CS4532 Concurrent Programming

Lab 2

Step 1

1. Created Member function, Insert function, and Delete function of the linked list.
2. Then created an array which contains 10000 operations according to the fractions given (number of reads, number of inserts, number of deletes).
3. Inserted 0's to the array to an equal amount of the number of member function calls. (Eg : *If m =10000 and fraction of member function calls is 0.99, then filled up the first 99000 slots of the array with 0's).* Then inserted 1's to the array to an equal amount of the insert function calls. (Eg : *If m =10000 and fraction of insert calls is 0.005 then filled up the next 50 slots of the array with 1's).* Finally, inserted 2's to the array to an equal amount of the delete function calls. (Eg : *If m =10000 and fraction of delete calls is 0.005 then filled up the next 50 slots of the array with 2's)*
4. Next, shuffled the elements of the array since it was required to make operation calls random.
5. Then divided the number of operations (m) by the number of threads present. This gave the number of operations to be allocated per thread. But in sequential execution, the entire operations array was iterated by the main thread.
6. Once threads were given the starting index and the size of the range where they are supposed to be operated, they did iterate the linked list according to their starting index and ending index.
7. By iterating the operations array, each thread got a shuffled sequence of 0's,1's, and 2's. When 0 occurred, member function was called. When 1 occurred, insert function was called. When 3 occured, delete function was called.
8. Generated random numbers by giving the the time at which the experiment ran as a seed.

Step 3

Results

Case 1

n = 1,000 and *m* = 10,000, m_{Member} = 0.99, m_{Indert} = 0.005, m_{Delete} = 0.005

implementation	No of threads							
	1		2		4		8	
	Average (s)	Std (s)	Average (s)	Std (s)	Average (s)	Std (s)	Average (s)	Std (s)
serial	0.0242	0.002215						
One mutex	0.025352	0.002078	0.054914	0.003356	0.064321	0.002918	0.066224	0.008832
Read-write lock	0.025408	0.003906	0.022398	0.004155	0.021892	0.001585	0.021792	0.004163

Case 2

n = 1,000 and *m* = 10,000, m_{Member} = 0.90, m_{Indert} = 0.05, m_{Delete} = 0.05

implementation	No of threads							
	1		2		4		8	
	Average (s)	Std (s)	Average (s)	Std (s)	Average (s)	Std (s)	Average (s)	Std (s)
serial	0.039819	0.001953						
One mutex	0.03154	0.004242	0.058222	0.004243	0.066954	0.008152	0.067153	0.006679
Read-write lock	0.032139	0.003634	0.023109	0.001153	0.022932	0.002381	0.026252	0.005928

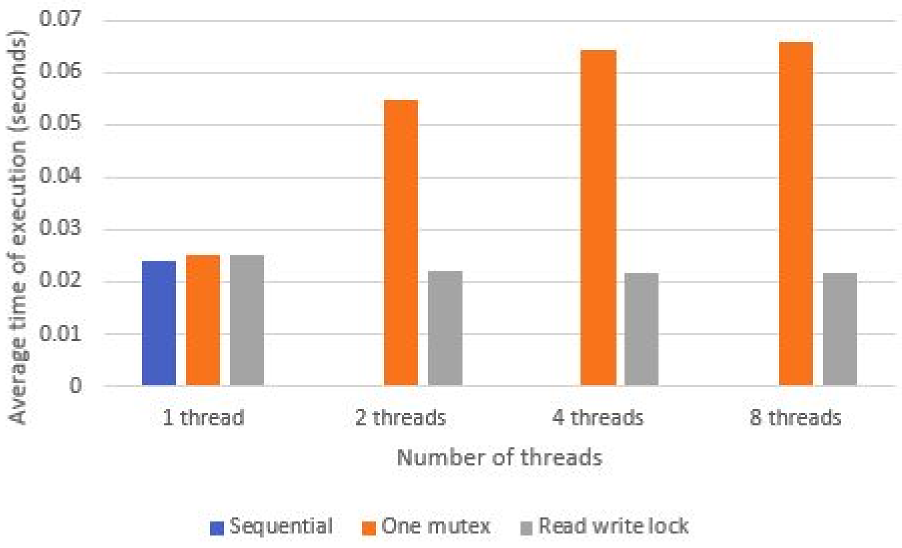
Case 3

n = 1,000 and *m* = 10,000, m_{Member} = 0.50 , m_{Indert} = 0.25, m_{Delete} = 0.25

