CS4532 Concurrent Programming

Take Home Lab 1

- ## Design

The Linked list and the Nodes are implemented as structs in the header files included. They follow the same implementation as present in the slides for a sorted linked list. The corresponding *Member, Insert, Delete* functions are also included in the header file.

```
typedef struct Node {
    int value;
    struct Node *next;
} Node;

typedef struct LinkedList {
    struct Node *head;
} LinkedList;
```

```
typedef struct Node {
    int value;
    struct Node *next;
} Node;

typedef struct LinkedList {
    struct Node *head;
    pthread_mutex_t mutex;
} LinkedList;
```

```
typedef struct Node {
    int value;
    struct Node *next;
} Node;

typedef struct LinkedList {
    struct Node *head;
    pthread_rwlock_t  rwlock;
} LinkedList;
```

The number of operations for each operation type *Member, Insert, Delete* is calculated at first proportionate to the percentiles *mMember, mInsert, mDelete*.

Then an array (*process_order[m]*) of size *m* is initialized with random order of operations where each type adds up to the proportionate values mentioned above. The algorithm is designed to randomize and distribute the processes evenly. rand()%3 is used to generate the random order and *0, 1, 2* in the switch case statement refers to *Member, Insert, Delete* respectively.

```
process_order[c] = rand()%3;
```

Afterward in the parallel executions each thread is assigned equal proportions of the process order to execute on their own time and rank is used to divide work. When executing *srand(time(0))* is used to initialize the random seed for each iteration and *rand() % 65535* is used to generate parameter values. The time complexity of rand is O (1) and used in all cases, hence the impact on timing is negligible.

