Mutex:

Analysis of the Code and Explanation of Lost Entries

An entry is considered "lost" in the hash table when a key-value pair that was inserted by one thread is no longer retrievable during the retrieval phase. This unintended behavior occurs due to **race conditions** during concurrent writes (`insert`) to the hash table by multiple threads.

Root Cause: Race Condition in `insert` Function

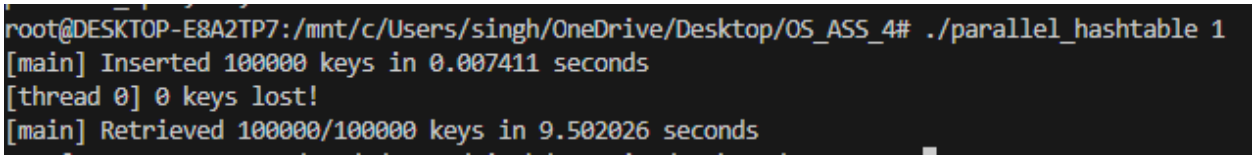
The `insert` function directly modifies the hash table (`table`) without any synchronization mechanism. Here's how this leads to a **race condition**:

1. **Concurrent Modifications**:
    - Multiple threads can attempt to insert entries into the same bucket at the same time (e.g., if two keys hash to the same bucket).
    - Because `insert` modifies the linked list (`table[i]`) without any locks or synchronization, threads can overwrite each other's updates.
2. **Overwriting Entries**:
    - Thread A inserts an entry into bucket `i` and sets `table[i]` to point to the new entry.
    - While Thread A is still processing, Thread B inserts another entry into the same bucket and updates `table[i]` to its new entry.
    - This causes the entry added by Thread A to be disconnected from the linked list, effectively "losing" that entry.

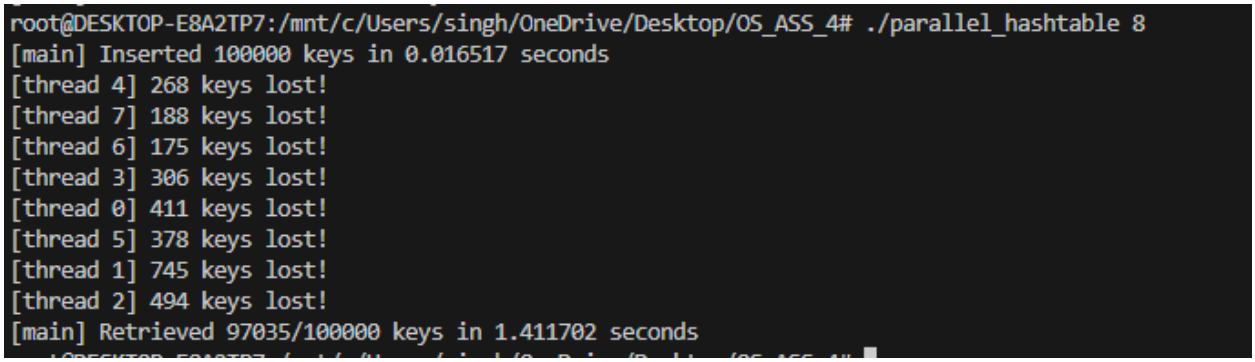
What Does It Mean for an Entry to Be "Lost"?

- An entry is "lost" when:
    - It was successfully inserted by a thread into the hash table.
    - It cannot be retrieved later because it was overwritten or disconnected from the bucket's linked list due to a race condition.

In the provided output:

- With a single thread (`./parallel_hashtable 1`), no entries are lost because there is no concurrent modification of the hash table.
- 

- With multiple threads (`./parallel_hashtable 8`), some entries are lost because threads overwrite each other's updates to the same bucket.

- 

Parts of the Code Causing the Issue

1. **`insert` Function**:
    - The lack of a synchronization mechanism (e.g., mutex or spinlock) for bucket-level access is the primary cause.
    - This allows multiple threads to simultaneously update `table[i]` and its linked list.
2. **Shared Resource: `table`**:
    - The hash table (`table`) is a shared resource, but there is no mechanism to prevent concurrent modification of the same bucket.

How to Fix the Issue?

To prevent lost entries, synchronization must be added to protect access to each bucket during the `insert` operation. For example:

- Use a **mutex** or **spinlock** per bucket to ensure only one thread can modify a bucket at a time.
- Lock the bucket while performing modifications in the `insert` function.

Lost entries occur due to **unsynchronized writes** to the shared hash table (`table`) when multiple threads modify the same bucket simultaneously. Adding bucket-level locks will ensure that each bucket is updated atomically, preventing entries from being lost.

