

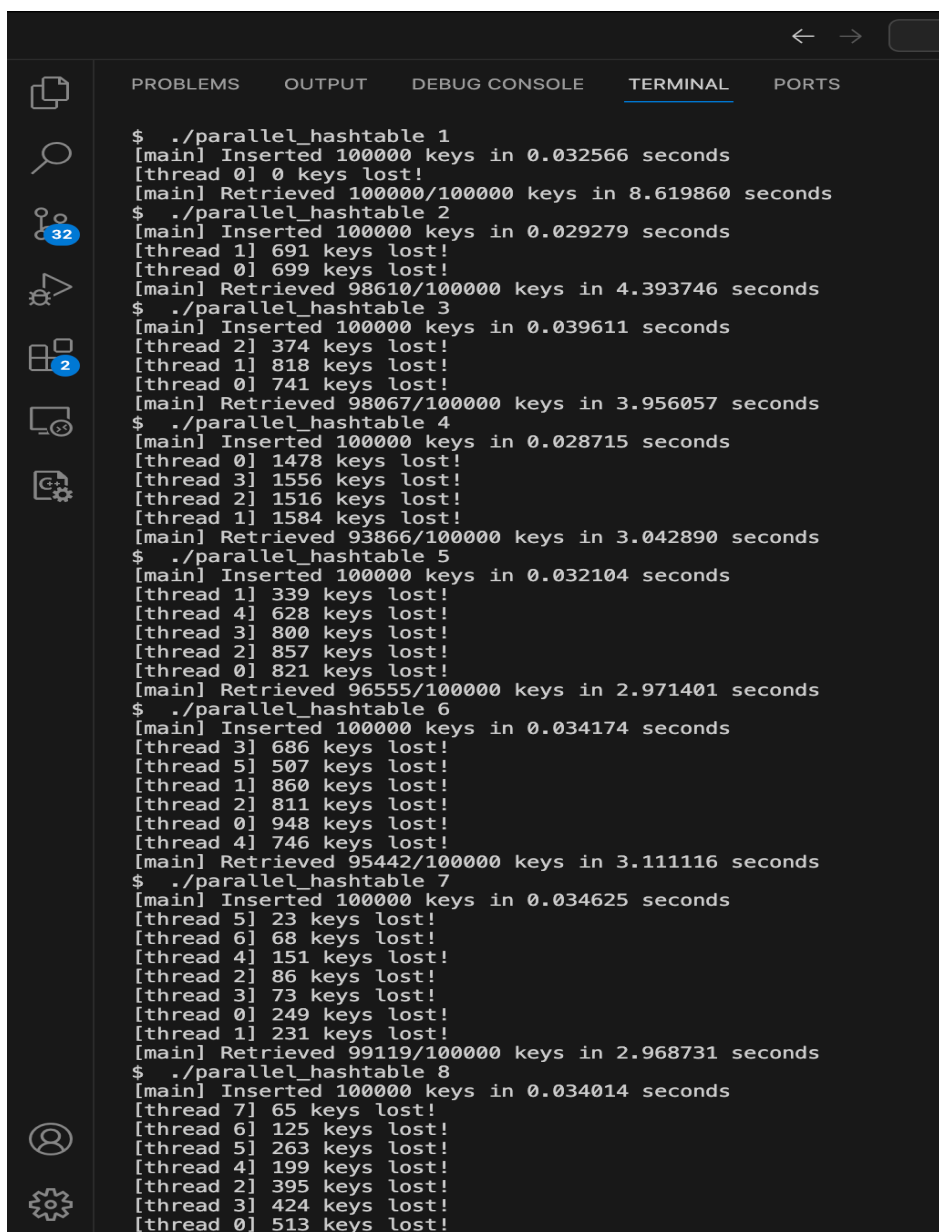
Explanations

Part-1

These are timetaken to complete using 1,2,3,4..8 threads.

