Comparison of Filter Lock and Bakery Locks

Tushita Sharva

September 23, 2024

1 Problem Statement

To implement the two locking algorithms, **Filter Lock & Bakery Lock**, and compare their performance with respect to waiting time and throughput.

Parameters

- n: Number of threads
- k: Number of times each thread should enter the critical section
- λ_1: Parameter for simulating the thread performing actions in the critical section.
- λ_2: Parameter for simulating the thread performing actions outside the critical section.

2 Program Design

The overall structure of both programs is the same, with differences only in how the lock and unlock functions are implemented in the Filter Lock and Bakery Lock classes. Below is the design outline for both programs.

2.1 Classes & Their Methods

Logger Class: This class is responsible for logging events such as entry into the critical section, exit, and other debugging information. It contains two methods:

- `DEBUG(...)`: Logs debugging information to a file.
- `OUTPUT(...)`: Logs important outputs related to entry and exit times of threads.

ThreadData Class: Stores information for each thread, such as its ID and waiting time. All members are private, and they are accessed through methods:

- `getThreadId()`: Returns the thread's ID.
- `setThreadId(int id)`: Sets the thread's ID.
- `getWaitingTime()`: Returns the waiting time for the thread.
- `incrementWaitingTime(int time)`: Increments the waiting time for the thread.

Lock Class (Bakery and Filter): Each locking mechanism is implemented as a separate class in their respective programs.

- **Bakery Lock:** This class uses an array of atomic flags and labels for each thread. It ensures that threads enter the critical section based on their ticket (label).
- **Filter Lock:** This class uses an array of atomic levels and victims, allowing threads to pass through levels to enter the critical section.

However, if we abstract it a level up, both have constructors and destructors that allocate and destroy dynamic memory required for locking algorithms. The lock and unlock functions are implemented as:

- `lock(int threadId)`: Implements the locking mechanism to allow a thread to enter the critical section.
- `unlock(int threadId)`: Releases the lock, allowing other threads to proceed.

2.2 Helper Functions

getSysTime: This function converts a `std::chrono::time_point` into a formatted string representing the system time. It is used to log timestamps of critical section entries and exits.

Timer: This function generates random delays using an exponential distribution, with parameters λ_1 for the critical section and λ_2 for the remainder section. It simulates the time spent by a thread inside and outside the critical section.

testCS: This function simulates a thread requesting entry into the critical section, performing its task, and then exiting. It calls the locking and unlocking functions of the respective algorithm (Bakery or Filter). It also logs the entry and exit times, along with the waiting time.

2.3 Program Flow

1. **Initialization:** The program starts by reading input values from a file. The number of threads, n, the number of critical section entries, k, and the exponential parameters λ_1 and λ_2 are set.
2. **Thread Creation:** A vector of threads and thread data objects is initialized. Each thread is assigned an ID and is executed using the `testCS` function.
3. **Locking Mechanism:** Each thread, when attempting to enter the critical section, calls the `lock` method of the appropriate lock class (Bakery or Filter). The thread then logs its entry, performs its task (with a random delay), and logs its exit.
4. **Completion:** Once all threads complete, the program calculates and logs the total waiting time, average waiting time, and throughput for each locking mechanism.

3 Program Correctness

The correctness of the implementation is ensured by examining several properties, using proof by contradiction where applicable to demonstrate the guarantees provided by the Bakery and Filter algorithms. Both algorithms rely on atomic operations, ensuring correct behavior across varying hardware architectures and memory consistency models.

1. **Mutual Exclusion:**

   - **Condition:** No two threads can be in the critical section at the same time.
   - **Proof by Contradiction:** Assume that mutual exclusion is violated, i.e., two threads T_i and T_j are in the critical section at the same time. In the Bakery algorithm, this implies that both threads must have the smallest label (ticket) simultaneously. However, this contradicts the fact that labels are strictly ordered and unique, as atomic operations on labels ensure that no two threads can hold the same label at the same time. Therefore, no two threads can have the smallest label, and mutual exclusion is preserved.

     In the Filter algorithm, assume two threads T_i and T_j are at the highest level simultaneously. This would imply that both threads have passed through all lower levels without one of them being assigned as the victim in those levels, which contradicts the logic of the Filter algorithm, where only one thread can pass to the next level due to the atomic victim assignment. Thus, mutual exclusion holds.

