CSCI5103
Project 2 Writeup
David Ma (maxxx818) and Andrew Walker (walk0655)

Overall Architecture

Lock

The Lock class implements a standard mutex synchronization mechanism. It contains a lock queue that allows the user to have multiple threads waiting on the lock at one time. There are public lock() and unlock() methods and private _unlock() and _signal() that are used for the Condvar.

Condvar

Condvar implements a conditional variable class based on Hoare Semantics. The main methods are wait, signal, and broadcast, each of which is according to provided specifications.

SpinLock

The SpinLock class is a lock that causes threads to busy wait while attempting to acquire a lock. This implementation makes use of C++'s atomic test_and_set() method.

Priority

We chose to implement priority boosting, which periodically boosts the priority of low-priority threads holding locks. Specifically, we add a static counter to the main uthread scheduler that counts the number of scheduler interrupts. Every x interrupts (currently we have x set to 20), the signal handler will iterate through the low priority queues and increase the priority of the first one that it finds holding a lock. As part of this implementation, we also had to make some slight adjustments to Lock and Condvar so that we could track the number of locks that a thread is holding.

Asynchronous I/O

Our async I/O implementation simply makes an async i/o request to the kernel with the aio_functions, and then immediately yields thread control. The async calling thread is polled as normal by the scheduler; if async I/O is completed, the thread continues. Otherwise, it simply yields thread control again.

Performance Tests (Functionality test descriptions located in readme)

The following tests are used to compare the behaviors and performances of our various mechanisms. Similar to the functionality tests, each test can be compiled with “make [testname]” and run with ./[test name]. All tests - both performance and functionality - can be compiled with “make alltests”.

- controltest (testControl.cpp)
    - Creates 5 tester threads and joins them to the main thread. Each thread prints “`{my_tid}\t012\n`” and then “`{my_tid}\t345\n`”. Between printing each digit, it does a lot of computationally heavy math that takes longer than the quantum size, meaning these numbers all print out of order. This test uses no synchronization mechanisms.
- locktest (testLock.cpp)
    - Creates 5 tester threads like above, except before printing “`{my_tid}\t012\n`” it locks, and before printing “`{my_tid}\t012\n`”, it unlocks and relocks. At the end, it finally unlocks again. The numbers should print in an ordered fashion.
- spintest (testSpin.cpp)
    - Creates 5 tester threads like locktest, except using a spinlock instead of a lock.
- sControlTest (shortTestControl.cpp)
    - Same as controltest except it uses a different function for its threads. This function uses much less time but has more iterations.
- sLockTest (shortTestLock.cpp)
    - Same as sLockTest but each critical section is shorter and repeated more times.
- sSpinTest (shortTestSpin.cpp)
    - Same as sLockTest but uses SpinLock instead of Lock.

- Suite a of I/O performance tests use long I/O with a single read, write, and other thread. Long in this case means 400 reads of size 10,000 for read and 1 write of size 10,000,000 for write.
    - asyncTestPerformance (asynctest.cpp)
    - nonasynctest (nonasynctest.cpp)

- Suite b of I/O performance tests uses the same other thread as suite a. It uses a singel read and a single write thread, each doing short I/O (size 10) repeated 1,000,000 times each.
    - sAsyncTest (sAsyncTest.cpp)
    - sNonAsyncTest (sNonAsyncTest.cpp)

- Suite c of I/O performance tests uses just read and write threads. There are 20 of each. The read threads read 400 reads of size 10,000. The write threads write once with a size of 3,000,000. Each write thread writes to a separate file.
    - tAsyncTest (tAsyncTest.cpp)
    - tNonAsyncTest (tNonAsyncTest.cpp)

- Suite d of I/O performance tests is the same as suite c but it reintroduces the same other thread that was used in suite a.
    - tAsyncTestMoreWork (tAsyncMoreWork.cpp)
    - tNonAsyncTestMoreWork (tNonAsyncMoreWork.cpp)

Performance Evaluation

Lock vs SpinLock

- Which lock provides better performance in your testing? Why do you think that is?
    - We timed the performance of `controltest`, `lockTestPerformance`, and `spinTestPerformance` five times each. These tests are described above in the "Tests" section. Below is the average user time of these 5 runs.

