Students: ym2977, ks7157
ASSIGNMENT-4 [WRITE-UP]

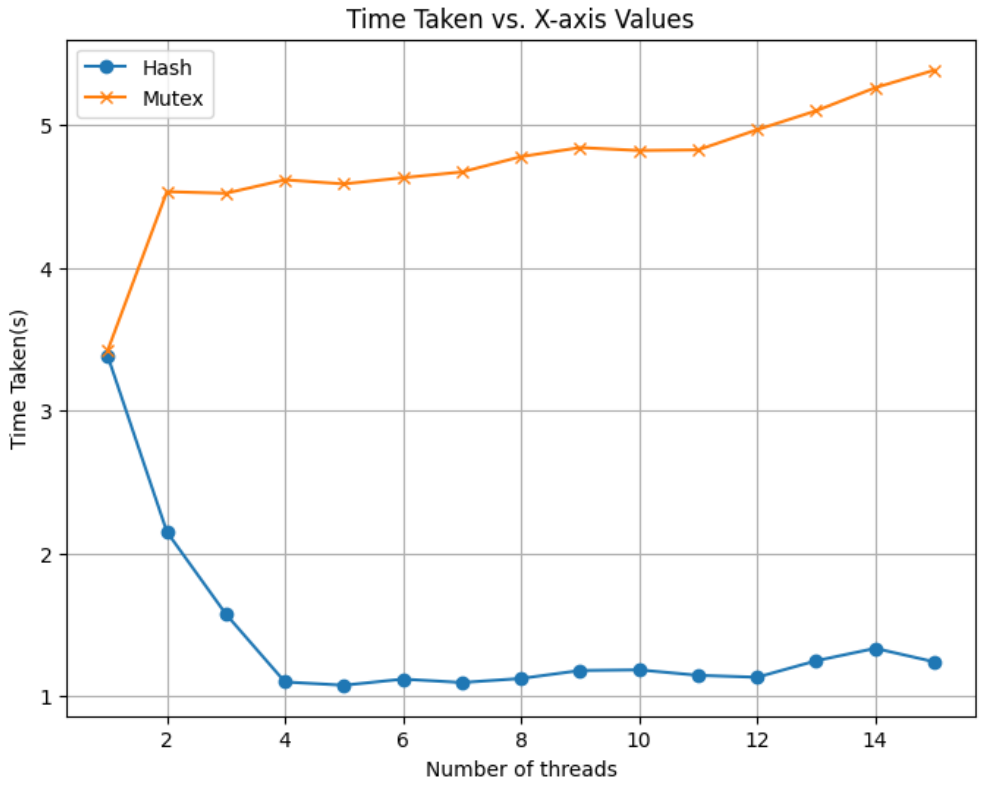
PART-2 [Mutex locks]

What circumstances cause an entry to get lost?

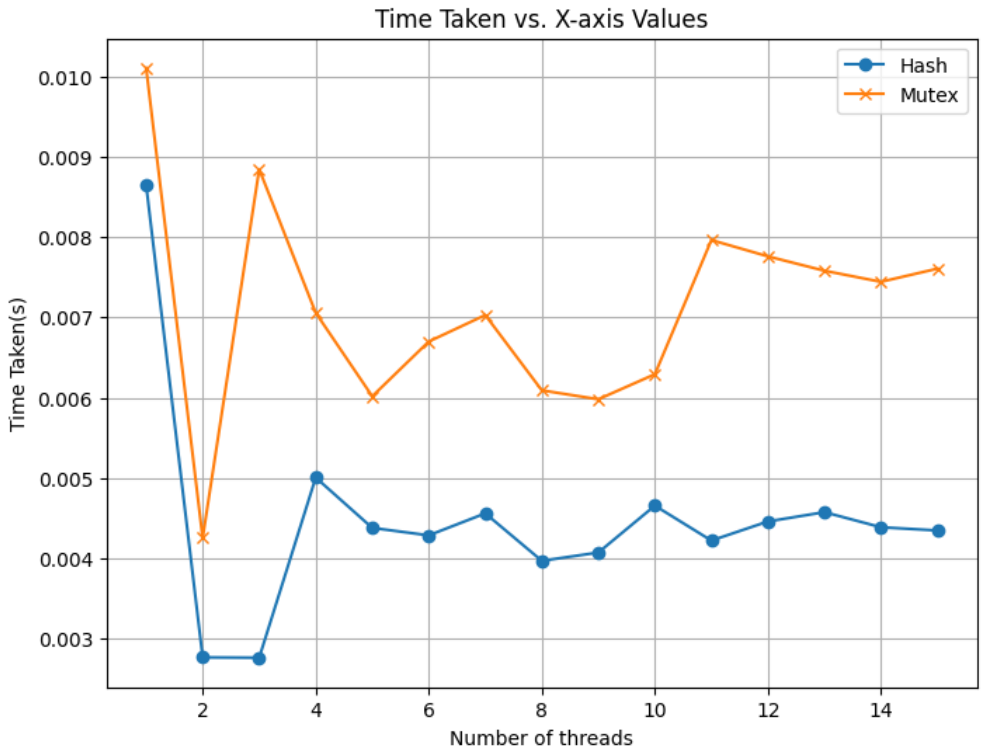
- An entry is considered to be lost if the key that should've been inserted in the hash table is found missing. This happens due to the lack of concurrency that happens due to the concurrent access and insertion of keys by multiple threads.
- The part of code that causes the unintended behavior are insert and retrieve functions such that multiple threads simultaneously insert keys into the hash table using the insert() function without synchronization. Retrieve() function doesn't affect the synchronization as such since it does not modify the keys but just retrieves them.

