**Report for COS226 Practical 5**

**Name:** Arnaud Zander Strydom  
**Student Number:** 23536013

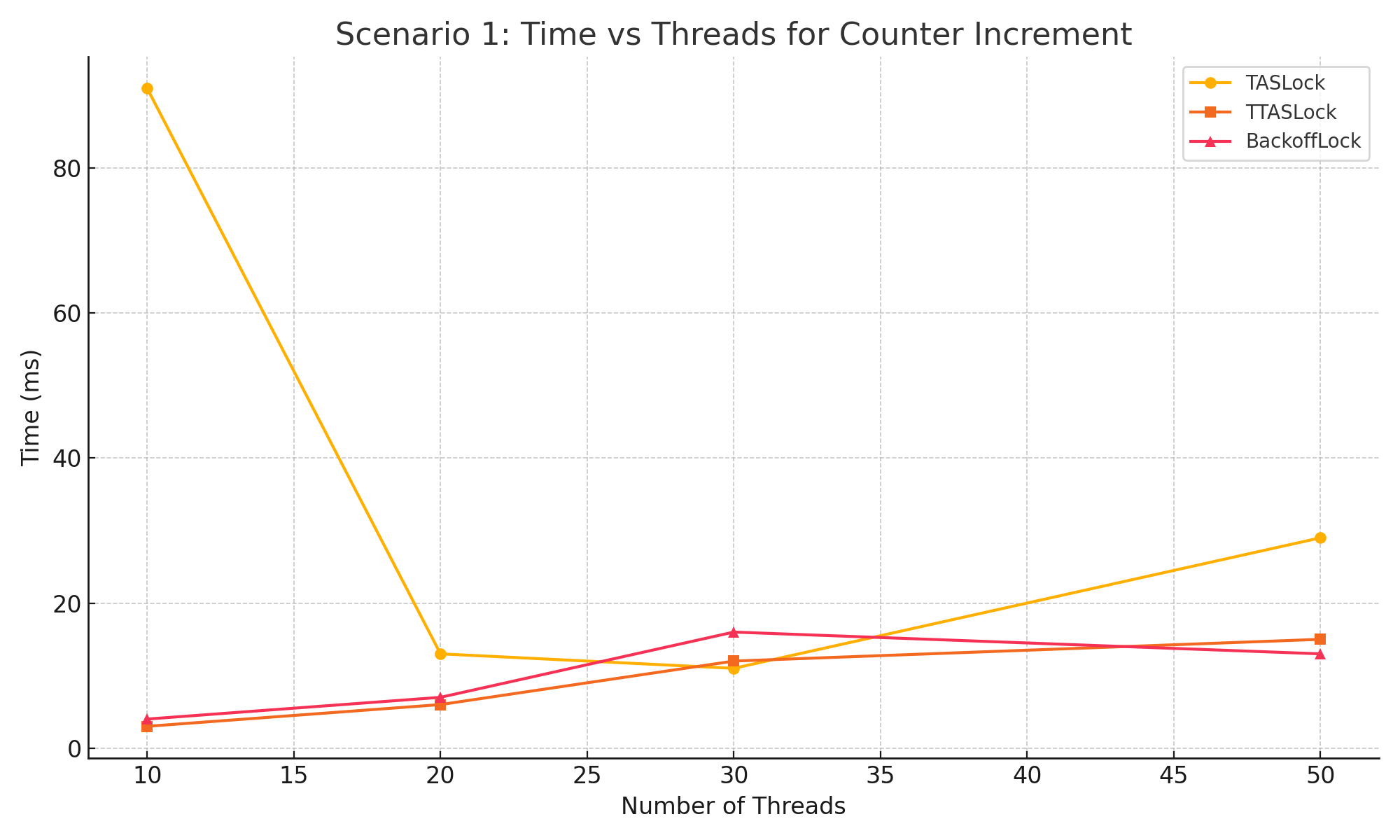
**Introduction**

Welcome to my SUPER detailed and captivating report on practical locks and their performance in relation to each other as well as their relation to different amounts of threads. I embarked on an exhilarating and captivating journey to test the TAS, TTAS, and Backoff locks under various thread loads and scenarios to evaluate the difference in performance.

**Graphs: Time vs Number of Threads**

The following graphs illustrate the time each lock took to complete tasks under different thread loads. Scenario 1 focuses on incrementing a shared counter, and Scenario 2 focuses on dequeuing from a shared queue.

