+------------------------+

| CS 212 |

| PROJECT 1: Threads |

| DESIGN DOCUMENT |

+------------------------+

---- GROUP ----

Karim Tarek Ibrahim
Hussein Khaled
Mohamed Anwar
Nagui Mostafa
Abdullah Taman

: task1 ALARM CLOCK

---- DATA STRUCTURES ----

:A1

we use this list to put in it all threads that call :(static struct list sleeping_list) .fun sleep

int64_waik_up_time → in struct thread and it use to know in what time the .thread will unblock

---- ALGORITHMS ----

.()A2-Briefly describe what happens in a call to timer_sleep

when any thread call fun timer_sleep(ticks) we check the number of ticks that the thread will sleep if it -ve or zero we return else we take this ticks and add it to the current time to know in what time will unblock this thread and then push this thread in list that is sorted with respect to the wake up time and then we block this thread. Each tick we check the list and if it is the .time to wake up thread we unblock it

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

Ans: we added thread to sorted list (wake up time) and search in it to the .first thread that wake up time > current time

---- SYNCHRONIZATION ----

A4-How are race conditions avoided when multiple threads calltimer_sleep() ?simultaneously

Ans: By disabling interrupts by built in fun inter_disable() and inter_set_level() and by using thelist_insert_ordered() function, threads are inserted into the sleep queue in order of their wake-up time, guaranteeing that the first thread to wake up is always the one with the earliest wake-up .time

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

()Ans: By disabling interrupts by built in fun inter_disable() and inter_set_level

---- RATIONALE ----

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

Ans: Because this approach optimizes the reading procedure of the sleeping threads, it reads the to-wake-up threads in O(1), which reduces the time consumed in thread-tick. However, the making-sleep, i.e. writing- procedure takes O(N) where n is the number of sleeping threads, but again reducing the .effort exerted in thread-tick is the top priority

---

Task 2: Priority scheduling

- **Data structures:**

In thread.h:

Added to struct thread:

```
int real_priority;
struct list Owned_locks;
struct lock *Waited_on_lock;
```

real_priority : used to store the actual priority of the thread

Owned_locks: a list is used to keep track of all the possible donating threads by keeping in record the locks the current thread is owning now

Waited_on_lock: is used to keep track of the lock the thread is waiting on

In synch.h:

Added to struct lock:

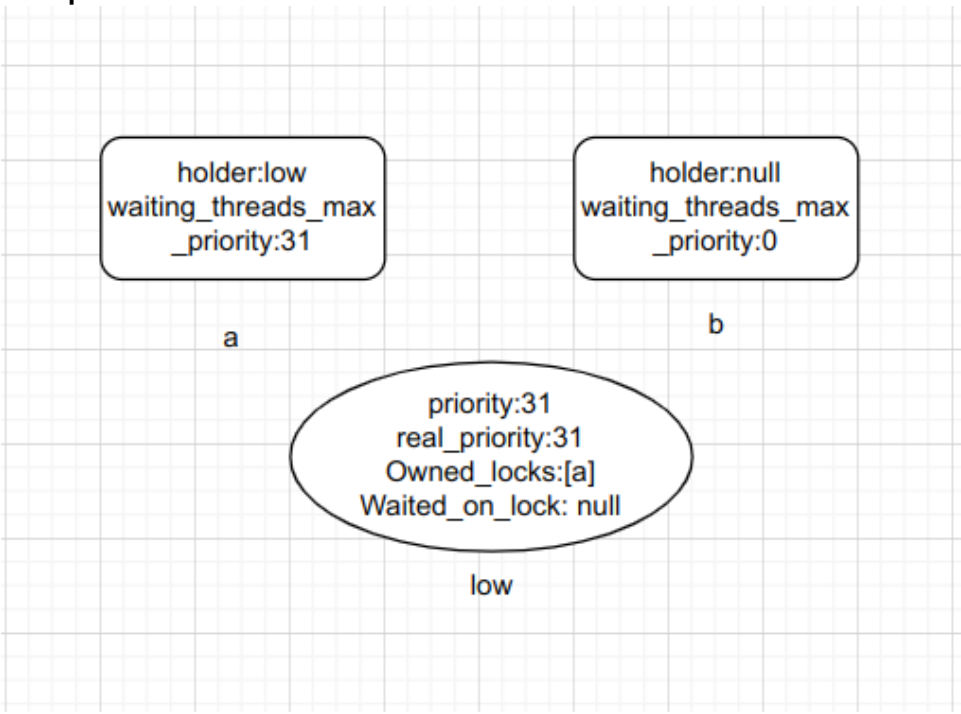
```
struct list_elem elem;
int Waiting_threads_max_priority;
```

elem: so it can be used in list

Waiting_threads_max_priority: to keep track of the maximum priority of the threads waiting on the lock

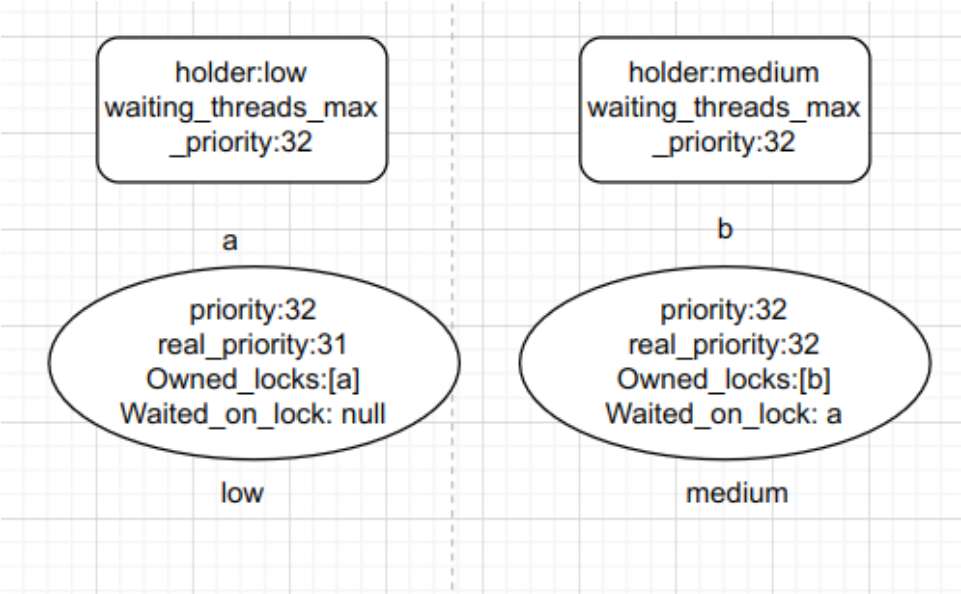
Nested donation:

1. The low thread initiates two locks a and b and then acquire lock a

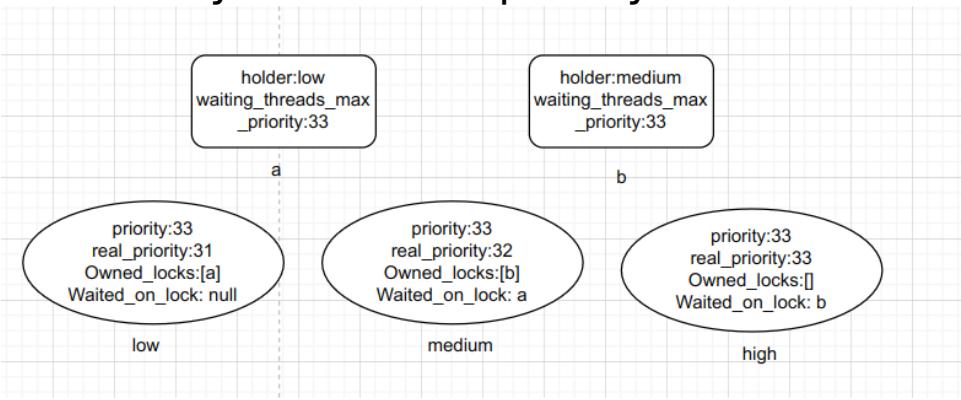


2. A medium thread priority is created and acquires lock b then lock a so it now holds b and waiting for low thread to release a so it donates its priority to

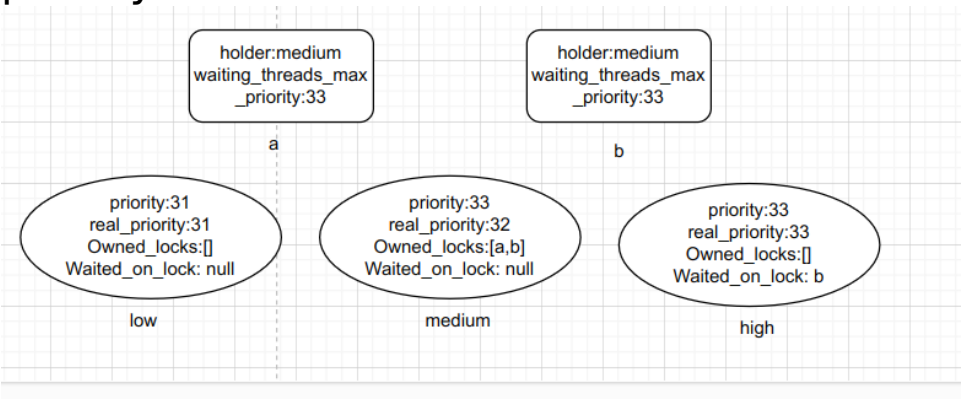
low thread



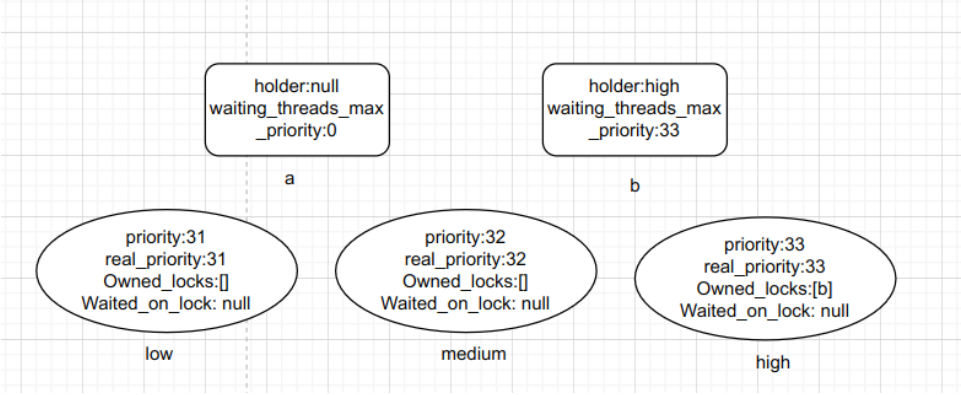
3. A high thread priority is created and acquires lock b but it is blocked waiting for medium thread to release it so it donates its priority to medium thread and recursively donates its priority to low thread



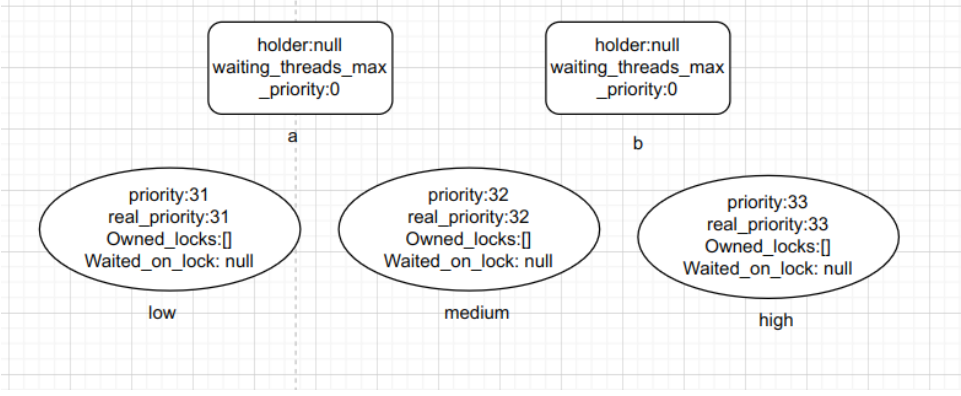
4. The low thread releases lock a and restores its actual priority then medium thread now holds lock a and b



5. The medium thread then release lock a then lock b and restores its actual priority and now high thread hold lock b



6. The high thread then releases lock b



- **Algorithms:**

Choosing the right thread: we need to ensure that the ready queue is ordered by priority from high to low, so we maintain this when inserting thread in the ready queue. We insert thread in 3 functions: init_thread, thread_unblock, thread_yield so instead of using list_push_back() we will use list_insert_ordered() and pass comparator function to it. When setting new thread priority or creating a new thread we call thread yield if we found that the priority is smaller than new priority or the priority of the newly created thread.

Changing thread's priority: when setting a new priority, we check if there is not a priority donation on this thread or if the donated priority is less than the new priority then we set the new priority and call thread_yield().

Lock acquire: When lock_acquire() function is called, the thread checks if there is a holder for the lock or not. If there is a holder, then we wait on this lock and check if lock priority is smaller than thread priority then updates lock priority and holder priority and recursively go through the other lock that holder is waiting on. After calling sema_down(), the lock is available, put the lock in the Owned_locks of the current thread and update priority of both the lock and thread.

Lock release: When calling lock_release() function, the lock is removed from the Owned_locks of thread and updating priority of thread and then setting holder of lock to null and then sema_up() is called

Donating priority: call thread_update_priority() and then check if the thread modified in the ready state, then we rearrange the ready queue.

Updating priority: We check our Owned_locks list. If it is empty, then the priority will be the real priority. Otherwise, we get the lock that has max priority and then the priority will be the largest between lock priority and real priority.

Semaphore priority scheduling: We will turn the semaphore queue into priority queue as when calling sema_down(), the waiting thread will be inserted in the semaphore->waiters using list_insert_ordered() function so the waiters list become sorted from high to low based on priority. When calling sema_up(), we will get the front of the waiters list and unblock it and then call thread_yield() so as if the unblocked thread has higher priority than the running thread.

Condition variable priority scheduling: We will turn the condition variable queue into priority queue by sorting the list each time calling cond_signal() and passing cond_cmp_priority as a comparator.

- **Synchronization:**

There will be a potential race when calling thread_set_priority() function as many threads can set the priority of another thread at the same time. However, because the operation of changing priority and yielding the CPU is under the condition of disabling the interrupt, these operations could be considered atomic. Then there will not be such a problem.

- **Rationale:**

This design is quite simple as it depends on sorting the lists according to their priority. However, it leads to some overhead as inserting threads or locks in order takes more time than pushing it back and leaving it unsorted. In priority donations, I don’t think there was a better way to solve this problem considering the nested donations and multiple donations.

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.

In struct thread in thread.h:

```
int nice;
struct real recent_cpu;
```

- nice value is used to determine the priority of a thread. The priority of a thread is inversely proportional to its nice value, i.e., a lower nice value corresponds to a higher priority.
- recent_cpu is the amount of time a thread has spent on the CPU recently. In the context of the BSD scheduler, it is used as one of the parameters for determining a process's priority.

In thread.c:

```
/* System load avgerage */
static struct real load_avg;
```

load avg, often known as the system load average, estimates the average number of threads ready to run over the past minute and is used as one of the parameters for determining a process's priority in the BSD scheduler.

All the above values were used to calculate the priority of a thread using these eqns:

recent_cpu = (2*load_avg) / (2*load_avg + 1) * recent_cpu + nice

load_avg = (59/60) *load_avg + (1/60) *ready_threads

priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)

---- 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	1	2	63	61	59	A
4	4	1	2	62	61	59	A
8	7	2	4	61	61	58	B
12	6	6	6	61	59	58	A
16	9	6	7	60	59	57	A
20	12	6	8	60	59	57	A
24	15	6	9	59	59	57	B
28	14	10	10	59	58	57	A
32	16	10	11	58	58	56	B
36	15	14	12	59	57	56	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?

- When priorities are equal like in the table above it cannot be determined whether to leave the current thread running or to make it ready and run the other thread with equal priority.
- The rule for resolving this conflict was to always yield the running thread and make the other one with equal priority run instead.
- Yes, it matches that behavior.

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

In interrupt context aka timer_ interrupt we made sure to not make many operations as to not affect the values of recent_ cpu and load_avg and more importantly the priority

Timer_interrupt had 3 main operations which are:

- Inc_recent_cpu()
- Thread_update_recent_cpu_and_load_avg()
- Thread_update_priority_mlfqs_all()

Inc_recent_cpu():

Runs in constant time as it only increments the recent cpu value of thread t.

Thread_update_recent_cpu_and_load_avg():

Runs in linear time as it iterates over each thread and updates recent_cpu and priority but it won't halt the system as often as it runs every 100 ticks and it performs basic calculations which doesn't take as much cpu time.

Thread_update_priority_mlfqs_all():

Runs in linear time and is probably the most time-consuming function as it runs every 4 ticks and iterates through each thread to update its priority.

---- 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?

Pros:

- Scheduling was done using one queue only which is the pintos ready_list
- Insertions in ready_list were ordered based on the priority of the threads to ensure that the thread with max priority is on top of the list.
- In try_yield function it is required to check if it is needed to reschdule and to do that as fast as possible insertion with order was required to make it in constant order operation
- Every 4 ticks it was required to update values of priorities of all threads, so it must be sorted after to maintain the property in the above points.
- To keep that property, we need to make sure each insertion in the system is ordered to not alter the max priority position in the list and thread_unblock and thread_yield was altered to do just that.
- In thread_set_nice priority of the current thread is updated based on new nice value and then it needs to try to reschedule using the above algorithm in constant time which is very efficient

Cons:

- Since we needed to sort ready_list each time we update the priority of all threads it's a time-consuming operation as it runs every 4 ticks.
- We needed to update recent_cpu of every thread also but it isn't as bad as the previous one as it only runs every 100 ticks.

Refinements:

- A refinement that would improve the design would be using multiple queues for each priority so we wouldn't need to sort or insert inorder and everything would run in constant time.

>> 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?

I decided to implement the fixed-point arithmetic functions as a set of functions that operate on a custom struct **real**, which holds the fixed-point number as an integer. I chose this implementation approach because it allowed for encapsulation and abstraction of the fixed-point arithmetic operations, making them easier to use and more modular.

In addition, by defining a struct to hold the fixed-point number, I was able to easily define functions to convert between integer and fixed-point representations, as well as provide functions for performing arithmetic operations such as addition, subtraction, multiplication, and division.

My implementation follows the conventions for fixed-point arithmetic by using a fixed number of bits for the fractional part (14 bits in this case) and defining a constant **f** as 2^14. This allows for efficient multiplication and division using shifts and avoids the overhead of floating-point arithmetic.