# CS4532 Concurrent Programming Lab 2

120095L P.M.T de Silva

Design ExplanationIn the assignment the fractions of member, insert and delete functions that have to be performed is given (m\_member, m\_insert, m\_delete). Total number of operations need to perform is also given. Then easy to find the individual number of sub operations to be performed by simple multiplication.  
**Example:**  
Number of insert operations = m\_insert \* mWith having the number of individual operations, it is important to design a method to allocate those operations to given threads.

**Solution**   
Allocate number of member, insert and delete operations per thread by dividing number of each operations by number of threads (n). So each threads have to perform m\_member / n member operations, m\_insert / n insertions and m\_delete / n delete operations.

When a thread call operation function, each time one of member, insert and delete operations will run. This will continue until number of operations reach allocate number of operations.   
For example:

Consider a thread that have 2000 member operations, 50 insert and delete operations to perform. When the operation function is called member, insert and delete functions run one time each and 2000 becomes 1999 and 50 becomes 49. This loop runs until all operations to perform becomes zero.

Thread Functionality

All operations need to perform is distributed within threads evenly. When defining a thread, define number of operations that thread need to do. Threads keep those values locally. Each time the thread does the operations, thread decrement the value of number of each operations need to do. At calling operation function, run a while loop until all number of member, insert and delete operations need to do becomes zero. At each loop one member, insert and delete function will run. Before running a certain operation, check the number of that operation need to perform. If this value is zero, then don’t run that operation.

**Case 1**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.99, *mIndert* = 0.005, *mDelete* = 0.005, *sample size = 350*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Implemetation | No of Threads | | | | | |
| 1 | | 2 | | 4 | |
| Average | Std | Average | Std | Average | Std |
| Serial | 0.019636 | 0.001480 |  |  |  |  |
| One mutex for entire list | 0.025701 | 0.001067 | 0.088160 | 0.006775 | 0.152245 | 0.030844 |
| Read Write Lock | 0.025581 | 0.001258 | 0.029498 | 0.004774 | 0.042053 | 0.015526 |

**Case 2***n* = 1,000 and *m* = 10,000, *mMember* = 0.9, *mIndert* = 0.05, *mDelete* = 0.05, *sample size = 350*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Implemetation | No of Threads | | | | | |
| 1 | | 2 | | 4 | |
| Average | Std | Average | Std | Average | Std |
| Serial | 0.034167 | 0.001826 |  |  |  |  |
| One mutex for entire list | 0.040205 | 0.001374 | 0.122479 | 0.009289 | 0.219762 | 0.059438 |
| Read Write Lock | 0.040211 | 0.001398 | 0.052753 | 0.005667 | 0.077019 | 0.025825 |

**Case 3***n* = 1,000 and *m* = 10,000, *mMember* = 0.5, *mIndert* = 0.25, *mDelete* = 0.25, *sample size = 350*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Implemetation | No of Threads | | | | | |
| 1 | | 2 | | 4 | |
| Average | Std | Average | Std | Average | Std |
| Serial | 0.070331 | 0.008424 |  |  |  |  |
| One mutex for entire list | 0.075251 | 0.001435 | 0.176952 | 0.032247 | 0.362530 | 0.129127 |
| Read Write Lock | 0.075293 | 0.001544 | 0.153908 | 0.026641 | 0.319878 | 0.148641 |

**Accuracy**

Initially selected sample size of 25 to select the sample size. Calculated sample size was about 320. Recieved that value for **Case 3 using Read Write Lock and 4 threads.** So for better results, selected the sample size of 350. Sample size is calculated for an accuracy of ±5% and 95% confidence level.

Machine Specifications

* CPU
  + Clock speed 1.2Ghz
  + No: of cores – 4
  + Logical cores - 8
  + Cache size 6MB
  + Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
* Memory – 4GB DDR3 Ram at 1600Mhz
* Operating system – Ubuntu 14.04 LTS no other user servicers were running

**Observations**

Case 1Expected result was linked list with mutex and read write lock with multiple threads show fast operations than serial processing of serial linked list. But reality observed opposite of expected result. This could be happened because of overhead of creating thread, destroying thread, locking and unlocking linked list.   
Consider the execution with 2 and 4 threads. Linked list with mutex take lots of time compared to read write lock. The reason for that is mutex lock the whole linked list such that no other thread can perform read or write operation. So only one thread can read or write at same time. But in read write lock multiple threads can perform read(member) operation at same time. But when threads perform reading operation no thread can perform write operation. They have to wait for read lock to be unlocked. In writing both mutex and read write lock doesn’t allow any other thread to access to linked list. As in case 1 have 9900 read operations out of 10000 total operations a significant time different between mutex and read write lock can be observed.   
Case 2Observation is quite similar to case 1, that serial processing perform better than parallel processing. Even though number of read operations is much greater than number of write operations time spent by mutex is higher than time spent by read write lock when executing 2 or 4 threads. But time gap between mutex and read write lock is closer than in case 1. This gap is decreased because of increase of number of write operations compared to case 1 so read write lock losing some of the advantage of being able to multiple read operations at once.

Case 3Considering serial processing vs parallel processing the observation is again similar to the above cases. But as number of read and write operations are equal read write lock cannot achieve the advantage of allowing multiple threads can read the linked list. Because of that in execution time of multiple threads reduction of the gaps between the read write lock and the mutex can be observed.