2. **Sequential Consistency:**

   - **Condition:** The program must maintain a globally consistent order of operations as observed by all threads.
   - **Gaurenteed:** Atomic operations in C++ ensure sequential visibility across all threads by enforcing memory fences that prevent such reordering.

3. **Fairness (Bounded Waiting):**

   - **Condition:** Every thread that requests to enter the critical section will eventually be granted access.
   - **Proof by Contradiction:** Assume that a thread T_i is never granted access to the critical section, implying that it waits indefinitely while other threads continue to enter. In the Bakery algorithm, T_i will eventually receive a ticket with a unique and larger value than any currently active ticket. Once all threads with smaller tickets enter and exit the critical section, T_i must become the thread with the smallest ticket, allowing it to enter. If T_i never enters, it would contradict the strictly increasing nature of ticket assignment. Thus, bounded waiting is ensured.

     In the Filter algorithm, assume T_i is stuck at some level indefinitely. This would imply that other threads are constantly overtaking T_i at the same level, which is impossible since only one thread can progress beyond each level due to the atomic victim assignment. Therefore, T_i must eventually reach the highest level and enter the critical section, proving bounded waiting.

4. **Deadlock Freedom:**

   - **Condition:** The system should not reach a state where all threads are stuck indefinitely waiting to enter the critical section.
   - **Proof by Contradiction:** Assume a deadlock occurs, meaning that all threads are stuck in a waiting state, unable to enter the critical section. In the Bakery algorithm, this would imply that each thread is waiting for another thread to proceed, creating a circular wait. However, this is impossible because each thread holds a unique label, and the thread with the smallest label is not waiting for any other thread. Thus, at least one thread will always be able to enter the critical section, contradicting the assumption of deadlock.

     In the Filter algorithm, a deadlock would require all threads to be stuck at various levels, waiting for another thread to proceed. However, due to the atomic victim selection at each level, there is always a thread that can progress to the next level. Therefore, deadlock cannot occur, and the assumption leads to a contradiction.

5. **Livelock Freedom:**

   - **Condition:** The system should not enter a state where threads are continuously trying to enter the critical section but none makes progress.
   - **Proof by Contradiction:** Assume that a livelock occurs, meaning that threads are continuously attempting to enter the critical section but are repeatedly blocked by each other, without making progress.

     In the Bakery algorithm, once a thread holds the smallest ticket, it is guaranteed to enter the critical section, as no other thread with a larger ticket can block it. This ensures that each thread will eventually make progress. If livelock occurred, it would contradict the strict ordering of ticket values, which guarantees forward progress. Therefore, livelock cannot occur.

     In the Filter algorithm, assume that a thread is stuck at a particular level while other threads repeatedly block it. However, since only one thread can progress beyond each level at any given time due to the atomic victim mechanism, it is impossible for a thread to be blocked without making progress. Thus, livelock cannot occur, and the assumption leads to a contradiction.

