Comparison of Filter and Bakery Locks

Om Dave (CO22BTECH11006)

September 23, 2024

1 Introduction

This report discusses the implementation and analysis of two locking algorithms—Filter Lock and Bakery Lock—using C++. The goal of this assignment is to evaluate the performance of these algorithms based on parameters such as throughput and average waiting time. Both locks are used to ensure mutual exclusion in concurrent programming environments.

2 Program Design and Implementation

2.1 System Specifications

Table 1: System Specifications

Detail	Specification
CPU(s)	12
Model name	AMD Ryzen 5 5600H
Thread(s) per core:	2
Core(s) per socket:	6

2.2 Code Structure and Workflow

2.2.1 Main Program Flow

The main program initializes a lock object and creates a set of threads that access the critical section multiple times. Each thread logs the request and actual entry/exit times for the critical section.

2.2.2 Input and Output

The program reads parameters from a file named `input.txt`, which contains the following values:

```
n k lambda1 lambda2
```

Where:

- `n` is the number of threads
- `k` is the number of times each thread enters the critical section
- `lambda1, lambda2` are time delays simulating task duration in milliseconds

The output logs each thread's entry and exit times, ensuring the correctness of lock usage.

2.3 Timing and Performance Measurement

To measure performance, the system time is recorded at the start and end of execution. Each thread also records the time it requests to enter and actually enters the critical section. The differences between these times help to calculate waiting times and throughput.

2.4 Mutual Exclusion Check

To ensure mutual exclusion, the program tracks the number of threads inside the critical section using an atomic counter, `cs_counter`. Each thread increments the counter when entering the critical section and decrements it upon exit. If more than one thread is detected inside the critical section simultaneously, the counter detects a violation of mutual exclusion, and a warning is logged. This ensures that only one thread can access the critical section at a time, verifying the correct implementation of the locking mechanism.

2.5 Fairness Check

Fairness is evaluated by maintaining two global vectors, `request_order` and `entry_order`, which track the order in which threads request and enter the critical section, respectively. After execution, these vectors are compared to check whether the critical section is accessed in a fair (FIFO) order. A fairness score is also computed by comparing how many threads entered the critical section in the same order as their request. This helps ensure that no thread is starved or unfairly delayed in accessing the critical section.

3 Lock Methods

3.1 Filter Lock

The Filter Lock algorithm ensures mutual exclusion through a multi-level filtering mechanism. Threads must pass through different levels, with each level reducing the number of contenders until only one thread reaches the final level and enters the critical section.

3.1.1 Working

Filter Lock organizes threads into several levels, where each thread competes to enter the next level. When a thread reaches the last level, it gains entry to the critical section. Each thread has an associated level and a victim variable. The victim variable at each level stores which thread is allowed to compete, while the level array keeps track of the highest level a thread is currently competing in. The key idea is that only one thread can reach the highest level because of the filter mechanism.

- Level Assignment: Every thread starts at level 0 and tries to move up to higher levels by checking if any other thread is already in the same or higher level.
- Victim Selection: At each level, a victim is selected to be blocked while other threads proceed.
- Spinning: If a conflict occurs (i.e., if another thread is in the same or higher level and the current thread is the victim of it's level), the thread waits ("spins") until the conflict is resolved.

This process ensures that only one thread passes through all the levels and enters the critical section.

