Hayley Dodkins u21528790

Introduction

In this practical, I've implemented the TAS Lock, TTAS Lock, and the Exponential Backoff Lock. The TAS Lock is sort of like JavaScript: simple, functional, but no one would call it a masterpiece. It's not great for handling contention, but it gets the job done.
TTAS is more like TypeScript. It introduces some structure into the chaos, adding local spinning to reduce cache interference.
Finally, the Exponential Backoff Lock, like perl, it's not fast or ideal but instead of throwing itself into a race it can't win, it chooses to wait, which leads to better performance under contention.

These locks were subjected to highly advanced test scenarios, such as dequeuing from a shared queue and incrementing a counter allowing for a comparison of their strengths and weaknesses.

Hayley Dodkins u21528790

Results

```
Backoff Lock with low contention:
Backoff Test completed in 7 milliseconds.
Backoff Test2 completed in 2 milliseconds.
Counter: 9
Backoff Lock with high contention:
Backoff Test completed in 5 milliseconds.
Backoff Test2 completed in 3 milliseconds.
Counter: 69
Backoff Lock with no contention:
Backoff Test completed in 0 milliseconds.
Backoff Test2 completed in 1 milliseconds.
Counter: 72
```

```
TAS Lock with no contention:
TAS Test completed in 1 milliseconds.
TAS Test2 completed in 1 milliseconds.
Counter: 75
TAS Lock with low contention:
TAS Test completed in 1 milliseconds.
TAS Test2 completed in 0 milliseconds.
Counter: 84
TAS Lock with high contention:
TAS Test completed in 2 milliseconds.
TAS Test2 completed in 11 milliseconds.
Counter: 144
```

```
TTAS Lock with no contention:
TTAS Test1 completed in 1 milliseconds.
TTAS Test2 completed in 1 milliseconds.
Counter: 147
TTAS Lock with low contention:
TTAS Test1 completed in 0 milliseconds.
TTAS Test2 completed in 1 milliseconds.
Counter: 156
TTAS Lock with high contention:
TTAS Test1 completed in 3 milliseconds.
TTAS Test2 completed in 3 milliseconds.
Counter: 216
```

