Homework-2

Sriamoghavarsha Beerangi Srinivasa

September 29, 2024

Problem 1

Solution:

Informal Proof of Mutual Exclusion

a) Initialization

- Initially, both $a = 1$, $b = 1$, and $k = 0$.

b) Process P0

- $P0$ sets $a = 0$ to indicate its intention to enter the critical section.
- $P0$ checks k. If $k == 1$, it waits in the inner loop until $b == 1$, which means $P1$ is not trying to enter the critical section. Once $b == 1$, $P0$ sets $k = 0$ and proceeds to enter the critical section.

c) Process P1

- $P1$ sets $b = 0$ to indicate its intention to enter the critical section.
- $P1$ checks k. If $k == 0$, it waits in the inner loop until $a == 1$, which means $P0$ is not trying to enter the critical section. Once $a == 1$, $P1$ sets $k = 1$ and proceeds to enter the critical section.

Verification of Mutual Exclusion

a) P0 enters the critical section first

- $P0$ sets $a = 0$ and proceeds to check k.
- Since k is initially 0, $P0$ will bypass the inner loop and enter the critical section.
- As $P0$ is in the critical section, it will not reset a to 1 until it exits the critical section.
- During this time, if $P1$ attempts to enter the critical section, it will set $b = 0$ and check k. But k is 0, so $P1$ will enter its inner loop and wait until $a == 1$.
- Since $a = 0$ while $P0$ is in the critical section, $P1$ will be blocked from entering the critical section.

b) P1 enters the critical section first

- $P1$ sets $b = 0$ and proceeds to check k.
- Since k is 0, $P1$ will bypass the inner loop and enter the critical section.
- As $P1$ is in the critical section, it will not reset b to 1 until it exits the critical section.
- During this time, if $P0$ attempts to enter the critical section, it will set $a = 0$ and check k. But k is now 1, so $P0$ will enter its inner loop and wait until $b == 1$.
- Since $b = 0$ while $P1$ is in the critical section, $P0$ will be blocked from entering the critical section.

This informal proof shows that the algorithm prevents both processes from being in the critical section simultaneously, thereby satisfying the mutual exclusion requirement.

Problem 2

Solution:

a) To determine if the lock is starvation-free, we need to check whether every process that tries to acquire the lock will eventually succeed in doing so.

The lock mechanism:

- **nextTurn** is a global counter that assigns a unique number to each process that wants to acquire the lock.
- **nowServing** tracks the number of the process currently holding the lock.
- When a process calls `Lock()`, it get a unique number (by atomically incrementing **nextTurn**) and waits until its unique number matches `nowServing`.
- The `Unlock()` function simply increments **nowServing** to allow the next waiting process to acquire the lock.

Starvation occurs if a process can be indefinitely delayed from acquiring the lock, even though it continuously requests it.

In this algorithm, each process gets a unique number, and processes are served in the order of their unique numbers. This ensures **FIFO (First-In-First-Out) ordering**.

Since every process is guaranteed to get a unique number and the lock is granted to the process with the next number in sequence, every process will eventually acquire the lock in the order it requested it. Therefore, **this lock is starvation-free**.

b) Doorway and Waiting Sections:

- Doorway Section - is the part of the code where a process commits to acquiring the lock. This is usually a small and fast section of code that ensures fairness. In this algorithm, the doorway section is:

  ```
  myTurn = AtomicFetchAndIncrement(nextTurn);
  ```

  This line gives the process a unique ticket number and ensures it will be served in the correct order. The use of `AtomicFetchAndIncrement` is crucial to prevent race conditions and ensure that each process gets a unique number.
- Waiting Section - is part of the code where a process waits for its turn to enter the critical section. In this algorithm, the waiting section is:

  ```
  while (myTurn != nowServing) {};
  ```

  Here, the process spins in a loop, waiting for its number (`myTurn`) to match the `nowServing` value, indicating that it is now its turn to enter the critical section.

The doorway and waiting sections in the lock algorithm are segregated into algorithms that ensure fairness guarantees because the quick and atomic doorway section allows threads to register for the lock without heavy competition, ensuring fairness by preventing delays once a thread gets a ticket. and the waiting section ensures threads wait in order.

