Spin Locks and Contention

Shivam Verma (2020CS50442)[a] **and Dr. Smruti Ranjan Sarangi**[a]

[a]Department of Computer Science and Engineering

Indian Institute of Technology, Delhi

Abstract*—This project presents an in-depth analysis of various spin lock implementations and their performance characteristics concerning contention among threads accessing critical sections in concurrent programs. The project focuses on understanding machine architecture's impact on spin locks' efficiency in multiprocessor environments. Different spin lock algorithms are evaluated and compared by measuring the time taken to access critical sections. The report discusses the significance of comprehending underlying machine architectures in designing efficient concurrent programs, emphasizing the importance of selecting appropriate locking mechanisms. Through empirical evaluation and comparison, insights are drawn into the suitability of different spin lock algorithms for minimizing contention and optimizing performance in multiprocessor systems.*

***Keywords*—** *Spin locks, multiprocessor programming, contention, critical section, concurrency, performance analysis.*

1. Introduction

We implemented 5 different types of spin locks.

1. Test-And-Set Lock (TAS)
2. Test-And-Test-And-Set Lock (TTAS)
3. Anderson Lock (ALock)
4. CLH Lock
5. MCS Lock

1.1. Test-And-Set Lock (TAS)

The Test-And-Set Lock (TAS) relies on an atomic instruction that tests a memory location and sets it to a new value based on the result of the test. When a thread wants to enter a critical section, it executes the Test-And-Set operation on the lock's memory location. If the location indicates that the lock is unlocked, the thread sets it to locked and enters the critical section. If the lock is already locked, the thread typically enters a busy-waiting loop until it successfully acquires the lock. Once the critical section is executed, the lock is released, allowing other threads to acquire it. While simple, Test-And-Set locks may suffer from inefficiencies due to busy-waiting.

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. It aims to enhance performance by reducing contention and overhead. TTAS operates by first performing a test to check the lock's state. If it's unlocked, a secondary test is conducted before attempting to set the lock, allowing other threads a chance to release it. If both tests indicate the lock is free, the thread acquires the lock. Otherwise, it enters a busy-waiting loop until the lock becomes available. TTAS improves performance by minimizing the number of threads actively spinning on the lock, but it may still face inefficiencies in heavily contended scenarios.

1.3. Anderson's Lock

Anderson's Lock, also known as the Anderson Queue Lock, involves maintaining an array of Boolean flags, each representing whether a thread is waiting for the lock. Threads atomically claim a position in the queue and set their corresponding flag to true before entering a busy-waiting loop. When a thread finishes executing its critical section, it releases the lock by setting its flag to false, allowing the next thread in the queue to proceed. Anderson's Lock offers scalability, fairness, and low overhead but may not perform optimally in scenarios with a large number of contending threads.

1.4. CLH Lock

The CLH (Craig, Landin, and Hagersten) lock maintains a queue of threads, where each thread waits its turn to acquire the lock. When a thread wants to enter a critical section, it creates a node indicating its intention and spins on its predecessor's node until it's its turn. Once a thread finishes its critical section, it releases the lock, allowing the next thread in the queue to proceed. The CLH lock provides scalability, fairness, and low overhead but may not perform optimally in certain hardware architectures.

1.5. MCS Lock

The MCS (Mellor-Crummey and Scott) lock operates by maintaining a queue of threads, each waiting its turn to acquire the lock. When a thread wants to enter a critical section, it adds itself to the end of the queue and spins on its own flag until its predecessor releases the lock. Once acquired, the thread proceeds to execute its critical section. When finished, it updates its successor's flag, allowing the next thread to proceed. The MCS lock provides scalability, fairness, and low overhead, making it suitable for scenarios with moderate contention among threads.

2. Implementation Details

We have created an abstract base class called **Lock**. The 5 different types of spin locks inherit this base class and override the *lock()* and *unlock()* functions.

I have added the *lock()* and *unlock()* functions of different spin locks below:

```
TASLock::TASLock() {
    state.store(false);
}

void TASLock::lock() {
    while (state.exchange(true)) {}
}

void TASLock::unlock() {
```

```
    state.store(false);
}

void TASLock::type() {
    std::cerr << "\e[1;33m \u26BF \e[0m";
    std::cerr << "TAS Lock used\n";
}
```

Code 1. *TAS Lock*

```
TTASLock::TTASLock() {
    state.store(false);
}

void TTASLock::lock() {
    while (state.load()) {}
    while (state.exchange(true)) {}
}

void TTASLock::unlock() {
    state.store(false);
}

void TTASLock::type() {
    std::cerr << "\e[1;33m \u26BF \e[0m";
    std::cerr << "TTAS Lock used\n";
}
```