**Graph 1: Time vs Threads for Counter Increment**

****

**Graph 2: Time vs Threads for Dequeue Operation**

**A graph with red and orange lines

Description automatically generated**

**Explanation of Trends**

From the results, we can observe the following:

**Scenario 1: Incrementing a Shared Counter**

**Findings:**

1. **TASLock**:
   * **Performance at 10 threads (91 ms)** was significantly slower compared to both TTASLock and BackoffLock, which suggests that TASLock suffers heavily from contention at lower thread counts. The constant retrying to acquire the lock without any backoff or testing causes a significant delay.
   * **Improvement at higher thread counts (20-50 threads)**: TASLock improves dramatically at higher thread counts, with **29 ms at 50 threads**. This improvement may result from more consistent access patterns where threads don't constantly hammer the lock. However, TASLock is still slower than TTASLock and BackoffLock at all thread levels.
2. **TTASLock**:
   * **Performance is excellent across the board**. With **3 ms at 10 threads**, and **6-12 ms** from 20 to 30 threads, TTASLock demonstrates minimal contention, making it the most efficient lock for this scenario.
   * **Stable performance under 50 threads**: Even with 50 threads, TTASLock only takes **15 ms**, making it the most scalable option for high concurrency.
3. **BackoffLock**:
   * **Moderately better than TASLock at lower thread counts**, performing similarly to TTASLock with only **4 ms at 10 threads**, and only **7 ms at 20 threads**. The backoff mechanism helps prevent contention when there are few threads.
   * **Degrades at 30 threads (16 ms)** but **recovers at 50 threads (13 ms)**: BackoffLock's performance becomes slightly inconsistent at mid-range thread counts, but it manages to maintain a respectable performance under high thread counts.

**Trend Observations:**

* **TTASLock is the most efficient** under almost any thread count. The extra check (testing if the lock is free before trying to acquire it) helps minimize cache coherence issues and prevents unnecessary contention.
* **BackoffLock shows more stability** than TASLock at lower and higher thread counts but struggles a bit at the mid-range (30 threads). The backoff mechanism works well in high-contention environments, but its performance hit shows up at intermediate loads.
* **TASLock improves** with higher thread counts but is still the slowest in all cases. It performs poorly at 10 threads due to heavy contention.

**Scenario 2: Dequeuing from a Shared Queue**

**Findings:**

1. **TASLock**:
   * **Performance with 10 threads (8 ms)** is decent, but **it deteriorates sharply at 20 and 30 threads** with **16 ms and 13 ms**, respectively. This suggests that TASLock experiences significant contention in scenarios where multiple threads access shared resources.
   * **TASLock does not scale well** in the dequeue scenario because of its inefficient locking strategy. Threads constantly trying to acquire the lock leads to heavy contention, slowing down performance.
2. **TTASLock**:
   * **Best performer again**. Even with 30 threads, TTASLock completes the task in just **14 ms**, compared to **TASLock's 13 ms**. Like in the counter increment scenario, TTASLock remains highly efficient at reducing contention and optimizing performance.
   * **Steady and scalable**: TTASLock performs better across all thread counts, as it reduces the overhead associated with constantly attempting to acquire the lock, leading to consistent results.
3. **BackoffLock**:
   * **Moderate performance** at **10 threads (6 ms)**, but struggles significantly at **20 threads (21 ms)** and **30 threads (26 ms)**. This indicates that while the backoff mechanism works well for small loads, it introduces too much overhead for mid-range and higher thread counts in the dequeue scenario.
   * **Degrades in high contention scenarios**: BackoffLock doesn't handle the dequeuing scenario well with more threads, as the random backoff delays become inefficient when multiple threads are competing for access.

**Trend Observations:**

* **TTASLock once again is the most efficient**: Its extra test before setting the lock keeps contention low, and it scales best across thread counts, even with many threads accessing a shared resource.
* **BackoffLock suffers** in this scenario due to increased overhead at higher thread counts. The exponential backoff becomes a burden, as the delays between lock attempts grow, leading to slower dequeue operations.
* **TASLock starts strong but slows**: It performs surprisingly well with **10 threads**, but as the thread count increases, contention leads to performance degradation.

**Lock Performance Comparison:**

* **TTASLock is consistently the best performer** in both scenarios. Its ability to reduce unnecessary lock contention makes it ideal for tasks that involve shared resource access in concurrent environments. TTASLock scales well with increasing thread counts, making it the most scalable and efficient lock across all tested scenarios.
* **BackoffLock performs well with fewer threads** but struggles with higher thread counts in the dequeuing scenario. The exponential backoff, while useful in high-contention situations, introduces inefficiencies as the number of threads grows. It is more stable than TASLock but less predictable.
* **TASLock performs poorly under low-thread conditions** and shows improvement with more threads in Scenario 1, but its overall performance is still inferior to both TTASLock and BackoffLock. TASLock's constant attempts to set the lock without testing leads to heavy contention and performance degradation.

**References**

* Data was generated using my Java test program which simulated locking scenarios with multiple threads.
* Geeks4Geeks & tutorialpoint for the theoretical part of the report.
* The COS226 textbook provided background information on locks and contention, as well as implementation for the various locks.

**Programming Joke**

Two threads walk into a bar. The barkeeper looks up and yells, “Hey, I want don’t any conditions race like time last!”