Aayush Kumar
CO21BTECH11002
Assignment 5

CLHLock Class:

The CLHLock class contains an atomic<QNode*> variable named tail, which represents the tail of the CLH lock queue. It is initialized with a new QNode and stored as the initial tail. The lock function is used to acquire the lock. myNode is a reference to the calling thread's node in the CLH queue. myPred is a reference to the calling thread's predecessor's node. qnode is a pointer to the calling thread's node. (*qnode).locked is set to true, indicating that the thread is attempting to acquire the lock. tail.exchange(qnode) atomically replaces the tail of the queue with qnode and returns the previous tail (pred). myPred is set to the predecessor (pred). The function then spins in a loop, waiting for the predecessor to release the lock. The unlock function is used to release the lock. myNode is a reference to the calling thread's node in the CLH queue. myPred is a reference to the calling thread's predecessor's node. (*qnode).locked is set to false, indicating that the thread is releasing the lock. Finally, myNode is set to myPred so that the myPred node can be used later as myNode if the same thread calls lock again.

MCSLock Class:

The MCSLock class contains an atomic<QNode*> variable named tail, which represents the tail of the MCS lock queue. It is initialized to NULL to signify that the queue is initially empty. The lock function is used to acquire the lock. myNode is a reference to the calling thread's node in the MCS queue. qnode is a pointer to the calling thread's node. pred is a pointer to the previous tail of the queue (the predecessor). If pred is not NULL, indicating that there is a predecessor, the calling thread sets its own locked field to true, updates the predecessor's next pointer to point to itself, and

then enters a spin loop, waiting for the predecessor to release the lock. The unlock function is used to release the lock. myNode is a reference to the calling thread's node in the MCS queue. If the next pointer of the calling thread's node is NULL, it means there is no successor, and the thread checks if the tail is still its own node. If so, it sets the tail to NULL and returns, indicating that the queue is now empty. If there is a successor, it spins in a loop until the successor's next pointer is set. Finally, it sets the successor's locked field to false and updates the calling thread's next pointer to NULL, indicating that it has released the lock.

TestCS() function:

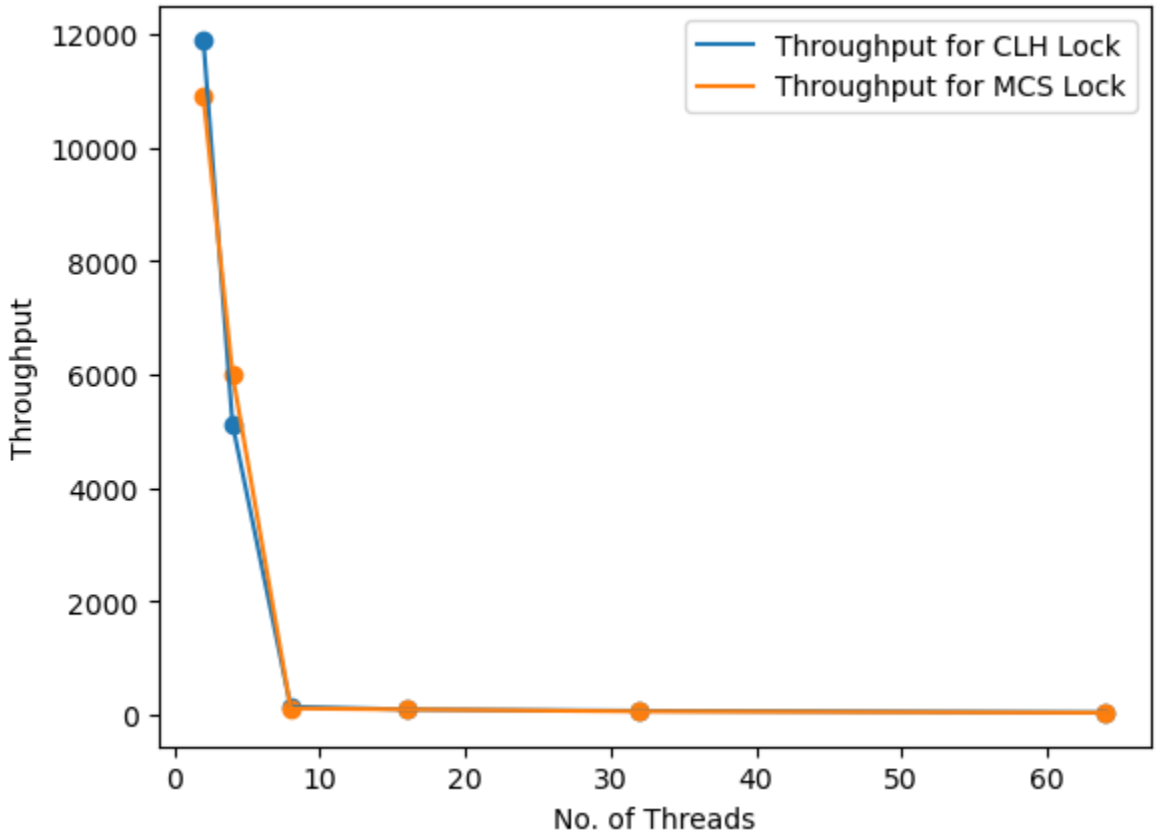
The function enters a loop to simulate the thread attempting to enter and exit the critical section k times. The function records the time when the thread requests entry to the critical section and logs this information to an output file (f_out). The thread then attempts to acquire the lock. If the lock is acquired, the thread records the time when it actually enters the critical section and logs this information. The time taken to enter the critical section is added to the total time spent to enter the critical section (csEnterTime). The thread sleeps for an exponentially distributed random time (t1) to simulate some processing within the critical section. The thread logs the time when it requests to exit the critical section. The thread then releases the lock. The thread records the time when it actually exits the critical section and logs this information. The thread sleeps for another exponentially distributed random time (t2) to simulate some processing outside the critical section.

