CSCI 5103 Project 2 Writeup

Anlei Chen & Shuo Jiang

1. For the grader:

The project can be compiled using `make` or `make all` in the top-level submission directory. Those commands will compile the library, unit tests, performance tests, and the HTTP server. Additional commands, such as `make debug` and `make run`, are also provided. `make debug` will recompile the entire project with debug statements enabled. `make run` can be used to run test cases, performance tests, and http server.

2. Test Cases:

There are two main test methods used to test the library's correctness: the unit tests located in tests.cpp and the HTTP server test located in tests/server/. Five unit tests are provided in tests.cpp: mutex lock, spin lock, condition variable, multiple condition variables, and asynchronous I/O. The two lock tests are standard cases of needing to lock a shared variable, with random yield statements added between each line to ensure that the locks work correctly. The two condition variable tests implement a barrier and a bounded buffer, which also include random yields to ensure correctness. The asynchronous I/O test uses a pipe and has the threads communicate with the main thread via the pipe. These tests can be run using `make run-tests`.

Additionally, there is a cumulative HTTP server test. We implemented a standard multi-threaded HTTP server using user threads instead of pthreads. Some key aspects of the HTTP server include a connection queue, which is a bounded buffer of accepted client connections. When the server accepts a client, a thread will handle reading and writing the HTTP request using asynchronous I/O. To test the server, `make run-server` can be used to start the server on the default port of 8000. The server can be connected to using a web browser or shell via localhost:8000/<server_file>. By requesting multiple files in separate tabs, the correctness can be tested. To shut down the server, send SIGINT.

3. Performance Evaluation:

3.a Mutex Lock vs. Spin Lock:

Which I/O type provides better performance in your testing?

In our testing, the spin lock will usually provide better performance than the mutex lock because critical sections are mostly kept as short as possible. Since the critical sections are so short, the spin lock avoids the overhead of managing lock queues and context switches, leading to better performance compared to the mutex lock.

How does the size of a critical section affect the performance of each type of lock?
Lock performance depends heavily on the critical section. As we increase the length of the critical section, the implementation of the spin lock can significantly bottleneck its performance. Each thread spins, waiting for the lock to be released, which may take a lengthy amount of time, especially with round-robin scheduling. This leads to threads unnecessarily consuming CPU cycles instead of yielding them.

Lock Type	Number of threads	Number of iterations/thread	Length of critical section (loops)	Wall time (ms)
mutex_lock	2	100	1000	1.29481
spin_lock	2	100	1000	0.68259
mutex_lock	2	100	10000	6.51624
spin_lock	2	100	10000	6.1986
mutex_lock	2	100	100000	62.8141
spin_lock	2	100	100000	92.6094
mutex_lock	10	100	100000	315.468
spin_lock	10	100	100000	1261.71

Table 1: The test was conducted with 5 different critical section sizes on a ubuntu docker container (testing setup). The wall time was calculated using an average of 5 trials. Unlisted is the quantum time, which was held constant at 10,000 usecs. The data was collected using the command: `make run-lock NLOOPS=<nloops>`.

Observing the results, the spin lock outperforms the mutex lock when the number of iterations in the critical section is relatively low. However, as the number of iterations increases, the spin lock becomes slower because it's much more likely that the thread is preempted while holding the lock. This results in other threads spending their entire quantum spinning for a lock that won't be released until much later. In contrast, the mutex lock voluntarily gives up its quantum, allowing other threads to perform their workload.

The drawback of spinning becomes clearer when the number of threads acquiring the lock increases. As more threads attempt to acquire the lock, it will take much longer for the scheduler to resume the thread that was preempted while holding the lock. This results in the remaining threads spinning for their entire quantum in a round-robin fashion until the lock is released.

How might the performance of the lock types be affected if they could be used in parallel by a multi-core system?
The performance of the lock types can be both improved and degraded in a multi-core environment. Performance may degrade due to lock contention and the cache line "ping-pong" issue. In a high lock contention setting, threads will frequently spin and context-switch, leading to wasted CPU cycles. The cache line "ping-pong" issue occurs when a modification to the shared state causes cache invalidation to all other cores that hold the shared state. The performance could be improved, specifically with spin locks, if the thread holding the spin lock is rescheduled to run sooner. For example, in the spin lock scenario mentioned in the previous part, if the thread holding the spin lock is preempted but rescheduled on another core, the other threads waiting for the lock wouldn't need to spin their entire quantum for a thread that isn't even running.