Filter Lock Implementation:

```
class FilterLock {
private:
    vector<atomic<int>> level; // Tracks the level of each thread
    vector<atomic<int>> victim; // Victim array for each level
public:
    FilterLock(int n) {
        level = vector<atomic<int>>(n); // Initial level of each thread is
            ↪ 0
        victim = vector<atomic<int>>(n); // No victim initially
        for(int i = 0; i < n; i++) {
            level[i].store(0);
            victim[i].store(-1);
        }
    }

    void lock(int id) {
        id--; // Convert to 0-based index
        for (int i = 1; i < n; i++) { // Competing through each level
            level[id].store(i, memory_order_relaxed); // Move to level i
            victim[i].store(id, memory_order_relaxed); // Set current
                ↪ thread as victim at this level

            // Spin while there is a conflict
            while (true) {
                bool conflict = false;
                for (int k = 0; k < n; k++) {
                    if (k == id) continue;
                    if (level[k].load(memory_order_acquire) >= i) {
                        conflict = true;
                        break;
                    }
                }
                if (victim[i].load(memory_order_acquire) != id || !
                    ↪ conflict) {
                    break;
                }
            }
        }
    }

    void unlock(int id) {
        id--; // Convert to 0-based index
        level[id].store(0, memory_order_release); // Reset level to 0
            ↪ after exiting critical section
    }
};
```

3.1.2 Implementation

The Filter Lock is implemented by maintaining two arrays: one for the level of each thread and another for tracking the victim at each level. The `lock()` function iteratively moves each thread through the levels, spinning (waiting) if a conflict occurs. The `unlock()` function resets the thread's level to 0, making it ready for the next entry into the critical section. In concurrent programs, multiple threads can attempt to read or modify shared variables simultaneously. This can lead to race conditions where the program behavior is unpredictable. The `atomic` type ensures that operations like reading and writing to shared variables (e.g., `level` and `victim`) are performed atomically, meaning without interruption. This guarantees thread safety by preventing two threads from simultaneously modifying the same value, which is critical for maintaining the correct functioning of the Filter Lock.

3.2 Bakery Lock

The Bakery Lock ensures mutual exclusion by assigning a unique token number to each thread, mimicking a bakery's system where the thread with the smallest token number gets access to the critical section.

3.2.1 Working

The Bakery Lock works by assigning a number (token) to each thread when it wants to enter the critical section. The thread with the smallest token number enters first. The algorithm ensures fairness as no thread can overtake another without waiting for its turn.

- Token Assignment: Each thread is given a unique token, which is assigned by incrementing the highest token value currently in use.
- Waiting Mechanism: Once a thread has a token, it waits for all other threads with smaller tokens to finish before entering the critical section.
- Fairness: If two threads have the same token number (unlikely but possible due to race conditions), the thread with the smaller ID goes first.

This process ensures that all threads eventually enter the critical section in the order they acquired their tokens, thus maintaining fairness and preventing starvation.

Bakery Lock Implementation:

```
class BakeryLock {
private:
    vector<atomic<bool>> flag; // Indicates if a thread wants to enter
        ↪ critical section
    vector<atomic<int>> ticket; // Token assigned to each thread
public:
    BakeryLock(int n) {
        flag = vector<atomic<bool>>(n); // No thread wants to enter
            ↪ initially
        ticket = vector<atomic<int>>(n); // All tickets are initialized to
            ↪  0
        for (int i = 0; i < n; i++) {
            flag[i].store(false);
            ticket[i].store(0);
        }
    }

    void lock(int id) {
        id--; // Convert to 0-based index
        flag[id].store(true, memory_order_relaxed); // Indicate thread
            ↪ wants to enter

        // Assign new ticket
        int max_ticket = 0;
        for (int i = 0; i < n; i++) {
            int t = ticket[i].load(memory_order_acquire);
            if (t > max_ticket) {
                max_ticket = t;
            }
        }
        ticket[id].store(max_ticket + 1, memory_order_release); //
            ↪ Increment ticket

        // Spin while other threads have lower ticket numbers
        for (int k = 0; k < n; k++) {
            if (k == id) continue;
            while (flag[k].load(memory_order_acquire)) {
                int tk = ticket[k].load(memory_order_acquire);
                int ti = ticket[id].load(memory_order_acquire);
                if (tk < ti || (tk == ti && k < id)) {
                    // Busy wait
                } else {
                    break;
                }
            }
        }
    }
    void unlock(int id) {
        id--; // Convert to 0-based index
        flag[id].store(false, memory_order_release); // Reset flag
           ↪ indicating thread no longer wants to enter
    }
};
```

