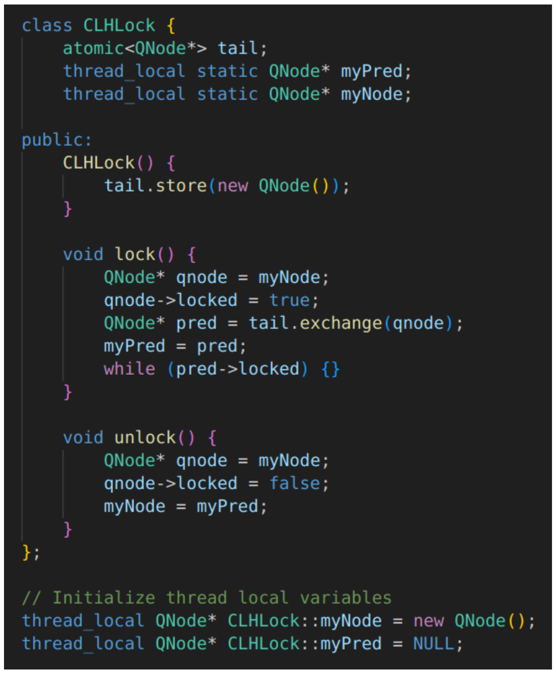
Comparison of CLH and MCS Queue Locks

Aaryan, CO21BTECH11001

Program Design:

1. **Thread local variables in c++:** We can declare thread local variables inside a class but the initialization of those variables has to be done outside the class. Also, we have to use the `static` keyword while declaring such variables. Here is the way to do so:



```
class CLHLock {
    atomic<QNode*> tail;
    thread_local static QNode* myPred;
    thread_local static QNode* myNode;

public:
    CLHLock() {
        tail.store(new QNode());
    }

    void lock() {
        QNode* qnode = myNode;
        qnode->locked = true;
        QNode* pred = tail.exchange(qnode);
        myPred = pred;
        while (pred->locked) {}
    }

    void unlock() {
        QNode* qnode = myNode;
        qnode->locked = false;
        myNode = myPred;
    }
};

// Initialize thread local variables
thread_local QNode* CLHLock::myNode = new QNode();
thread_local QNode* CLHLock::myPred = NULL;
```

2. **Time measurements:** To measure time, `chrono` library is used. Program finds the current time at various points for the following purposes:
    a. To check the correctness: The times when a thread acquires the lock and when it requests to exit are measured and printed in the log file. To check the correctness, their difference from the starting time (time at which thread creation started) is taken and printed in the log file. Since multiple threads can acquire the lock within a second, the order of accuracy of this difference is nanoseconds.

b. <u>To find the average waiting time</u>: A variable named `csEnterTime` is declared globally and initialized to `0`. It measures the total waiting time in microseconds. Every thread measures its waiting time and increments this variable.
In the end, average waiting time (in microseconds) is calculated by `csEnterTime/(n*k)`.

c. <u>Random sleep time</u>: To ensure that all the threads produce a different sequence of random numbers for sleep time , the random number generator is seeded with a value equal to the product of the thread-id and the current time. This also ensures that different random numbers are generated in different executions of the program.

d. <u>Throughput calculation</u>: To find throughput, the start time and end time are measured and their difference is calculated in order of accuracy of microseconds. Then throughput (in operations/s) is calculated by `(n*k*1e6)/(time_taken_in_microseconds)`.