After updating the code by introducing mutex locks, a comparison with the original hash table code with mutex code is shown in the table below:
Command used to execute with multiple threads:
for i in {1..15}; do ./parallel_mutex $i; done
for i in {1..15}; do ./parallel_hashtable $i; done

Retrieval time taken comparison:



Insertion time taken comparison:



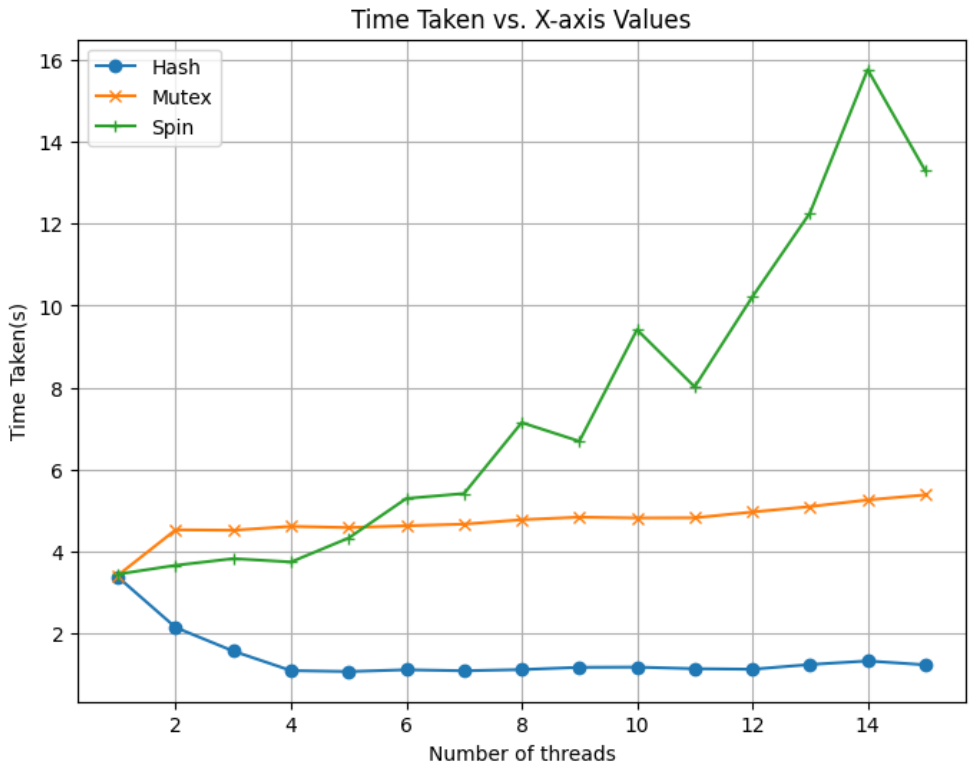
Time Overhead:
It was noticed that after inserting mutex locks, the runtime for both insertion as well as retrieval was increased and noted by calculating the time difference after and before the operations using the now() function and printing it. Now, after introducing locks, there may be a large overhead because of threads waiting and fighting for access in high contention situations where numerous threads frequently compete for the same lock. Longer execution delays may result from the extra time required to acquire and release locks as a result of this contention.
The values are provided in the table.

***Time overhead table in PART-3*

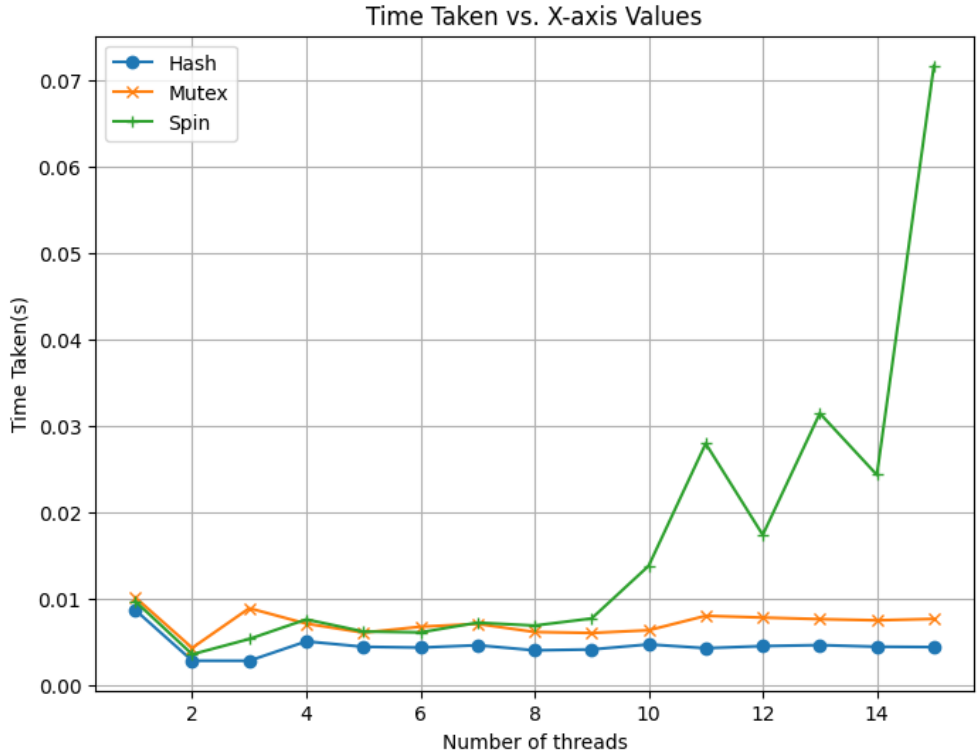
PART-3 [Spinlock]