3.2.2 Implementation

The Bakery Lock is implemented using two arrays: `flag`, which indicates whether a thread wants to enter the critical section, and `ticket`, which stores the current ticket for each thread. The `lock()` function assigns a unique ticket to the thread and makes it wait for its turn to enter the critical section, while the `unlock()` function resets the flag to indicate the thread has exited. Both the `flag` and `ticket` arrays are accessed by multiple threads simultaneously. The `atomic` type ensures that these shared variables are updated in a thread-safe manner. For example, atomicity is necessary when assigning a token or checking whether a thread wants to enter the critical section to avoid race conditions, ensuring that no two threads can interfere with each other's updates.

4 Performance Analysis

4.1 Experiment 1: Throughput with Varying Threads (n)

4.1.1 Methodology

In this experiment, the throughput of both the Filter Lock and Bakery Lock was measured as the number of threads (n) increased. The value of n, the number of threads, was varied from 2 to 64 in powers of 2 (i.e., $n = 2^1, 2^2, \ldots, 2^6$). The number of times each thread entered the critical section (k) was fixed at 15.

Throughput is defined as the total number of critical section entries completed per unit of time. The system measures the time taken for all threads to complete their entries into the critical section and calculates throughput using the formula:

$$\text{Throughput} = \frac{k \times n}{\text{Total Time Taken}}$$

where:

- $k = 15$ (fixed number of critical section entries per thread),
- n is the number of threads,
- Total Time Taken is the difference between the start and end times of the test.

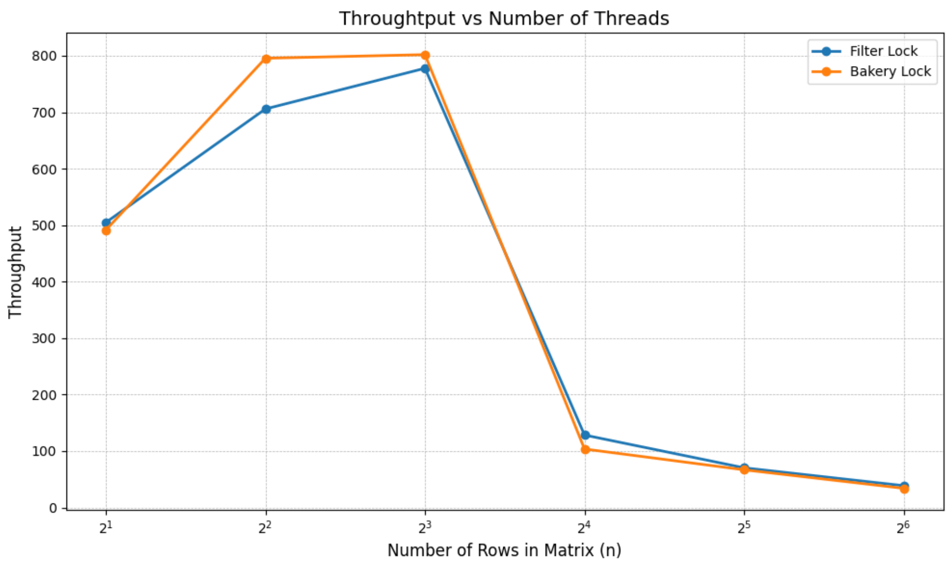

Figure 1: Throughput vs Number of Threads (n)

4.1.2 Observations and Reasoning

The results of the experiment, depicted in Figure 1, reveal several key insights:

- **Initial Increase in Throughput (n = 2 to 8):** Both Filter Lock and Bakery Lock show an increase in throughput as the number of threads increases from 2 to 16. This is expected since adding more threads increases the number of tasks executed per unit time. However, Bakery Lock generally performs slightly better than Filter Lock due to its simpler token-based mechanism, which reduces the overhead compared to the multi-level filtering approach in Filter Lock.
- **Peak Throughput (n = 8):** Both locking mechanisms reach their peak throughput around $n = 8$. This is likely due to the optimal balance between the number of threads and the system's ability to handle concurrent threads. Beyond this point, the system may start experiencing contention between threads.
- **Degradation in Throughput (n = 16 to n = 64):** As the number of threads increases beyond 16, throughput for both locks drops significantly. This can be attributed to the increased contention among threads as they attempt to enter the critical section. More threads competing for the lock lead to higher waiting times, which reduces overall throughput. The impact is more pronounced for larger thread counts. Also the hardware limitations kick in as there a limited number of threads at hardware level, but creating more threads can lead to significant overhead of thread management.

4.2 Experiment 2: Throughput with Varying Critical Section Entries (k)

4.2.1 Methodology

In this experiment, the throughput was measured by varying the number of times each thread enters the critical section (k), while keeping the number of threads (n) constant at 16. The value of k, the number of critical section entries per thread, was varied from 5 to 25.

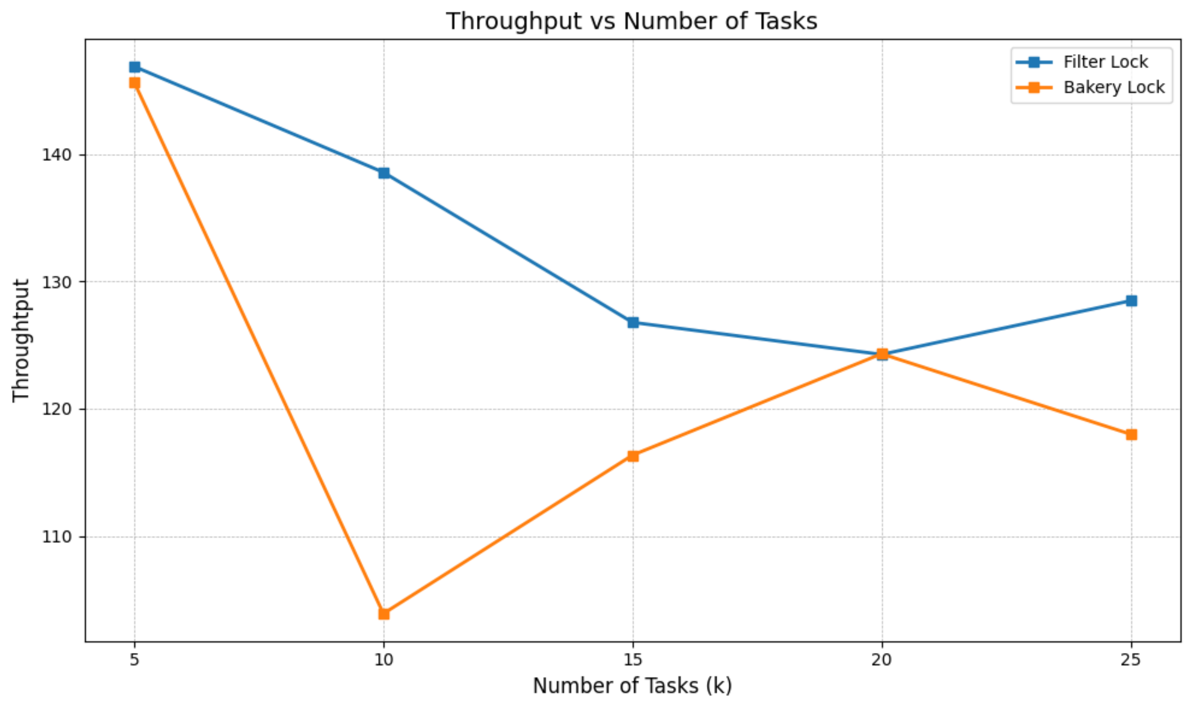

Figure 2: Throughput vs Number of Tasks (k) for Filter Lock and Bakery Lock.

