Homework 4 Explanation

Mutex

An entry is considered “lost” when a key-value pair is inserted by one thread but cannot be found by other threads during retrieval. This unintended behaviour arises due to **data races**, where multiple threads access and modify shared data (hash table buckets) without proper synchronization.

Causes of Lost Entries

1. Concurrent Modifications in insert

Code Block:

```
// Inserts a key-value pair into the table
void insert(int key, int val) {
  int i = key % NUM_BUCKETS;
  bucket_entry *e = (bucket_entry *) malloc(sizeof(bucket_entry));
  if (!e) panic("No memory to allocate bucket!");
  e->next = table[i];
  e->key = key;
  e->val = val;
  table[i] = e;
}
```

Issue:

- When multiple threads call insert concurrently, they modify the same bucket's linked list (table[i]) without synchronization.
- The line e->next = table[i]; and table[i] = e; assume exclusive access to table[i]. Without locking, two threads might:
  - Read the current head of the bucket (table[i]).
  - Update table[i] to point to their new entry, overwriting changes made by the other thread.
- This results in one of the inserted entries being “lost” because its reference (next) is overwritten.

2. Data Races in retrieve

Code Block:

```
// Retrieves an entry from the hash table by key
// Returns NULL if the key isn't found in the table
bucket_entry * retrieve(int key) {
  bucket_entry *b;
  for (b = table[key % NUM_BUCKETS]; b != NULL; b = b->next) {
    if (b->key == key) return b;
  }
  return NULL;
}
```

Issue:

- When threads access the same bucket concurrently during retrieval, the linked list (table[i]) may be in an inconsistent state.
- For example:
    - One thread might be modifying the list (inserting or updating entries) while another is traversing it (b = b->next).
    - This can cause the traversing thread to skip entries or encounter invalid pointers, leading to entries being missed during retrieval.

NOTE:

1. The system on which the following tests are run is a Linux VM and has **4 physical cores.**
2. The running time mentioned below for a given thread count is the sum of the insertion time plus the retrieval time