`time ./controltest` (s)	`time ./lockTestPerformance` (s)	`time ./spinTestPerformance` (s)
1.5678	1.5978	4.7428

    - We also timed the performance of `sControlTest`, `sLockTest`, and `sSpinTest` five times each. These tests are described above in the "Tests" section. Below is the average total, user, and system times of each.

	`time ./sControlTest` (s)	`time ./sLockTest` (s)	`time ./sSpinTest` (s)
total	5.96	14.13333	7.9496
user	1.14125	5.427	3.5544
system	4.515	8.662667	3.8408

    - We address "which lock provides better performance" in the next section.
- How does the size of a critical section affect the performance of each type of lock? Explain with results.
    - Based on the above results, it seems that a longer critical section means that a normal mutex lock would be more appropriate as it has better performance. Additionally, it seems that a shorter critical section would mean a spinlock is more appropriate. We can reconcile these results by remembering that spinlocks can be better for shorter critical sections since they avoid context switching and other overhead. This is why many operating system kernels use spinlocks -- they often don't have to wait for very long. On the other hand, with longer critical sections, it is more important to not waste time busy waiting.
- *uthread* is a uniprocessor user-thread library. How might the performance of the lock types be affected if they could be used in parallel by a multi-core system?
    - As a general rule, spinlocks are worthless on uniprocessor systems, because after a thread acquires a spinlock, it just does nothing until another thread is scheduled. Normal locks are significantly better on uniprocessor systems because instead of waiting on processor time, the acquiring thread is just sent to a wait queue and the scheduler can immediately schedule another thread.

However, spinlocks are significantly more useful on multiprocessor systems, because then a thread on one processor can release the spinlock, instantly giving it over to another thread on a different processor. This avoids the overhead of context switching and thus can be more efficient.

- Are there any other interesting results from your testing?
    - As expected, no locks goes fastest of the three options (no locks, mutex, spinlock). Also as expected, as we increase the number of threads, this difference in speeds becomes more pronounced.

Synchronous vs. Asynchronous I/O

- Which I/O type provides better performance in your testing? Why do you think that is?
    a. For long I/O with singular threads, async seems to go faster. In our test case, it goes marginally faster with averages of 0.8795s total time for `asyncTestPerformance` and 2.2860s total time for `nonasynctest`.
    b. For short I/O with many threads, async seems to go slower. In our test case (`sAsyncTest` and `sNonAsyncTest`), it took async 1m22s to complete, and the nonasync version took just 0.402s.
    c. For long I/O with a medium amount of threads, async went faster again. In our test case (`tAsyncTest` and `tNonAsyncTest`), it took async 1.033s on average with 2.142s on average for nonasync.
    d. For long I/O with a medium amount of threads and with other (non-I/O) worker threads, async went faster again. In our test case (`tAsyncTestMoreWork` and `tNonAsyncTestMoreWork`), it took async 6.591s on average, and it took nonAsync 7.914s.
    e. "Why do you think that is?" is answered in the below two prompts.
- How does the amount of I/O affect the performance of each type of I/O? Explain with results.
    a. This question depends on how much I/O each thread has to do and also on how many threads there are. The longer each thread's I/O takes, the better the performance of async. This is obviously because we don't need to wait for the I/O to finish, and can instead take this time to do other work. However, if a thread's I/O is shorter, async's performance time starts to suffer from its inherent overhead and it becomes slower than synchronous. The benefits or drawbacks of this are multiplied by the number of threads there are. We see this from looking at test suites a, b, and c above.
- How does the amount of other available thread work affect the performance of each type of I/O? Explain with results.
    a. If we look at test suites c and d, we see that adding extra worker threads slightly increases the difference between async and synchronous from 2.142-1.033=1.109s to 7.914-6.591=1.323. Generally, the higher amount of other available thread work there is, the more efficient async will become. This is because while async is inherently less efficient than synchronous, it allows other threads to do work instead of waiting. This means that if we can do the I/O in

parallel with other work, it becomes more efficient because we're doing multiple things at once. If there isn't other work to do in parallel with async I/O, it's effectively just a inferior read/write.