4.2.2 Observations and Reasoning

The experiment results, depicted in Figure 2, reveal the following key insights:

- **Filter Lock vs Bakery Lock Performance**: For smaller values of k, Filter Lock demonstrates superior performance in terms of throughput compared to Bakery Lock. The token-based mechanism of the Bakery Lock introduces more overhead when the number of tasks is small, leading to lower throughput.

- **Decreased Throughput with Increased Tasks (k)**: For both locks, throughput initially decreases as the number of tasks (k) increases. This is because the time spent by threads in the critical section increases, reducing the overall throughput. The overhead of managing locks in each thread increases with more tasks.

- **Fluctuations in Bakery Lock Performance**: The Bakery Lock experiences more significant fluctuations in throughput compared to the Filter Lock. The token assignment mechanism of the Bakery Lock may introduce delays for threads with higher token numbers, which leads to greater variability in throughput.

4.3 Experiment 3: Average and Worst-Case Entry Time with Varying Threads (n)

4.3.1 Methodology

In this experiment, the entry time was measured by varying the number of threads (n), while keeping the number of critical section entries per thread (k) fixed at 10. The average and worst-case entry times were recorded for each lock configuration.

The entry time is defined as the time elapsed between a thread requesting access to the critical section and the actual entry into it. Two metrics were measured:

- **Average Entry Time:** The average time it takes for a thread to enter the critical section across all threads.
- **Worst-Case Entry Time:** The maximum time observed for any thread to enter the critical section.

The number of threads, n, was varied from 2 to 64 in powers of 2 (i.e., $n = 2^1, 2^2, \ldots, 2^6$). The results were plotted as entry time versus the number of threads, as shown in Figure 3.