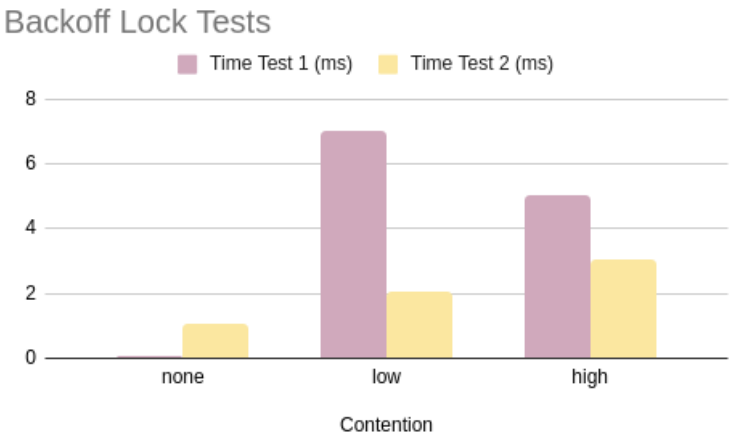
Hayley Dodkins u21528790

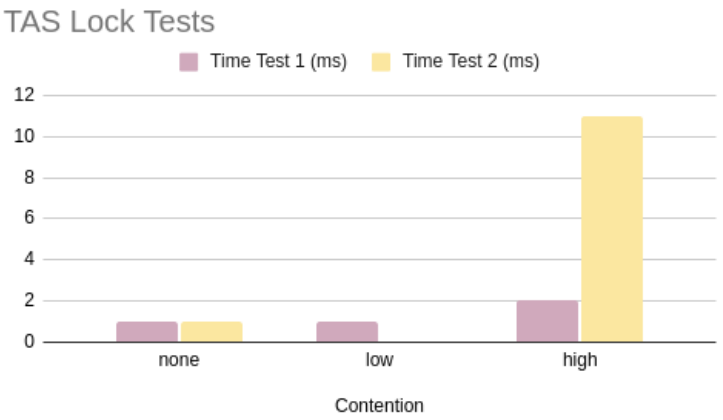
Graphs

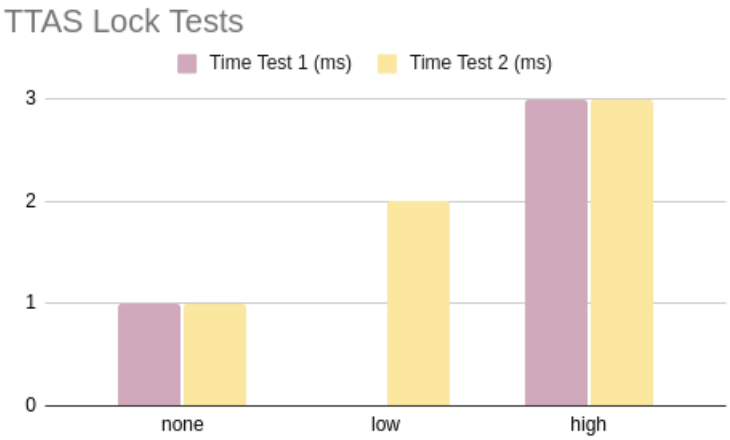
No contention = 1 thread

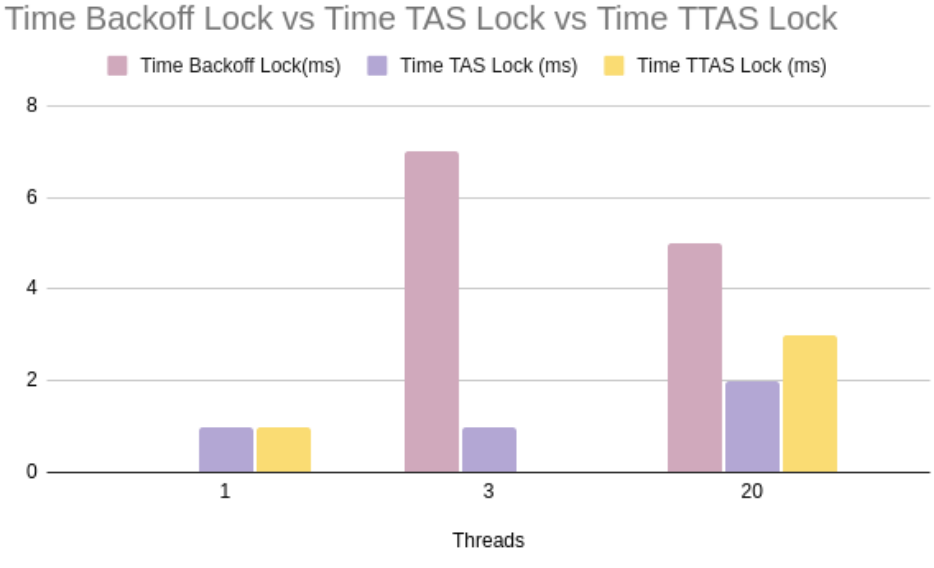
Low Contention = 3 threads

High contention = 20 threads









Hayley Dodkins u21528790

Report

The report summarizes the observations made during Test 1 : Dequeuing an integer from a shared queue and Test 2 : Incrementing a counter where 3 versions of each test were performed. No contention used a single thread, low contention used 3 threads and high contention used

TAS Lock

Under no contention, the TAS Lock performs efficiently, completing both Test 1 and Test 2 in just 1ms. This indicates that the lock works well when there is no contention.

Under low contention, the TAS lock performs consistently for Test 1, completing in 1ms, but Test 2 is faster, completing in 0ms. The TAS lock handles the queue dequeue operation with no delay, but the counter increment operation is significantly faster, reflecting how simple operations benefit from the lock’s simplicity.

Under high contention, Test 1 increases to 2ms, indicating that contention had an effect on the performance. Test 2 increases to 11ms, showing the lock's struggles under high contention. The TAS lock is simplistic and lacks mechanisms like local spinning or backoff. Thus, when there is contention, it becomes less efficient.

TTAS Lock

Under no contention, the TTAS Lock performs efficiently in both Test 1 and Test 2, with a time of 1ms. This suggests that the additional check before acquiring the lock helps prevent unnecessary cache invalidations without adding a delay in the absence of contention.

In the low contention scenario, Test 1 improves to 0ms, while Test 2 takes 2ms. The TTAS Lock's performance in Test 1 benefits from its local spinning mechanism, which helps to reduce overhead by checking the lock state before attempting to acquire it. However, Test 2 experiences an increase due to the overhead of waiting for the lock as counter increment operations are quick.

In high contention, Test 1 increases to 3ms, while Test 2 remains at 3ms. TTAS incorporates local spinning to reduce cache invalidation traffic, which results in a better performance in Test 1.

Backoff Lock Analysis

When there is no contention the backoff lock performs optimally, completing Test 1 in 0ms and Test 2 in 1ms. Without any contention, the backoff mechanism is not used, leading to minimal time taken for both tasks. This is consistent with expectations, as the lock doesn't need to back off when there is no contention.

Under low contention the time increases slightly for both tests. Test 1 takes 7ms, and Test 2 takes 2ms. The introduction of contention begins to slow down the backoff lock's performance. Although the effect is still minor, the backoff strategy comes into play, adding a small delay to handle contention. This delay increases as threads wait to access the critical section, causing the slight performance drop in both tests.

Under high contention, Test 1 shows an improvement, dropping to 5ms, indicating that the backoff mechanism is effective at reducing CPU strain and cache interference. By allowing threads to back off and reduce contention, the lock helps improve the time for dequeuing from the shared queue.
However, Test 2 shows less of an improvement. The counter increment operation is quick and requires little overhead, so using backoff leads to a performance cost. Threads spend more time waiting for the lock than performing the task. This leads to underutilization of the critical section, as threads are forced to wait excessively, reducing the overall efficiency of the counter increment operation.

In conclusion, while the backoff lock helps in high contention scenarios, it may lead to excessive waiting and inefficiency when the critical section is simple. The TAS Lock is efficient under low contention but suffers from performance issues under high contention. Under higher contention, it causes cache invalidation traffic which adds overhead and reduces performance. The TTAS Lock outperforms the TAS Lock in high

contention scenarios due to the local spinning reducing the cache invalidations. This reduces the overhead during times of high contention leading to improved performance.

Joke

Rust iterators are the best thing since &Bread[..]

References

Kirk, P. (n.d.). *Locks*. Computer Systems Fundamentals. James Madison University. Available at: https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Locks.html (Accessed: 3 October 2024).