3.b Asynchronous I/O vs. Synchronous I/O:

Which I/O type provides better performance in your testing?
Asynchronous I/O provides better performance during testing because threads issue the asynchronous I/O and voluntarily give up their quantum, allowing for more efficient use of CPU cycles. In contrast, synchronous I/O blocks and prevents the thread library from scheduling other user-level threads until I/O completion.

How does the amount of I/O affect the performance of each type of I/O?
The amount of I/O affects both synchronous and asynchronous I/O types. When the I/O size is small, I/O can finish almost instantly, making context switching unnecessary. In our tests, an arbitrary-length workload (busy waiting) was implemented after every write operation. This was done because, without a write operation, asynchronous and synchronous I/O were essentially the same, with very negligible differences in time. Asynchronous I/O consistently outperforms synchronous I/O in our tests. This is because the thread issues the asynchronous I/O, allowing another thread to start its workload. The time saved by the thread in computing its workload far outweighs the overhead needed to perform a context switch. However, as the size of the I/O increases, more of the total time is spent waiting for the I/O to complete. This reduces the speedup that asynchronous I/O has over synchronous I/O.

I/O Type	Threads	I/O Operations	I/O Size (bytes)	Workload (iterations)	Wall Time (ms)
asynchronous	10	5	512	50000	12.4481
synchronous	10	5	512	50000	15.1840
asynchronous	10	5	4096	50000	12.3024
synchronous	10	5	4096	50000	17.0423

asynchronous	10	5	8192	50000	13.2452
synchronous	10	5	8192	50000	19.3184
asynchronous	10	5	32768	50000	15.8260
synchronous	10	5	32768	50000	23.6279
asynchronous	10	5	65536	50000	28.9172
synchronous	10	5	65536	50000	32.5241

Table 3: The test was conducted with 5 different I/O sizes on a ubuntu docker container (testing setup). The wall time was calculated using an average of 5 trials. Unlisted is the quantum time, which was held constant at 10,000 usecs. The data was collected using the command: `make io`.

How does the amount of other availability thread work affect the performance of each type of I/O?

The amount of workload per thread will substantially impact the performance of the asynchronous I/O type. With a large number of iterations in the workload, the asynchronous I/O can switch to another thread that is not waiting on I/O, allowing it to utilize its quantum to perform its useful work. However, with low workloads, the time it takes to perform a context switch outweighs the time saved of having another thread perform work. Additionally, at high workloads, the workload consumes so much more time compared to the I/O that there is very little improvement in performance.

I/O Type	Threads	I/O Operations	I/O Size (bytes)	Workload (iterations)	Wall Time (ms)
asynchronous	10	5	4096	10000	14.3175
synchronous	10	5	4096	10000	13.9482
asynchronous	10	5	4096	50000	10.9302
synchronous	10	5	4096	50000	19.4579
asynchronous	10	5	4096	100000	18.0949
synchronous	10	5	4096	100000	31.0652
asynchronous	10	5	4096	500000	85.2685

synchronous	10	5	4096	500000	104.79
asynchronous	10	5	4096	1000000	1632.6
synchronous	10	5	4096	1000000	1661.03

Table 4: The test was conducted with 5 different workload sizes on a ubuntu docker container (testing setup). The wall time was calculated using an average of 5 trials. Unlisted is the quantum time, which was held constant at 10,000 usecs. The data was collected using the command: `make io`.

The performance of synchronous I/O remains linearly scaled with the amount of workload per thread. The synchronous write is a system call that blocks the calling thread, thus the thread library must wait until the I/O is completed. Therefore, in a uniprocessor environment, synchronous I/O behaves much like a sequential program, where tasks are performed one after another — no task can be done while the I/O is busy.

Are there any other interesting results from your testing?

Lock Type	Number of threads	Number of iterations/thread	Length of critical section (loops)	Wall time (ms)
mutex_lock	20	100	100000	646.01
spin_lock	20	100	100000	4821.74

Table 5: Mutex lock vs. spin lock when time in critical section exceeds quantum. Wall time was calculated as an average of 5 trials on a ubuntu docker container (testing setup). Unlisted is the quantum time, which was held constant at 10,000 usecs. The data was collected using the command: `make run-lock NTHREADS=20 NLOOPS=1000000`.

As mentioned in the lock performance evaluation section, if a thread holding the spin lock is preempted, any threads waiting for the lock will spin until the scheduler reschedules the thread holding the spin lock. The performance of this behavior worsens with the number of threads, as a round-robin scheduler is used.