4 Experiments

4.1 Throughput Analysis with varying threads

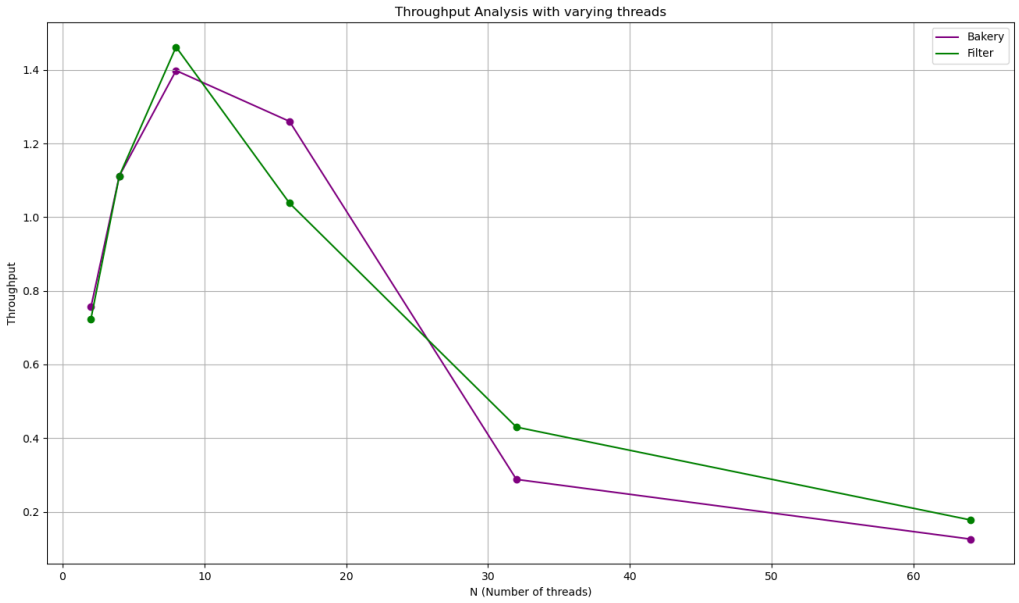

Figure 1: Throughput Analysis with varying threads

Throughput is calculated as $(k \times n)/\text{time\_taken\_for\_execution}$. We can see that the throughput is decreasing as the number of threads increases. This is primarily due to increased contention for shared resources like the critical section. As more threads are introduced, they must spend more time waiting to enter the critical section, which increases the overall execution time. Additionally, the overhead caused by context switching and synchronization further reduces throughput.

We notice a spike in throughput in the middle of the graph, which may be attributed to optimizations in OS scheduling, such as a temporary reduction in context switching overhead or better cache locality. However, on average, throughput decreases as thread count increases due to resource contention, context switching, and synchronization overhead.

4.2 Throughput Analysis with varying k

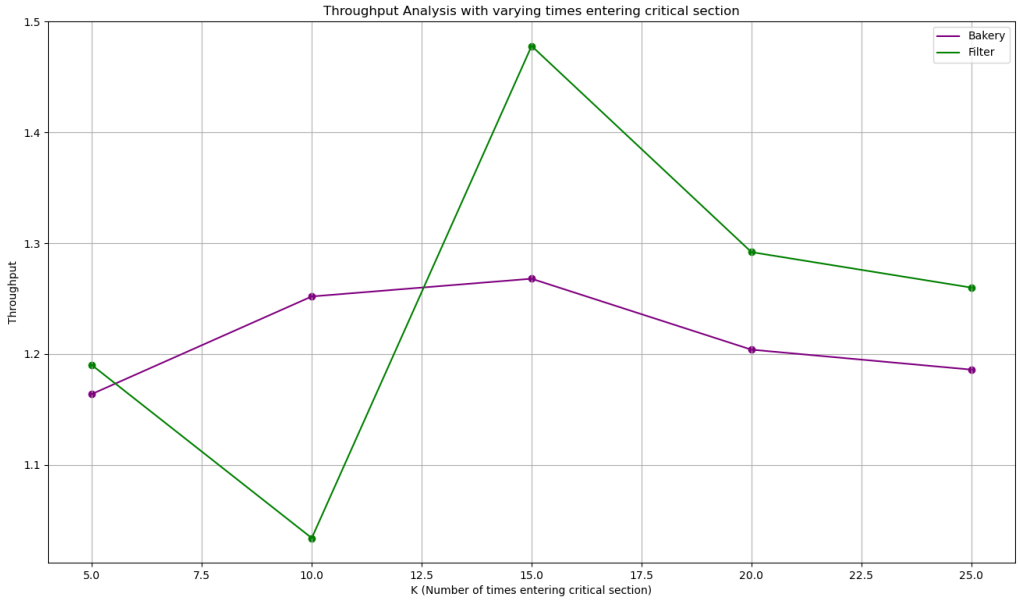

Figure 2: Throughput Analysis with varying k

We observe that throughput generally increases as the value of k increases. The reason for this is that increasing k allows each thread to access the critical section more times, thereby reducing the proportion of time spent on synchronization and overhead (such as lock acquisition and release).

When k is small, threads spend more time on synchronization tasks rather than in the critical section, leading to lower throughput. As k increases, the threads perform more useful work (i.e., executing in the critical section) relative to the overhead, leading to improved throughput.