Problem 3

Solution:

a) Mutual exclusion ensures that no two threads can be in the critical section simultaneously. When a thread i sets $x = i$, it signals its intention to enter the critical section. The critical section is entered only if:

   i) $y = 0$ (indicating no other thread is attempting to enter the critical section).

   ii) After setting $y = 1$, the thread checks if x has changed to the other thread's ID. If it hasn't, it proceeds to the critical section.

   Let us consider two threads, $T1$ and $T2$. If both attempt to enter the critical section simultaneously:

   - $T1$ sets $x = 1$, and $T2$ sets $x = 2$.
   - Both threads check y. If $y = 0$, one of them (say $T1$) will proceed to set $y = 1$.
   - $T2$ will then observe $y = 1$ and will be forced to wait.
   - The condition `if (x != i)` ensures that only one thread can proceed to the critical section based on the value of x.

   Thus, **mutual exclusion** is satisfied because only one thread can be in the critical section at a time.

   Deadlock-freedom means that if some threads are trying to enter the critical section, one will eventually succeed. In this case, the algorithm ensures that a thread that fails the `x != i` check will reset $y = 0$ and wait for the other thread to finish (i.e., wait for $x = 0$). Once the other thread exits the critical section and resets $x = 0$, the waiting thread can proceed.

   Thus, the algorithm is **deadlock-free** because each thread will eventually enter the critical section, provided the other thread exits.

b) Starvation-freedom means that any thread that wants to enter the critical section will eventually be able to do so. In this algorithm, if one thread is delayed, it will eventually get its turn once the other thread completes the critical section. Since the threads alternate their attempts to enter the critical section based on the values of x and y, no thread will be indefinitely postponed.

   Thus, the algorithm **guarantees starvation-freedom** for two threads.

c) To prove deadlock-freedom with three threads

   - The same logic applies as with two threads, the structure of the algorithm still ensures that only one thread can hold $y = 1$ and enter the critical section at a time.
   - If one thread fails the `x != i` check, it waits for the others to finish by waiting for $x = 0$, after which it retries.

   Thus, the algorithm is **deadlock-free** for three threads as well because it ensures that one thread can make progress at a time, while others wait their turn.

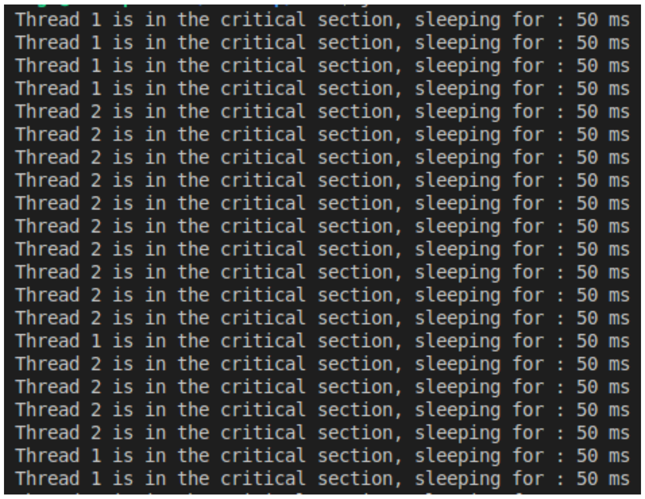
d) To prove mutual exclusion for three threads:

   - For a thread to enter the critical section, it must set $y = 1$ and check if $x == i$. If $x \neq i$, it waits for $x = 0$.

- While one thread is in the critical section, $y = 1$, ensuring that no other thread can enter until the current thread exits and resets y and x.

