User-Level Thread Library

Names:
Christopher Naporlee - cmn134 Section 3
Michael Nelli - mrn73
Section 3

Thread Functions:

1. rpthread_create()
   a. If the scheduler context has not been initialized, sets up scheduler and the timer for preemption.
   b. Assigns an id to the pointer passed into the function.
   c. Allocates space for the new thread stack.
   d. If previous steps successful, calls a thread function, which when thread function returns it goes to pthread_exit()
2. rpthread_yield()
   a. Disables preemption and takes note of the thread yielding.
   b. Swaps context to the scheduler.
3. rpthread_exit()
   a. Stores the return value passed into rpthread_exit into the thread control block.
   b. Sets the status of the thread to STOPPED.
   c. Releases all threads waiting to join in on it, and sets their status to be READY so they can continue running and or collect the return value of the STOPPED thread.
   d. Gives control back to the scheduler to find the next thread to run.
4. rpthread_join()
   a. Look up the thread by the id passed into the function.
   b. If the thread is not found, returns -1.
   c. If the thread to search for is not STOPPED
      i. Add the id of the call thread to the waitlist of the thread to join in on.
      ii. Set call thread to BLOCKED.
      iii. Give control back to the scheduler while we wait for the thread to finish.
   d. If, and when, the thread we are looking to join is STOPPED
      i. Retrieve the return value from the STOPPED thread's control block.
      ii. Set the thread from STOPPED to JOINED, so the scheduler knows that we are done with it and it's resources can be cleaned up.

Mutex Functions:

1. rpthread_mutex_init()
    a. Assigns the mutex an id
    b. Set the owner of the mutex and it's waitlist to NULL
2. rpthread_mutex_lock()
    a. Checks to see if the mutex's owner field is valid.
    b. If valid
        i. Add the calling thread's id to the mutex waitlist.
        ii. Give control back to the scheduler to wait for the mutex to become unlocked again.
    c. If, and when, the mutex is unlocked
        i. Assign the mutex's owner field to be the calling thread.
3. rpthread_mutex_unlock()
    a. If the calling thread is not the owner of the mutex lock, returns -1
    b. If the calling thread is the owner
        i. Temporarily blocks the preempt signal to allow all waiting threads to be set into the READY state.
        ii. Once finished setting threads to READY, reenables preemption to interrupt the thread.
        iii. If the thread was signaled during the blocking period, hands control to the scheduler to give one of the released threads a chance to grab the lock.
4. rpthread_mutex_destroy()
    a. Destroys all properties of a mutex
        i. Owner
        ii. Wait list

Schedule Functions:

1. schedule()
    a. Calls sched_rr() or sched_mlfq(), depending on which scheduler is set in the Makefile.
2. sched_rr()
    a. Receives a circular, doubly linked list which represents a single queue.
        i. We used a circular list to simplify adding to the head and tail by also using doubly linked nodes; a doubly linked list allows us to add a thread to the end of the list by using head->prev.
    b. Passes the queue to a helper function which searches for the next ready thread. While searching, it will free any joined threads it comes across. The function will return a thread that is ready to sched_rr.
    c. Enables preempt and swaps context to the next thread.
3. sched_mlfq()
    a. Checks the priority level of the running thread.
        i. If the priority is greater than the lowest priority and the thread uses its whole timeslice, then it will be removed from its current queue and added to the one below it. Otherwise it won’t move down.
    b. Cycle through all queues, highest to lowest priority, until the first non-null queue is found.
        i. Send this queue to sched_rr().
            1. If we go through the entire list and don't find a “ready” thread, then sched_rr returns -1, signaling that we should drop to the next layer as everything is blocked.

Benchmarks

All testing done on snow.cs.rutgers.edu

*all runtime averages are across 5 samples with a 5ms timeslice.

Vector Multiply			
Thread count	*rpthread RR* avg. runtime(ms)	*rpthread MLFQ* avg. runtime(ms)	*pthread* avg. runtime(ms)
1	2940	2347	51
2	2936	2339	175
10	2935	2346	302
50	3006	2399	353
100	3015	2416	364

Parallel Calculate			
Thread count	*rpthread RR* avg. runtime(ms)	*rpthread MLFQ* avg. runtime(ms)	*pthread* avg. runtime(ms)
1	2677	2365	2654
2	2654	2366	1351
10	2656	2365	673
50	2655	2365	595
100	2665	2368	600

External Calculate			
Thread count	*rpthread RR* avg. runtime(ms)	*rpthread MLFQ* avg. runtime(ms)	*pthread* avg. runtime(ms)
1	14139	13932	6364
2	14012	13997	3620

10	13981	14011	2431
50	13963	14001	2394
External Cal			
Thread count	*rpthread RR* avg. runtime(ms)	*rpthread MLFQ* avg. runtime(ms)	*pthread* avg. runtime(ms)
100	13994	14120	2440

From the data above, we can see that our implementation of threads is, obviously, outperformed by the pthread library. We suspect this to be for a couple of reasons:

1. The pthreads can map to multiple kernel threads, in contrast to our threads which will at most use one kernel thread.
2. The pthread library most likely has much better and more matured algorithms then what we have implemented.

Point 1, is definitely the most compelling because the ability to have multiple threads running in true parallel allows the workloads to be finished much faster than a single kernel thread can.