Ryan Hamamoto
rkhamamo
5/9/18
CS111
Professor Harrison

Thread.c/h

- Added elements to thread data structure: it now has lock_wait, which stores the pointer to the lock the thread is waiting for, old_priority, for storing the thread's old priority after priority acceptance from another thread's donation, donation_elem to act as elements to the donation_list of another thread, as well as the thread's donation list itself. These elements are unique to each thread and thus must follow them around in their execution.
- Deleted all the unordered insert sorts from thread_unblock and thread yield, instead replaced with insert ordered so that threads are placed in the ready list in order of priority, facilitated by the ordered_priority function that makes it so that threads with highest priority are at the front of the ready queue. This priority ordered implementation is also true for the donation lists of threads.
- thread_create() was modified to see if the thread currently on the CPU should be kicked off because the element at the front of the ready list has a higher priority. It does this by calling priority_yield(), which I created. Priority_yield() is different than thread_yield() because it only calls thread_yield() if the current thread's priority is less than the thread at the front of the ready queue (if the ready queue is not empty). If that is not the case, nothing happens.
- thread_set_priority() disables interrupts so that thread priority can be safely changed. It utilizes 3 functions: update_priority(), priority_yield(), and donate_priority(). It first updates the current thread's priority, then calls priority_yield() if it is lower than its original priority, and calls donate_priority() if it is higher than its original priority. An important note is that just because priority_yield() and donate_priority() are called does not necessarily mean the thread yielded or donated its priority, its merely a check to see if it needs to happen with the right circumstances (i.e. yield only if the thread's updated priority is now suddenly less than the priority of the thread at the front of the ready list, and donate only if the thread's updated priority is suddenly greater than the thread that holds the lock the current thread is waiting for).
- update_priority() changes the current thread's priority; it first resets its priority to its original state, then checks if it has a thread in its donor list with a higher priority than it, accepting the donation if it does.
- Donate_priority() looks at the current thread's lock_wait element, which is the lock the current thread is waiting for, and checks to see if the lock holder's priority is less than that of the current thread. If it is, the current thread donates its priority to the lock holder. Priority inversion is no longer an issue and donation test now passes.
- Updated init_thread to also initialize the elements I added, setting the lock variable to a null pointer, indicating that it is not yet waiting on a lock, initialize its donation list, setting its old_priority variable to its current priority, since this function is called upon thread creation
- Remove_lock is called by lock_release in lock.c, it increments through the current thread's donation list until it finds the matching lock to the lock inputted into the function. Once it finds it, it removes the element from the list, and leaves.

Most of my errors were a result of negligence, like not having the correct inequality sign in place, forgetting to dereference one of the many pointers correctly. I also did not completely understand the

instruction “you may need to reorder the ready queue, and preempt if a thread with higher priority that the currently executing thread is now waiting” in the thread_set_priority function, and failed the priority_change and priority_fifo tests quite often. Implementing the checks outlined in the thread_set_priority bulletpoint above eventually solved the issue.

Lock.c/h

- Lock_acquire is now implemented so that the current thread sets its lock wait variable to the input lock (called lock), the current thread adds itself to the donation_list of the lock holder (in order of priority of course), all of which only if the lock is held by someone. All lock holdings are updated so that the current thread is in full possession of the lock.
- Lock_try_acquire now disables interrupts, tries to decrement the semaphore, and returns a Boolean indicator that it was either successful or unsuccessful. If successful, changes the lock’s holder element to the current thread, sets the current thread’s lock wait element back to initial value (NULL) because it is no longer waiting to acquire a lock (it just did), return 1.
- Lock_release does the inverse of lock acquire, resets necessary parameters of the lock, updates the priority of the current thread, puts the semaphore back up so that it can be acquired again, etc.

Semaphore.c/h

- Semaphore_down had the while loop modified so that it checks for the need of a priority donation of the current thread, to make sure that the thread about to acquire the semaphore and access the critical section is in fact the one with the highest priority. Originally I had implemented a priority ordered list of the semaphore’s waiters in this function, but they would always be out of order in semaphore_up no matter what. As stated in lecture, it was much easier to just sort the waiters after the fact in semaphore_up, calling list_sort on them. I kept missing the priority-donate-sema test until I resorted them in semaphore_up with list_sort. This may have been a crude implementation, but it works.

Condvar.c/h

- Condvar_wait was not modified for the same reason semaphore_down was not modified, changing the list_push_back call to an ordered priority call is useless because the waiters are always out of order coming out the other end. Calling list_sort after the fact (in condvar signal) is optimal.
- A new sorter ordered_semaphore_priority is implemented similar to ordered_sleep and ordered_priority. It takes the two semaphores, sorts the waiter lists the after-the-fact way (with list_sort), and using ordered_priority so that the highest priority element is at the front of the list, checks the first element in the two semaphore’s waiter list, and returns either 1 or 0 depending on which thread has the highest priority and therefore should go first.