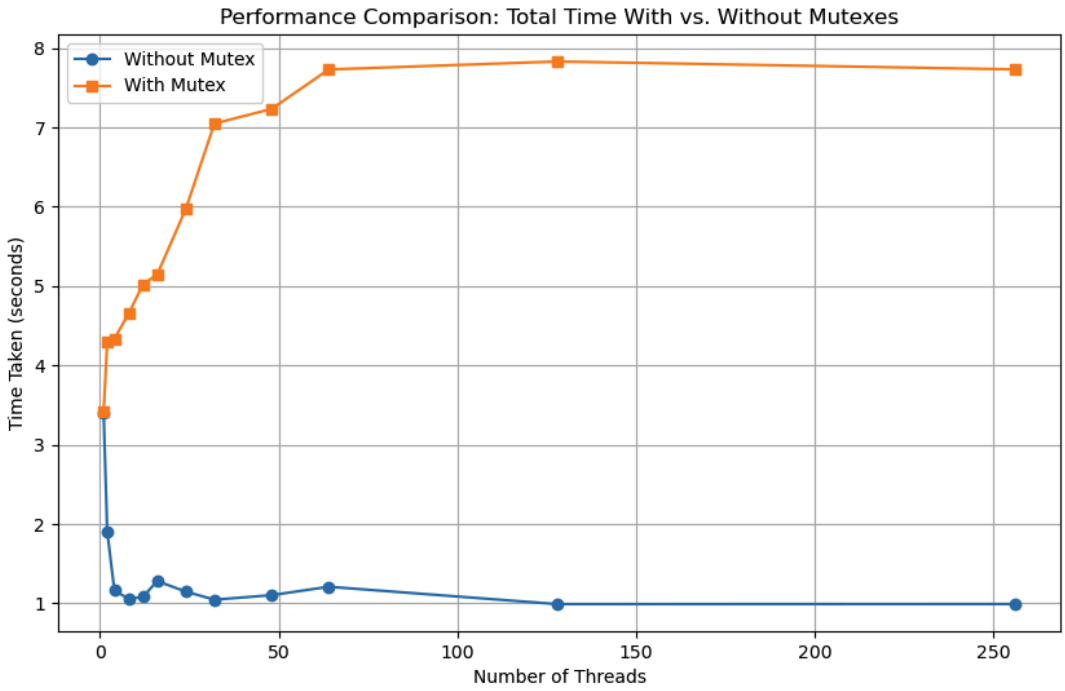
Original vs Mutex Running Times -

We considered thread counts of [1, 2, 4, 8, 12, 16, 24, 32, 48, 64, 128, 256] for comparing the running time between the original implementation and the mutex-based implementation.

The overhead can be calculated as the mean of the percentage change in time taken for each thread count for the mutex code over the original code. We sum up all the times taken with and without mutex across all thread counts and calculate the percentage.

Average Overhead = ∑ ((Time with Mutex – Time without mutex)*100 / Time without mutex)

Following is the graph which compares the running times between the original implementation and the mutex-based implementation.



The time overhead of the mutex-based implementation is 405%, indicating the additional time required to ensure correctness compared to the unsafe, non-thread-safe hash table. This overhead is caused by the use of a global mutex, which serializes all access to the hash table. As a result, threads cannot operate on different buckets concurrently, even if their operations are independent. Every thread must acquire the mutex before performing an insert or retrieve operation, causing contention and delays, especially as the number of threads increases. This serialization eliminates data races and ensures that no keys are lost, maintaining the integrity and consistency of the hash table. However, the trade-off is a significant reduction in performance due to the lack of parallelism, highlighting the cost of guaranteeing thread safety with a global lock.

Spinlock

Replacing mutexes with spinlocks will likely increase running time under high contention because spinlocks rely on busy waiting, where threads continuously check if the lock is available, consuming CPU cycles. Unlike mutexes, which block threads and allow the operating system to schedule other work, spinlocks keep the CPU busy even when the thread cannot proceed. While spinlocks can be faster in scenarios with low contention or short critical sections (due to avoiding the overhead of context switching), their inefficiency in high-contention environments may lead to significant performance degradation, especially as the number of threads increases.

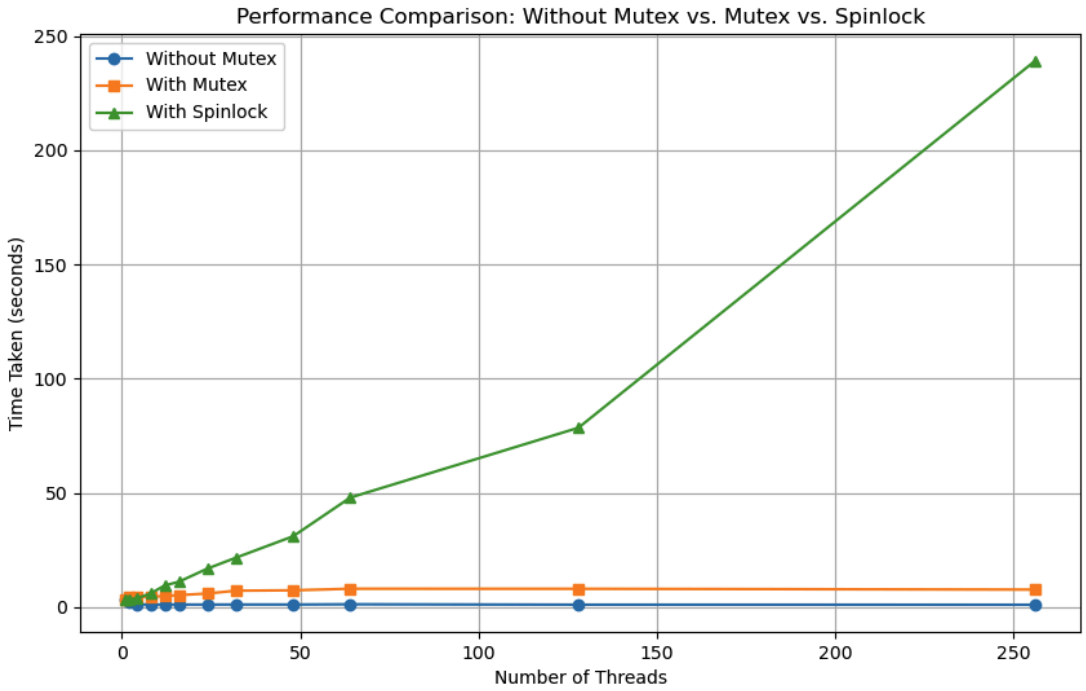
Original vs Mutex vs Spinlock Running Times-