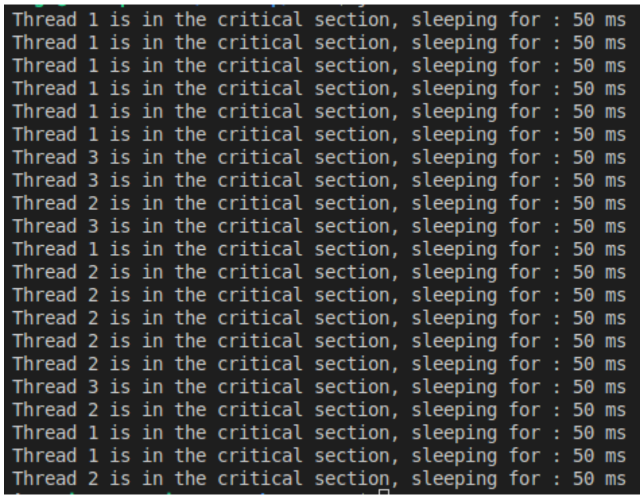
Thus, the algorithm **satisfies mutual exclusion** even for three threads, ensuring that no two threads can be in the critical section at the same time.

Illustration of algorithm with two and three threads



```
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
```

Figure 1: Execution with 2 threads.



```
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 3 is in the critical section, sleeping for : 50 ms
Thread 3 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 3 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 3 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 1 is in the critical section, sleeping for : 50 ms
Thread 2 is in the critical section, sleeping for : 50 ms
```

Figure 2: Execution with 3 threads.

Problem 4

Solution:

In the original Bakery algorithm, the lexicographical comparison $(label[k], k) \prec (label[i], i)$ ensures that if two threads have the same *label*, the tie is broken by comparing their thread IDs (k). This guarantees that only one thread can enter the critical section at a time.

a) Less-Than Comparison on Labels Only:

   By changing the comparison to $label[k] < label[i]$, we are no longer considering the thread IDs in case of a tie. This could lead to a scenario where two threads with the same label both believe they can enter the critical section.

   This modification could lead to **mutual exclusion violations**. Two threads with the same label might both decide to enter the critical section simultaneously, as there is no longer a mechanism to enforce a strict order between them.

b) Less-Than or Equal-To Comparison on Labels:

   Changing the comparison to $label[k] \leq label[i]$ means that if two threads have the same label, they will each see the other's label as less than or equal to their own and will wait for the other to proceed.

   This modification could lead to a **deadlock** situation. When two threads have the same label, they will both wait for each other, causing neither to enter the critical section, as there is no longer a mechanism to select betweem two threads when they share the same label.

Problem 5

Solution:

Figure 2 Algorithm

- Mutual Exclusion - The main issue with mutual exclusion in this algorithm is that each thread sets its respective `flag[]` to `true` after the `while` loop has already checked the other thread's flag. This means both threads could see the other's flag as `false` initially, which would allow both to proceed into the critical section simultaneously, violating mutual exclusion.

  Hence, this algorithm does **not** provide mutual exclusion.

- Deadlock Freedom - Deadlock would occur if both threads were stuck waiting indefinitely, but in this case, if one thread sets its `flag` to `true`, the other will eventually notice and exit the `while` loop. Thus, both threads will always eventually make progress and exit the `while` loop. The random delays and setting `flag[]` to `false` after the `while` loop ensure that the threads won't block each other indefinitely.

  Hence, this algorithm does **provide** deadlock freedom.

- Starvation Freedom - Starvation could occur if one thread consistently finishes its random delay faster and thus repeatedly re-enters the critical section before the other thread gets a chance. However, this depends on the nature of the random delays and the timing.

  Hence, this algorithm does **not guarantee** starvation freedom, as one thread could theoretically be starved if it consistently has a longer delay.

Figure 3 Algorithm

- Mutual Exclusion - The `turn` variable ensures that only one thread can enter the critical section at a time. Specifically, `P0` waits while `turn` is `1`, and `P1` waits while `turn` is `0`. This mechanism guarantees that the threads take turns accessing the critical section, ensuring that they do not enter simultaneously.

  Hence, this algorithm **provides** mutual exclusion.

- Deadlock Freedom - Deadlock Freedom is ensured because each thread sets the `turn` variable to allow the other thread to proceed after it exits the critical section. This prevents both threads from waiting indefinitely. Once a thread completes its critical section, it allows the other thread to enter, ensuring that both can make progress.

  Hence, this algorithm **provides** deadlock freedom.