Main function:

The main function reads the input from the file and creates n threads. It then waits for all the threads to finish and then computes the average time taken to enter the Critical Section and throughput.

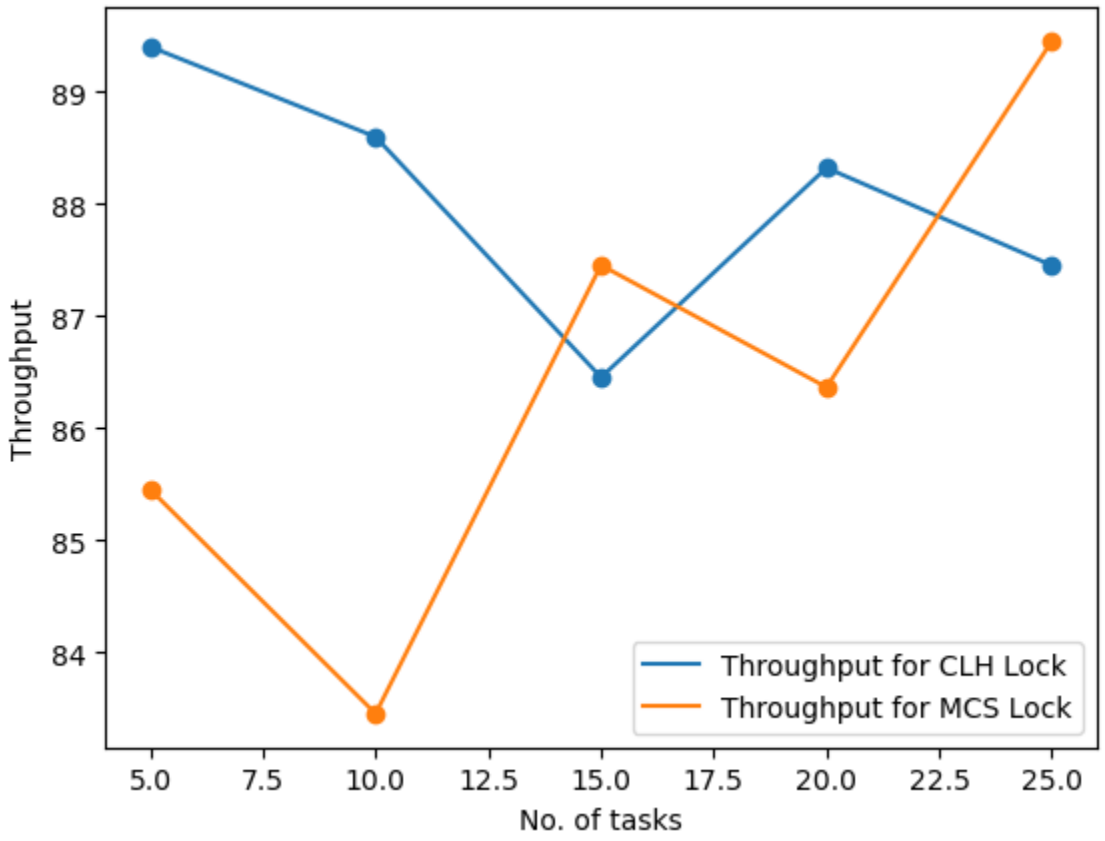
Results:

No. of threads vs Throughput (keeping k = 15, lambda1 = 1, lambda2 = 2)



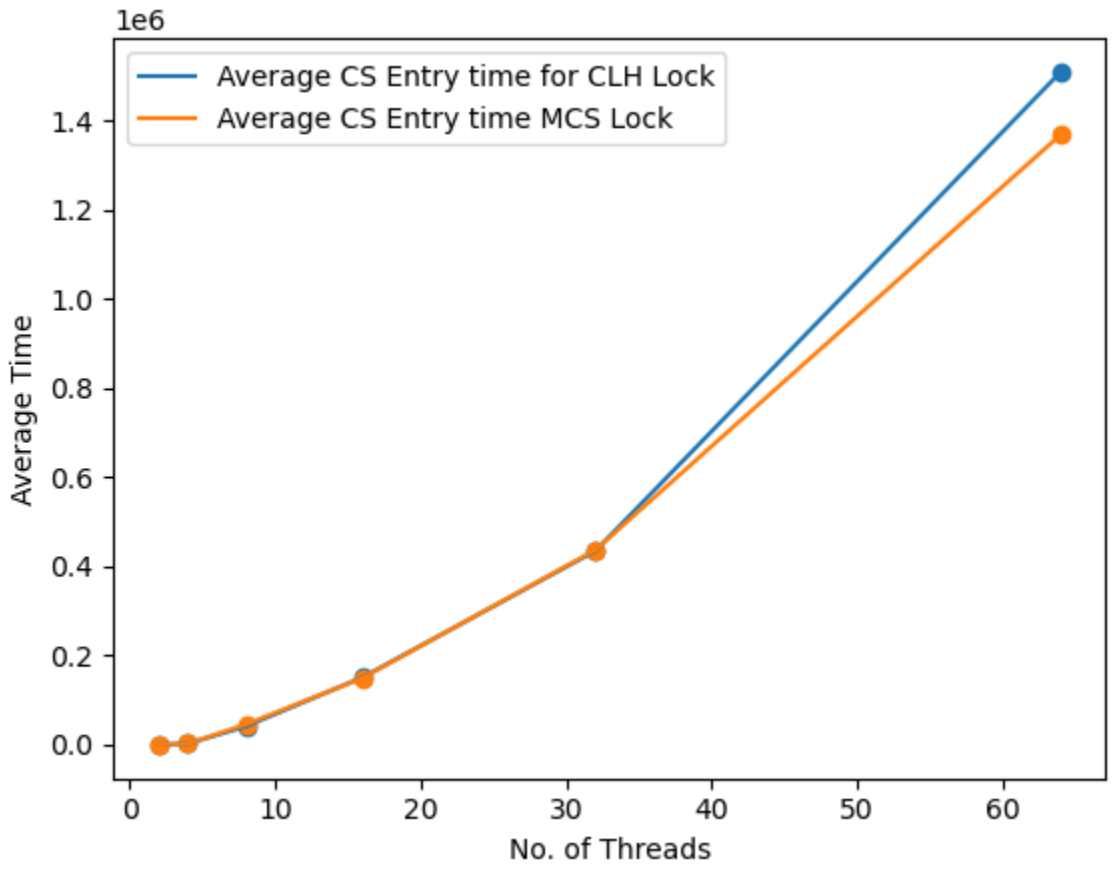
- Throughput decreases as the number of threads increases.
- As the number of threads increases, the contention on the lock increases and each thread has to wait longer to obtain the lock. Hence, they perform less operations per second.
- CLH and MCS Locks perform similarly.