Analysis of Prior Code's Unintended Behaviors

In the original `parallel_hashtable.c`:

1. **Race Conditions During Insertion**:
    - Multiple threads could modify the same bucket simultaneously without synchronization.
    - This led to overwriting or disconnecting entries in the bucket's linked list, causing keys to be "lost."
2. **Unprotected Access**:
    - Both `insert` and `retrieve` operated on the shared hash table (`table`) without any locking mechanisms, resulting in inconsistent states during concurrent operations.

How It Was Addressed in `parallel_mutex.c`

1. **Use of Mutexes**:
    - Introduced a **mutex (`pthread_mutex_t`) per bucket** to synchronize access.
    - During `insert`, the mutex ensures that only one thread can modify a bucket at a time, preventing overwriting or disconnecting entries.
    - During `retrieve`, the mutex ensures consistent reads, blocking other threads from modifying the bucket while a thread is retrieving a key.
2. **Thread-Safe Access**:
    - Both `insert` and `retrieve` now lock the corresponding bucket before accessing or modifying it, ensuring atomic operations and maintaining the integrity of the hash table.

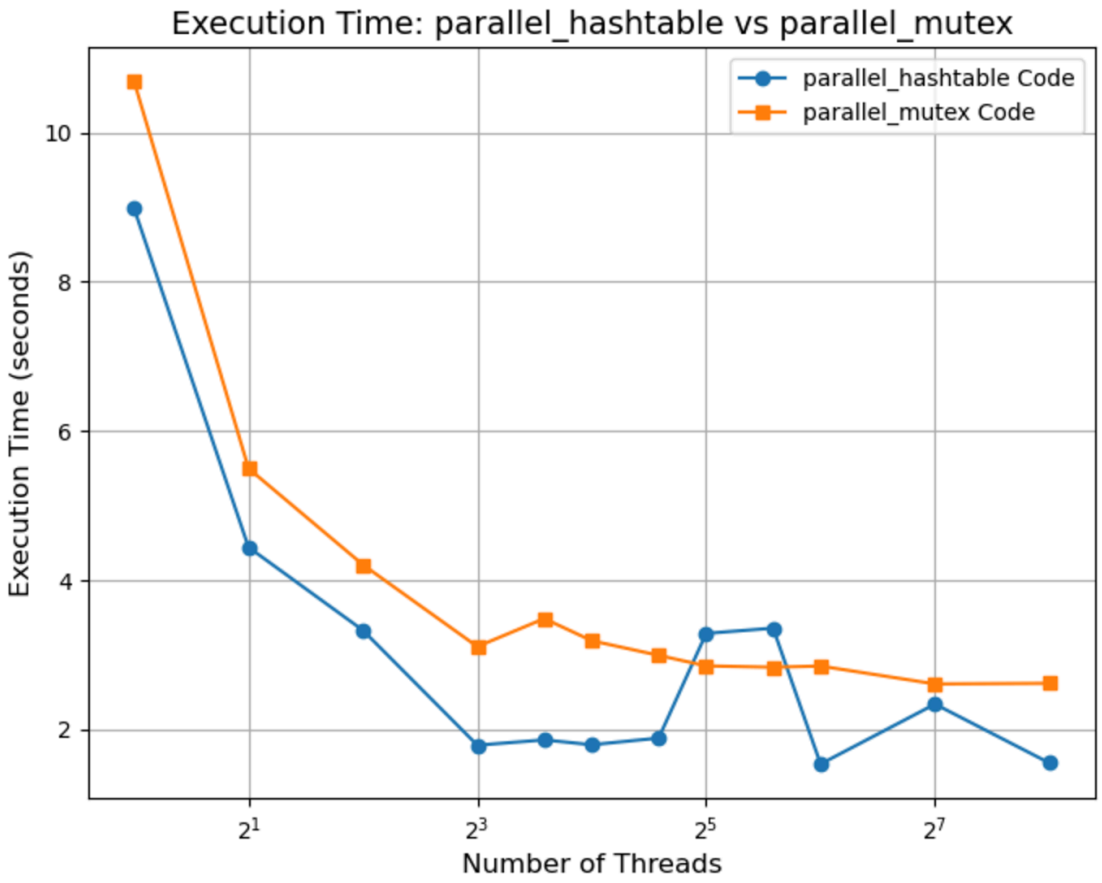
Key Improvements

- No entries are "lost" during concurrent insertions or retrievals.
- The program successfully handles multiple threads while maintaining data consistency and correctness.

```
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4# ./parallel_mutex 1
[main] Inserted 100000 keys in 0.009274 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 7.375242 seconds
[main] Total running time: 7.384516 seconds
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4# ./parallel_mutex 8
[main] Inserted 100000 keys in 0.016445 seconds
[thread 2] 0 keys lost!
[thread 7] 0 keys lost!
[thread 5] 0 keys lost!
[thread 6] 0 keys lost!
[thread 3] 0 keys lost!
[thread 4] 0 keys lost!
[thread 0] 0 keys lost!
[thread 1] 0 keys lost!
[main] Retrieved 100000/100000 keys in 3.416721 seconds
[main] Total running time: 3.433166 seconds
```

