Indian Institute of Technology Delhi
Computer Science Department

COL818 A2: LOCKS

Implementation of multiple locks in C++ using std::atomic

Author
Yatharth Kumar, 2020CS10413

April 8, 2024

Contents

1 Introduction

In this assignment, we implement and compare 5 different types of spin locks.

- Test-And-Set Lock (TAS)
- Test-And-Test-And-Set Lock (TTAS)
- Anderson Lock (ALock)
- CLH Lock
- MCS Lock

1.1 Test-And-Set Lock (TAS)

The Test-And-Set Lock (TAS) relies on an atomic operation that combines testing and setting a designated memory location. This operation is crucial for managing access to a critical section of code. When a thread attempts to enter this critical section, it executes the Test-And-Set operation on the lock's memory location. If the lock is detected as being unlocked, the thread then proceeds to set it to a locked state, indicating that it has gained entry into the critical section. However, if the lock is already in a locked state, the thread enters a busy-waiting loop, repeatedly attempting the Test-And-Set operation until it successfully acquires the lock. After executing the critical section, the thread releases the lock by resetting its state to unlocked, allowing other threads to acquire it. While the Test-And-Set mechanism is simple and effective for synchronization, it can introduce inefficiencies due to the busy-waiting nature, where threads continually try to acquire the lock by performing attempt to write operations on the lock's memory location.

1.2 Test-And-Test-And-Set Lock (TTAS)

The Test-And-Test-And-Set Lock (TTAS) is a synchronization primitive used in concurrent programming to achieve mutual exclusion, similar to the Test-And-Set (TAS) lock, with the aim of reducing contention and overhead to enhance performance. In the case of the Test-And-Test-And-Set Lock (TTAS), the efficiency is attributed to the strategy of performing a read operation (test) first before attempting to write (set) the lock. This approach capitalizes on the relatively lower cost of reading compared to the more resource-intensive write operation involved in setting the lock. By checking the lock's state with a read operation initially, TTAS can often avoid the immediate overhead associated with attempting to acquire the lock (setting) as in the traditional Test-And-Set (TAS) approach.

1.3 Anderson's Lock (ALock)

Anderson's Lock, also referred to as the Anderson Queue Lock, operates by maintaining an array of Boolean flags, with each flag representing whether a thread is currently waiting to acquire the lock. Threads utilize an atomic operation to claim a position in this queue and then set their corresponding flag to true before entering a busy-waiting loop. When a thread completes execution of its critical section, it releases the lock by resetting its flag to false allowing other threads, if any, to proceed and acquire the lock. Anderson's Lock is appreciated for its scalability, fairness, and minimal overhead. However, its performance

might be impacted in scenarios where a large number of threads contend for the lock simultaneously.

1.4 CLH Lock

The CLH (Craig, Landin, and Hagersten) lock manages a queue of threads, ensuring that each thread waits its turn to acquire the lock. When a thread desires entry into a critical section, it creates a node indicating its intention and then spins on its predecessor's node until it becomes its turn to acquire the lock. Upon finishing its critical section, a thread releases the lock, thereby allowing the next thread in the queue to proceed. The CLH lock is valued for its scalability, fairness, and low overhead in synchronization. However, its performance may not be optimal on certain hardware architectures due to factors such as memory architecture or cache coherence protocols.

1.5 MCS Lock

The MCS (Mellor-Crummey and Scott) lock operates by managing a queue of threads, where each thread patiently waits for its turn to acquire the lock. When a thread wishes to enter a critical section, it adds itself to the end of the queue and then spins on its own flag (in CLH, predecessor's flag was used for spinning), awaiting the release of the lock by its predecessor in the queue. Once the lock is acquired, the thread proceeds to execute its critical section. Upon completion, the thread updates the flag of its successor in the queue, signaling that the successor thread can proceed. The MCS lock is known for its scalability, fairness, and low overhead in synchronization, making it particularly suitable for scenarios with moderate contention among threads. However, like other lock mechanisms, the performance of the MCS lock can be influenced by factors such as hardware architecture and specific system characteristics.

2 Implementation

We have implemented an abstract base class named `Lock`, which declares pure virtual functions `lock()`, `unlock()`, and a virtual destructor. All the five locks derive from this base class and implement the `lock()` and `unlock()` functions. This design leverages runtime polymorphism in the main file for testing purposes.

```
1 class Lock {
2   public:
3   virtual void lock() = 0;
4   virtual void unlock() = 0;
5   virtual ~Lock() = default;
6 };
```

Listing 1. Base `Lock` class

The implementations in the following subsections each provide a specific lock type. These implementations utilize the `std::atomic` library available in C++ and strictly adhere to the C++20 standard. Each lock type is designed to ensure thread safety and synchronization using atomic operations provided by `std::atomic`.

