Implementing Atomic MRMW Register

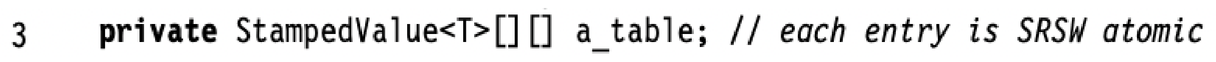
Kushagra Indurkhya
(CS19B1017)

Introduction

For the implementation of Atomic MRMW, I have used the algorithms from implementations shown in fig 4.10(Stamped Value),4.11(SRSW),4.12(MRSW),4.14(MRMW) of the book which were discussed in class as well.

In atomic SRSW as mentioned in the problem statement i have considered c++ variables as regular registers

In fig 4.12

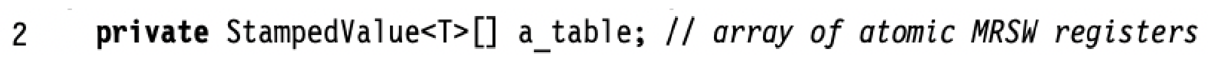


```
3    private StampedValue<T>[][] a_table; // each entry is SRSW atomic
```

Is Implemented as:

```
AtomicSRSWRegister<StampedValue<T> > **a_table; // each entry is SRSW atomic
```

In fig 4.14



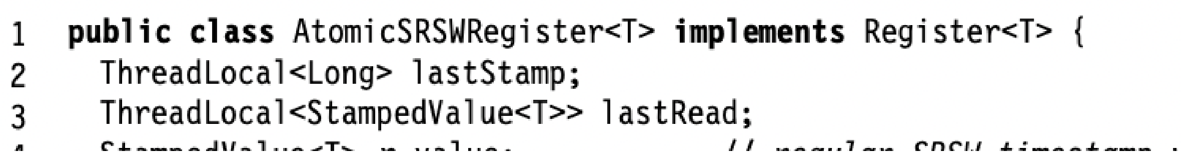
```
2    private StampedValue<T>[] a_table; // array of atomic MRSW registers
```

Is implemented as:

```
AtomicMRSWRegister<StampedValue<T> > *a_table;
```

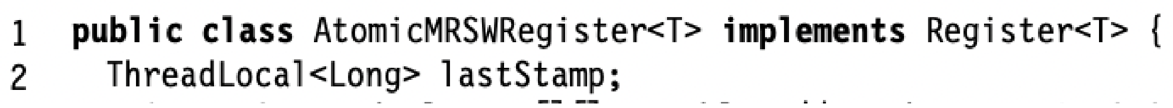
ThreadLocal Implementation

A major roadblock in the implementation of the Atomic SRSW and MRSW was they required ThreadLocal<T> object:
Atomic SRSW: Line 2,3 for lastStamp lastRead



```
1   public class AtomicSRSWRegister<T> implements Register<T> {
2     ThreadLocal<Long> lastStamp;
3     ThreadLocal<StampedValue<T>> lastRead;
```

Atomic MRSW: Line 2 for the last stamp



```
1  public class AtomicMRSWRegister<T> implements Register<T> {
2    ThreadLocal<Long> lastStamp;
```

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:
   T *arr;
   /**
    * Create an array of type T and initialize it to the default value of T
    *
    * @param n the number of threads that will be using the array.
    */
   ThreadLocal(int n)
   {
       arr = new T[n];
   }
   /**
    * Create an array of type T and initialize it with the value val
    *
    * @param val The value that will be assigned to each thread-local variable.
    * @param n the number of threads that will use this ThreadLocal object.
    */
   ThreadLocal(T val, int n)
   {
       arr = new T[n];
       for (int i = 0; i < n; i++)
           arr[i] = val;
   }
   /**
    * Create an array of type T for each thread
    */
   ThreadLocal()
   {
       arr = new T[MAX_THREADS];
   }