Code 2. *TTAS Lock*

```
ALock::ALock (int capacity) {
    size = capacity;
    tail.store(0);
    flag.assign(size, false);
    flag[0] = true;
}

void ALock::lock() {
    int slot = (tail.fetch_add(1)) % size;
    mySlotIndex = slot;
    while (!flag[slot]) {};
}

void ALock::unlock() {
    int slot = mySlotIndex;
    flag[slot] = false;
    flag[(slot + 1) % size] = true;
}

void ALock::type() {
    std::cerr << "\e[1;33m \u26BF \e[0m";
    std::cerr << "Anderson Lock used\n";
}

thread_local int ALock::mySlotIndex = 0;
}
```

Code 3. *Anderson's Lock*

```
CLHLock::CLHLock() {
    tail = std::shared_ptr<QNode>(new QNode());
}

CLHLock::~CLHLock() {}

void CLHLock::lock() {
    myNode->locked = true;
    std::shared_ptr<QNode> pred = tail.exchange(
    myNode);
    myPred = pred;
    while (pred->locked) {}
}

void CLHLock::unlock() {
    myNode->locked = false;
    myNode = myPred;
}

void CLHLock::type() {
    std::cerr << "\e[1;33m \u26BF \e[0m";
    std::cerr << "CLH Lock used\n";
}

thread_local std::shared_ptr<QNode> CLHLock::
    myPred(nullptr);
thread_local std::shared_ptr<QNode> CLHLock::
    myNode(new QNode());
```

Code 4. *CLH Lock*

```
MCSLock::MCSLock() {
    tail = nullptr;
}

MCSLock::~MCSLock() {}

void MCSLock::lock() {
    auto pred = tail.exchange(myNode);
    if (pred != nullptr) {
        myNode->locked = true;
        pred->next = myNode;
        while (myNode->locked) {}
    }
}

void MCSLock::unlock() {

    if (myNode->next == nullptr) {
        auto f = myNode;
        if (tail.compare_exchange_strong(myNode,
     nullptr, std::memory_order_acq_rel)) {
            return;
        }
        myNode = f;
        while (myNode->next == nullptr) {}
    }
    myNode->next->locked = false;
    myNode->next = nullptr;
}

void MCSLock::type() {
    std::cerr << "\e[1;33m \u26BF \e[0m";
    std::cerr << "MCS Lock used\n";
}

thread_local std::shared_ptr<QNode> MCSLock::
    myNode(new QNode());
```

Code 5. *MCS Lock*

The *std::atomic<T>* class in C++ is used for writing the atomic instructions. Although the **aux/** folder in the project directory also contains the atomic instructions in assembly using ASM directives.

In the main testing file, different numbers of threads are spawned and each thread tries to access the critical section where a shared counter is incremented.

```
auto criticalSection = [&](Lock* f) {
    f->lock();
        for (int i = 0; i < 1E7; i++) {
            int f = 0;
        }
        for (int i = 1; i <= 10000; i += 1) {
            val += 1;
```

```
        }
        for (int i = 0; i < 1E7; i++) {
            int f = 0;
        }
    f->unlock();
};

```

Code 6. *Critical Section*

3. Results

The table 1 on the next page illustrates the average time taken by a thread to access the critical section when utilizing different locks. This evaluation was conducted for $n = 1, 2, \ldots, 7$ threads, and the time is measured in microseconds.

Furthermore, plot 1 presents the number of threads on the X-axis versus the average time taken by a thread to complete executing the critical section on the Y-axis. The times for different locks are depicted using various colors.

4. Analysis

As more threads join in, it takes longer on average for each thread to finish doing its job in the critical section. This happens because more threads are competing for the same resources at the same time. This competition, called contention, causes delays as threads wait for their turn to use the resources.

In the case of spin locks, when a thread tries to get a lock that another thread is already using, it keeps checking repeatedly until the lock becomes available. When there are lots of threads competing, more of them end up stuck in this checking loop, using up the computer's resources without getting much done. This wastes time and slows down the whole system.

Threads trying to get locks may also have to wait longer if the locks are already in use. How long they wait can depend on things like how the operating system decides which threads to run, how many threads are competing, and how long the locks are held by other threads.

When we compare different types of locks, we notice that Test-and-Set (TAS) locks have the most contention, meaning more threads are fighting for them. As we move to other types like Test-and-Test-and-Set (TTAS), Anderson's, CLH, and MLS queue spin locks, the performance gets better because there's less contention.