4.3 Average Entry Time and Worst-Case Entry Time Analysis with varying threads

We can see that as the number of threads increases, both the average and worst-case entry times increase. This is because as more threads contend for the lock, the waiting time for each thread to acquire the lock increases due to the higher level of contention. Also, the overhead of synchronization mechanisms like busy-waiting (in the Filter algorithm) and ticket generation (in the Bakery algorithm) becomes more as the number of threads increases, contributing to the increase in entry times.

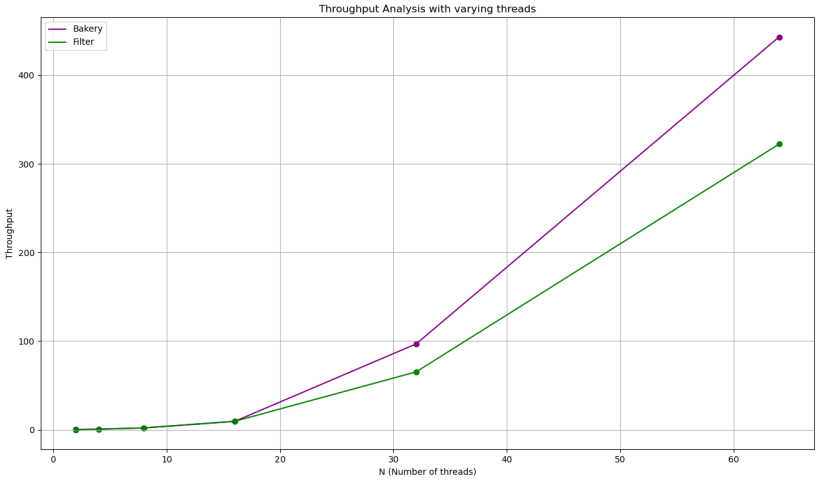

Figure 3: Average Entry Time with varying threads

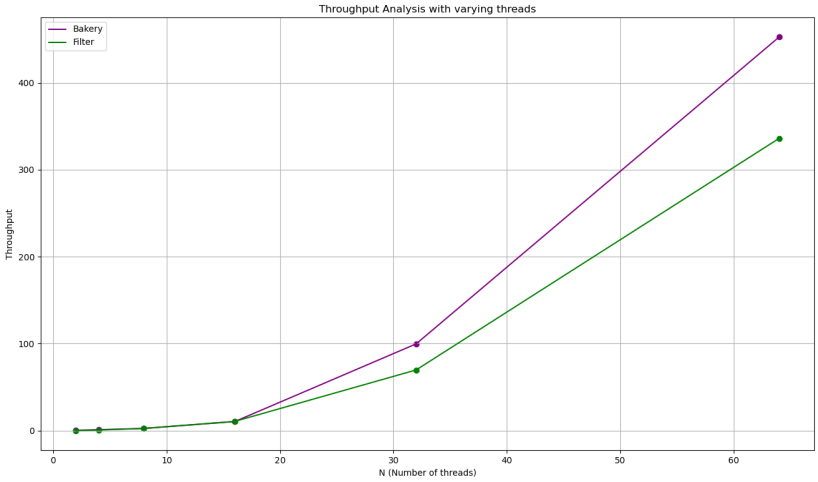

Figure 4: Worst-Case Entry Time with varying threads

4.4 Average Entry Time and Worst-Case Entry Time Analysis with varying k

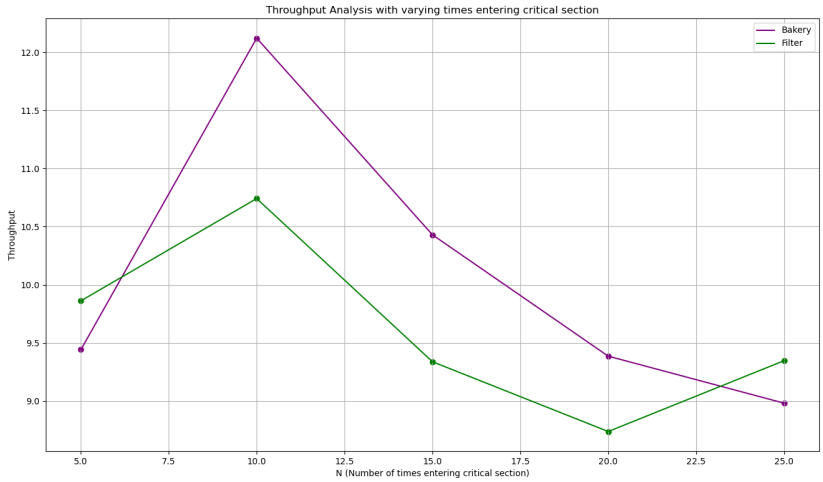

Figure 5: Average Entry Time with varying k

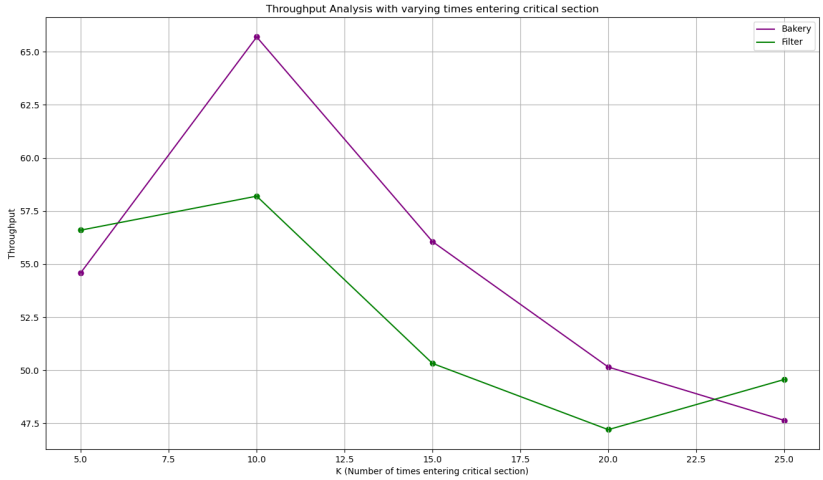

Figure 6: Worst-Case Entry Time with varying k

We observe that as the value of k increases, both the average and worst-case entry times tend to decrease after a split at around $k = 10$. This can be explained by the fact that as k increases, each thread enters the critical section more frequently, reducing the relative waiting time for each individual entry. In other words, the threads are able to enter the critical section more frequently, which reduces the overall wait time between critical section entries.

The decrease after $k = 10$ suggests that beyond this point, the threads are entering the critical section often enough that the overhead associated with waiting for the lock becomes amortized over more frequent critical section entries.

5 Conclusion: Filter vs. Bakery Lock

After analyzing the results from multiple experiments, including throughput, average entry time, and worst-case entry time, we can draw the following conclusions about the performance of the **Filter Lock** and **Bakery Lock** algorithms:

- **Throughput:** In most experiments, the Filter lock demonstrates slightly higher throughput compared to the Bakery lock, especially as the number of threads increases. This is due to the fact that the Filter lock generally has lower synchronization overhead, allowing it to handle a larger number of threads more efficiently. The Bakery lock, on the other hand, tends to have higher contention due to its ticket-based system (which need to atomically assigned), which slows down throughput as the number of threads grows. The results might vary if sequential consistency is ensured by hardware and atomics are not required.
- **Average Entry Time:** The Filter lock often achieves a lower average entry time than the Bakery lock. This is because the Bakery algorithm requires threads to compare and update their labels more frequently, which adds overhead. The Filter algorithm's level-based approach allows threads to progress through the levels faster, especially when the number of threads is high.
- **Worst-Case Entry Time:** In terms of worst-case entry time, the Bakery lock tends to show higher values, particularly when there is a large number of threads or when the value of k increases. The Filter lock, while also experiencing an increase in worst-case entry times as the number of threads grows, generally maintains a lower worst-case time compared to Bakery, making it more predictable in high contention environments.

Final Conclusion: Based on the experimental results, the **Filter lock** consistently performs better than the Bakery lock in terms of throughput, average entry time, and worst-case entry time, particularly when the number of threads increases. The Filter lock is thus the preferred choice for scenarios with a high number of threads or when lower contention and faster entry times are required.