Assignment-5:Implementing TAS,TTAS,CLH,MCS Locks

Submitted By: Kushagra Indurkhya
CS19B1017

Implementation Designs

Files:

```
tree | grep .cpp
├── CLH-CS19B1017.cpp
├── MCS-CS19B1017.cpp
├── TAS-CS19B1017.cpp
├── TTAS-CS19B1017.cpp
├── qnode.cpp
├── test_app.cpp
└── thread_local.cpp
```

- test_app.cpp: contains the code for the test application implemented as described in the problem statement common to each Lock file called in the main of each individual Lock file
- qnode.cpp : Contains the class of a queue node being used in CLH and MCS locks

```cpp
/* A QNode is a node in a queue */
class QNode
{
public:
   QNode *next;
   QNode *prev;
   bool locked;
/**
* A constructor for the QNode class. It initializes the next and prev pointers to NULL and the locked
* variable to false.
*/
   QNode()
   {
       next = NULL;
       prev = NULL;
       locked = false;
   }
};
```

- CLH-CS19B1017.cpp,MCS-CS19B1017.cpp,TAS-CS19B1017.cpp,TTAS-CS19B1017.cpp are the implementations of the TAS,TTAS,CLH and MCS lock as described in the book and discussed in the class
- thread_local.cpp:
    - In java, ThreadLocal<T> provides thread-local storage for class members while in c++ the implementation thread_local keyword is for static class members only.So to use for our purposes I implemented my own ThreadLocal class:

```
/* This code is for implementing ThreadLocal functionality for class members. */
template <typename T>
class ThreadLocal
{
public:
   /**
    * It creates a new thread local variable.
    */
   ThreadLocal() {}
   T *arr;
   /**
    * `ThreadLocal` is a template class that takes a type `T` as a template parameter.
It has a
    * constructor that takes an integer `n` as a parameter and creates an array of
type `T` of size `n`
    *
    * @param n The number of elements in the array.
    */
   ThreadLocal(int n)
   {
       arr = new T[n];
   }
   /**
    * The function takes a value and an integer as arguments and creates an array of
the given size and
    * initializes it with the given value
    *
    * @param val The value to be assigned to each element of the array.
    * @param n The number of threads that will be accessing the ThreadLocal object.
    */
   ThreadLocal(T val, int n)
   {
       arr = new T[n];
       for (int i = 0; i < n; i++)
           arr[i] = val;
   }
   /**
    * Set the value of the element at index id to val.
    *
    * @param id The id of the element you want to access. This is a 0-indexed number.
    * @param val The value to set the element to.
    */
   void set(int id, T val)
   {
       arr[id] = val;
   }
   /**
    * Returns the value of the element at the given index.
    *
    * @param id The id of the object you want to get.
    *
    * @return The value of the array at the index of the id.
    */
   T get(int id)
   {
       return arr[id];
   }
};
```

- The working principle of this solution is when a ThreadLocal object is created it creates an array of elements of length n where n is the number of threads that

may access this object(can be made fixed to MAX_THREADS that in our problem statement is defined as 100)

- The get and set takes an id as argument and access arr[i] location only
- I is a custom thread id for our use case which ranges from 0 to n-1
- I have used array because it has a very clear implementation with contiguous memory allocation so we know for sure no concurrency issues will take place.
- To prevent using custom id we would have needed some sort of concurrent map structure and used thread.GetId() as a key but that would have required a concurrent map data structure to avoid those complexities i have used an array
- Due to the usage of local id for thread_lcoal, the interface of lock class i have customized to receive thread_id:

```
/* Lock is an abstract class that defines a lock and unlock method. */
class Lock
{
public:
   /* These are pure virtual function,used to make the class abstract. */
   virtual void lock(int) = 0;
   virtual void unlock(int) = 0;
};
```

Algorithms

TAS

```
class TASLock : public Lock
{
   atomic<bool> state;

public:
   TASLock() : state(false) {}
   void lock(int t_id)
   {

       while (state.exchange(true))
       {
       }
   }
   void unlock(int t_id)
   {
       state.store(false);
   }
};
```

TTAS

```
class TTASLock : public Lock
{
   atomic<bool> state;

public:
   TTASLock() : state(false) {}
   void lock(int n)
   {
       while (true)
       {
           while (state.load())
           {
           };
           if (!state.exchange(true))
               return;
       }
   }
   void unlock(int n)
   {
       state.store(false);
   }
};
```

