David Porter (davidpor@)
Wei Lin (veilam@)

Design

From a high level design of the threading library, we use two queues, “ready_list” and “dead_queue”. The ready list contains the threads that can be switched to, and the dead queue contains threads which have called sthread_exit(). Threads on the dead queue are periodically freed.

In terms of our sthread data design, the basic thread struct has a few properties. The thread stores basic information such as tid (for debugging), the start routine, the arg the user passed in, the return value, and a flag for keeping track if the thread is finished or is joinable.

We also use a few global variables, for example, an alive_thread count, which stores the current number of threads created and not exited or joined yet. We decided to keep this count in order to reclaim memory from the underlying queue library. When the number of alive threads reaches zero, we call sthread_queue_clear_free_list to reclaim memory from the free list and avoid having memory leaks.

In terms of design decisions for our threads, one of the areas that took some time to think about and implement was the overall structure of the memory management and when it’s ok to free threads. We came to the conclusion that we can classify threads into two categories, joinable and not joinable and treat their memory management a bit differently. For threads that are marked as joinable, we free the thread and it’s underlying resources as soon as it has been joined with another thread. For threads that are not joinable, we add them to the dead_queue in the sthread_user_exit function and then have a separate method clean_dead_threads() which is called periodically to free the threads as our garbage collector. We also kept track of the alive thread count as mentioned above to avoid having memory leaks due to the queue free list.

Preemption was also a bit challenging to implement correctly and we spent a good amount of time fixing issues as a result of adding preemption. We added disabling of interrupts in between sections of the code where we were modifying the ready_list and other global resources. We also made use of the atomic_test_and_set function when using a lock in the mutex and condition variable implementations.

As for the design of our synchronization primitives (mutexes and condition variables), our mutexes had three elements in the struct, the underlying lock_t, a queue of blocked threads, and a lock for the blocked_threads_queue. In mutex_lock, if the lock was set we would add the thread to the blocked queue and then switch away from it. In mutex_unlock, we would clear the lock and then move all the the threads from the blocked queue to the ready list. The condition

variable, similarly, contained a list of waiting threads and an underlying lock_t. The cond_wait method would move the thread to the waiting queue, while signal() would remove one thread from the waiting queue. Broadcast would remove all of the threads from the waiting queue, waking them all up.

Functionality

We believe our implementation works, as we tested it heavily against the included tests as well as with our test-burger program.

Conclusion

Working on this project was definitely very interesting and help us to gain some insights on how threads work and what threading libraries like p_threads do under the hood. The nature of multithreading required a good amount of thinking through various interleavings and debugging to narrow down any race conditions that occurred. It was definitely good experience to learn about how multithreaded code works and the challenges when writing it.

In terms of what we would have differently, we might have adapted our memory management, for example by freeing threads as soon as they exit if they're joinable and keeping track of the return values. We might also have changed our synchronization primitives a bit, such as the mutex, relying more on disabling interrupts rather than having lock associated with the blocked_threads_queue for example to reduce the complexity in their implementations.

Overall, we learned a lot about threads and concurrency with this project, which will definitely be useful down the road.