Graph parallel_hashtable vs parallel_mutex:

Here the hashtable is executing faster but we have to keep in mind that the keys are also getting lost.



Spinlock:

What Happens When Replacing Mutexes with Spinlocks?

Replacing mutexes with spinlocks affects the running time because of the fundamental differences in how spinlocks and mutexes manage thread synchronization. Here's what we can expect:

Expected Impact on Running Time

1. **Low Contention (e.g., 1 thread):**
    - **Expected:** The performance should be similar to mutexes.
        - With only one thread, there is no contention, and the overhead of acquiring/releasing locks is minimal for both spinlocks and mutexes.
    - **Why:** Spinlocks do not incur any additional waiting overhead because no threads are contending for locks.
2. **High Contention (e.g., multiple threads):**
    - **Expected:** Spinlocks may perform faster than mutexes in **short critical sections** because spinlocks avoid the overhead of putting threads to sleep and waking them up.
    - However, if the critical section is long, spinlocks can lead to higher CPU usage due to busy-waiting, potentially degrading performance when many threads compete for the same lock.
    - **Why:** Mutexes put threads to sleep if the lock is unavailable, saving CPU cycles, whereas spinlocks continuously consume CPU cycles while waiting.

Comparison of Observed Results

Threads	parallel_mutex Running Time	parallel_spin Running Time
1	7.384516 seconds	7.167404 seconds
8	3.43366 seconds	1.312867 seconds

1. **For 1 Thread**:
    - The running times are nearly identical because there is no contention. Both mutexes and spinlocks incur minimal overhead for a single thread.

2. **For 8 Threads**:

    - Spinlocks significantly outperform mutexes.
    - This suggests that the critical sections in this program are relatively short, and spinlocks benefit from avoiding the context-switching overhead of mutexes.

Why Does This Happen?

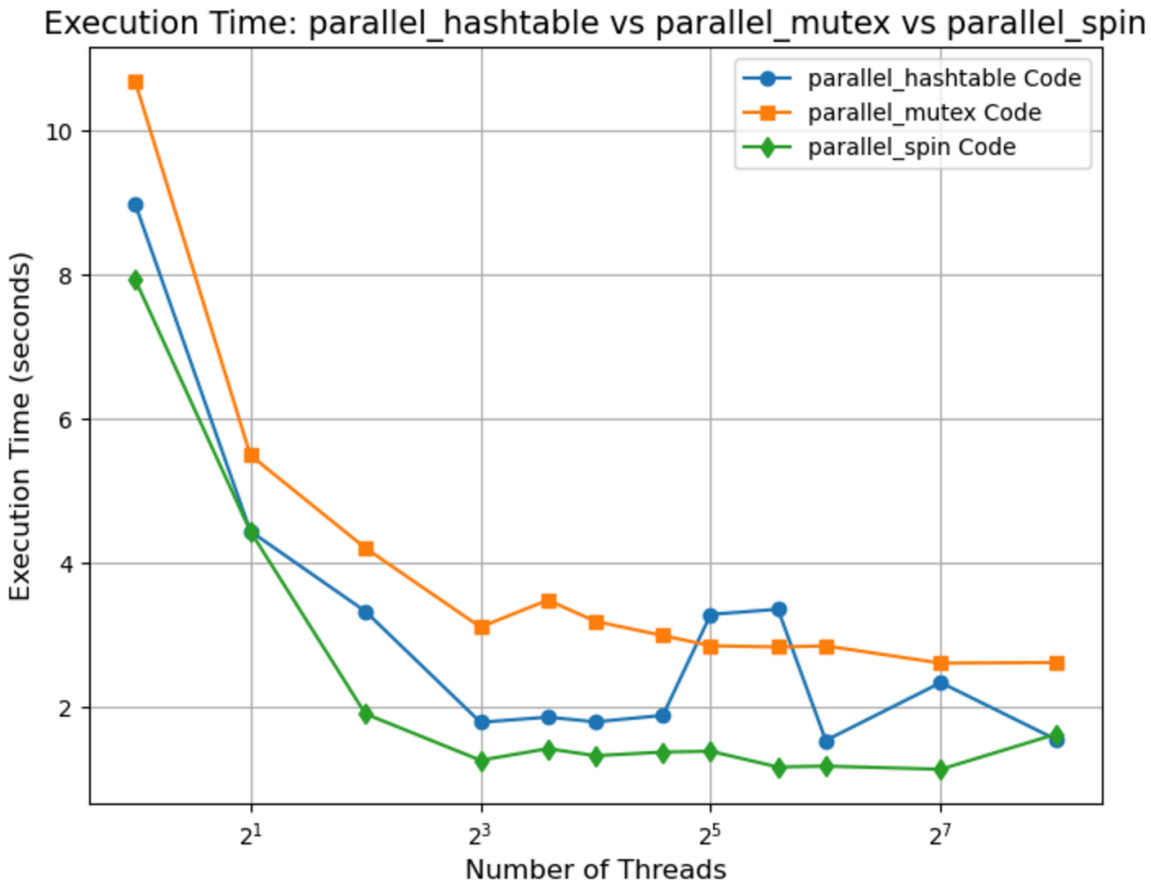
1. **Spinlocks:**
    - Spinlocks continuously check for the lock in a loop (busy-waiting). This makes them efficient in low-contention or short-critical-section scenarios because threads avoid the overhead of being put to sleep and waking up.
    - However, they consume CPU cycles while waiting, potentially degrading performance in high-contention or long-critical-section cases.
2. **Mutexes:**
    - Mutexes put threads to sleep if the lock is unavailable, which conserves CPU cycles but incurs overhead for the context switch (putting a thread to sleep and waking it up).
    - They perform better in scenarios with long critical sections or high contention, where busy-waiting would waste significant CPU cycles.