8.619860 4.393746 3.956057 3.042890 2.971401 3.111116 2.968731 2.706698

Execution of parallel_hashtable.c:



```
$ ./parallel_hashtable 1
[main] Inserted 100000 keys in 0.032566 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 8.619860 seconds
$ ./parallel_hashtable 2
[main] Inserted 100000 keys in 0.029279 seconds
[thread 1] 691 keys lost!
[thread 0] 699 keys lost!
[main] Retrieved 98610/100000 keys in 4.393746 seconds
$ ./parallel_hashtable 3
[main] Inserted 100000 keys in 0.039611 seconds
[thread 2] 374 keys lost!
[thread 1] 818 keys lost!
[thread 0] 741 keys lost!
[main] Retrieved 98067/100000 keys in 3.956057 seconds
$ ./parallel_hashtable 4
[main] Inserted 100000 keys in 0.028715 seconds
[thread 0] 1478 keys lost!
[thread 3] 1556 keys lost!
[thread 2] 1516 keys lost!
[thread 1] 1584 keys lost!
[main] Retrieved 93866/100000 keys in 3.042890 seconds
$ ./parallel_hashtable 5
[main] Inserted 100000 keys in 0.032104 seconds
[thread 1] 339 keys lost!
[thread 4] 628 keys lost!
[thread 3] 800 keys lost!
[thread 2] 857 keys lost!
[thread 0] 821 keys lost!
[main] Retrieved 96555/100000 keys in 2.971401 seconds
$ ./parallel_hashtable 6
[main] Inserted 100000 keys in 0.034174 seconds
[thread 3] 686 keys lost!
[thread 5] 507 keys lost!
[thread 1] 860 keys lost!
[thread 2] 811 keys lost!
[thread 0] 948 keys lost!
[thread 4] 746 keys lost!
[main] Retrieved 95442/100000 keys in 3.111116 seconds
$ ./parallel_hashtable 7
[main] Inserted 100000 keys in 0.034625 seconds
[thread 5] 23 keys lost!
[thread 6] 68 keys lost!
[thread 4] 151 keys lost!
[thread 2] 86 keys lost!
[thread 3] 73 keys lost!
[thread 0] 249 keys lost!
[thread 1] 231 keys lost!
[main] Retrieved 99119/100000 keys in 2.968731 seconds
$ ./parallel_hashtable 8
[main] Inserted 100000 keys in 0.034014 seconds
[thread 7] 65 keys lost!
[thread 6] 125 keys lost!
[thread 5] 263 keys lost!
[thread 4] 199 keys lost!
[thread 2] 395 keys lost!
[thread 3] 424 keys lost!
[thread 0] 513 keys lost!
```

Part-2

What circumstances cause an entry to get lost? Analyze the initial code and write a short answer to describe what it means for an entry to be "lost," and which parts of the program are causing this unintended behavior when run with multiple threads.

In the initial code, an entry is considered "lost" when a thread attempts to retrieve a key from the hash table and fails to find it, even though it was previously inserted. This can happen due to a race condition between multiple threads accessing and modifying the hash table concurrently. The primary cause of this unintended behavior is the lack of proper synchronization mechanisms to ensure the atomicity of critical operations, such as inserting and retrieving elements from the hash table.

When multiple threads concurrently execute the `insert` function without proper synchronization, they might interfere with each other, leading to race conditions. For instance, if one thread is in the process of updating the hash table, another thread might concurrently read from or modify the same location, resulting in the loss of data.

Similarly, during the `retrieve` operation, if one thread is retrieving an entry while another thread is concurrently modifying the hash table, inconsistencies may arise, causing the entry to appear as if it were never inserted.

If I can summarize then the absence of synchronization mechanisms, such as mutexes or locks, allows multiple threads to interfere with each other's access to the hash table, leading to race conditions and, consequently, the loss of entries.

Time Overhead = time taken for parallel_hashtable – time taken for parallel_mutex

These are times taken to complete using 1,2,3,4..8 threads.

8.681130 9.131634 9.147779 9.295456 9.189071 9.226427 9.194912 9.283103

1. Overhead for 1 thread: $9.131634 - 8.619860 = 0.511774$ seconds
2. Overhead for 2 threads: $9.131634 - 4.393746 = 4.737888$ seconds
3. Overhead for 3 threads: $9.147779 - 3.956057 = 5.191722$ seconds
4. Overhead for 4 threads: $9.295456 - 3.042890 = 6.252566$ seconds
5. Overhead for 5 threads: $9.189071 - 2.971401 = 6.217670$ seconds
6. Overhead for 6 threads: $9.226427 - 3.111116 = 6.115311$ seconds
7. Overhead for 7 threads: $9.194912 - 2.968731 = 6.226181$ seconds
8. Overhead for 8 threads: $9.283103 - 2.706698 = 6.576405$ seconds

