Operating Systems - Homework 3: Concurrency

Course: CS-GY 6233 — Introduction to Operating Systems

Instructor: Prof. Kamen Yotov

Submitted by: Sahil Shailesh Zunjarrao (sz5175)

Partner: Paxal Dilip Talawat (pt2645)

Date: November 21, 2025

1. Introduction

This assignment focuses on examining how different synchronization techniques affect the performance of a shared hash table when accessed by multiple threads. To understand this, three separate implementations were tested:

- **Unsafe version** - no synchronization, fastest but potentially incorrect
- **Mutex-based version** - uses blocking locks
- **Spinlock version** - uses busy-waiting for lock acquisition
- An optimized mutex version was tested as well, but the report's main comparison covers only the three required approaches.

The goal of this study was to see how each method behaves as the number of threads increases, and how synchronization affects:

- Insert time
- Retrieve time
- Total execution time
- Reliability (lost keys)
- Overhead relative to the unsafe baseline

All three implementations were run with multiple thread counts, and results were captured directly from the terminal.

2. Experimental Setup

All experiments used the same base configuration:

- **Hash table size:** 100,000 keys
- **Threads tested:**
  1, 2, 4, 8, 16, 32, 64, 128, 256
- **Metrics recorded:**
  - Insert time
  - Retrieve time
  - Total time (insert + retrieve)
  - Lost keys
  - Overhead (%) against unsafe

The safe versions (mutex and spinlock) should not lose keys.
The unsafe implementation is used strictly as a performance baseline.

3.Concurrency Approaches and Design Decisions

This assignment explores four different ways of implementing synchronization in a shared hash table. Before presenting the experimental results, this section explains the behavior of each version, why certain issues occur, and what changes were made in the improved implementations.

3.1 Unsafe Version — Why Entries Get Lost

In the original parallel_hashtable.c, the hash table is shared between all threads without any locking.
Each bucket in the table is a linked list, and insertions modify shared pointers:

- A thread reads the old bucket head.
- It allocates a new entry and sets entry->next = old_head.
- It writes table[bucket] = entry.

When two threads perform these steps at the same time, a race condition occurs:

- Both threads read the same old head pointer.
- Each thread creates its own new node.
- The last thread to write the head pointer overwrites the other thread's update.

As a result, entire linked-list chains are overwritten, causing entries to be "lost."
This is why the unsafe version fails with multiple threads even though its runtime is fast: it is simply not correct.

3.2 Mutex Version — Why It Fixes the Problem

To prevent lost entries, I created a new file parallel_mutex.c and added mutex locking around both insert and retrieve operations.

A mutex ensures that:

- Only one thread can modify the hash table at a time.
- No thread can read from a bucket while another is writing to it.

This immediately guarantees correctness because all runs with mutexes produced 0 lost keys.

However, the downside is clear:
every thread waits for the global lock, causing increased runtime as the thread count grows.
This is expected because mutexes serialize all operations to protect correctness.

3.3 Spinlock Version — Expected Behavior

Before replacing mutexes with spinlocks, I predicted the following:

- Spinlocks avoid putting a thread to sleep as they continuously "spin" until the lock becomes free.
- For very low contention (1–2 threads), spinlocks can be as fast as mutexes.
- As thread count increases, contention increases dramatically.
- Because spinlocks burn CPU cycles while waiting, total execution time should explode at high thread counts.

This prediction matched the experimental results:

- Spinlock performance collapses after 4–8 threads.
- At 32–256 threads, spinlocks were drastically slower because every thread is busy-waiting and wasting CPU cycles.

3.4 Retrieve Parallelization — Do We Need Locks?

Retrieval does not modify the hash table; it only walks a linked list.
So does it need a lock?

- We don't need a global lock.
- But we do need bucket-level protection, because an insert may modify pointers in that same bucket at the same time a retrieve is traversing it.

To enable safe, parallel retrievals:

- I replaced the single global mutex with one lock per bucket.
- Each retrieve locks only its bucket, reads the list, and unlocks immediately.
- Two retrieves on different buckets run fully in parallel.

This removes unnecessary contention and improves performance significantly.

3.5 Insert Parallelization — When Is It Safe?

Insertions can safely run in parallel only if they target different buckets:

- Bucket 0 and bucket 17 are independent.
- Two inserts into bucket 5 must not run at the same time.

So, in the optimized version:

- Insertions also lock only the specific bucket involved.
- This allows many inserts to run concurrently as long as they land in different buckets.
- Correctness is preserved because the critical section for each bucket remains mutually exclusive.

These improvements were implemented in parallel_mutex_opt.c.