It is expected that switching to spinlocks from mutexes may lengthen the running time because of avoidance of context switches resulting in busy waits, which waste CPU resources when a lock is held for an extended amount of time. This may affect concurrent threads or processes attempting to use the same CPU core.

Retrieval time taken comparison:



Insertion time taken comparison:



Time Overhead:
It was noticed that after inserting spinlocks, the runtime for both insertion as well as retrieval was increased as compared to no locks and noted by calculating the time difference after and before the operations using the now() function and printing it. Compared to the mutex runtime, it was noticed that the runtime was faster with less threads and then there was a spike in the spinlock runtime.
Spinlocks may perform better than mutexes in low contention scenarios since spinlocks reduce lock acquisition latency by avoiding the overhead of sleeping and awakening threads.
The values are provided in the table.

Insertion:

No. of Threads	No lock - runtime (s)	Mutex - runtime (s)	Spinlock - runtime (s)	Mutex overhead (s)	Spinlock overhead (s)
1	0.008648	0.010107	0.008648	0.001459	0.001009
2	0.002766	0.004254	0.009657	0.001488	0.000741
3	0.002762	0.008846	0.003507	0.006084	0.002519
4	0.005006	0.007058	0.005281	0.002052	0.002545
5	0.004379	0.006018	0.007551	0.001639	0.001767
6	0.004286	0.0067	0.006146	0.002414	0.001762
7	0.00456	0.00703	0.006048	0.00247	0.00261
8	0.003969	0.006092	0.00717	0.002123	0.002867
9	0.004072	0.005981	0.006836	0.001909	0.003583
10	0.004656	0.006293	0.007655	0.001637	0.009089
11	0.004221	0.007964	0.013745	0.003743	0.023684
12	0.004459	0.007761	0.027905	0.003302	0.012871
13	0.004575	0.007582	0.01733	0.003007	0.026826
14	0.004387	0.007444	0.031401	0.003057	0.019894
15	0.004347	0.007608	0.024281	0.003261	0.06729

Retrieval:

No. of	No lock -	Mutex -	Spinlock -	Mutex	Spinlock

Threads	runtime (s)	runtime (s)	runtime (s)	overhead (s)	overhead (s)
1	3.379096	3.42668	3.452964	0.047584	0.073868
2	2.150868	4.53434	3.668203	2.383472	1.517335
3	1.574785	4.523218	3.83165	2.948433	2.256865
4	1.099809	4.617437	3.749461	3.517628	2.649652
5	1.076957	4.588739	4.334524	3.511782	3.257567
6	1.119878	4.632289	5.298473	3.512411	4.178595
7	1.095862	4.672438	5.418683	3.576576	4.322821
8	1.124347	4.779795	7.148832	3.655448	6.024485
9	1.17999	4.843198	6.694098	3.663208	5.514108
10	1.184666	4.822803	9.41414	3.638137	8.229474
11	1.146368	4.827112	8.021115	3.680744	6.874747
12	1.132865	4.968412	10.215691	3.835547	9.082826
13	1.248493	5.101118	12.265348	3.852625	11.016855
14	1.335444	5.262707	15.75988	3.927263	14.424436
15	1.240819	5.385305	13.289284	4.144486	12.048465

PART-4

A lock is only required when fetching an item from the hash table if doing so modifies shared data or poses a risk of data inconsistency from concurrent accesses. The retrieve() function in the present scenario searches a bucket's linked list for a particular key without changing any shared data. Hence, a lock is not necessary in this particular scenario in order to ensure the accuracy of the retrieval process.

Regarding the modifications to the code (parallel_mutex_opt.c):

No mutex lock is required in the retrieve() function hence it remains as it was as the function in the original hashtable code.

PART-5

Unlike the locks applied in the parallel_mutex.c which were applied on rows (on the table), here, we can apply the locks on the bucket i.e. 5 buckets can be used for insertions in parallel. To further increase parallelism, we can also increase the number of buckets as much as the number of keys.