- Starvation Freedom - Starvation is guaranteed because the `turn` variable alternates between the two threads. Once a thread finishes its critical section, it sets the `turn` variable to allow the other thread to enter next. This ensures that each thread eventually gets its turn to enter the critical section, preventing any thread from being starved.

  Hence, this algorithm **provides** starvation freedom.

Problem 6

Solution:

a) **Peterson's lock** can be modified to be a reentrant lock by checking the value of its own flag variable before setting it to `true`. If the flag is already `true`, the thread already owns the lock, so it should return immediately.

b) **Bakery lock** can also be modified to be a reentrant lock in a similar manner by checking the value of its own flag variable. If the flag is already set, the thread already owns the lock and should return without further acquiring.

c) **Filter lock** cannot be modified to be a reentrant lock just by checking the existing flags or registers. The hierarchical structure of the Filter lock requires more than simply checking flag variables for reentrancy.

d) **The given lock** can be modified to be a reentrant lock similarly to Bakery or Peterson's lock by checking the flag before acquiring the lock.

Part II Problem 1

Solution:

a) I integrated the timestamp functionality directly into the `lock()` method using two key functions: `chooseTicket()` and `waitForTurn()`. Each thread selects a unique ”ticket” by scanning the current maximum ticket among all threads and then choosing the next available one. This ensures proper ordering for entry into the critical section.

The `waitForTurn()` function ensures lexicographical ordering based on ticket values. A thread can only proceed if it has the smallest ticket, or if it shares a ticket value but has a lower ID than any other competing thread. Once done, the thread resets its ticket value to 0 to release the lock.

c) Performance tests using both the Bakery Lock and Filter Lock across different thread counts (2, 4, 8, 16, 20, and 32).

Thread Count	Avg Bakery Lock (ms)	Avg Filter Lock (ms)
2	4.8	4.0
4	4.6	10.2
8	7.8	21.4
16	10.0	46.2
20	23.4	56.4
32	64.8	154.0

Table 1: Average Runtime per Thread for Bakery Lock and Filter Lock

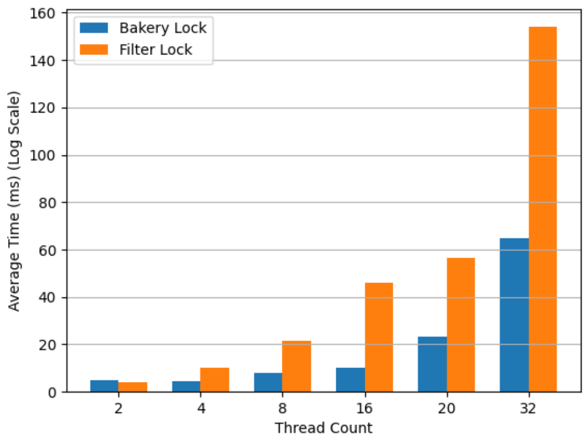


Figure 3: Average Time vs Thread Count for Bakery and Filter Locks (Bar Graph)

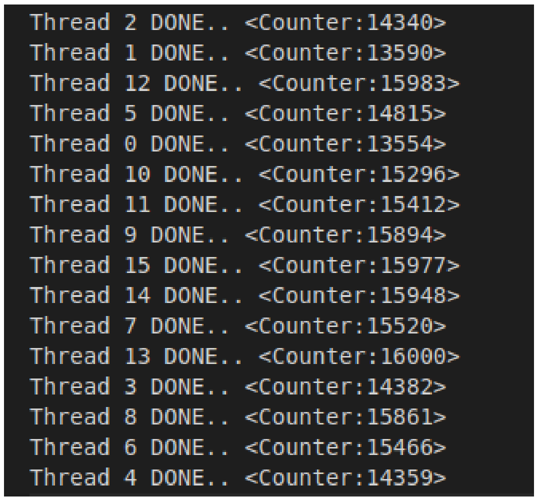
d) The Bakery Lock appears to be the more efficient algorithm because of its fair, ticket-based system, which handles contention better and incurs less overhead. The Filter Lock may be slower due to the overhead of maintaining the multi-level filtering structure, which adds complexity as more threads compete for the lock.