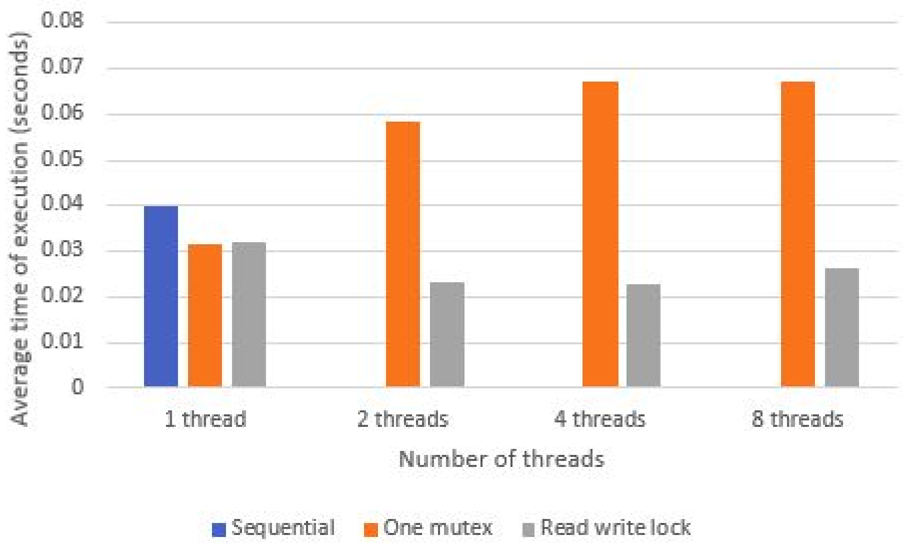
implementation	No of threads							
	1		2		4		8	
	Average (s)	Std (s)	Average (s)	Std (s)	Average (s)	Std (s)	Average (s)	Std (s)
serial	0.112284	0.002921						
One mutex	0.100241	0.003788	0.103317	0.004320	0.114649	0.011953	0.135297	0.010584
Read-write lock	0.101368	0.003648	0.104711	0.005688	0.106840	0.006163	0.100280	0.002505

Analysis

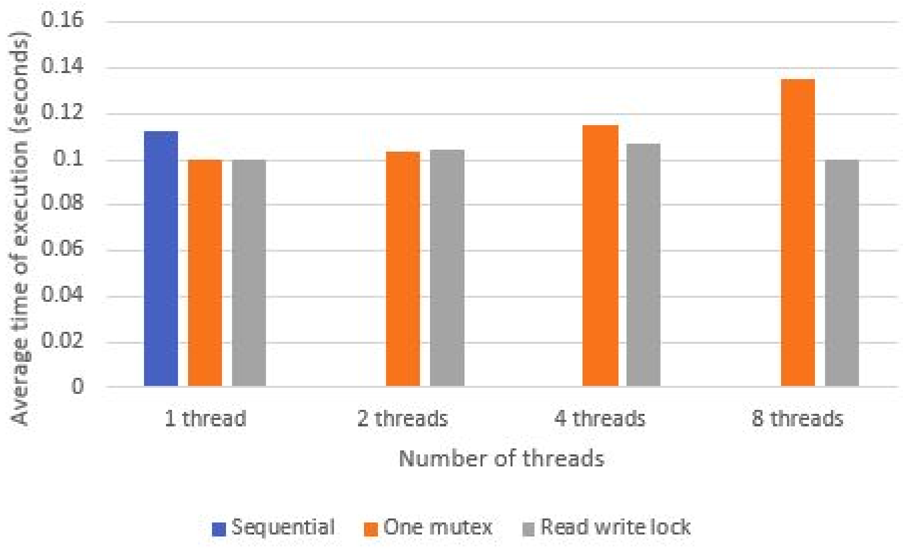
Case 1 - Average time taken



Case 2 - Average time taken



Case 3 - Average time taken



Specification of the machine

- CPU
    - Clock speed
        - Base frequency - 2.00 GHz
        - Max frequency - 3.10 Ghz
    - No of physical cores - 2
    - Cache
        - Level 1 - 32 KB
        - Level 2 - 256 KB
        - Level 3 - 4096 KB
    - CPU model - Intel core i7 4510U
- Memory
    - Capacity - 8GB
    - Speed - DDR3, 1600 MHz
- Operating system
    - OS name - Ubuntu
    - OS family - linux
    - OS architecture - 64 bit
    - Version - 16.04.3

Step 5

Observation

In all 3 cases, 1 thread scenario of serial implementation, one mutex for entire list implementation, and read-write lock implementation showed similar behaviour. But mutex implementation and read-write lock implementation took some additional time other than the serial implementation since they both needed to acquire a lock which was unwanted step in single threaded application.

2 Threads/4 Threads/8 Threads:

In all the cases, except for 1 thread scenario, one mutex for entire linked list implementation took more time than the read-write lock implementation. Since only one mutex used for the entire linked list, before each time a thread operates on the linked list it acquires the lock resulting sequential behavior. Other reasons for one mutex for entire linked list to consume the highest time are overhead of creating threads, switching threads and acquiring locks.

Read-write lock implementation is supposed to show better performance than the one mutex implementation and serial implementation as it allows multiple member operations parallely. But we ran our simulations in 2 Core machine where logical threads were available. This has more overhead in scheduling threads, context switching, and creating and terminating threads. Therefore we could only see performance improvements in 1 thread and 2 threads scenarios.