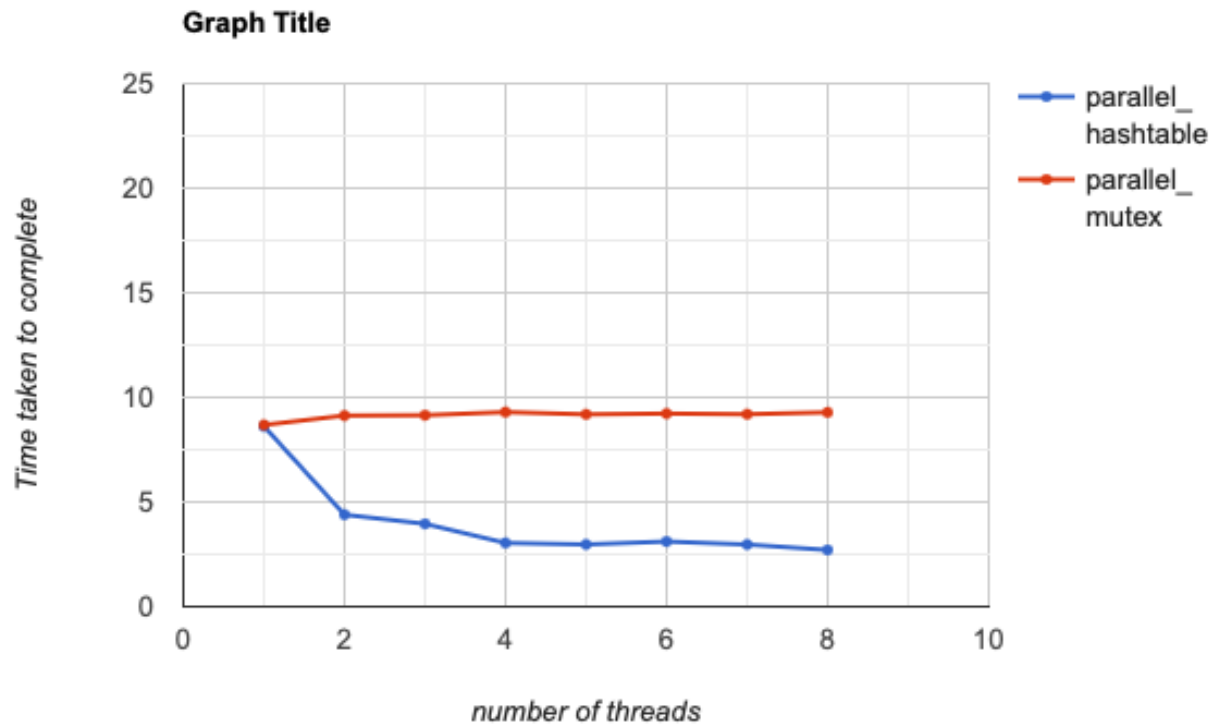
Cumulative overhead = 41.829517

Execution of parallel_mutex.c:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ ./parallel_mutex 1
[main] Inserted 100000 keys in 0.038213 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 8.681130 seconds
$ ./parallel_mutex 2
[main] Inserted 100000 keys in 0.040001 seconds
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[main] Retrieved 100000/100000 keys in 9.131634 seconds
$ ./parallel_mutex 3
[main] Inserted 100000 keys in 0.047560 seconds
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[thread 2] 0 keys lost!
[main] Retrieved 100000/100000 keys in 9.147779 seconds
$ ./parallel_mutex 4
[main] Inserted 100000 keys in 0.048247 seconds
[thread 0] 0 keys lost!
[thread 3] 0 keys lost!
[thread 1] 0 keys lost!
[thread 2] 0 keys lost!
[main] Retrieved 100000/100000 keys in 9.295456 seconds
$ ./parallel_mutex 5
[main] Inserted 100000 keys in 0.050977 seconds
[thread 4] 0 keys lost!
[thread 2] 0 keys lost!
[thread 3] 0 keys lost!
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[main] Retrieved 100000/100000 keys in 9.189071 seconds
$ ./parallel_mutex 6
[main] Inserted 100000 keys in 0.044059 seconds
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[thread 3] 0 keys lost!
[thread 2] 0 keys lost!
[thread 5] 0 keys lost!
[thread 4] 0 keys lost!
[main] Retrieved 100000/100000 keys in 9.226427 seconds
$ ./parallel_mutex 7
[main] Inserted 100000 keys in 0.045579 seconds
[thread 3] 0 keys lost!
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[thread 2] 0 keys lost!
[thread 5] 0 keys lost!
[thread 4] 0 keys lost!
[thread 6] 0 keys lost!
[main] Retrieved 100000/100000 keys in 9.194912 seconds
$ ./parallel_mutex 8
[main] Inserted 100000 keys in 0.042317 seconds
[thread 2] 0 keys lost!
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[thread 5] 0 keys lost!
[thread 3] 0 keys lost!
[thread 4] 0 keys lost!
[thread 6] 0 keys lost!
```

Graph of time taken for parallel_hashtable vs parallel_mutex



We can see that parallel_mutex execution has remained almost the same regardless of increase in number of threads.

Whereas the time execution in parallel_hashtable has decreased significantly by increasing the number of threads.

Part -3

If you were to replace all mutexes with spinlocks, what do you think will happen to the running time? Write a short answer describing what you expect to happen, and why the differences in mutex vs. spinlock implementations lead you to that conclusion

When replacing mutexes with spinlocks, I would expect the running time to decrease compared to using mutexes. Spinlocks are generally considered more lightweight and have lower overhead than mutexes.

Mutexes involve blocking the thread when it tries to acquire a lock, and the thread is only unblocked when the lock is acquired. This blocking and unblocking operation introduces additional overhead, especially in situations with high contention.

On the other hand, spinlocks work by repeatedly "spinning" in a busy-wait loop until the lock is acquired. This approach is more efficient when lock contention is brief because it avoids the context-switching overhead associated with blocking and unblocking threads.

In summary, I expect that using spinlocks will result in reduced running time compared to mutexes, especially in scenarios with short critical sections and low contention but not sure about more critical or introducing more no of threads.

Time Overhead = time taken for parallel_hashtable – time taken for parallel_spin

These are timetaken to complete using 1,2,3,4..8 threads.

8.578704 8.665593 10.545755 10.754553 11.163455 16.094853 20.432355 22.897329

8.578704 - 8.614970 = -0.036266 seconds

8.665593 - 8.619860 = 0.045733 seconds

10.545755 - 8.614970 = 1.930785 seconds

10.754553 - 9.131634 = 1.622919 seconds

11.163455 - 9.189071 = 1.974384 seconds

16.094853 - 9.226427 = 6.868426 seconds

20.432355 - 9.194912 = 11.237443 seconds

22.897329 - 9.283103 = 13.614226 seconds

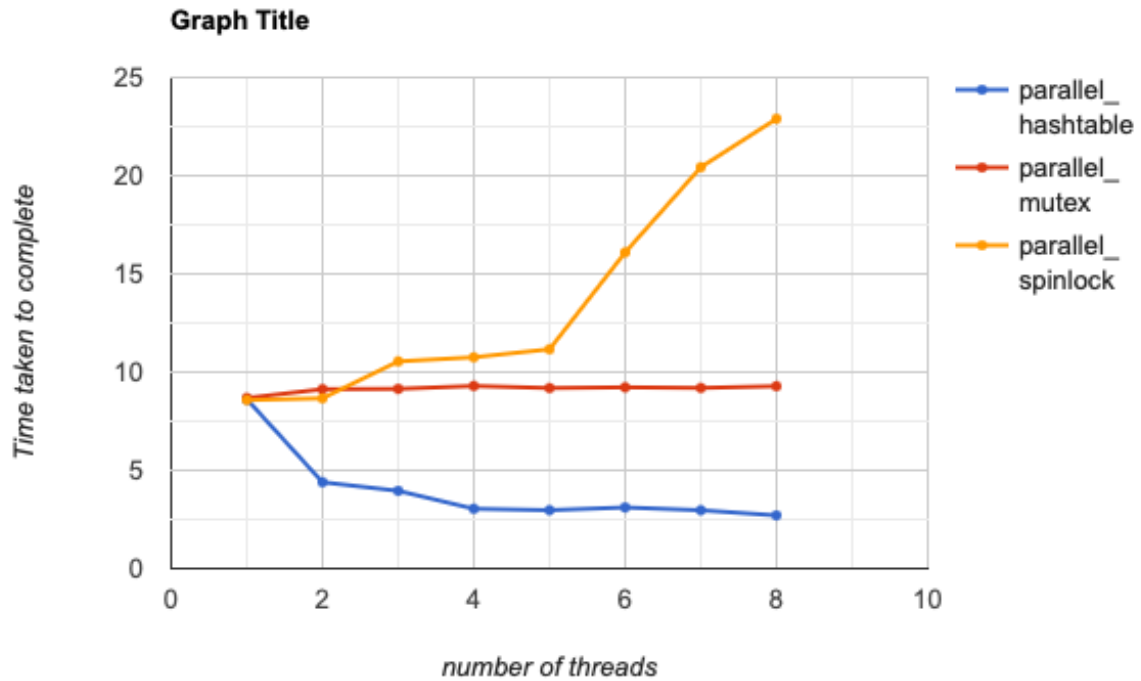
Total cumulative overhead = 37.25765

Execution of parallel_spin:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ ./parallel_spin 1
[main] Inserted 100000 keys in 0.038270 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 8.578704 seconds
$ ./parallel_spin 2
[main] Inserted 100000 keys in 0.033884 seconds
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 8.665593 seconds
$ ./parallel_spin 3
[main] Inserted 100000 keys in 0.034144 seconds
[thread 1] 0 keys lost!
[thread 2] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 10.545755 seconds
$ ./parallel_spin 4
[main] Inserted 100000 keys in 0.033933 seconds
[thread 3] 0 keys lost!
[thread 0] 0 keys lost!
[thread 2] 0 keys lost!
[thread 1] 0 keys lost!
[main] Retrieved 100000/100000 keys in 10.754553 seconds
$ ./parallel_spin 5
[main] Inserted 100000 keys in 0.040241 seconds
[thread 2] 0 keys lost!
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[thread 4] 0 keys lost!
[thread 3] 0 keys lost!
[main] Retrieved 100000/100000 keys in 11.163455 seconds
$ ./parallel_spin 6
[main] Inserted 100000 keys in 0.045086 seconds
[thread 4] 0 keys lost!
[thread 3] 0 keys lost!
[thread 5] 0 keys lost!
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[thread 2] 0 keys lost!
[main] Retrieved 100000/100000 keys in 16.094853 seconds
$ ./parallel_spin 7
[main] Inserted 100000 keys in 0.034635 seconds
[thread 0] 0 keys lost!
[thread 2] 0 keys lost!
[thread 6] 0 keys lost!
[thread 5] 0 keys lost!
[thread 3] 0 keys lost!
[thread 4] 0 keys lost!
[thread 1] 0 keys lost!
[main] Retrieved 100000/100000 keys in 20.432355 seconds
$ ./parallel_spin 8
[main] Inserted 100000 keys in 0.039997 seconds
[thread 2] 0 keys lost!
[thread 0] 0 keys lost!
[thread 3] 0 keys lost!
[thread 6] 0 keys lost!
[thread 7] 0 keys lost!
[thread 1] 0 keys lost!
[thread 5] 0 keys lost!
```

Graph:



We can see that spinlock execution has increased significantly when we have increased the number of threads but has consumed less time with 1 thread when compared to others.

My hypothesis was correct for one thread but not when more number of threads are introduced.

Part-4

Let's revisit your mutex-based code. When we retrieve an item from the hash table, do we need a lock? Write a short answer and explain why or why not.

When considering whether to use a lock during the retrieval of items from the hash table, it's crucial to assess the nature of the concurrent operations happening in the program.

In situations where multiple threads are exclusively performing read operations (retrievals) without any concurrent write operations (such as insertions or deletions), omitting a lock during retrieval might seem feasible. This is grounded in the understanding that reading from shared data structures is generally considered safe when there are no concurrent writes.

However, it's essential to recognize that the decision to forego a lock during retrieval is contingent on the absence of any concurrent write operations. If there's a possibility of simultaneous write operations occurring, such as insertions or deletions, it becomes imperative to introduce a lock. This precautionary measure is crucial for ensuring data consistency and preventing race conditions that could lead to inaccurate or partially updated results.

The necessity of a lock during retrieval hinges on the specific requirements of the application and the potential for concurrent write operations. When there's any chance of concurrent writes, opting for a lock becomes a prudent choice to uphold data integrity and prevent unintended inconsistencies.

Changes I made:

I just left the insert function same as in the `parallel_mutex.c` and have removed mutex locks in retrieve function.

Execution of parallel_mutex_opt_1.c:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ gcc -pthread parallel_mutex_opt_1.c -o parallel_mutex_opt_1
$ ./parallel_mutex_opt_1 1
[main] Inserted 100000 keys in 0.038280 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 8.665619 seconds
$ ./parallel_mutex_opt_1 2
[main] Inserted 100000 keys in 0.038347 seconds
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 4.410507 seconds
$ ./parallel_mutex_opt_1 3
[main] Inserted 100000 keys in 0.046224 seconds
[thread 2] 0 keys lost!
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 3.430454 seconds
$ ./parallel_mutex_opt_1 4
[main] Inserted 100000 keys in 0.051823 seconds
[thread 3] 0 keys lost!
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[thread 2] 0 keys lost!
[main] Retrieved 100000/100000 keys in 2.484234 seconds
$ ./parallel_mutex_opt_1 5
[main] Inserted 100000 keys in 0.048700 seconds
[thread 4] 0 keys lost!
[thread 2] 0 keys lost!
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[thread 3] 0 keys lost!
[main] Retrieved 100000/100000 keys in 3.200293 seconds
$ ./parallel_mutex_opt_1 6
[main] Inserted 100000 keys in 0.049645 seconds
[thread 2] 0 keys lost!
[thread 5] 0 keys lost!
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[thread 3] 0 keys lost!
[thread 4] 0 keys lost!
[main] Retrieved 100000/100000 keys in 2.695721 seconds
$ ./parallel_mutex_opt_1 7
[main] Inserted 100000 keys in 0.048095 seconds
[thread 6] 0 keys lost!
[thread 5] 0 keys lost!
[thread 2] 0 keys lost!
[thread 3] 0 keys lost!
[thread 1] 0 keys lost!
[thread 4] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 2.491166 seconds
$ ./parallel_mutex_opt_1 8
[main] Inserted 100000 keys in 0.056539 seconds
[thread 5] 0 keys lost!
[thread 3] 0 keys lost!
[thread 2] 0 keys lost!
[thread 6] 0 keys lost!
[thread 4] 0 keys lost!
[thread 1] 0 keys lost!
```

Part-5

Last, let's consider insertions. Describe a situation in which multiple insertions could happen safely (hint: what's a bucket?).

In the context of a hash table, a bucket is a slot or container that holds key-value pairs. Each bucket corresponds to a specific hash value, and multiple keys can map to the same bucket due to hash collisions. The objective is to design the hash table in a way that allows multiple insertions to different buckets to happen safely and concurrently.

A situation in which multiple insertions could happen safely involves ensuring that each insertion operation targets a distinct bucket. Since each bucket is independent and doesn't share data with other buckets, concurrent insertions to different buckets won't result in data races or conflicts.

To achieve this, the hash function used to determine the bucket index for a given key should be designed such that different keys are more likely to hash to different buckets. This reduces the likelihood of collisions and ensures that threads inserting different keys can safely operate on their respective buckets without interfering with each other.

In summary, for safe concurrent insertions, the key is to design the hash function to distribute keys evenly across buckets, minimizing the chance of collisions and allowing threads to independently insert into different buckets without the need for locks or synchronization.

Multiple insertions can happen safely in parallel when they target different buckets in the hash table. In a hash table, a bucket is a container that holds key-value pairs. Each bucket is identified by a unique index, typically determined by the hash value of the keys.

Since different keys can hash to different indices, threads inserting key-value pairs with different keys (hashing to different buckets) can safely operate concurrently without conflicting with each other. This is because they are modifying different parts of the hash table (different buckets), and thus, there is no contention for the same memory location.

In other words, as long as the threads are inserting key-value pairs into distinct buckets, there won't be any data dependencies or conflicts, and the insertions can happen safely in parallel. This scenario allows for better utilization of multiple threads, promoting parallelism and potentially improving the overall performance of the hash table operations.

Explanations on changes I made:

The modifications made to the code involve introducing individual mutexes for each bucket in the hash table, thereby replacing the global mutex. Specifically, for each bucket, a mutex is initialized, and before performing any operations on the bucket (insertion or retrieval), the corresponding mutex is locked. After the operation is completed, the mutex is unlocked.

Here are the key changes made to the code:

1. Mutex Initialization for Each Bucket:

For each bucket in the hash table, a separate mutex is initialized. This is achieved by introducing an array of mutexes.

