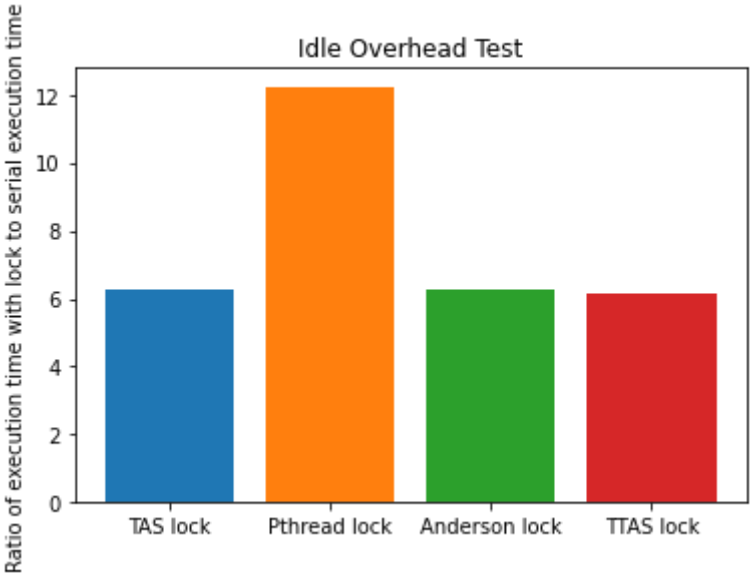
Changes to Implementation

I more or less followed the implementation guide I described in my design doc. The only major difference was for my A_lock. In addition to padding as suggested, I now reset where my tail pointer is using bit-manipulation instead of the modulo operator. This is because the modulo operator runs on divide, which hardware does not optimize for (it takes 60 cycles apparently). I address this by setting my flags array as a power of 2 always, then

Experiment Results: Idle Overhead



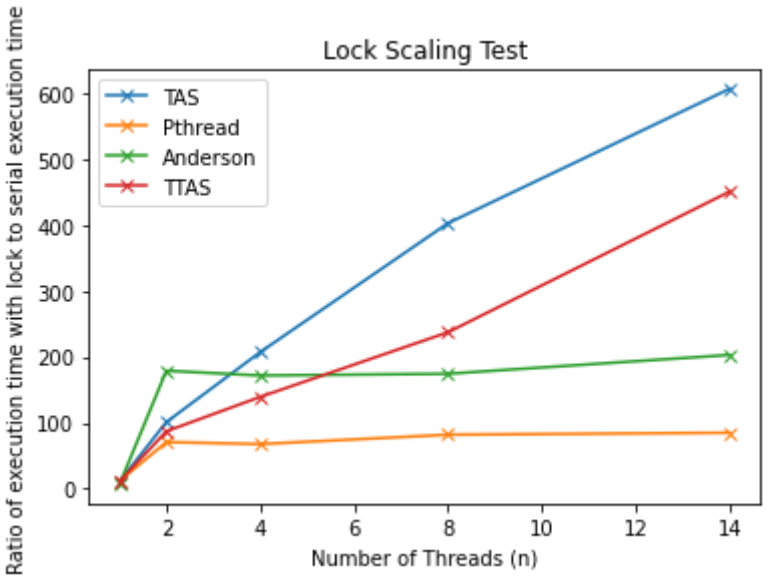
You can see here that in the idle overhead experiment, where we have a thread lock, increment, then unlock until the counter reaches big, that all the other locks do better than the default Pthread lock.

To me, it makes sense that the TAS, TTAS, and Anderson lock would all take about the same time because they do the same thing: lock with an atomic operation, do some cheap work changing some pointers (in the case of the ALock), then call unlock which is a cheap change of state.

It would be a different story if the different locks had a different number of atomic calls. For example, I mistakenly set the unlock method in TAS and TTAS to be atomic at first, and that doubled the time. This means that the bulk of the work for the lock method for these two locks is in the atomic operation.

It is not surprising that the pthread lock might not do well here, because we are trivializing using a lock for what is essentially a serial program that has only one thread. Thus, the pthread implementation may not be optimized for this use case.

Experiment Results: Lock Scaling



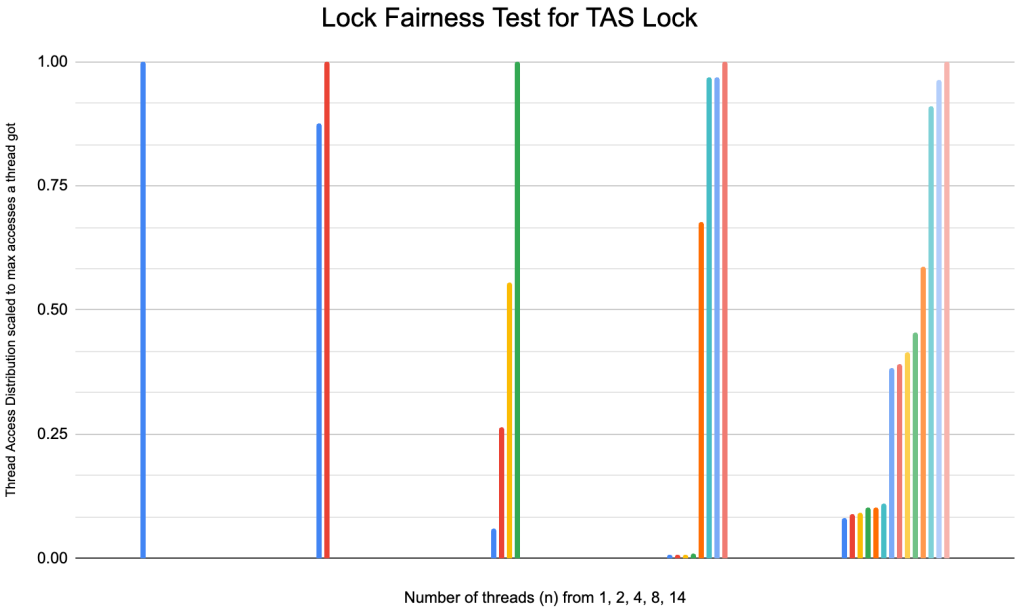
This graph also makes sense to me. First, TTAS does better than TAS overtime because it has a check-step, which leads to less cache invalidations. However, the A_lock does better because its cache invalidation only applies to the thread that is waiting on the lock, instead of doing a broadcast which the TTAS lock does. This is also why the effects become more noticeable as the number of threads increase: the cache invalidations of TTAS and TAS increase (whether linearly, quadratically, who knows, but it does all the same). However, the ALock has a constant number of cache invalidations for its lock/unlock operations regardless of the number of threads used.

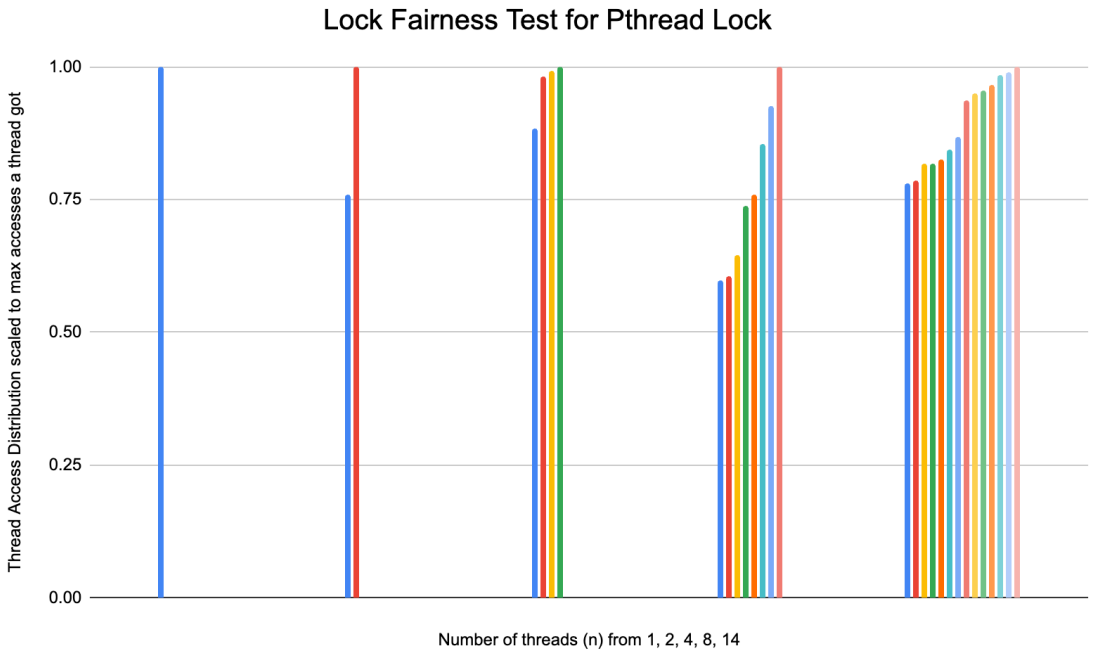
It's not surprising too that pthread outperforms the anderson lock. Designing for the many threads case should be what the pthreads implementers should be optimizing for, so they must use some implementation that has lock/unlock be O(1) to the number of threads used. It just seems then that pthreads has a lower overhead than the Anderson lock, but it must use the same principle that there is no broadcasting of cache invalidations, which preserves the flat-lineness we see above.

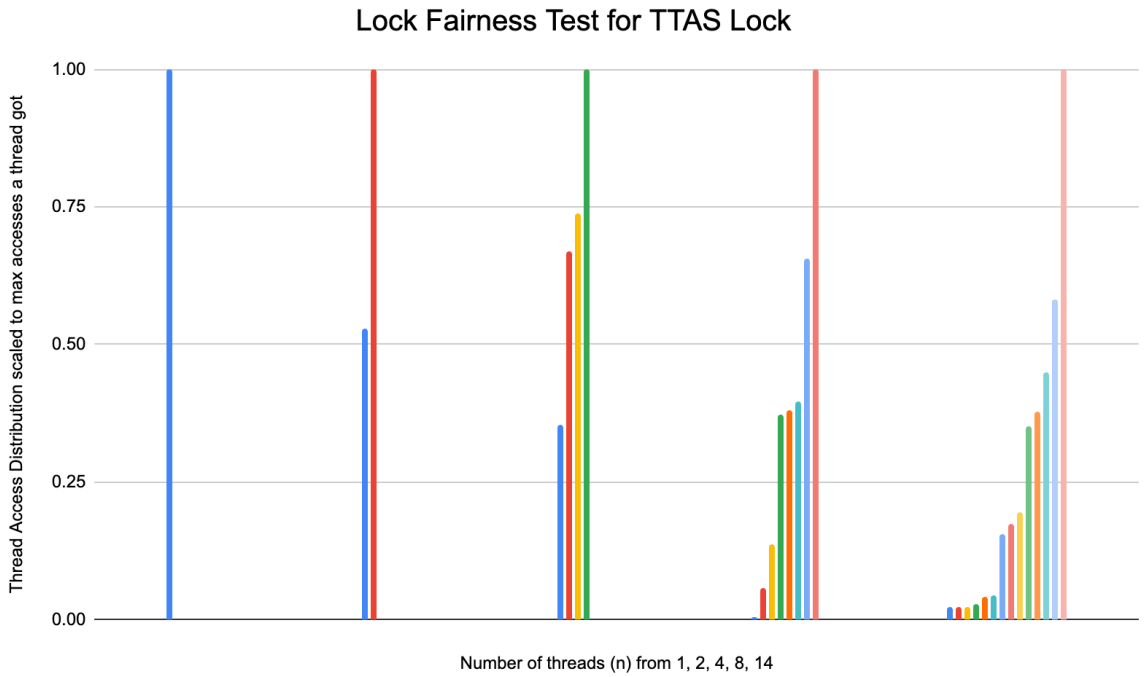
In other words, the Alock performs best in contended situation b/c other locks all spin on one memory address. But in the Alock, every thread is spinning on its own memory location → whenever a thread writes to its state variable, the other thread's caches are not invalidated.