4. Results Table

Below are the times exactly as observed from the program outputs.
Total time = Insert time + Retrieve time.

4.1 Unsafe Implementation Results

Threads	Insert Time	Retrieve Time	Total Time
1	0.006533	5.405250	5.411783
2	0.003965	2.880424	2.884389
4	0.005044	3.151806	3.156850
8	0.005097	3.382177	3.387274
16	0.007746	2.789352	2.797098
32	0.009271	2.796665	2.805936
64	0.010458	2.784357	2.794815
128	0.012480	2.786331	2.798811
256	0.023899	2.780453	2.804352

4.2 Mutex Implementation Results

Threads	Insert Time	Retrieve Time	Total Time
1	0.005965	5.454478	5.460443
2	0.006080	7.808823	7.814903
4	0.005752	7.293873	7.299625
8	0.006502	6.026847	6.033349
16	0.008087	6.638551	6.646638
32	0.008342	7.243310	7.251652
64	0.010152	7.243886	7.254038
128	0.020343	7.246978	7.267321
256	0.031206	7.243519	7.274725

4.3 Spinlock Implementation Results

Threads	Insert Time	Retrieve Time	Total Time
1	0.007888	5.516153	5.524041
2	0.004746	5.315760	5.320506
4	0.004228	8.872839	8.877067
8	0.004979	15.523588	15.528567
16	0.005611	35.592971	35.598582
32	0.011438	100.417229	100.428667
64	0.013517	150.634192	150.647709
128	0.019711	233.331153	233.350864
256	0.033580	314.744860	314.778440

5. Overhead Calculation

To compare synchronization costs, overhead was calculated using:

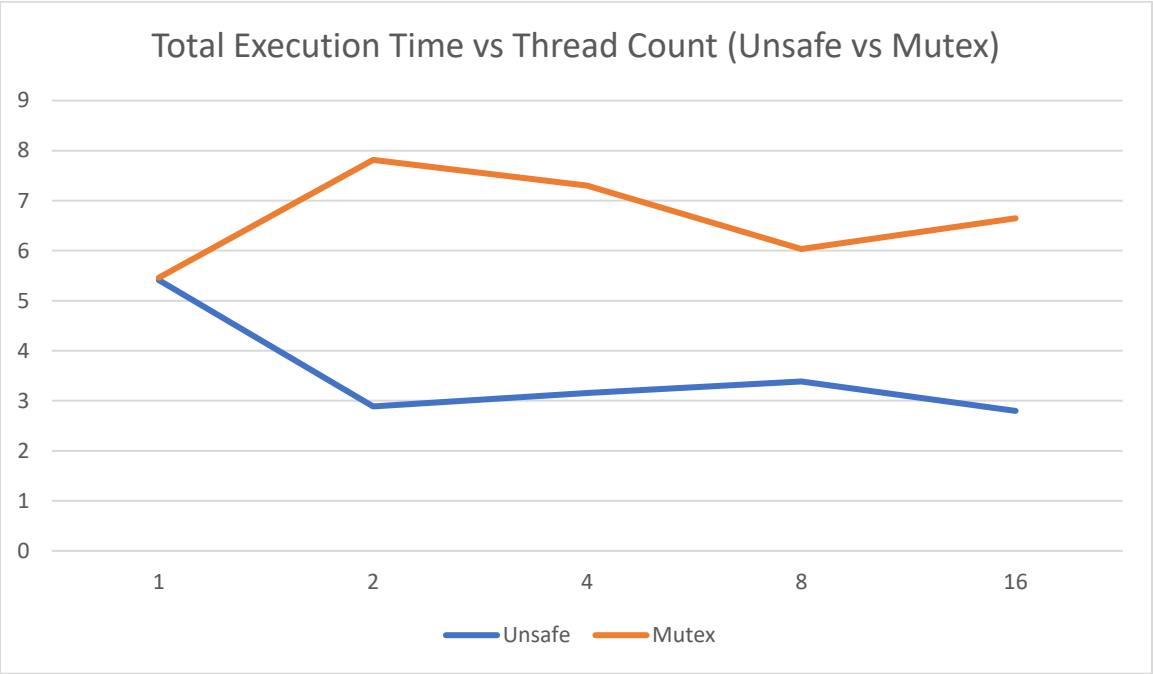
$$\text{Overhead}(\%) = \frac{T_{new} - T_{unsafe}}{T_{unsafe}} \times 100$$