CLH

```
class CLHLock : public Lock
{
   atomic<QNode *> tail;
   ThreadLocal<QNode *>* myNode;
   ThreadLocal<QNode *>* myPred;

public:
   CLHLock(int n)
   {
       tail.store(new QNode());
       myNode = new ThreadLocal<QNode *>(n);
       for (int i = 0; i < n; i++)
       {
           myNode->arr[i]=new QNode();
       }
       myPred = new ThreadLocal<QNode *>(nullptr, n);
   }
   void lock(int t_id)
   {
       QNode *qnode = myNode->get(t_id);
       qnode->locked = true;
       QNode *pred = tail.exchange(qnode);
       myPred->set(t_id, pred);
       while (pred->locked){}
   }
   void unlock(int t_id)
   {
       QNode *qnode = myNode->get(t_id);
       qnode->locked = false;
       myNode->set(t_id, myPred->get(t_id));
   }
};
```

- Here my node is a thread_local which is an array of qnodes for each thread to access initialized with a new Qnode
- While myPred is a thread_local initialized with nullptr

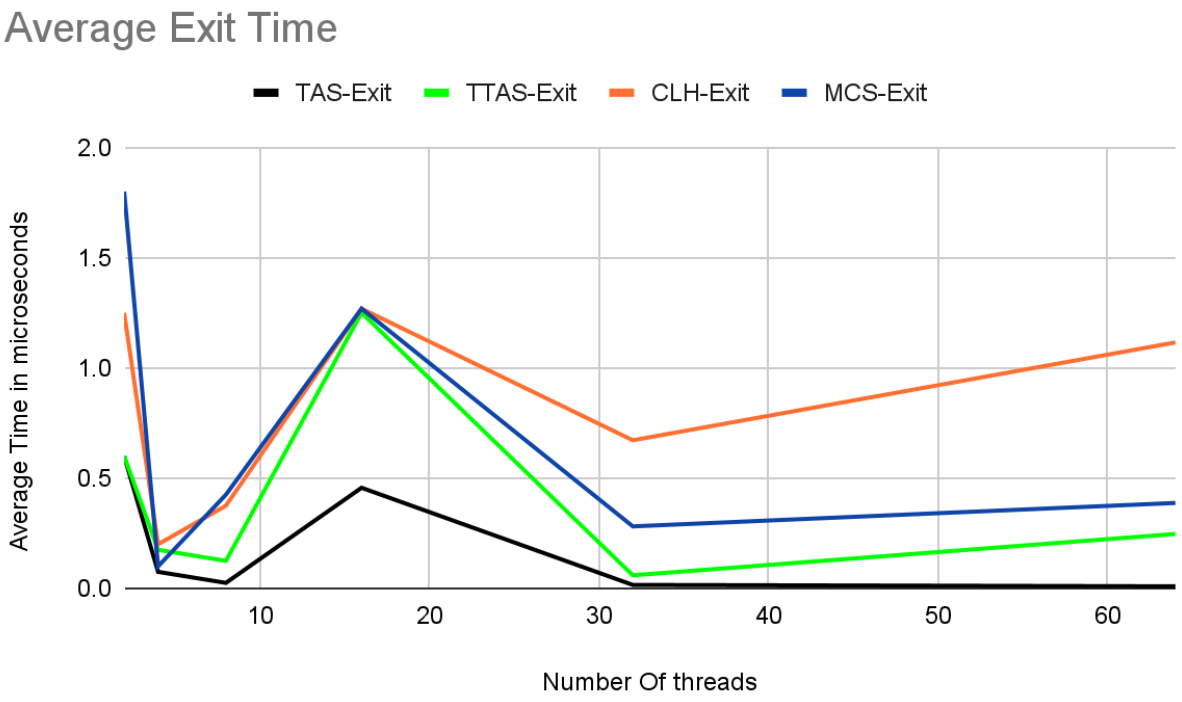
MCS

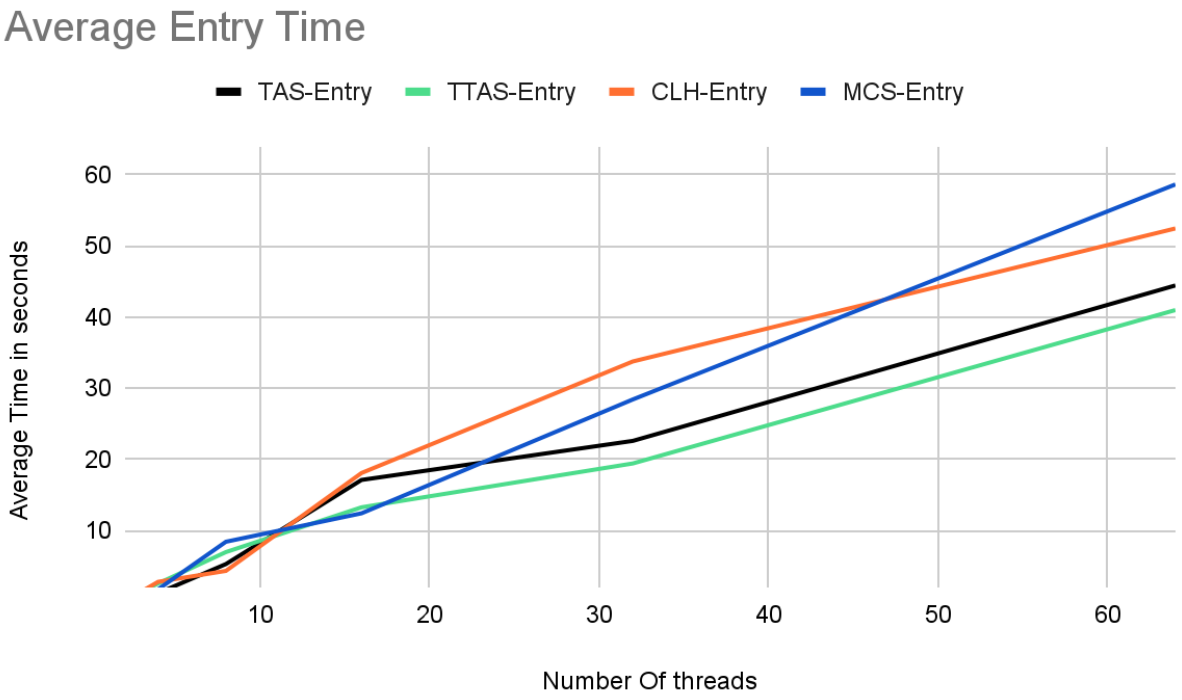
```
class MCSLock : public Lock
{
   atomic<QNode *> tail;
   ThreadLocal<QNode *> *myNode;

public:
   MCSLock(int n)
   {
       tail.store(nullptr);
       myNode = new ThreadLocal<QNode *>(n);
       for (int i = 0; i < n; i++)
       {
           myNode->arr[i] = new QNode();
       }
   }
   void lock(int t_id)
   {
       QNode *qnode = myNode->get(t_id);
       QNode *pred = tail.exchange(qnode);
       if (pred != nullptr)
       {
           qnode->locked = true;
           pred->next = qnode;
           while (qnode->locked)
           {
           }
       }
   }
   void unlock(int t_id)
   {
       QNode *qnode = myNode->get(t_id);
       if (qnode->next == NULL)
       {
           if (tail.compare_exchange_strong(qnode, NULL))
               return;
           while (qnode->next == NULL)
           {
           }
       }
       qnode->next->locked = false;
       qnode->next = NULL;
   }
};
```

- Here my node is a thread_local which is an array of qnodes for each thread to access initialized with a new Qnode

Results and Comparisons





We observe the following from the results :

- TTAS performs better than TAS for almost all the case when number of threads is large (>4),this was expected as TTAS's performance is better than TAS because the in TAS we are repeatedly using exchange() calls while in TTAS we are using exchange() only when we see load() returns the call,this improves the performance as load() call is inherently only reading hence makes use of the cache on the processor level while exchange needs memory access.
- MCS and CLH are performing worse than TAS and TTAS because of the simulated implementation of thread local which is directly accessing the memory locations which is a simulation of thread_local thus the **implementation is giving correct functionality but not using the processor cache at hardware level thus its performance is hampered.** as we are effectively using these algorithm on a cacheless architecture.