We considered thread counts of [1, 2, 4, 8, 12, 16, 24, 32, 48, 64, 128, 256] for comparing the running time between the original implementation and the spinlock-based implementation.

The overhead can be calculated as the mean of the percentage change in time taken for each thread count for the spinlock code over the original code. We sum up all the times taken with and without mutex across all thread counts and calculate the percentage.

Average Overhead = ∑ ((Time with Spinlock – Time without mutex and spinlock) *100 / Time without mutex and spinlock)

Following is the graph which compares the running times between the original implementation, the mutex-based implementation and the spinlock-based implementation.



The time overhead of the spinlock-based implementation is 3642.61%, which highlights the significant performance penalty incurred compared to the unsafe, non-thread-safe hash table. This overhead is due to the use of a global spinlock, which serializes all access to the hash table and causes threads to engage in busy waiting. Unlike a mutex, which puts threads to sleep when the lock is unavailable, spinlocks continuously check for the lock to be released, consuming CPU resources even while idle. As the number of threads increases, contention for the single global spinlock grows exponentially, leading to excessive spinning and wasted CPU cycles. This behaviour is especially detrimental in scenarios where critical sections (like the insert and retrieve functions) take longer to execute, as threads waiting for the lock consume CPU time unnecessarily. While the spinlock ensures correctness by preventing data races, its poor performance under high contention makes it unsuitable for workloads with multiple threads, highlighting the cost of busy waiting as compared to blocking-based synchronization mechanisms like mutexes.

Mutex, Retrieve Parallelization

No, we do not need a lock for retrieval if the operation only reads data and does not modify it. The retrieve function traverses the linked list of a bucket to find a matching key. Since the retrieval process does not modify the structure of the hash table or the bucket's linked list, it can run safely in parallel with other retrievals, as long as no other thread is writing to the same bucket.

In parallel_mutex_opt.c, the retrieve function was modified to be lock-free by removing the pthread_mutex_lock and pthread_mutex_unlock calls. Since retrieve only reads data without modifying the hash table, locking is unnecessary, as multiple threads can safely traverse the same or different buckets concurrently. This change enables parallel retrievals, significantly improving performance in read-heavy scenarios by eliminating contention among threads accessing the same bucket.

Mutex, Insert Parallelization

Multiple insertions can happen safely if they target different buckets in the hash table, as each bucket is independent of the others. The bucket for a key is determined by the hash function (key % NUM_BUCKETS), so keys with different hash results will map to different buckets. Since each bucket maintains its own separate linked list, insertions into different buckets do not interfere with one another. In the code, this is achieved by using per-bucket locks (i.e., one mutex per bucket), ensuring that only threads inserting into the same bucket are serialized, while threads inserting into different buckets can proceed in parallel.

In parallel_mutex_opt.c, the insert function was updated to lock only the specific bucket being modified (pthread_mutex_lock(&bucket_mutexes[i])) instead of locking the entire hash table. This allows for efficient and safe concurrent insertions across different buckets.