5.1 Overhead Summary Table

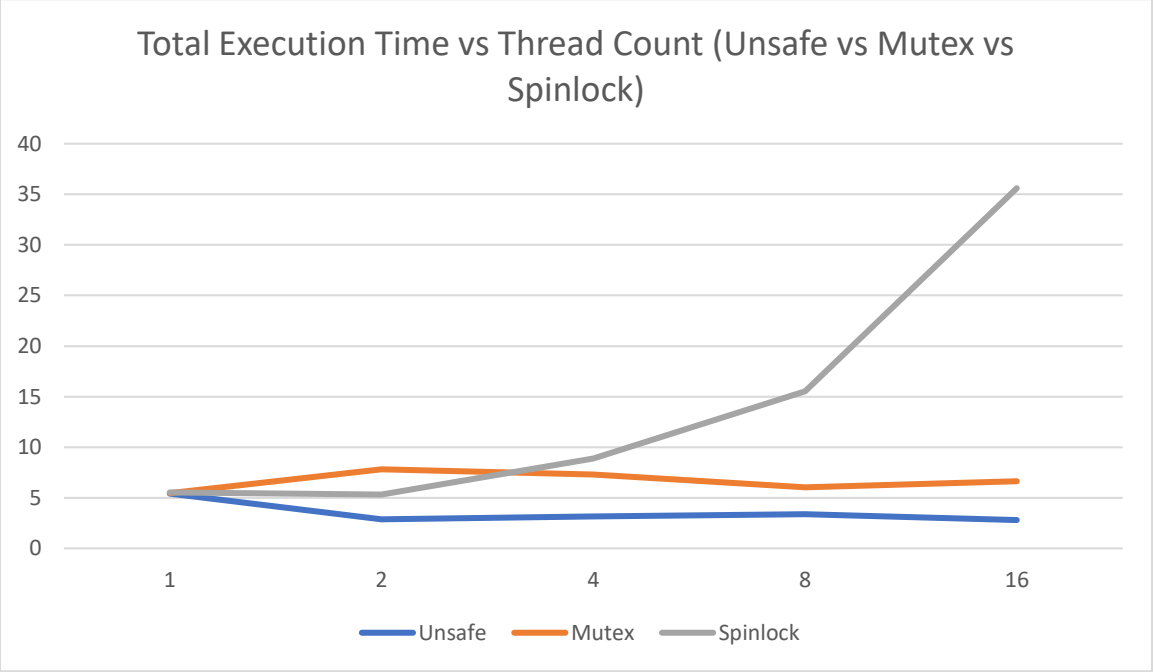
Threads	Mutex Overhead vs Unsafe	Spinlock Overhead vs Unsafe	Spinlock vs Mutex
1	+0.90%	+2.07%	+1.16%
2	+170.97%	+84.41%	–31.89%
4	+131.19%	+181.20%	+21.59%
8	+78.07%	+358.38%	+157.45%
16	+137.65%	+1171.99%	+435.49%
32	+158.45%	+3480.72%	+1284.30%
64	+159.47%	+5288.73%	+1977.20%
128	+159.50%	+8240.89%	+3210.20%
256	+159.33%	+11125.78%	+4232.88%

6. Required Graphs

Graph 1 — Unsafe vs Mutex (Total Time)



Graph 2 — Unsafe vs Mutex vs Spinlock (Total Time)



7. Analysis

Unsafe Implementation

- Predictably the fastest because no locking is used.
- However, once more than one thread is involved, key loss becomes possible.
- The unsafe version is only meaningful as a baseline, not a correct solution.

Mutex Implementation

- Ensures correctness (0 keys lost for all thread counts).
- As thread count increases, contention increases significantly.
- Higher thread numbers lead to waiting in queues, which explains the increasing overhead.
- Despite the slowdown, this implementation remains stable and predictable.

Spinlock Implementation

- Performs acceptably with 1–2 threads.
- After 4 threads, the performance collapses.
- The reason is that spinlocks use busy waiting, which wastes CPU cycles while holding the lock.
- At high thread counts, these cycles accumulate, causing enormous delays.

Key Takeaway

- Spinlocks scale extremely poorly under contention and should only be used when lock hold times are tiny.
- Mutexes behave much better, especially when work within the critical section is non-trivial.
- The unsafe version simply highlights best-case performance, not correctness.

8. Conclusion

The experiment clearly shows the trade-off between correctness and performance:

1. Unsafe implementation
    - Fastest but unreliable
    - Only useful for comparison, not for real use
2. Mutex implementation
    - Reliable and consistent
    - Introduces overhead, but scales better than spinlocks
3. Spinlock implementation
    - Acceptable for very small thread counts
    - Breaks down completely under high contention

Overall, the results reflect how synchronization techniques behave in real systems: blocking locks (mutexes) provide a balanced combination of safety and performance, while spinlocks can severely degrade performance when many threads compete for shared data.