Part II Problem 2

Solution:

a)
- **Mutual Exclusion:** Each thread acquires locks from the leaf to the root, and each 2-thread Peterson lock allows only one thread to pass at a time. The final counter value of 16,000 confirms that no two threads were in the critical section simultaneously.
- **Deadlock Freedom:** Since each thread holds one lock at a time and Peterson's lock is deadlock-free, the tree structure ensures deadlock freedom. The test results show that all threads completed without being stuck.
- **Starvation Freedom:** Peterson's lock guarantees no indefinite blocking, and each thread progresses through the tree. The test results confirm that all threads contributed to the final counter, showing no starvation occurred.
- **Fairness:** Peterson's lock treats threads fairly. The similar average runtimes (18 ms to 30 ms) indicate that no thread was significantly delayed.

The tree-based Peterson lock satisfies mutual exclusion, deadlock freedom, starvation freedom, and fairness, as demonstrated by the final counter value of 16,000 and similar average runtimes across threads.



```
Thread 2 DONE.. <Counter:14340>
Thread 1 DONE.. <Counter:13590>
Thread 12 DONE.. <Counter:15983>
Thread 5 DONE.. <Counter:14815>
Thread 0 DONE.. <Counter:13554>
Thread 10 DONE.. <Counter:15296>
Thread 11 DONE.. <Counter:15412>
Thread 9 DONE.. <Counter:15894>
Thread 15 DONE.. <Counter:15977>
Thread 14 DONE.. <Counter:15948>
Thread 7 DONE.. <Counter:15520>
Thread 13 DONE.. <Counter:16000>
Thread 3 DONE.. <Counter:14382>
Thread 8 DONE.. <Counter:15861>
Thread 6 DONE.. <Counter:15466>
Thread 4 DONE.. <Counter:14359>
```

Figure 4: Execution SharedCounter using TreePeterson Lock for 16 Threads

b) The worst-case number of acquisitions and releases a thread might face is proportional to the number of competing threads at each level. For N threads, a thread could potentially be delayed up to $N/2$ times at the first level, $N/4$ at the second level, and so on.

Thus, the worst-case upper bound considering contention can be approximated by:

$$2 \cdot \sum_{i=1}^{\log_2(N)} \frac{N}{2^i}$$

This accounts for both acquiring and releasing locks at each level, with possible contention from other threads.

c) For the benchmarks, my local machine has a core count of 20, and I used the Rlogin cluster (hostname: pawpaw), which supports up to 64 threads. I ran the tests for 128,000 iterations on both machines.

	12AM	6AM	12PM	6PM	9PM	Local Machine
Iteration 1	101	113	98	99	111	12
Iteration 2	87	88	90	86	87	7
Iteration 3	74	76	75	75	69	8
Iteration 4	74	79	82	74	80	7
Iteration 5	80	81	83	80	80	7
Average	83.2	87.4	85.6	82.8	85.4	8.2

Table 2: Runtime data for 64 threads at different times of the day, including local machine.

	12AM	6AM	12PM	6PM	9PM	Local Machine
Iteration 1	111	118	113	116	119	27
Iteration 2	131	111	116	132	111	13
Iteration 3	115	108	100	115	108	14
Iteration 4	133	113	117	134	113	30
Iteration 5	110	107	96	111	101	11
Average	120	111.4	108.4	121.6	110.4	19

Table 3: Runtime data for 32 threads at different times of the day, including local machine.

	12AM	6AM	12PM	6PM	9PM	Local Machine
Iteration 1	113	111	102	123	111	28
Iteration 2	105	107	110	111	107	18
Iteration 3	101	114	106	107	108	21
Iteration 4	109	99	94	100	115	21
Iteration 5	96	101	104	103	93	21
Average	104.8	106.4	103.2	108.8	106.8	21.8

Table 4: Runtime data for 16 threads at different times of the day, including local machine.

	12AM	6AM	12PM	6PM	9PM	Local Machine
Iteration 1	100	103	111	74	108	30
Iteration 2	103	104	119	104	100	26
Iteration 3	112	110	95	109	100	20
Iteration 4	92	91	99	89	92	17
Iteration 5	113	103	102	103	101	20
Average	104	102.2	105.2	95.8	100.2	22.6

Table 5: Runtime data for 8 threads at different times of the day, including local machine.

	12AM	6AM	12PM	6PM	9PM	Local Machine
Iteration 1	52	106	55	57	104	25
Iteration 2	55	94	84	89	102	22
Iteration 3	75	70	64	78	95	24
Iteration 4	91	77	68	69	98	16
Iteration 5	97	76	93	80	114	23
Average	74	84.6	72.8	74.6	102.6	22

Table 6: Runtime data for 4 threads at different times of the day, including local machine.

	12AM	6AM	12PM	6PM	9PM	Local Machine
Iteration 1	39	52	41	32	37	20
Iteration 2	24	26	23	19	25	11
Iteration 3	25	25	75	27	26	13
Iteration 4	27	24	19	28	25	16
Iteration 5	36	25	32	26	25	15
Average	30.2	30.4	38	26.4	27.6	15

Table 7: Runtime data for 2 threads at different times of the day, including local machine.

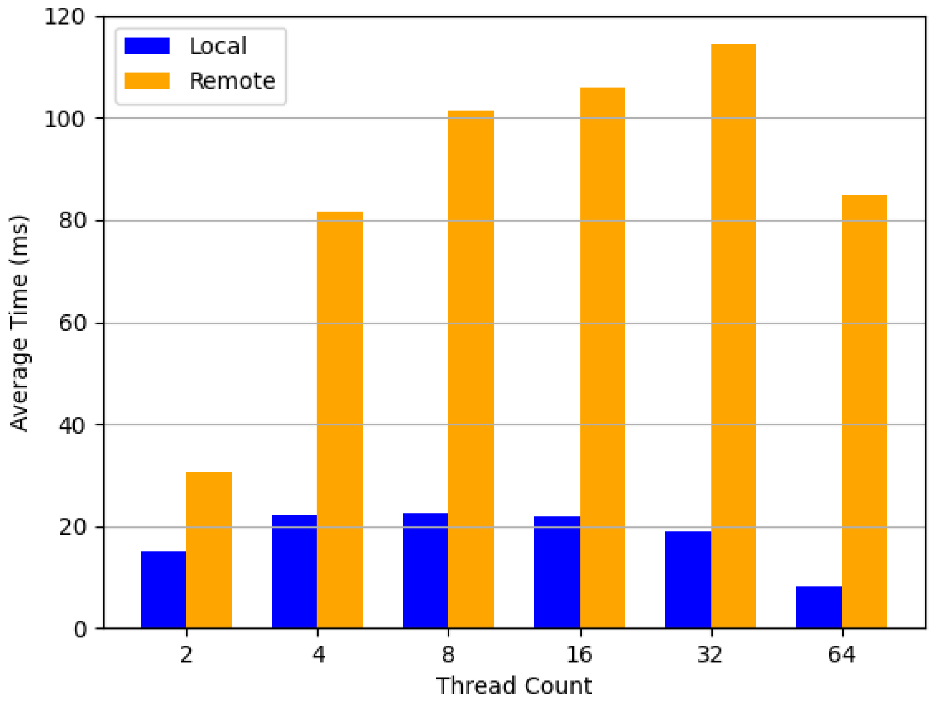
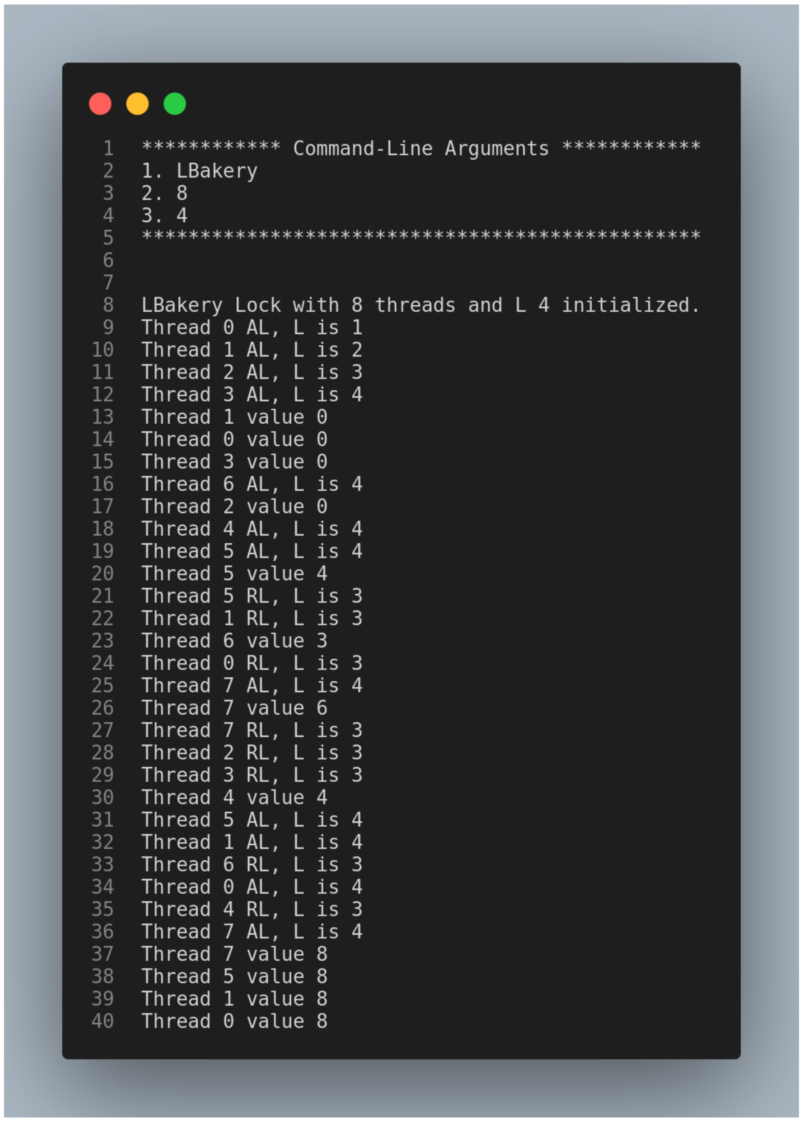