Conclusion

- **Low Contention or Short Critical Sections:** Spinlocks perform better due to reduced overhead.
- **High Contention or Long Critical Sections:** Mutexes perform better because they avoid wasting CPU cycles on busy-waiting.
- In this program, spinlocks excel because the critical sections (hash table bucket access) are short, and contention is moderate.

```
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4# ./parallel_spin 1
[main] Inserted 100000 keys in 0.008302 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 7.167404 seconds
[main] Total running time: 7.175706 seconds
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4# ./parallel_spin 8
[main] Inserted 100000 keys in 0.013251 seconds
[thread 7] 0 keys lost!
[thread 5] 0 keys lost!
[thread 4] 0 keys lost!
[thread 2] 0 keys lost!
[thread 3] 0 keys lost!
[thread 6] 0 keys lost!
[thread 1] 0 keys lost!
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 1.299616 seconds
[main] Total running time: 1.312867 seconds
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4#
```

Graph of parallel_hashtable vs. mutex running time vs. spinlock running time:



Mutex, Retrieve Parallelization

Do we need a lock for retrieval?

Yes, but only under certain conditions:

- **We need a lock for writers:** If there are concurrent writes to a bucket, a lock is necessary during retrieval to ensure consistency.
- **We don't need a lock for multiple readers:** Multiple retrieval operations (reads) can happen concurrently as long as no thread is modifying the same bucket.

What changes were made?

To allow multiple retrieval operations to run in parallel:

1. **Reader-Writer Synchronization**:
    - Introduced a `reader_lock` and a `readCount` for each bucket.
    - The first reader locks the bucket (`lock`), and subsequent readers increment the `readCount`.
    - The last reader unlocks the bucket.
2. **Concurrent Reads**:
    - Multiple threads can now read a bucket simultaneously without blocking each other, as long as no write is happening to the same bucket.
3. **Code Updates**:
    - Added a reader-writer synchronization model in the `retrieve` function using `reader_lock` and `readCount`.

Mutex, Insert Parallelization

When can multiple insertions happen safely?

Multiple insertions can occur safely if:

- Each insertion targets a **different bucket**. Since buckets are independent, threads modifying different buckets do not interfere with each other.

What changes were made?

1. **Per-Bucket Locking**:
    - Each bucket has its own lock (`lock`), ensuring that only one thread modifies a specific bucket at a time.
    - Different threads can insert into different buckets concurrently.
2. **Parallel Insertions Across Buckets**:
    - By locking only the specific bucket being modified, multiple threads can safely perform insertions into distinct buckets in parallel.
3. **Code Updates**:
    - Updated the `insert` function to lock only the relevant bucket (`lock[i]`), allowing concurrent writes to different buckets.