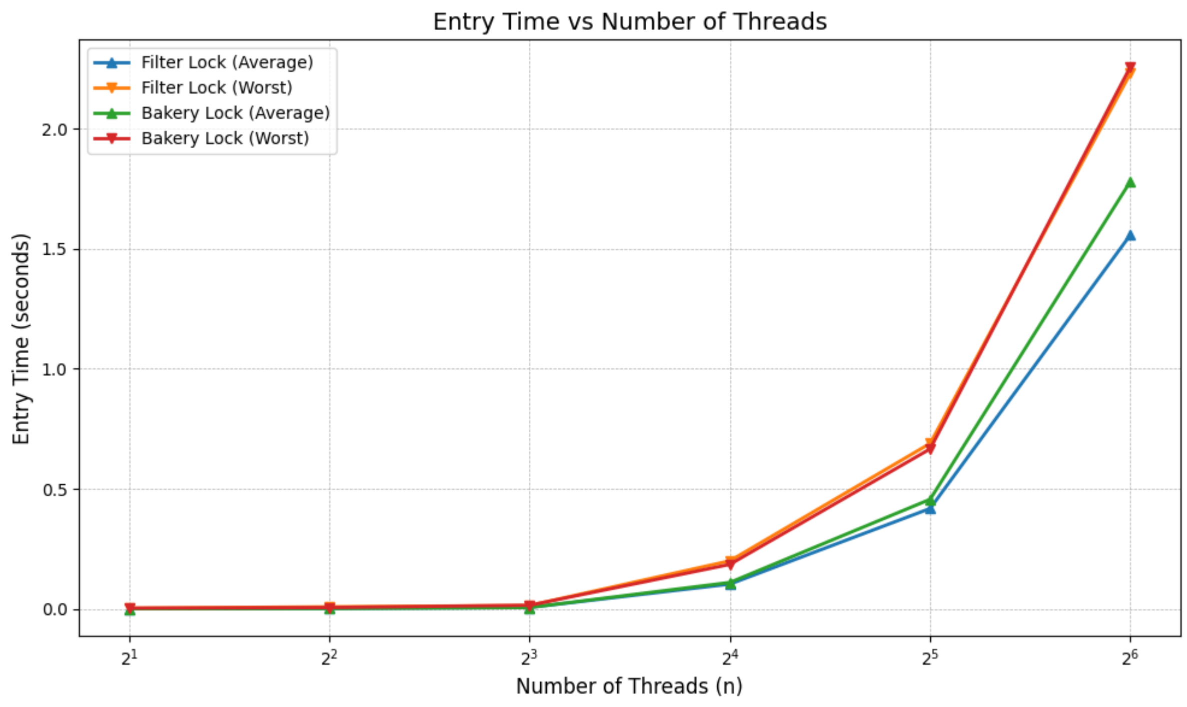

Figure 3: Entry Time vs Number of Threads for Filter Lock and Bakery Lock.

4.3.2 Observations and Reasoning

The experiment results, depicted in Figure 3, provide the following insights:

- **Low Entry Times with Fewer Threads** $(n \leq 16)$ **:** For both the filter lock and the bakery lock, the average and worst-case entry times remain low for smaller thread counts (n less than 16). The lower contention between threads results in faster entry into the critical section. The Filter Lock has slightly lower entry times in these configurations due to its multi-level mechanism that filters threads more efficiently with fewer competitors.
- **Increase in Entry Times with Higher Threads** $(n > 16)$ **:** As the number of threads increases beyond 16, both the average and worst-case entry times increase significantly. This is because, as more threads compete for access to the critical section, the likelihood of contention grows, leading to longer wait times before a thread can enter the critical section.

- **Filter Lock vs Bakery Lock:** The worst-case entry time for the Filter Lock is notably higher than the average entry time at higher thread counts. This can be explained by the filter's multiple levels, which introduce delays for the thread that gets caught at a higher level. In contrast, the Bakery Lock's worst-case time remains closer to the average because all threads are served based on their token number, making the performance more predictable but still prone to delays at higher thread counts.

- **Drastic Increase at n = 64:** The entry times for both locks increase drastically at $n = 64$. This suggests that, at very high thread counts, the contention becomes too intense for either lock to handle efficiently, leading to severe performance degradation.

- **Hardware Limitation (12 Threads):** As the system used for testing has a maximum of 12 physical threads, the performance observed for $n > 12$ involves logical cores. This may contribute to the rapid increase in entry times beyond $n = 16$, as the system struggles to manage higher levels of contention using logical cores.

4.4 Experiment 4: Average and Worst-Case Entry Time with Varying Critical Section Entries (k)

4.4.1 Methodology

In this experiment, the entry time was measured by varying the number of critical section entries per thread (k), while keeping the number of threads (n) fixed at 16. Both the average and worst-case entry times were recorded for each configuration of the locks. The number of critical section entries per thread, k, was varied from 5 to 25. The results were plotted as entry time versus the number of tasks k, as shown in Figure 4.

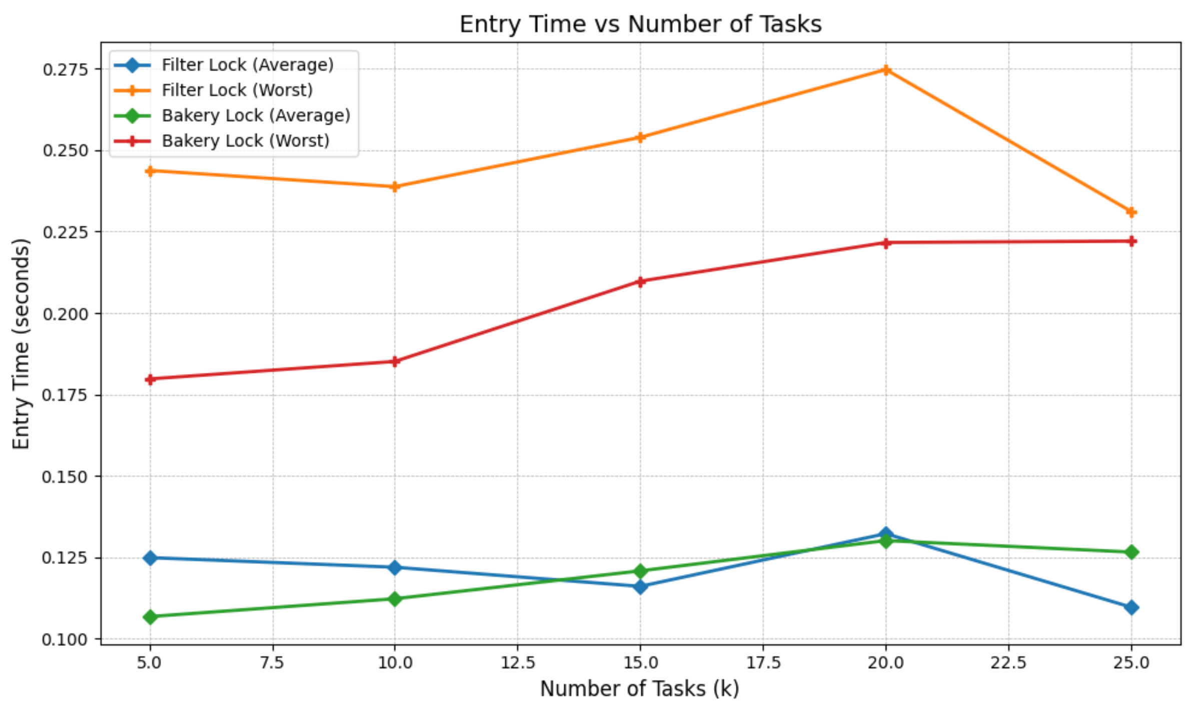

Figure 4: Entry Time vs Number of Tasks (k) for Filter Lock and Bakery Lock.

4.4.2 Observations and Reasoning

The experiment results, depicted in Figure 4, show the following observations:

- **Filter Lock:** The average entry time for the Filter Lock remains consistent with minor fluctuations as k increases. The worst-case entry time for the Filter Lock also follows a similar trend, with a slight increase as the number of tasks increases but without any dramatic spikes.

- **Bakery Lock:** The Bakery Lock shows a similar trend, with consistent average and worst-case entry times, though the fluctuations are more noticeable compared to the Filter Lock. The worst-case entry time for the Bakery Lock tends to increase slightly as k rises but remains manageable across all values of k.

- **Consistency Across Task Range:** Both locks maintain consistent entry times as the number of critical section entries increases, with no significant degradation in performance. There are slight fluctuations in both average and worst-case entry times, but these are expected due to the increasing contention for access to the critical section as k rises.

- **Comparison of Worst-Case Times**: The Filter Lock consistently exhibits higher worst-case entry times compared to the Bakery Lock, as shown in the graph. The reason for this is that, while the filtering mechanism is designed to reduce contention, it can also introduce additional delays in the worst case due to possibility of starvation of some thread. In contrast, the Bakery Lock's token-based mechanism, although simpler, results in more consistent and lower worst-case times because threads are processed strictly in the order of their tokens, avoiding the potential bottlenecks seen in the Filter Lock.

5 Conclusion

The Bakery Lock demonstrated more consistent performance in terms of both average and worst-case entry times, particularly as the number of threads and tasks increased. Its token-based mechanism ensured fairness and lower worst-case delays. On the other hand, the Filter Lock showed better performance with smaller thread counts and tasks but exhibited higher worst-case entry times, especially under heavier contention, due to its multi-level filtering system.

Overall, the choice of lock depends on the specific use case: Bakery Lock is preferable in environments where fairness and predictability are critical, while Filter Lock may be more suitable in scenarios with lower thread counts and contention levels.