our implementation disables interrupts (intr_disable()) before modifying the thread's base priority, priority, and interacting with the ready list. Disabling interrupts guarantees atomicity, ensuring that no other thread or interrupt handler can interfere during this critical section.

Using a lock would not be appropriate here because acquiring a lock could cause the current thread to sleep, which is not allowed when modifying scheduler-critical structures like the ready list. Therefore, disabling interrupts is the correct and necessary approach for race avoidance in this context.

B7: Why did you choose this design? In what ways is it superior to another design you considered?
Ans:
We chose this design because it cleanly supports nested priority donation, which is essential for correctly handling chains of blocked threads. By adding a **waiting_lock** pointer to each thread and a **max_priority** field to each lock, we allow priority donation to propagate recursively through the lock-holder dependency chain.

The advantages of our design are modular and efficient. Each thread tracks only the lock it's currently waiting on (waiting_lock), and each lock knows the maximum priority of any thread waiting on it (max_priority), so donation is localized and efficient. The thread's effective priority is recalculated dynamically based on the original priority and the max priority of all held locks, allowing priority revocation to happen naturally when locks are released.

Compared to other designs, such as maintaining a full donation graph or history, our approach is simpler and less memory-intensive, avoiding the complexity of managing and updating a full graph structure. To be more specific, our design requires less bookkeeping, as the list of locks a thread holds (**thread->locks**) and **lock->holder** are already available. This design fits naturally with the existing list structure used for scheduling and lock management in PintOS.

Overall, our design strikes a balance between simplicity, correctness, and performance, and integrates seamlessly with the existing thread and lock framework.

Part.3 Advanced Scheduler

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.

Ans:

- typedef int **fixed_point_number**: defines a type used for fixed-point arithmetic, where values are represented by shifting integers by a fixed number of bits. According to the documentation, we use 14 bits
- struct **thread**
    - **int nice:** an integer that stores the thread's nice value, which affects its priority in the BSD scheduler.
    - **fixed_point_number recent_cpu:** is a fixed-point variable that tracks the amount of CPU time recently used by the thread, as required for implementing the BSD scheduler.

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:

Ans:

Following the MLFQS priority rule written in thread_mlfqs_update_priority():
priority = PRI_MAX - (recent_cpu/4) - (nice*2), Where PRI_MAX=63 in pintos
If the highest-priority queue contains multiple threads, they run in "round robin" order.

timer	recent_cpu			priority			thread
ticks	A	B	C	A	B	C	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?

Ans:

There are two ambiguities in the scheduler specification.

(1) The specification does not clearly state whether recent_cpu should be updated before or after updating priorities.
our solution: we update recent_cpu first and then update the priorities, which ensures that the new priority values reflect the most recent CPU usage.

(2) It is not specified whether the scheduler should preempt the current thread when its priority becomes equal to another thread's priority.
our solution:We follow the round-robin policy as used in the BSD scheduler. When multiple threads share the highest priority, the scheduler selects the one that has been waiting the longest (i.e., least recently run) to preempt the currently running thread. This ensures fairness among threads with equal priority.

This behavior matches the expected outcomes observed in the scheduling table and ensures consistency in thread scheduling decisions.
The above matches the behavior of our scheduler.

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

Ans:

Our implementation performs the majority of MLFQS scheduling updates (e.g., updating recent_cpu, load_avg, and thread priorities) inside the timer interrupt handler, which is called at every tick. Although the majority of MLFQS updates are done in the timer interrupt handler, we also handle priority changes on-demand — for example, when nice is set via **thread_set_nice()**, the thread's priority is immediately recalculated, and the thread yields the CPU if needed. This balances periodic background updates with responsiveness to dynamic changes.
Our approach provides consistency and predictability. Running scheduling updates in the timer interrupt ensures consistent timing, which aligns well with MLFQS's time-based scheduling model. However, there are some drawbacks. Our approach will increase interrupt latency. This method will traverse all_list inside the interrupt context (especially once per second), which leads to increased interrupt latency, impacting the responsiveness of the system. This is particularly problematic in high-load scenarios with many threads.

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?
Ans:
One limitation in our current design is the implementation of the sleep list. Although it is more efficient than traversing all threads, it still requires scanning the entire sleep list, which may introduce significant overhead as the number of sleeping threads increases. A possible optimization would be to maintain the sleep list in an ordered manner based on each thread's wake-up tick. This way, we would only need to check the head of the list during each timer tick, reducing the operation to O(1) in the best case.

Another weakness we observed is the poor performance under the mlfqs test. This suggests that our scheduler may not be handling multi-level feedback queue scheduling as efficiently as expected, possibly due to suboptimal priority recalculation or update mechanisms.

If we had more time to refine our design, we would first improve the sleep list structure to reduce overhead, perhaps by using a priority queue or a sorted linked list. Additionally, we would conduct profiling on the mlfqs test to identify bottlenecks and fine-tune our priority update strategy to improve overall scheduler performance. For instance, we may want to implement other kinds of data structures such as binary heap to enhance the performance of our design.

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?
Ans:
To implement fixed-point arithmetic for the BSD scheduler (MLFQS), we created a dedicated abstraction layer using a set of **inline functions** and a **typedef** for the **fixed-point** type. We did so for **clarity, readability, avoiding repetition and bugs, and ease of maintenance**.
By abstracting fixed-point operations into named functions like **fixed_add_fixed()** or **fixed_multiply_fixed()**, the core scheduling code in thread.c, timer.c, and synch.c becomes easier to read and reason about. For example, instead of writing complex

shifting and casting operations inline, we can just call fixed_multiply_fixed(x, y) and immediately understand its purpose.
Fixed-point arithmetic involves a lot of bit-shifting and type-casting, which are easy to get wrong. Centralizing the logic ensures consistent behavior and reduces the risk of mistakes — especially in multiplication/division where **int64_t** is needed to prevent overflow.
If the scaling factor (currently 1 << 14) or internal representation needs to change later (e.g., higher precision), we only have to modify the implementation in fixed-point.h, without touching the logic in the rest of the kernel.
Performance via static inline
We used static inline functions rather than macros to avoid common macro pitfalls (like multiple evaluation), to preserve type safety, and to let the compiler optimize the function calls (inlining them during compilation) with zero runtime overhead.

We avoided using macros or more complex abstractions like structs or classes for the following reasons:
C Simplicity: PintOS is written in C, and the kernel codebase is simple and low-level. Introducing struct-based fixed-point types or opaque types might complicate integration with existing kernel code and lead to unnecessary overhead.
Efficiency: Plain int with inline functions gives us fast, predictable performance without abstraction penalties.

Part.4 Survey Questions

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?
<u>Ans:</u>
The three tasks are appropriate for students to implement. However, we found the assignment somewhat tedious. Additionally, debugging was challenging because printf becomes unusable when interrupts are disabled.

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?
<u>Ans:</u>
We believe that all three tasks offered valuable insights into operating system design, although they were quite time-consuming.

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?

Ans:

It would be helpful if future students could receive more support from TAs on certain aspects of the assignment. For example, offering additional debugging tips or giving a live demonstration on how to use debugging tools when interrupts are disabled would be extremely beneficial for completing the assignment.

Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

Ans:

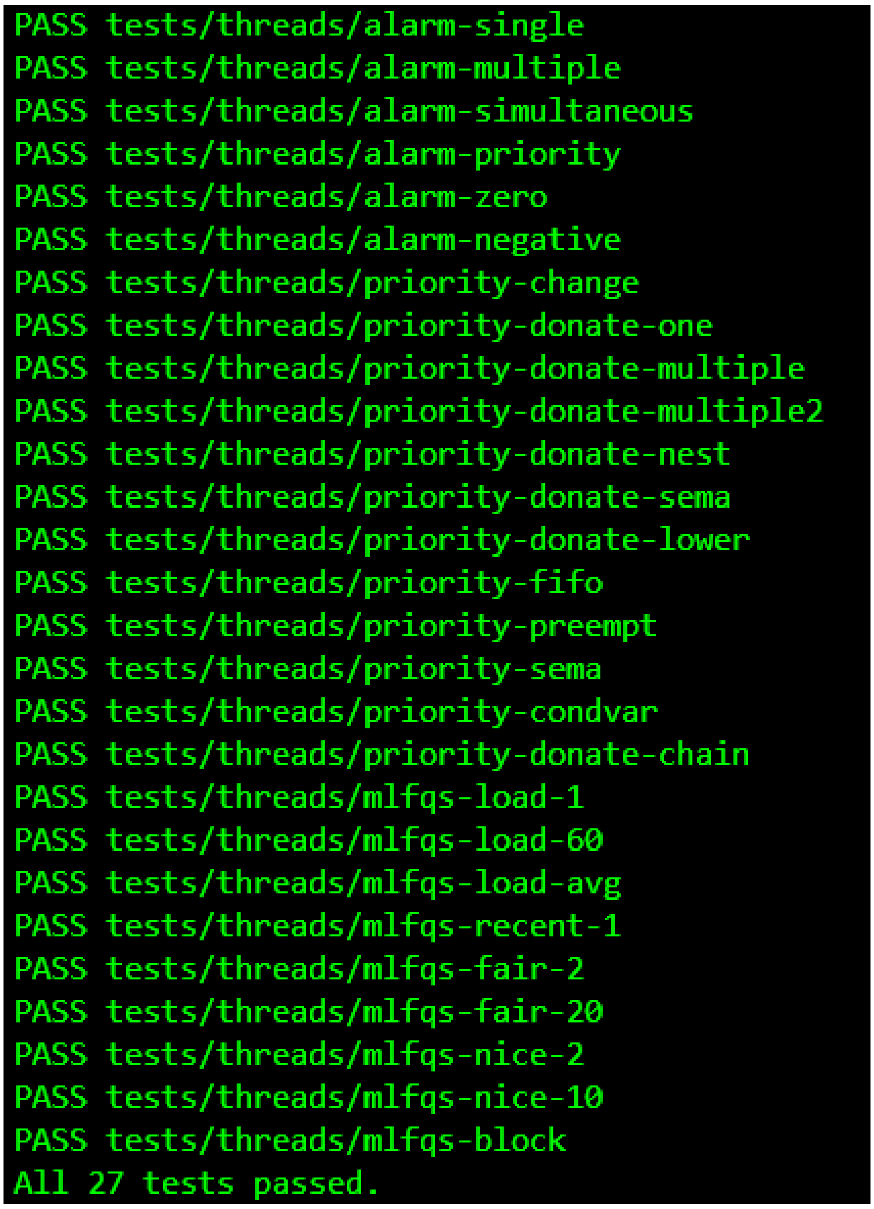
We believe it would be helpful if the TAs could provide an optimal or reference solution after the assignment deadline. This would greatly enhance our understanding of the operating system's structure.

Any other comments?

Ans:

No.

II. Test results



```
PASS tests/threads/alarm-single
PASS tests/threads/alarm-multiple
PASS tests/threads/alarm-simultaneous
PASS tests/threads/alarm-priority
PASS tests/threads/alarm-zero
PASS tests/threads/alarm-negative
PASS tests/threads/priority-change
PASS tests/threads/priority-donate-one
PASS tests/threads/priority-donate-multiple
PASS tests/threads/priority-donate-multiple2
PASS tests/threads/priority-donate-nest
PASS tests/threads/priority-donate-sema
PASS tests/threads/priority-donate-lower
PASS tests/threads/priority-fifo
PASS tests/threads/priority-preempt
PASS tests/threads/priority-sema
PASS tests/threads/priority-condvar
PASS tests/threads/priority-donate-chain
PASS tests/threads/mlfqs-load-1
PASS tests/threads/mlfqs-load-60
PASS tests/threads/mlfqs-load-avg
PASS tests/threads/mlfqs-recent-1
PASS tests/threads/mlfqs-fair-2
PASS tests/threads/mlfqs-fair-20
PASS tests/threads/mlfqs-nice-2
PASS tests/threads/mlfqs-nice-10
PASS tests/threads/mlfqs-block
All 27 tests passed.
```