Time is calculated using the following formula:

```
clock_t start, end;
double cpu_time_used;

start = clock();
//-------execution------
end = clock();

cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

- ## Observations

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

Implementation	**No of threads**							
	1		**2**		**4**		**8**	
	Average	**Std**	**Average**	**Std**	**Average**	**Std**	**Average**	**Std**
Serial	0.016489	0.00407						
One mutex for entire list	0.023711	0.007873	0.023189	0.003137	0.022122	0.002681	0.023856	0.003037
Read-Write lock	0.025144	0.007708	0.011344	0.002284	0.007933	0.002808	0.006556	0.001492

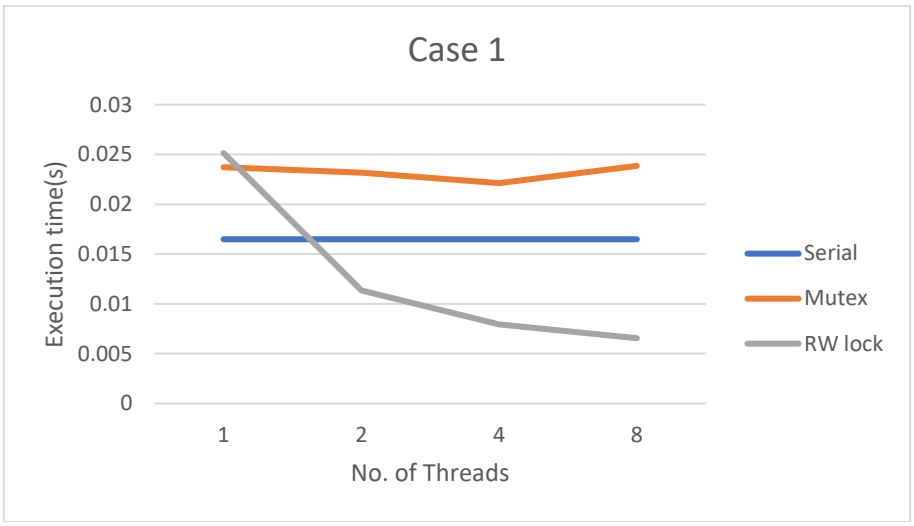
Case 2: *n* = 1,000 and *m* = 10,000, *mMember* = 0.90, *mIndert* = 0.05, *mDelete* = 0.05

Implementation	**No of threads**							
	1		**2**		**4**		**8**	
	Average	**Std**	**Average**	**Std**	**Average**	**Std**	**Average**	**Std**
Serial	0.0189	0.005714						
One mutex for entire list	0.028256	0.006821	0.022911	0.002989	0.024367	0.00412	0.025967	0.00331
Read-Write lock	0.027522	0.008499	0.0147	0.002159	0.011478	0.001651	0.009878	0.001792

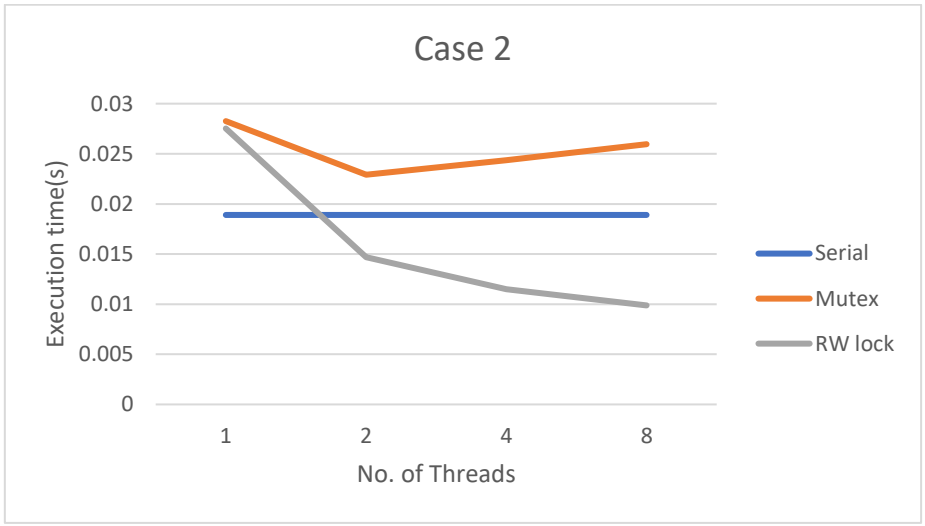
Case 3: *n* = 1,000 and *m* = 10,000, *mMember* = 0.50, *mIndert* = 0.25, *mDelete* = 0.25

Implementation	**No of threads**							
	1		**2**		**4**		**8**	
	Average	**Std**	**Average**	**Std**	**Average**	**Std**	**Average**	**Std**
Serial	0.028067	0.005662						
One mutex for entire list	0.038011	0.009578	0.044378	0.006363	0.033544	0.009657	0.024622	0.010778
Read-Write lock	0.036244	0.008045	0.057011	0.004531	0.034844	0.004958	0.019089	0.006458

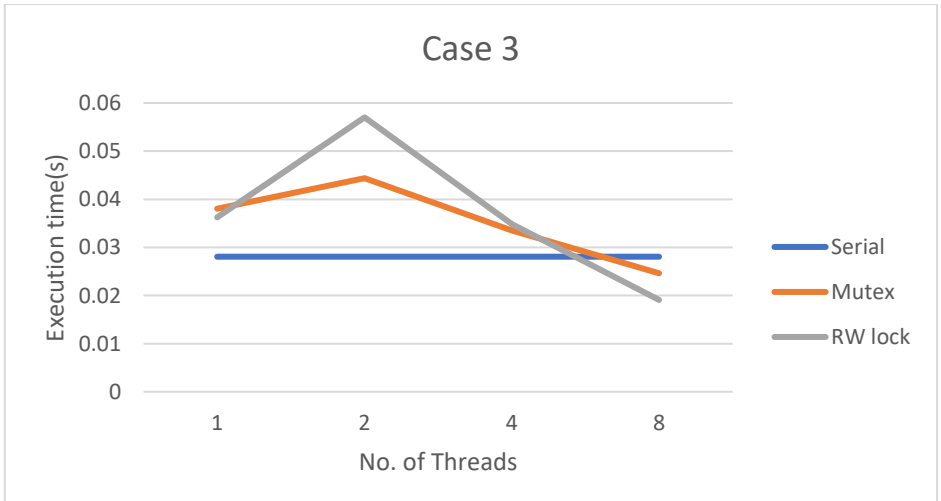
- **Graphs**
  - **Average execution time against the No. of Threads**



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, *mIndert* = 0.05, *mDelete* = 0.05

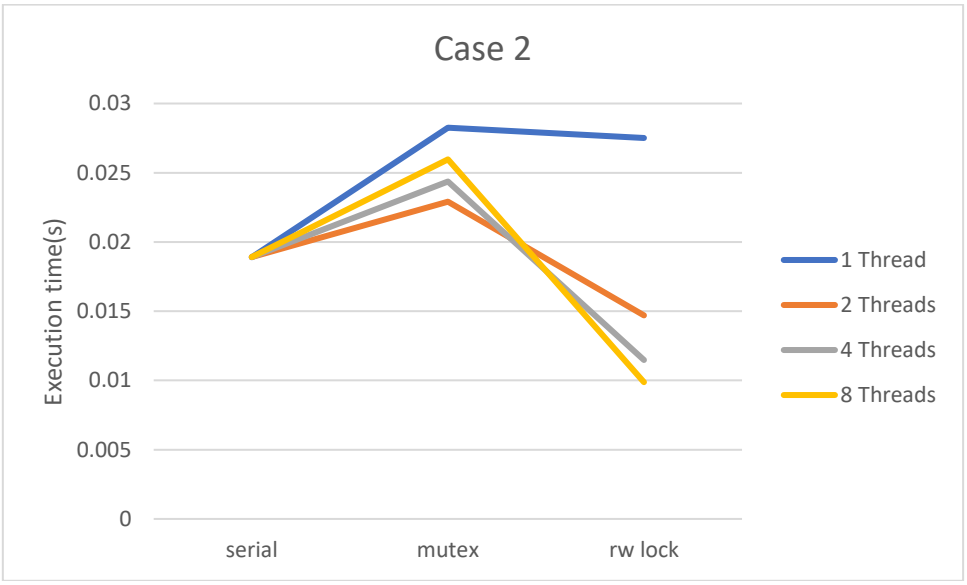


Case 3: *n* = 1,000 and *m* = 10,000, *mMember* = 0.50, *mIndert* = 0.25, *mDelete* = 0.25

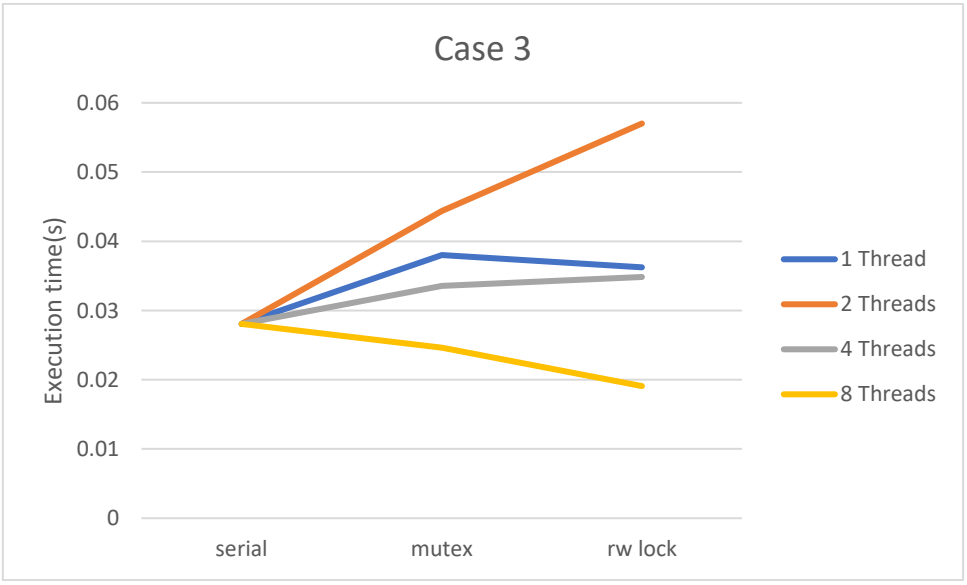
