**CS4532 Concurrent Programming**

**Lab 02**

**120314K, 120337H**

**Step 3:**

**Case 1**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.99, *mInsert* = 0.005, *mDelete* = 0.005

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Implementation** | **Number of threads** | | | | | |
| **1** | | **2** | | **4** | |
| **Average** | **Std** | **Average** | **Std** | **Average** | **Std** |
| **Serial** | 132.8 | 13.2367715555 |  | | | |
| **One mutex for entire list** | 137.4 | 10.4668789787 | 209.39 | 16.9829950601 | 209.68 | 11.9332148293 |
| **Read-Write lock** | 140.95 | 13.2278975173 | 82.55 | 6.63990842093 | 76.9 | 4.73008851648 |

**Case 2**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.90, *mInsert* = 0.05, *mDelete* = 0.05

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Implementation** | **Number of threads** | | | | | |
| **1** | | **2** | | **4** | |
| **Average** | **Std** | **Average** | **Std** | **Average** | **Std** |
| **Serial** | 239.41 | 12.8339000268 |  | | | |
| **One mutex for entire list** | 242.33 | 12.4884229217 | 315.95 | 21.1485247951 | 324.61 | 18.4975537558 |
| **Read-Write lock** | 243.96 | 13.955861156 | 146.7 | 13.0604654425 | 158.32 | 18.6519855687 |

**Case 3**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.50, *mInsert* = 0.25, *mDelete* = 0.25

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Implementation** | **Number of threads** | | | | | |
| **1** | | **2** | | **4** | |
| **Average** | **Std** | **Average** | **Std** | **Average** | **Std** |
| **Serial** | 579.22 | 19.291794456 |  | | | |
| **One mutex for entire list** | 607.98 | 40.9142932719 | 683.59 | 39.9403836039 | 711.96 | 50.4090179522 |
| **Read-Write lock** | 622.44 | 57.9204121309 | 586.76 | 60.8459090364 | 582.37 | 50.0316374655 |

Sample size determination equation is 𝑛= ()2

We need an initial standard deviation and average of the distribution in order to find the minimum number of samples. Initial average and standard deviations were calculated using 100 samples and required samples with an accuracy of ±5% and 95% confidence level are shown in the following table.

**Number of Samples for each case**

**Case 1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Number of threads** | | |
| **1** | **2** | **4** |
| **Number of samples** | **Number of samples** | **Number of samples** |
| **Serial** | 16 |  | |
| **One mutex for entire list** | 9 | 11 | 5 |
| **Read-Write lock** | 14 | 10 | 6 |

**Case 2**

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Number of threads** | | |
| **1** | **2** | **4** |
| **Number of samples** | **Number of samples** | **Number of samples** |
| **Serial** | 5 |  | |
| **One mutex for entire list** | 5 | 7 | 5 |
| **Read-Write lock** | 6 | 13 | 22 |

**Case 3**

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Number of threads** | | |
| **1** | **2** | **4** |
| **Number of samples** | **Number of samples** | **Number of samples** |
| **Serial** | 2 |  | |
| **One mutex for entire list** | 7 | 6 | 8 |
| **Read-Write lock** | 14 | 17 | 12 |

Since the required number of samples for all the test cases are below 100, our results are valid with an accuracy of ±5% and 95% confidence level.

**Specification of the Machine**

* Fedora 24 x64 bit
* Memory: 8 GB
* CPU: Intel(R) Core(TM) i7 -5500U CPU @ 2.40GHz - 4 cores
* L1d cache:32K
* L1i cache:32K
* L2 cache:256K
* L3 cache:3072K

**Step 4:**

**Case 1**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.99, *mInsert* = 0.005, *mDelete* = 0.005

**Case 2**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.90, *mInsert* = 0.05, *mDelete* = 0.05

**Case 3**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.50, *mInsert* = 0.25, *mDelete* = 0.25

The simulation results are plotted in the above graphs. This is a discrete event because possible values for the thread counts are positive integers. Only for those values, we can get the execution time. So in the graph those points are denoted with different kind of pointers according to their data category. The connecting line is only for reveals the trend. Since serial operation is only applicable with 1 thread there is only one data point in the graph and no tread line for that.

**Step 5**

**Observations**

In all three cases with single thread, serial linked list code gives the better performance. Because if there is only one thread to access the link list, then there is no need for locks. But those parallel codes still use the locks there. Due to those lock acquire and release overhead those parallel codes give poor performance than serial code in single thread scenario.

In all three cases Read-write lock gives the better performance than one mutex for entire list except in the single thread scenario. This is because in the one mutex scenario if one thread is doing any operation with the link list, no other threads are allowed to do any operations with that link list. But in Read-write locks when one thread is doing read operation with the link list, another thread is allowed for only read operation. Due to this improvement Read-write lock gives better performance. Case 1 to Case 3 the read operation percentage decrease, this resulted as the deviation between mutex and Read-write lock also converged from Case1 to Case 3 (in the graph those two trend line come closer from case 1 to case 3). In the single thread scenario, one mutex for entire list shows better performance than the Read-write linked list, because in single thread Read-Write lock mechanism cause additional overhead compare to one mutex linked list code.

In Case 1 and Case 2, Read-write lock could be able to give better performance than the serial when thread count increases, Even though it fails with single thread. But in Case 3 it fails even we increase the thread count. This is due to, in case 3 only 50% operations are read other 50% are insert and delete. In the Read-write lock, when a thread is doing insert/delete operation, no other threads can access the link list. But when a thread is doing read operation, another read operation is allowed. Because of this, when read operation percentage falls its performance become poor.

In all three cases, one mutex for entire list fails to give better performance than serial code even we increase the thread up to 4. This is due to the overhead of accruing and releasing the lock. Since there is only one mutex for entire list, when the thread count increase the threads starvation time increase so it gives bad performance. Since no other operations are allowed in the entire list when an operation is performed with the list, it gives poor performance than serial code.