```
pthread_mutex_t mutex[NUM_BUCKETS];
```

Initialization is done in the main function:

```
for (i = 0; i < NUM_BUCKETS; i++) {  
    if (pthread_mutex_init(&mutex[i], NULL) != 0) {  
        perror("Mutex initialization failed");  
        exit(1);  
    }  
}
```

2. Locking and Unlocking Mutexes in Insert and Retrieve Functions:

The critical sections in the `insert` and `retrieve` functions, where operations are performed on the hash table buckets, are protected by locking and unlocking the corresponding mutex.

```
// Lock the mutex before performing operations on the bucket  
pthread_mutex_lock(&mutex[i]);
```

```
// ... Perform operations on the bucket ...
```

```
// Unlock the mutex after the operations are completed  
pthread_mutex_unlock(&mutex[i]);
```

3. Mutex Destruction:

After all the operations are completed, the mutexes are destroyed in the main function.

```
for (i = 0; i < NUM_BUCKETS; i++) {  
    pthread_mutex_destroy(&mutex[i]);  
}
```

These changes address the issues related to concurrent access to the hash table buckets. Each bucket is now protected by its own mutex, ensuring that insertions and retrievals are performed atomically and preventing race conditions that could lead to data corruption. The use of individual mutexes for each bucket enhances parallelism by allowing multiple threads to operate on different buckets simultaneously, reducing contention compared to a single global mutex.

Execution of parallel_mutex_opt.c:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ gcc -pthread parallel_mutex_opt.c -o parallel_mutex_opt
$ ./parallel_mutex_opt 1
[main] Inserted 100000 keys in 0.038962 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 8.644975 seconds
$ ./parallel_mutex_opt 2
[main] Inserted 100000 keys in 0.035725 seconds
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 4.624151 seconds
$ ./parallel_mutex_opt 3
[main] Inserted 100000 keys in 0.036477 seconds
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[thread 2] 0 keys lost!
[main] Retrieved 100000/100000 keys in 3.043021 seconds
$ ./parallel_mutex_opt 4
[main] Inserted 100000 keys in 0.035476 seconds
[thread 3] 0 keys lost!
[thread 2] 0 keys lost!
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 3.300220 seconds
$ ./parallel_mutex_opt 5
[main] Inserted 100000 keys in 0.041079 seconds
[thread 2] 0 keys lost!
[thread 3] 0 keys lost!
[thread 4] 0 keys lost!
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 3.794860 seconds
$ ./parallel_mutex_opt 6
[main] Inserted 100000 keys in 0.038144 seconds
[thread 5] 0 keys lost!
[thread 4] 0 keys lost!
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[thread 3] 0 keys lost!
[thread 2] 0 keys lost!
[main] Retrieved 100000/100000 keys in 2.487802 seconds
$ ./parallel_mutex_opt 7
[main] Inserted 100000 keys in 0.039894 seconds
[thread 5] 0 keys lost!
[thread 6] 0 keys lost!
[thread 0] 0 keys lost!
[thread 3] 0 keys lost!
[thread 4] 0 keys lost!
[thread 2] 0 keys lost!
[thread 1] 0 keys lost!
[main] Retrieved 100000/100000 keys in 2.598037 seconds
$ ./parallel_mutex_opt 8
[main] Inserted 100000 keys in 0.038463 seconds
[thread 0] 0 keys lost!
[thread 7] 0 keys lost!
[thread 3] 0 keys lost!
[thread 6] 0 keys lost!
[thread 4] 0 keys lost!
[thread 1] 0 keys lost!
```