   /**
    * Create an array of type T and initialize it with the value val
    *
    * @param val The value to be assigned to each thread.
    */
   ThreadLocal(T val)
   {
       arr = new T[MAX_THREADS];
       for (int i = 0; i < MAX_THREADS; i++)
           arr[i] = val;
   }
   /**
    * Set the value of the element at index id to val
    *
    * @param id The index of the array element to set.
    * @param val The value to be set.
    */
   void set(int id, T val)
   {
       arr[id] = val;
   }
   /**
```

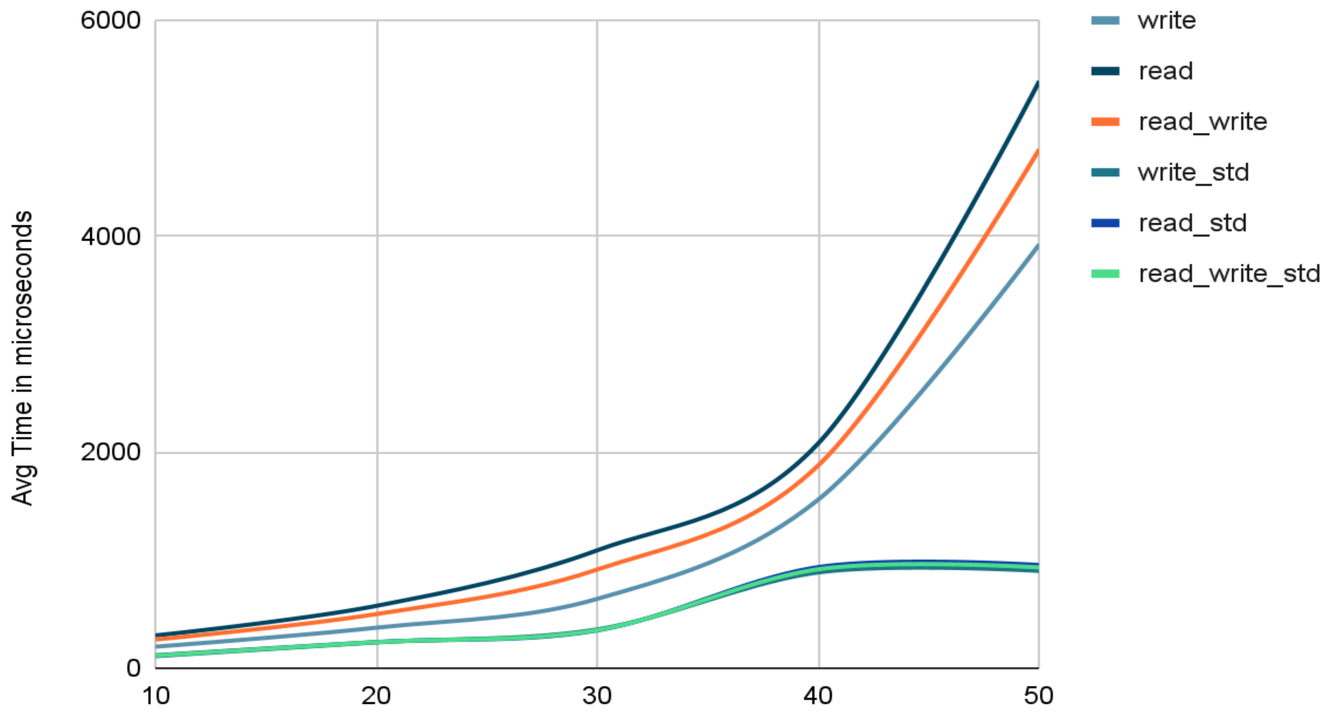
```
     * Return the value of the element with the given id
     *
     * @param id The id of the element you want to get.
     *
     * @return The object at the given index.
     */
    T get(int id)
    {
        return arr[id];
    }
};
```

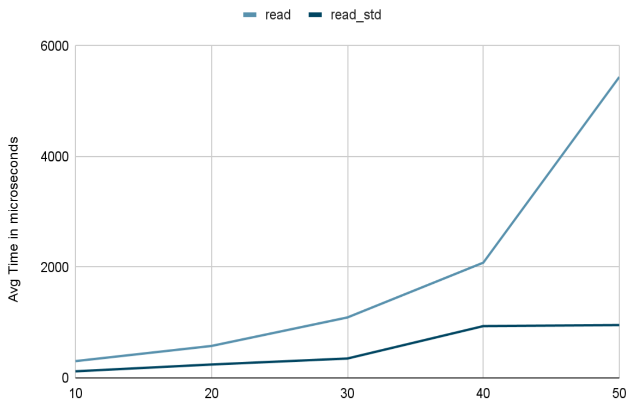
- The working principle of this solution is when a ThreadLocal object is created it creates an array of elements of length n where n is the number of threads that may access this object(can be made fixed to MAX_THREADS that in our problem statement is defined as 100)
- The get and set takes an id as 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.

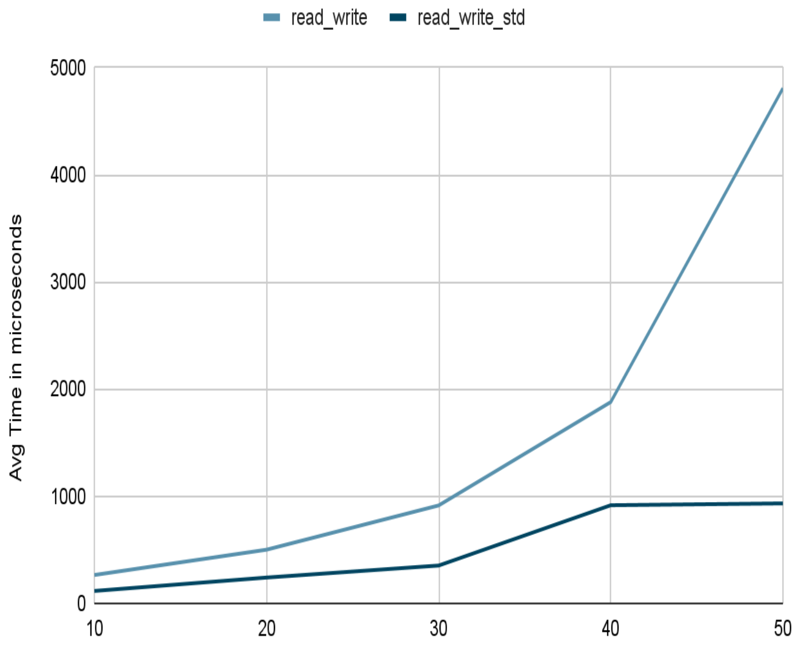
Due to the usage of local id for thread_lcoal, the interface of register class i have customized to receive thread_id:

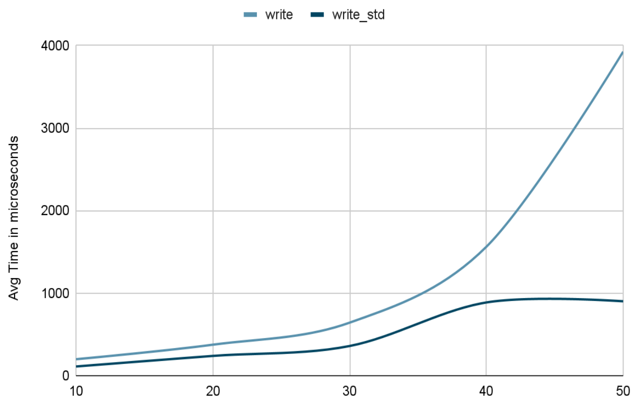
```
/* The above code is defining a class template called Register. */
template <typename T>
class Register
{
public:
    /* Defining a pure virtual function. */
    virtual T read(int) = 0;
    /* Defining a pure virtual function. */
    virtual void write(T, int) = 0;
};
```

Analysis


6000
4000
2000
0
Avg Time in microseconds
10
20
30
40
50
write
read
read_write
write_std
read_std
read_write_std


read
read_std
6000
4000
2000
0
Avg Time in microseconds
10
20
30
40
50


read_write
read_write_std
5000
4000
3000