Summary of Modifications

1. **Reader-Writer Parallelism**:
    - Enabled concurrent reads with a reader-writer synchronization model using `reader_lock` and `readCount`.
    - Ensures thread safety by locking a bucket only when the first reader enters or when a writer accesses the bucket.
2. **Insert Parallelism Across Buckets**:
    - Allowed multiple threads to insert into different buckets concurrently by using per-bucket locks.

Parallel Mutex Optimized Code Explanation

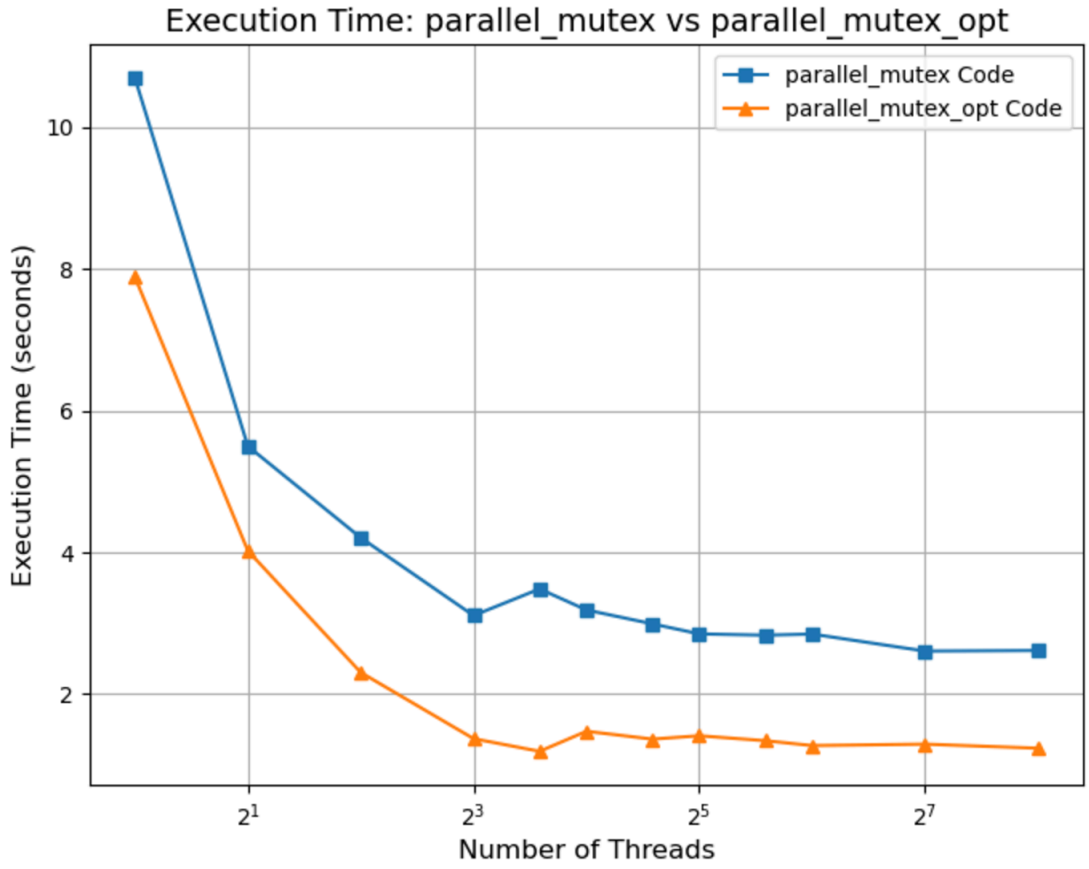
The changes in `parallel_mutex_opt.c` allow:

- **Efficient Retrieval**: Multiple threads can retrieve keys from the hash table simultaneously if they access different buckets or there are no ongoing writes.
- **Efficient Insertion**: Multiple threads can insert keys into different buckets concurrently, improving performance for multi-threaded workloads.

By leveraging bucket-level locks and reader-writer synchronization, this version achieves better parallelism without sacrificing correctness.

```
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4# ./parallel_mutex_opt 1
[main] Inserted 100000 keys in 0.009500 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 7.694634 seconds
[main] Total running time: 7.704134 seconds
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4# ./parallel_mutex_opt 8
[main] Inserted 100000 keys in 0.016329 seconds
[thread 4] 0 keys lost!
[thread 7] 0 keys lost!
[thread 6] 0 keys lost!
[thread 5] 0 keys lost!
[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 1.195021 seconds
[main] Total running time: 1.211350 seconds
root@DESKTOP-E8A2TP7:/mnt/c/Users/singh/OneDrive/Desktop/OS_ASS_4#
```

Graph parallel_mutex vs parallel_mutex_opt:



All four combined graph: