---- GROUP ----

>> Fill in the names and email addresses of your group members.

Douglas Johnston <johnston.5douglas@gmail.com> <douglasj>
Heng Xu <xu.he@husky.neu.edu> <hengx>
Wanting Jiang <jiang.wa@husky.neu.edu> <wanting>

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.

struct thread in thread.h
-has new field, int64_t wake_time, keeping track of how many ticks it should be woken up in.

enum thread_status in thread.h
now includes a sleeping state, for when the thread is asleep

thread_sleeping
-ordered list of threads that are asleep rather than blocked or ready.

sleeping_lock
-a lock need to be acquired before when calling timer_sleep(), make sure no multiple thread can call timer_sleep() simultaneously.

---- ALGORITHMS ----

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

A call to timer_sleep() checks that interrupts are on, and if given a number of ticks higher than 0, will call thread_sleep(). This disables interrupts, sets the wake time of the current thread to the current timer value + the given number of ticks,

and then inserts the thread into the sleeping list, sleeping list is ordered, sorted by the thread's wake_time. Finally, it calls schedule to select a thread to run, and then reactivates interrupts.

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?
Timer interrupt handler needs to do two things, thread_wake() and thread_tick(). We are trying to minimize the amount of time spent in the thread_wake(). As we mentioned before, thread_sleeping list is ordered, it is sorted by the thread's wake_time. The first thread in this list always has the smallest wake up time. So every time we call thread_wake(), we will go through the sleeping list, when we meet a thread whose wake_time is bigger than current tick, that means left threads in this list should still be sleeping, so break the loop and return. Thus, we needn't go through all the threads in the list and reduce the time spent on the thread_wake().

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
As mentioned in A1, we added a global struct sleeping_lock to avoid these race conditions.
When multiple threads call timer_sleep() simultaneously, only one thread can hold sleeping_lock and change its status to sleeping. Thus, race conditions are solved.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
Interrupts are disabled during the call to thread_sleep before timer_sleep makes any modifications, so a second thread cannot interfere with the results of the first thread in the middle of modifications, regardless of when it is called.

---- 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.

struct thread in thread.h
-has new int original_priority, to store the original priority not including donated priority
-has new struct lock *required_lock, storing a pointer to the lock that the thread is waiting for.
-has new struct list holding_locks, storing a list of locks held by the thread.

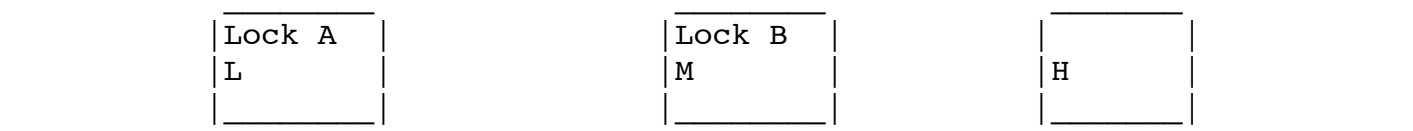
struct lock in synch.h
-has new int lock_ priority, storing the priority of the highest-priority thread waiting for the lock. If there is no thread waiting for the lock, this is set to PRI-MIN.
-has new struct list_elem elem, allowing locks to be connected in a list.

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

Priority donation is kept track of by four fields: int priority in struct thread, int original_priority in struct thread, lock_required * in struct thread, int lock_priority in struct lock.
priority stores the current priority of the thread, and is altered when priority is donated. It is referenced to select which thread should receive a lock.
original_priority stores what the priority of the thread would be without any priority donation occurring, and is referenced to reset the threads priority when returns donated priority.
lock_required stores which lock the thread is waiting on. It is referenced to check which lock, if any, the thread should donate priority to.
lock_priority is equal to the highest priority among threads waiting for the lock, and is referenced to determine if the priority donation should occur. If lock_priority is higher than the priority of the holder thread of the lock, then the priority of the holder thread should be raised to be equal to that of the lock_priority.