2.1 Test-And-Set Lock (TAS)

The `TAS` class is designed as a Test-and-Set (TAS) lock implementation that inherits from the `Lock` abstract base class. It utilizes a `std::atomic<bool>` variable named `locked` to represent the lock's state. In the `lock()` function, a busy-waiting loop (while loop) continuously attempts to set `locked` to true atomically using `locked.exchange(true)`, effectively acquiring the lock when `locked` was previously false. Conversely, the `unlock()` function simply sets `locked` back to false using `locked.store(false)`, releasing the lock. This implementation ensures thread safety through atomic operations and aligns with the C++20 standard.

```
1  class TAS : public Lock {
2    std::atomic<bool> locked = false;
3
4    public:
5    void lock() {
6      while (locked.exchange(true));
7    }
8
9    void unlock() {
10     locked.store(false);
11   }
12 };
```

2.2 Test-And-Test-And-Set Lock (TTAS)

The `TTAS` class is another lock implementation inheriting from the `Lock` abstract base class. It also utilizes a `std::atomic<bool>` variable named `locked` initialized to false to represent the lock's availability. In the `lock()` function, a busy-waiting loop first checks if `locked` is false with `locked.load()`, and if not, it spins until `locked` becomes false. Once `locked` is observed as false, it attempts to atomically set `locked` to true using `locked.exchange(true)`. If successful (meaning the lock was acquired), the function returns. The `unlock()` function simply sets `locked` back to false using `locked.store(false)`, releasing the lock. This design ensures thread safety through atomic operations and aligns with the C++20 standard.

```
1  class TTAS : public Lock {
2    std::atomic<bool> locked = false;
3
4    public:
5    void lock() {
6      while (true) {
7        while (locked.load());
8
9        if (!(locked.exchange(true))) {
10         return;
11       }
12     }
13   }
```

```
14
15    void unlock() {
16      locked.store(false);
17    }
18  };
```

2.3 Anderson's Lock (ALock)

The `ALock` class is designed to implement a custom array-based locking mechanism inheriting from the `Lock` abstract base class. It uses a combination of `std::atomic<int>` (`tail`) to manage the position in the queue and a `std::vector<bool>` (`flag`) to represent the state of each slot in the lock. Upon initialization using the constructor `ALock(int capacity)`, the `flag` vector is initialized with a size specified by `capacity`, with the first element (`flag[0]`) set to `true` to indicate the initial lock availability. In the `lock()` function, each thread calculates its `slot` by incrementing and taking the modulus of `tail` with `size`, where `size` is the capacity of the lock. The thread then enters a busy-wait loop (`while (!flag[slot])`) until the `flag` at its assigned `slot` becomes `true`, allowing the thread to proceed and acquire the lock. To release the lock, the `unlock()` function accesses the thread's assigned `slot`, sets the `flag` of that `slot` to `false` (indicating the lock is released), and sets the `flag` of the next slot (wrapped around using modulus) to `true`, signaling the next waiting thread to acquire the lock. This approach ensures mutual exclusion among concurrent threads using atomic operations and vector flags for efficient synchronization. Each thread's state is managed locally using `thread_local` storage for thread safety.

```
 1  class ALock : public Lock {
 2    thread_local static int myslot;
 3    std::atomic<int> tail {0};
 4    std::vector<bool> flag;
 5    int size;
 6
 7    public:
 8      ALock(int capacity) : size(capacity) {
 9        flag.assign(size, false);
10        flag[0] = true;
11      }
12
13      void lock() {
14        int slot = (tail++) % size;
15        myslot = slot;
16        while (!flag[slot]);
17      }
18
19      void unlock() {
20        int slot = myslot;
21        flag[slot] = false;
22        flag[(slot + 1) % size] = true;
23      }
24  };
25  thread_local int ALock::myslot = 0;
```

