Report for Nwen301 Project 2

Data Structures:

The members of data structures that I changed were only in the header files.

In thread.h, (the thread struct) I added:

struct lock *lockWaitiingOn - the lock on which the thread was currently waiting.

struct semaphore *semaWaitingOn - the semaphore on which the thread was currently waiting.

struct thread *receiver - the thread that received this thread's priority so that I could undo the donate after that thread was finished.

int oldPriority - the priority this thread had before it got donated to.

In synch.h (the semaphore struct) I added:

struct lock *lock - a pointer to the lock that contained this semaphore.

Algorithms:

I introduced new algorithms for donating and undoing donations, in the thread.c file. thread_donate_priority takes two arguments; a 'donor' thread, and a 'receiver' thread. This method then facilitates the donation of 'donor's priority to 'receiver'. thread_undo_donate, only takes an argument of 'receiver' and undoes the donation by reverting the receiver's priority back to its original priority. I also added a comparator method to compare two list elements and return the one with the highest priority.

I also modified a whole lot of algorithms in thread.c. I initialized all of the new struct members (as mentioned above), and then, in thread_create, I added a check for yielding the current running thread if it has a lower priority than the newly created thread. In thread_block I added a check for the priority donation, to check if the current thread is waiting on something (lock or semaphore) and if it is, then get the thread currently holding the lock/semaphore, and call the thread_donate_priority method to donate the current thread's priority to the holder. In thread_unblock I sort the ready queue, so that the highest priority is always the first to be popped off the front, and I undo a priority donation, if this has occurred. In thread_yield I, again, sort the ready queue. In thread_set_priority if a donation has occurred, then I set 'oldPriority' to the updated priority level, otherwise I just set 'priority' to that level. I then pop off the first entry in the ready queue and compare it to the new priority, running the thread with the higher one. In thread_get_priority I simply return the priority of the current running thread.

Ben Watson, 300317744

In synch.c I introduced new comparative algorithms for comparing the priorities of two list elements (compareElements) and comparing the priorities of two semaphore list elements (compareSemaElements). The former simply compares the two priorities of the elements, but the latter first gets the two lists of waiters from the elements, orders them to find the highest priority of each, and then calls compareElements on the first element in each list and returns that.

I also, in the while loop of sema_down, push the current thread's element onto the sema waiting queue, and sort the queue so that the first element in it has the highest priority. I then set the '*waitingOn' field of the current thread to the current semaphore/lock. After the while loop, I yield to the highest priority thread. In sema_up I, again, sort the waiters list, and set the *waitingon values to null. I then yield to the highest thread again. I sort the waiters list in cond_wait and cond_signal as well, so that condition variables are also prioritised.

Synchronisation and other:

I avoided race conditions by turning off interrupts in a number of places that needed to be atomic operations, and turning them on again when the operation was complete. This meant that during that operation, no other thread could take the cpu and do some other work which could mess with the work of this thread, causing some odd results (a race condition).

I added the pointer to a lock in semaphore, and which lock and sema a thread is waiting on in thread, because this way I wouldn't have to search through every lock and semaphore every time I wanted to see which one a thread was waiting on or which lock held this semaphore, thus increasing the performance.

I designed my queue to be just a single queue, ordered in priority, rather than a separate queue for each priority because I felt like this would be a lot easier to maintain, and that it would be more optimal in terms of memory usage (it takes up less memory), and in access time, as I wouldn't have to search through multiple queues, I could just pop the front list_elem from that single queue. This does, however, have a downside of constant ordering, but that's not so time consuming as I don't do it too often so it doesn't have much of an impact.

My solution failed the priority-donate-multiple and priority-donate-multiple2 tests as I didn't really attempt them. I didn't want to rework the entirety of my code to allow for multiple donations, although I gave it some thought and it would mostly just be reworking my code to pop off the front of a list of priorities, rather than just thread_current()->priority, and, when undoing a donation, making sure I still maintained the original priority.