- **Throughput analysis with varying threads:**

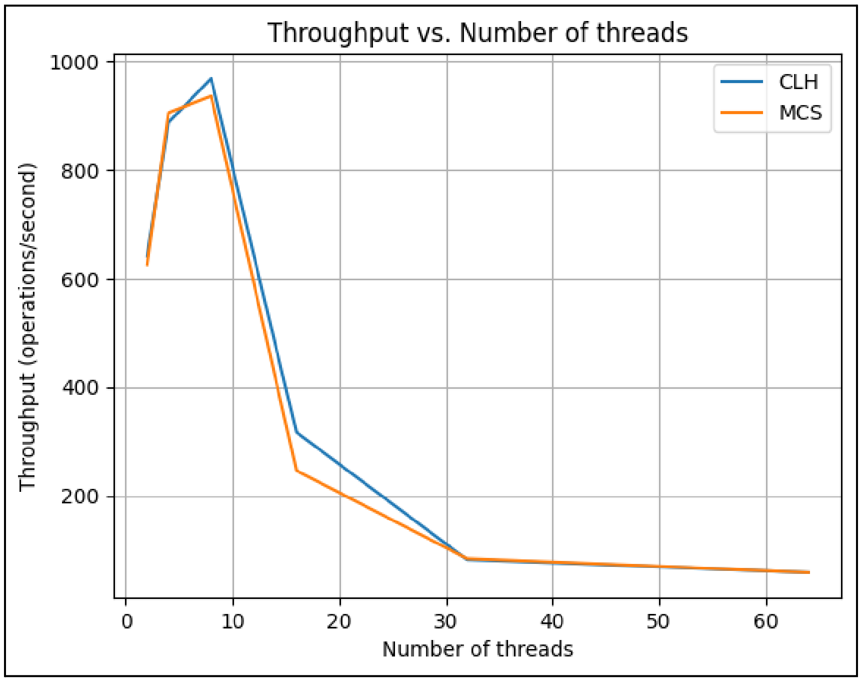


Figure 1: Throughput vs. Number of threads, k= 15 is kept constant

- **Average Entry Time Analysis with varying threads:**

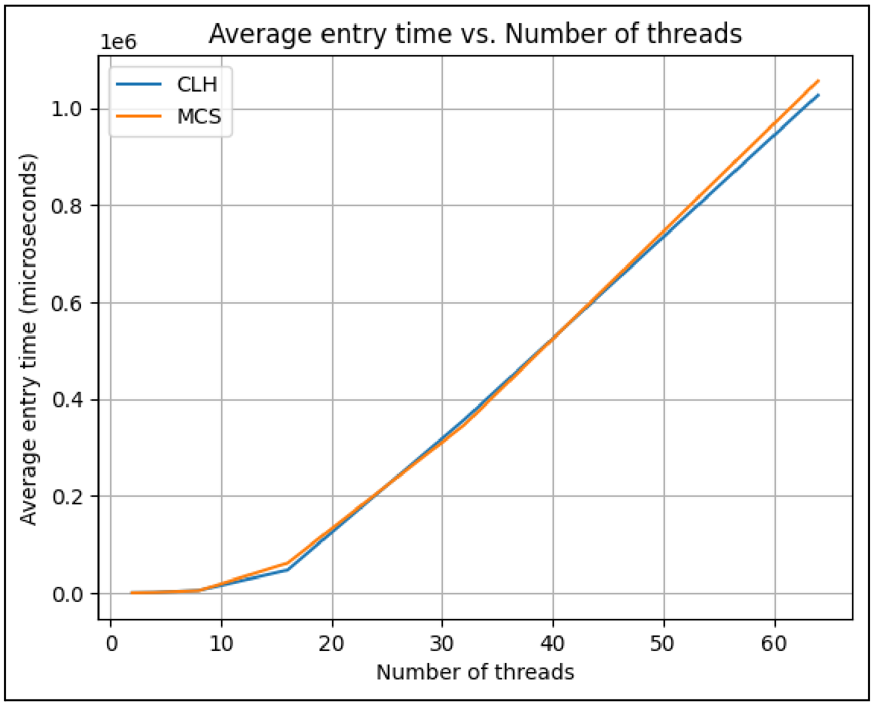


Figure 2: Average entry time vs. Number of threads, k= 10 is kept constant

- **Throughput Analysis with varying k:**

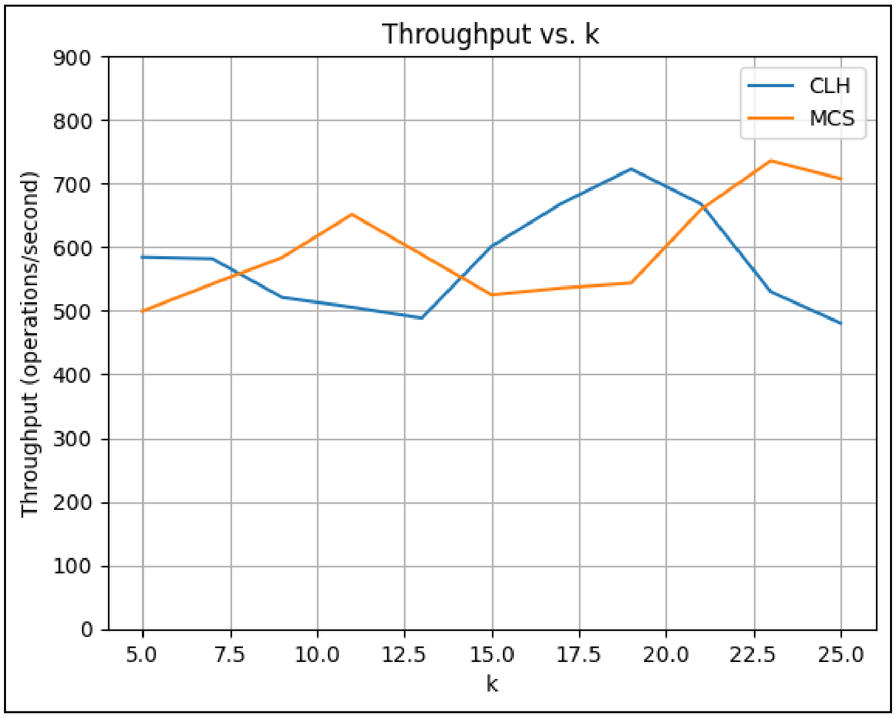


Figure 3: Throughput vs k, n = 16 is kept constant

- **Average Entry Time Analysis with varying k :**

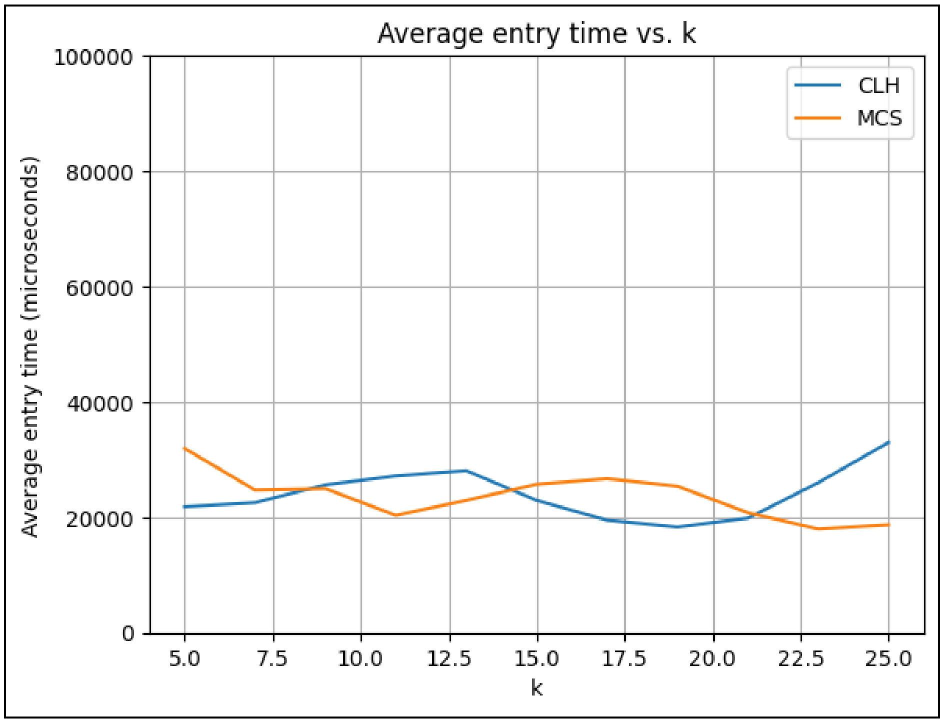


Figure 4: Average entry time vs. k, n = 16 is kept constant

- **Observations and Conclusion:**

1. When the number of threads are varied and k is kept constant, waiting time increases (refer Figure 2). The reason is that more threads means more nodes in the queue. Therefore, in case of a CLH lock, a thread will spend more time till the locked value of its predecessor is set to false. And in case of an MCS lock, a thread will spend more time till its own locked value is set to false by its predecessor.

2. The number of cores in my computer is 4. Therefore, throughput increases till the number of threads is 8 and decreases after that (refer Figure 1). The reason for the decrease in throughput is the overhead of managing the threads when they exceed the suitable limit of threads (which is 8 in this case).

3. When k is changed and the number of threads is kept constant, the average waiting time should remain almost the same because it depends on the amount of contention which only depends on the

number of threads. In Figure 4, we can see that the average waiting time remains practically constant when k is changed and the number of threads is kept constant.

4. When k is changed and the number of threads is kept constant, the throughput should remain almost the same because it depends on the amount of contention and the number of cores in the system. Contention depends only on the number of threads. In Figure 3, we can see that the throughput remains practically constant when k is changed and the number of threads is kept constant.

5. The performance of MCS and CLH locks is almost the same. The reason why CLH lock is performing as good as MCS lock is because the cost of spinning on some other thread's memory is very less in a normal computer. The book titled The Art of Multiprocessor Programming quotes "*Perhaps the only disadvantage of this (CLH) lock algorithm is that it performs poorly on cacheless NUMA architectures. Each thread spins waiting for its predecessor's node's locked field to become false. If this memory location is remote, then performance suffers. On cache-coherent architectures, however, this approach should work well.*".
   Since we are running the threads on cache-coherent architecture based computers, CLH performs as good as MCS.