Figure 5: Average runtime vs thread count for local and remote server execution

Part II Problem 3

Solution:



```
************ Command-Line Arguments ************
1. LBakery
2. 8
3. 4
************************************************


LBakery Lock with 8 threads and L 4 initialized.
Thread 0 AL, L is 1
Thread 1 AL, L is 2
Thread 2 AL, L is 3
Thread 3 AL, L is 4
Thread 1 value 0
Thread 0 value 0
Thread 3 value 0
Thread 6 AL, L is 4
Thread 2 value 0
Thread 4 AL, L is 4
Thread 5 AL, L is 4
Thread 5 value 4
Thread 5 RL, L is 3
Thread 1 RL, L is 3
Thread 6 value 3
Thread 0 RL, L is 3
Thread 7 AL, L is 4
Thread 7 value 6
Thread 7 RL, L is 3
Thread 2 RL, L is 3
Thread 3 RL, L is 3
Thread 4 value 4
Thread 5 AL, L is 4
Thread 1 AL, L is 4
Thread 6 RL, L is 3
Thread 0 AL, L is 4
Thread 4 RL, L is 3
Thread 7 AL, L is 4
Thread 7 value 8
Thread 5 value 8
Thread 1 value 8
Thread 0 value 8
```

Figure 6: Execution Log of L-Bakery Lock with 8 Threads and L = 4 Showing Proper L-Exclusion Enforcement

The design enforces **L-exclusion** using an atomic counter (`threadsinCriticalSection`) to ensure that no more than **L threads** can be in the critical section simultaneously. The key check is in the `waitForTurn` method, where the thread not only checks its ticket against others for lexicographical order but also waits until fewer than **L threads** are in the critical section:

```
while (ticket[j] != 0 && ... && threadsinCriticalSection.get() >= L) { }
```

Atomic operations are used to safely increment and decrement the counter when threads enter and exit the critical section, ensuring correctness and avoiding race conditions.