User-level-Thread-Library-with-Synchronization

For this project we utilized the instructor provided versions of uthread and TCB.

To run all test run '''./test.sh''' from the main directory This is shell script that will make clean, make all, then run all tests Consider redirecting output into a log because it is long '''./test.sh | log.txt'''

Lock vs Spinlock

Lock Tests

```
=============================
Critical section time set to 1000
=============================
lock time: 14930293 microseconds
spinlock time: 15819360 microseconds
spinlock was faster by  94.3799 percent

=============================
Critical section time set to 10
=============================
lock time: 10066642 microseconds
spinlock time: 5817716 microseconds
spinlock was faster by  173.034 percent
```

- **Which lock provides better performance in your testing? Why do you think that is?** The spinlock provided a better performance than the lock in our testing. This is because with a small critical section, it is faster for a thread to spin rather than spend the resources making a context switch.

- **How does the size of a critical section affect the performance of each type of lock? Explain with results.** As we increase the size of the critical section, the spinlock becomes less and less effective compared to the lock. As seen in the tests above, when we increased the time in the critical section, the percent speedup of the spinlock was greatly reduced. This is because a context switch becomes more worth the resources if we are going to be spinning for a long period of time.

- **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?** Spinlocks should become more effective on multiprocessors because it no longer blocks any and all threads from executing while it is spinning. It will still waste CPU cycles, but there are other cores available to run other threads.

Asynchronous vs Synchronous IO

IO Tests

```
Running tests...
==============================
 Message Length: 5000 bytes
 Number of Threads: 50
==============================
Sync I/O time: 159265 microseconds
Async I/O time: 71773 microseconds
Percent speedup for asynchronous IO: 54.9349%
==============================
 Message Length: 10 bytes
 Number of Threads: 50
==============================
Sync I/O time: 139642 microseconds
Async I/O time: 72400 microseconds
Percent speedup for asynchronous IO: 48.1531%
==============================
 Message Length: 5000 bytes
 Number of Threads: 3
==============================
Sync I/O time: 8202 microseconds
Async I/O time: 16976 microseconds
Percent speedup for asynchronous IO: -106.974%
==============================
 Message Length: 10 bytes
 Number of Threads: 3
==============================
Sync I/O time: 7973 microseconds
Async I/O time: 16155 microseconds
Percent speedup for asynchronous IO: -102.621%
```

- **Which I/O type provides better performance in your testing? Why do you think that is?**

  In a typical setting, asynchronous IO performed better in our testing. Instead of threads blocking during IO operations, they instead yield the processor while they are waiting. This allows for more work to be done when there are large amounts of IO and other available threads to perform their own work.

- **How does the amount of I/O affect the performance of each type of I/O? Explain with results.**

As you can see from the first 2 tests with 50 threads. Asynchronous IO provides a greater speedup when there is a larger message length. When there is more IO, yielding provides a greater benefit, since there is more time spent waiting on our IO operations to finish.

- **How does the amount of other available thread work affect the performance of each type of I/O? Explain with results.**

  Asynchronous IO performs much better when there are many available threads, while synchronous IO performs better when there are few threads. You can see this in the tests, where tests 3 and 4 show poor asynchronous performance when there are only 3 threads, while in tests 1 and 2 it provides a speedup. When there are few threads, yielding while waiting for IO operations doesn't provide much benefit because there are less free threads on average and instead adds the overhead of context switching to a new thread.