**Serial Program**

When implementing the serial program following are taken as inputs from the user.

* Number of items Inserted to the Link list at the beginning (n/inserts\_in\_main\_p)
* Number of operations to be performed on the Link list (m/total\_ops)
* Member operations fraction (member\_percent)
* Insert operations fraction (insert\_percent)

delete\_percent = 1.0 - (member\_percent + insert\_percent)

After taking the fractions member\_num, insert\_num and delete\_num is calculated according to the total number of operations.

After that the Link list is populated with n random values in the range of 0 and 216 – 1 with the use of my\_rand() function.

In order to perform Insert(), Member() and Delete() functions a random value between 0-2 is generated using my\_rand() function. Using this random value, the program chooses which operation to be performed on the Link list unless the total number of operations for that particular operation is not fulfilled. The uniqueness of the insertions are assured by checking its availability before inserting in into the linked list.

Get-time() function is used to take the clock time to measure the time spent for the operations. Free\_list() function is used to free the elements of the Link list.

**One Mutex for the entire Link list**

In this implementation also same input variables are taken from the user as in the serial program. Apart from that the number of threads (thread\_count) is taken as an additional parameter.

In addition to the serial program, a mutex is initialized. As well as using the pthread\_create function, thread\_count number of thread are initialized. After In addition to that, a mutex is initialized and a user defined number of threads are created using pthread\_create() function.

In the Thread\_work() function:

The global variables member\_total, insert\_total and delete\_total were used to calculate the total number of operations for a particular thread with unique rank. Local variables my\_total\_ops, my\_insert\_count, my\_delete\_count and my\_member\_count keep the number of operations each thread carried out locally. The implemented algorithm fairly divides the operations among the threads.

Critical part of the program is accessing the Link list by many threads. The access of the shared resource is limited by using mutex. In order to access the Link list a particular thread should acquire the mutex lock first. One thread has to wait until another thread releases the mutex lock in order to access the Link list.

Inside the mutex, program executes the critical part which is accessing the Link list using Member(), Insert() and Delete() functions. For all these operations program maintains a one mutex lock.

Once thread\_work() function finishes executing, the mutex is destroyed and all the elements in the Link list are freed.

**Read Write locks for the entire Link list**

The implementations of input arguments, linked list population, randomly selecting the operations, creating threads are same as the Mutex program. Apart from that read-write lock is initiated to be used in the Thread.

In the Thread\_work() function:

As same as mutex program member\_total, insert\_total, delete\_total and toal\_ops are shared globally among all the threads.

Same as mutex program, the operations a thread should perform is fairly divided among all the threads with the same algorithm. My\_total\_ops, my\_insert\_count, my\_delete\_count and my\_member\_count are stored locally in order to update them locally with the use of read-write lock.

In this program critical section is covered by read-write locks. The thread with read lock can perform Member() function while the thread with write lock can perform either Insert() or Delete() functions. The Link list can be accessed by the threads which has one of these two locks.

Once thread\_work() function finishes executing, the read-write locks are destroyed and all the elements in the Link list are freed.