Number of tasks per thread vs Throughput (keeping n = 16, lambda1 = 1, lambda2 = 2)



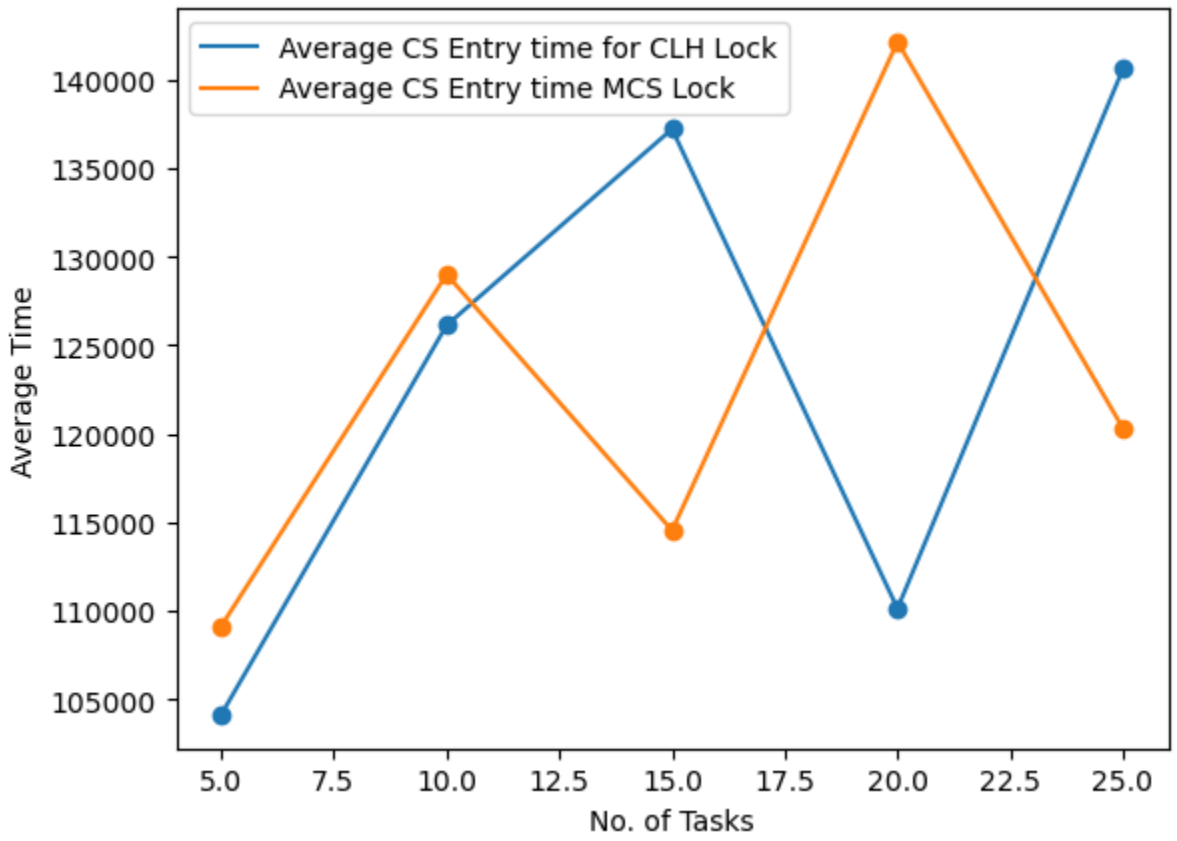
- Throughput does not follow a fixed trend on increasing the number of tasks. But, it consistently maintains an almost constant value as its value always lies around 80 to 90.
- CLH and MCS locks perform similarly.

Average CS entry time in (microseconds, scaled by 1e6) vs number of threads (keeping k = 10, lambda1 = 1, lambda2 = 2)



- Average CS entry time increases as the number of threads increases.
- As the number of threads increases, the contention on the lock increases and each thread has to wait longer to obtain the lock. Hence, they have to wait longer to enter the critical section.
- CLH and MCS Locks perform similarly.

Average CS entry time (in microseconds) vs number of tasks per thread (keeping n = 16, lambda1 = 1, lambda2 = 2)



- Average CS entry time does not follow a fixed trend on increasing the number of tasks. But, it consistently maintains an almost constant value as its value always lies around 0.1 to 0.14 seconds.
- CLH and MCS locks perform similarly.