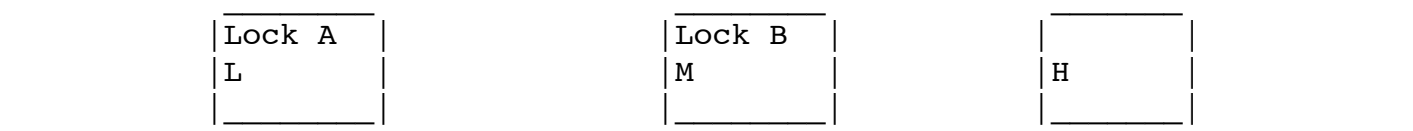
initial state: three threads.
Lowest priority thread L, its priority is 1, it is holding lock A, ;

Medium priority thread M, its priority is 2, it is holding lock B, but waiting for lock A;
Highest priority thread H, its priority is 3, it needs lock B.



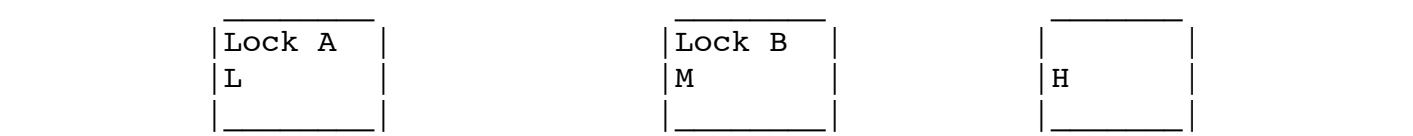
Priority of Locks
Lock A: PRI_MIN Lock B: PRI_MIN

state 1: Thread H donates its priority to Lock B and thread M.



Priority of Locks:
Lock A: PRI_MIN Lock B: 3

final state: Thread M donates its priority to Lock A and thread L.



Priority of Locks:
Lock A: 3 Lock B: 3

---- ALGORITHMS ----

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

When inserting threads into waiters list of a lock, semaphore or condition, we changed the list_push_back() to list_insert_ordered(), and created several compare function. So the threads waiting for a lock or semaphore are now stored in an ordered list for the appropriate semaphore, where the list is ordered according to their priority. So, popping off the first entry in the waiting list for a given lock or semaphore will return the highest priority thread waiting for it, and this is what is done by next_thread_to_run() when a lock or semaphore becomes available.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?
1.First check whether the lock holder is NULL or not.
2.If the lock holder is not NULL:
     (1) Set the lock as current thread's required_lock.
     (2) Compare the lock's priority with current thread's priority.
     (3) If lock's priority is smaller than current thread's priority, that means we might need to donate current thread's priority to the lock holder.
     (4) So set lock's priority to current thread's priority.
     (5) Call thread_update_priority() to update the lock holder's priority.
     (6) If current lock's holder also has a required lock, go back to step (2), otherwise break the loop, priority donation is finished.
3. Call sema_down().
4. If sema_down() successfully, current thread can hold the lock, call thread_get_lock().

The detail information about nested donation has been described in B2.

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.
First, interrupts are disabled to prevent races from occurring. Then set the holder field of the lock to be NULL. After that, the lock is removed from its current list (the list being the set of locks held by a given thread), indicating that the lock is no longer held by the thread that held it up to this point. The releasing thread then updates its priority, removing any donated priority that was given to it by the higher priority thread waiting for the lock. Finally, sema_up() is called to unblock the highest priority thread waiting for the lock, and

interrupts are then reenabled.

---- 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?

thread_set_priority, called in thread A and given int new_priority as an argument, first alters the original_priority of the current thread t to be equal to new_priority. After that, it checks if t's priority is less than new_priority, and checks whether t holds any locks. If either of these checks evaluates to true, it sets t's priority to new_priority. If the priority is changed, we should call thread_yield() to reschedule it.
So, if another thread B donates priority to t in the middle of thread_set_priority, after it checks whether to alter t's priority but before the alteration occurs, then the change to priority made by B could be overwritten by the change made by A. This is avoided by disabling interrupts before A makes any changes, preventing B's changes from being overwritten, as it cannot alter A until A has finished making changes.
A lock could be used to prevent this, if we had A hold a lock when making changes to itself, and requiring that lock be held by any thread that would make changes to A's donation. However, this would require additional structure creation and referencing to handle priority donation, lowering performance compared to simply disabling interrupts.

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?
Another design considered was, instead of having a original priority and current priority for each thread, to have only a priority and a pointer or identifier designating a second thread, where the second thread is the thread donating priority to the first. This was decided against due to the higher complexity of incorporating it into the existing design, and because it could require multiple function calls to look at other threads whenever a thread's priority is being examined, degrading performance.