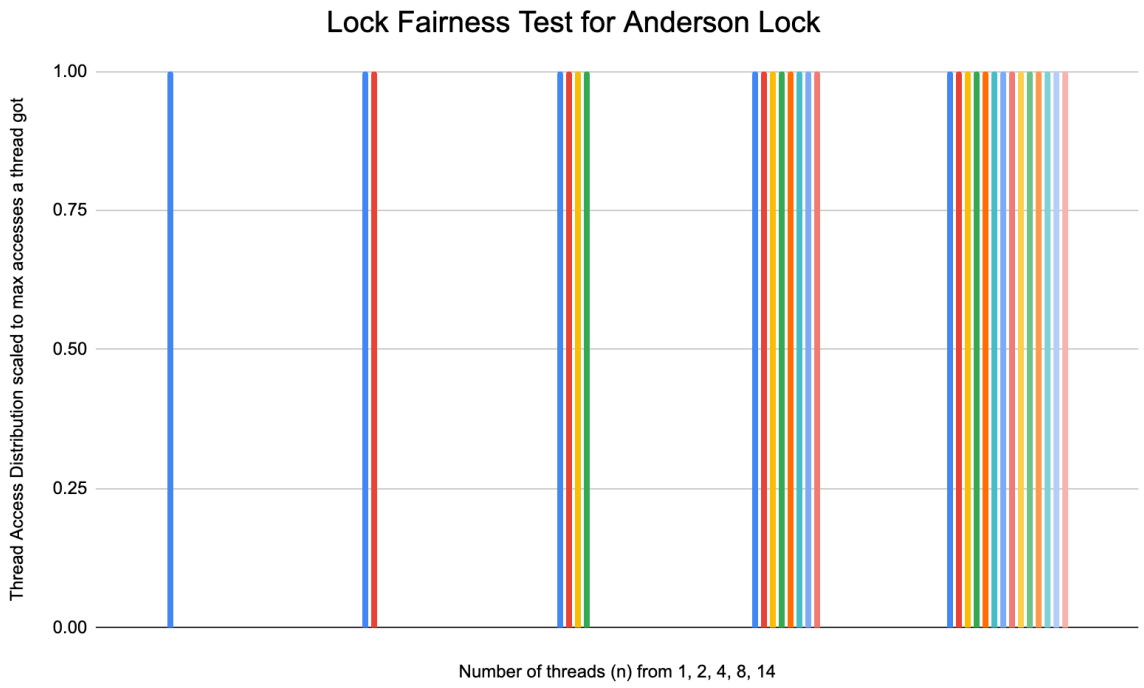
Experiment Results: Lock Fairness Experiment

Below are the distributions of thread accesses (scaled as a percentage of the max accesses any one thread got) for each of the locks for each thread count.


Lock Fairness Test for TAS Lock
Thread Access Distribution scaled to max accesses a thread got
1.00
0.75
0.50
0.25
0.00
Number of threads (n) from 1, 2, 4, 8, 14


Lock Fairness Test for Pthread Lock
Thread Access Distribution scaled to max accesses a thread got
1.00
0.75
0.50
0.25
0.00
Number of threads (n) from 1, 2, 4, 8, 14





The lock fairness results also make sense. First, the A_lock has a uniform distribution, which is reassuring. Secondly, the pthread distribution is relatively balanced, where in all cases of the trials, the most starved out thread enters half as much as does the most lucky thread. Fairness seems like something that I would design for if I were shipping pthreads, so that makes sense.

For the TAS lock and TTAS lock, it makes sense how we could get a very skewed distribution. There's no mechanism in the code that would make one think that no thread gets

starved out, and the result here is clear. In the most extreme case for n = 14, 6 threads in TTAS get less than 10% of what the max thread gets, and a similar story holds for TAS.

Correctness Testing

Our performance test for lock fairness also serves as a correctness test for the Anderson lock. For the ALock, we want to ensure the invariant that the ALock is FIFO. We could've tested for this by saying that certain threads enter in a certain order, and then checking the state of the queue, but this would be labor-intensive and there is no guarantee that we would be able to run an execution that would break this and show it.

Instead, we can say that if the ALock is acquired about equally among all locks, then it means threads are going in and out of the ALock queue in such a way that they're about equal, which could suggest FIFO is preserved for a hotly-contested lock.

I also followed through with my test plan for the exclusivity of the critical section. It passes, so I have reasonable confidence that my lock is mutually exclusive.