But sometimes, when we look at the plots showing performance, we see some areas where the lines overlap. This usually happens because the tests weren't done in perfect conditions.

> **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.*

5. References

1. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Revised First Edition. Morgan Kaufmann, 2012.

Table 1. *Time Taken (in microseconds) for Different Locks with Varying Threads*

Threads	TAS	TTAS	ALock	CLH	MCS
1	44922	44873	44480	44655	43887
2	45284	45131	45056	45023	45002
3	46194	46036	46002	45727	45714
4	46255	46051	46046	46005	46170
5	47114	47017	47059	46586	46105
6	47629	47609	47273	47190	46430
7	48969	48695	48456	48120	47825

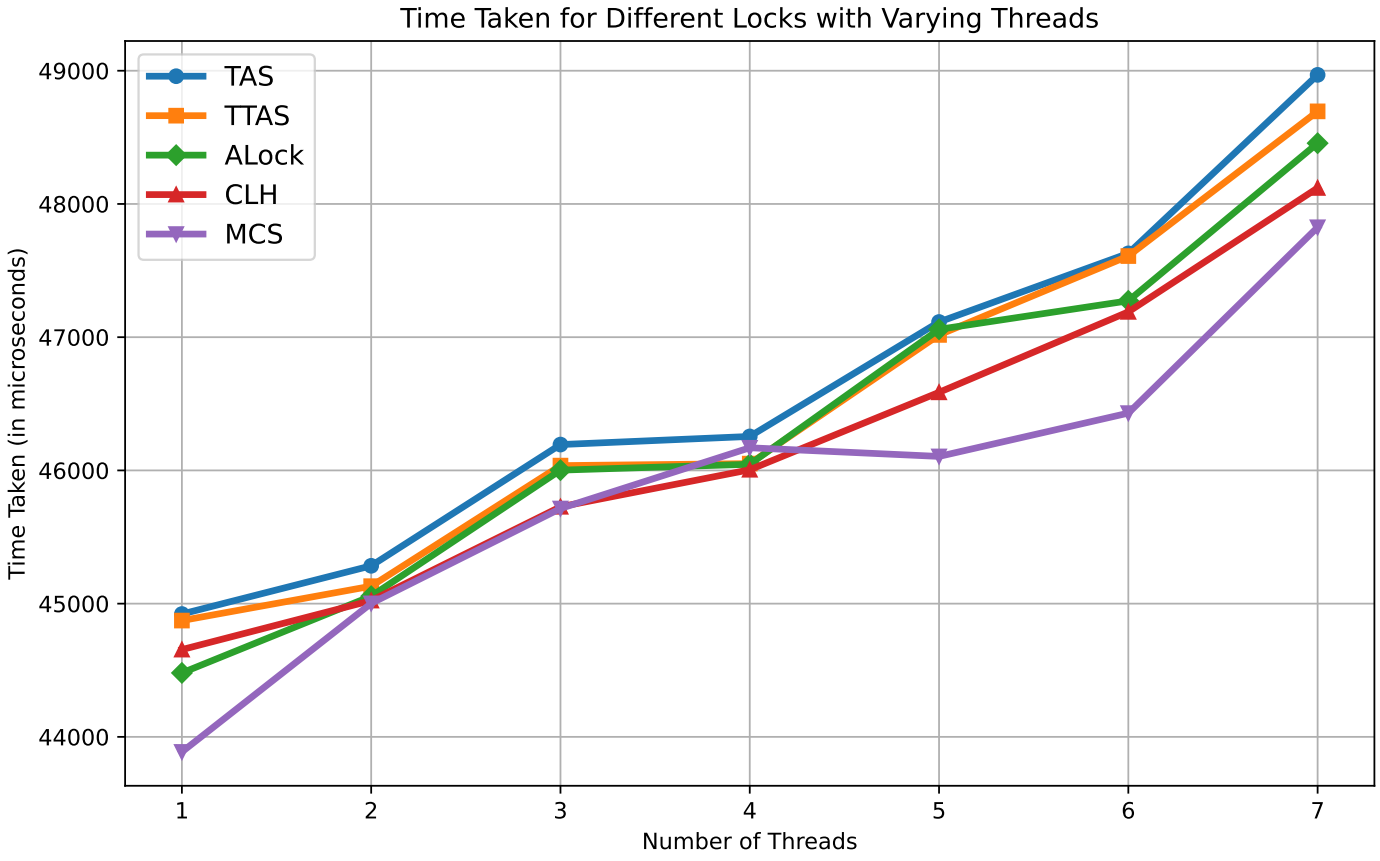

Figure 1. *Contention in Spin Locks*