2.4 CLH Lock

The `CLHLock` class extends the `Lock` abstract base class and implements the CLH (Craig, Landin, and Hagersten) lock algorithm using a queue of nodes (`QNode`). The class maintains a `std::atomic<std::shared_ptr<QNode»` variable `tail` to represent the last node in the queue, and utilizes `thread_local` static variables `myPred` and `myNode` to keep track of each thread's predecessor and node within the queue. Upon initialization using the constructor `CLHLock()`, the `tail` is initialized with a new `QNode`, which represents the end of the queue. The `lock()` function sets the `locked` flag of the current thread's `myNode` to `true`, atomically exchanges the `tail` with the current `myNode`, receives the previous `tail` as `pred`, updates `myPred` to point to `pred`, and enters a busy-wait loop (`while (pred->locked) `) to wait until the predecessor node (`pred`) indicates it has released the lock (`locked` is `false`). The `unlock()` function sets the `locked` flag of the current thread's `myNode` to `false`, indicating the lock is released, and resets `myNode` to the predecessor node (`myPred`), effectively removing the current node from the queue.

```
1  class CLHLock : public Lock {
2    std::atomic<std::shared_ptr<QNode>> tail;
3    thread_local static std::shared_ptr<QNode> myPred;
4    thread_local static std::shared_ptr<QNode> myNode;
5
6    public:
7    CLHLock() {
8      tail = std::shared_ptr<QNode>(new QNode());
9    };
10
11   void lock() {
12     myNode->locked = true;
13     std::shared_ptr<QNode> pred = tail.exchange(myNode);
14     myPred = pred;
15     while (pred->locked) {}
16   }
17
18   void unlock() {
19     myNode->locked = false;
20     myNode = myPred;
21   }
22 };
23
24 thread_local std::shared_ptr<QNode> CLHLock::myPred(nullptr);
25 thread_local std::shared_ptr<QNode> CLHLock::myNode(new QNode());
```

2.5 MCS Lock

The `MCSLock` class extends the `Lock` abstract base class and implements the MCS (Mellor-Crummey and Scott) lock algorithm using a linked list of nodes (`QNode`). Key components of this implementation include a `std::atomic<std::shared_ptr<QNode»` variable `tail` to represent the last node in the queue and a `thread_local` static variable `myNode` to store the node specific to each thread. Upon initialization in the constructor `MCSLock()`, `tail` is set

to `nullptr`. In the `lock()` function, a thread atomically exchanges `tail` with `myNode` using `std::atomic::exchange()` with `std::memory_order_acq_rel`. If `tail` was not `nullptr` (indicating other nodes are in the queue), the thread sets its `locked` flag to `true`, links itself to the end of the queue by setting `pred->next = myNode`, and spins on `myNode->locked` until the predecessor node indicates that the thread can proceed. The `unlock()` function first checks if `myNode` has a successor (`myNode->next`). If not, the thread attempts to remove itself from the queue by trying to set `tail` to `nullptr` using `std::atomic::compare_exchange_strong()` with `std::memory_order_acq_rel`. If unsuccessful (indicating there is still a successor in the queue), the thread spins until its successor node is set (`myNode->next != nullptr`). Finally, the thread unlocks by setting the `locked` flag of its successor (`myNode->next->locked`) to `false` and unlinking itself from the queue (`myNode->next = nullptr`).

```
1  class MCSLock : public Lock {
2    std::atomic<std::shared_ptr<QNode>> tail;
3    thread_local static std::shared_ptr<QNode> myNode;
4
5    public:
6    MCSLock() : tail(nullptr) {}
7
8    void lock() {
9      std::shared_ptr<QNode> pred = tail.exchange(myNode,
       std::memory_order_acq_rel);
10     if (pred != nullptr) {
11       myNode->locked = true;
12       pred->next = myNode;
13
14       while (myNode->locked) {}
15     }
16   }
17
18   void unlock() {
19     if (myNode->next == nullptr) {
20       auto store = myNode;
21       if (tail.compare_exchange_strong(myNode, nullptr,
       std::memory_order_acq_rel))
22         return;
23
24       myNode = store;
25       while (myNode->next == nullptr) {}
26     }
27
28     myNode->next->locked = false;
29     myNode->next = nullptr;
30   }
31 };
32
33 thread_local std::shared_ptr<QNode> MCSLock::myNode(new QNode());
```

Note

Both `MCS` and `CLH` Lock implementations utilize `std::shared_ptr` to manage memory and prevent memory leaks. They rely on the objects of type `std::atomic<std::shared_ptr>` objects, which is permissible starting from the C++20 standard. Therefore, compiling and executing the source codes requires using the `-std=c++2a` flag to ensure compatibility with the C++20 features utilized in these implementations.

3 Results

We tested these locks on the following critical section,

```
1 void compute(Lock* lock, int limit) {
2   lock->lock();
3   for (int i = 1; i <= limit; i += 1) {
4     counter += 1;
5   }
6   lock->unlock();
7 }
```

Every thread acquires a lock and increments a counter shared by multiple threads. The limit defines how much work a single thread has to do. We have written the testing code in such a way that no matter the number of threads, the sum of total work done by all the threads remains constant. The table 1 illustrates the average total time taken by program to complete the fixed amount of work with varying number of threads. The average time is obtained by running a shell script that runs the same testing code multiple times and computes the average of all the recorded time taken.