2000
1000
0
Avg Time in microseconds
10
20
30
40
50


write
write_std
4000
3000
2000
1000
0
Avg Time in microseconds
10
20
30
40
50

For the Test Application, I have used c++ atomic<T> to compare read and write operations.
I have plotted out for custom atomic MRMW register implementation :

- Read: average of average time taken by each thread in a read operation
- Write: average of average time taken by each thread in a write operation
- Read_write: average of average time taken by each thread in a read or write operation

I have plotted out for standard c++ atomic:

- Read_std: average of average time taken by each thread in a read operation
- Write_std: average of average time taken by each thread in a write operation
- Read_write_std: average of average time taken by each thread in a read or write operation

The time is in microseconds everywhere
The x-axis is the number of threads
On comparing their performances we observe that:

- With the increase in the number of threads all the operations take more time:
  - Growth in this time is much faster in our custom implementation
  - Although there's is a growth in std atomic that growth is slow and almost converging
- Read takes more time than writing, both in std as well as our custom implementation
- C++ std atomic performs much better than our custom implementation every time:
  - As mentioned in the book as well as discussed in class the implementation of registers discussed are correct but vastly different and inefficient both in terms of space and time than the ones that are actually used in the real world
  - For example, Atomic MRMW registers we are using an array of MRSW registers which in Turn is using a 2-d array of SRSW registers which is using a regular register, and iterating over them writing values takes a lot of time as it gets O(N^2) as we increase N.

References

- The Art of Multiprocessor Programming Maurice Herlihy, Nir Shavit