Note – Serial *average execution time* is extended for *no. of threads* for ease of comparison.

- **Average execution time against the implementation**

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, *mIndert* = 0.05, *mDelete* = 0.05



Case 3: *n* = 1,000 and *m* = 10,000, *mMember* = 0.50, *mIndert* = 0.25, *mDelete* = 0.25

Note – Serial *average execution time* is extended for *no. of threads* for ease of comparison.

- ## Discussion

In Case 1 the majority of processes *Member*() operations which is read only. In that case read-write lock should perform better since it allows parallel execution of read tasks and have decreased average execution time with thread count as expected. Although mutexes guarantee mutual exclusion they serialize the operations regardless of their operation type (read or write). Also, mutexes are efficient for lower no of threads, when thread count increases, more resources are consumed so the execution time increases.

In Case 2 similar observations can be seen for all three. The key difference is mutexes have started to drop in performance earlier than before with more of the process proportion being biased towards write operations. Read write locks have gained performance over thread count as expected but not as much as in Case 1. This is due to the increased no. of write operations.

In Case 3 as expected mutexes have performed better than read-write locks for the most part (before thread count is 8) since the amount of write operations have increased further. The initial increase in performance time of both mutexes and read-write locks must be due to the longer waiting introduced by the write operations (*Insert, Delete*) and increased competition for the locks. They have however gained more performance as the work is distributed to more threads which is the expected result.

- ## Device Specifications

Processor – Intel(R) Core (TM) i7-10510U CPU @ 1.8GHz 10^{th} Gen (4 Physical Processors)

RAM – 20 GB

Operating System (OS) – Windows 10 Home