Threads	TAS	TTAS	ALock	CLH	MCS
1	.00137985	.00136906	.00137917	.00137758	.00137496
2	.00141829	.00138010	.00138188	.00138514	.00138427
3	.00143334	.00138936	.00138690	.00138640	.00137270
4	.00155760	.00145621	.00144435	.00145735	.00146042
5	.00151701	.00143975	.00146244	.00147403	.00146861
6	.00146052	.00145205	.00148813	.00149284	.00146278
7	.00151429	.00150594	.00157472	.00154802	.00154943
8	.00157573	.00157697	.00159653	.00159782	.00157763

Table 1. Time taken (in seconds) for different locks with varying threads.

The plot shown in Figure 1 illustrates the performance comparison of different locking mechanisms based on the provided data. When evaluating the performance of these locks, Test-and-Test-and-Set (TTAS) emerges as the most efficient locking method. In contrast, the Test-and-Set (TAS) lock exhibits significant contention, evident from a sharp increase in runtime at thread counts 2, 3, and 4, indicating heightened competition among threads for lock acquisition. Transitioning to queue-based locks such as Anderson's (ALock), CLH,

and MCS, we observe comparable and sometimes superior performance relative to TTAS. These queue-based approaches demonstrate more consistent performance compared to TAS, which shows greater variability in runtime across different thread counts.

Note

The observed plots may exhibit deviations from anticipated behavior due to suboptimal testing conditions. The presence of context switches during testing could potentially mitigate the effects of contention, introducing variability in the results.

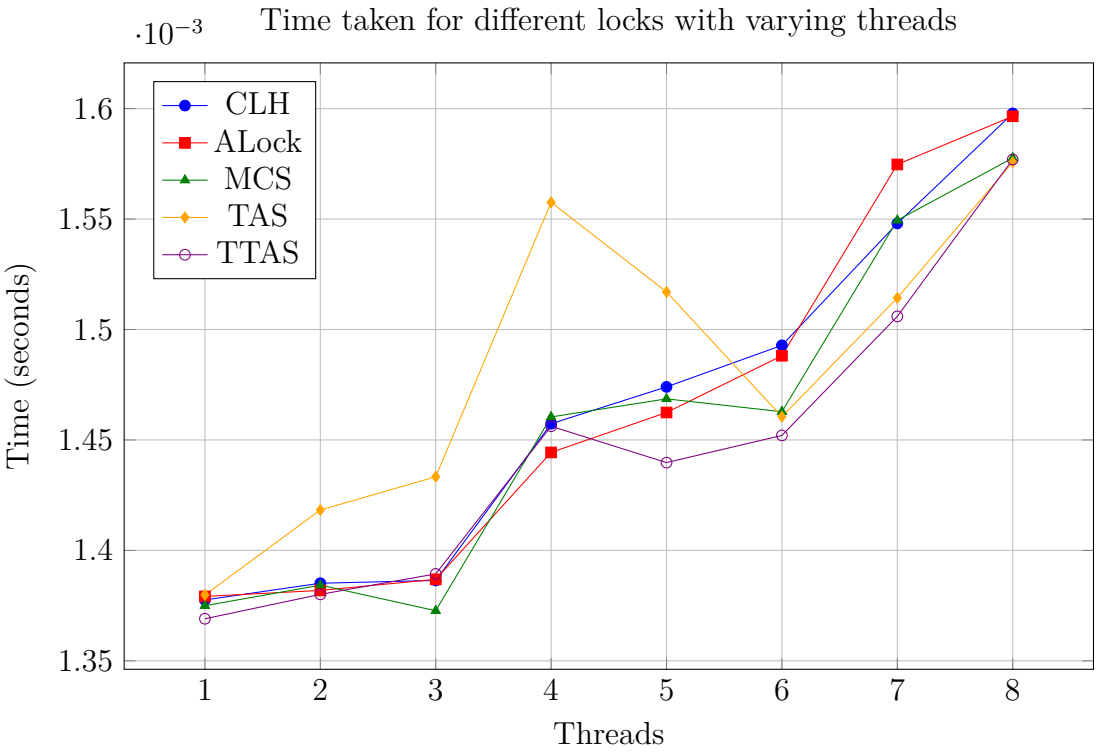

Figure 1. Time taken (in seconds) for different locks with varying threads.

References

[1] Maurice Herlihy and Nir Shavit. "The Art of Multiprocessor Programming, Revised Reprint". 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375.

[2] Stack Overflow Community. "C++ Atomic CAS". Stack Overflow. 2020. URL: https://stackoverflow.com/a/62867549.