Analysis of different Mutual Exclusion strategies over Shared-Data structure

Danh Nguyen, 2022

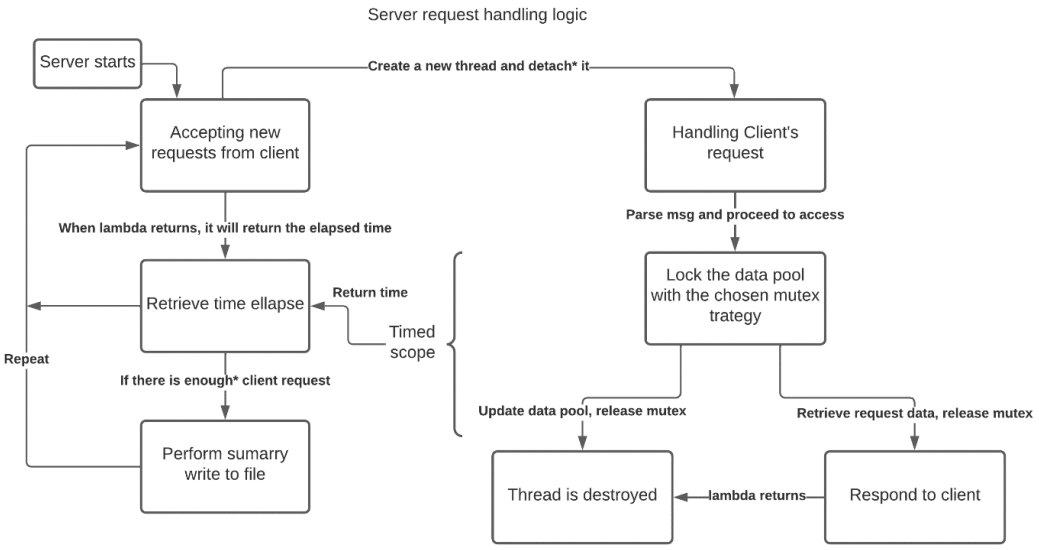
Description of Implementation

1. Introduction:

This project explores 4 different ways to achieve mutual exclusion (mutex) over a shared data pool. The data pool is an array of strings (array of arrays of characters). Via TCP/IP protocol, multiple concurrent clients will request access (either read or write) to the data pool, which is located at the server. We will protect the access to this data pool with 4 different mechanisms: a single mutex lock, an array of mutexes (each protects a single element in the array), a single read-write (RW) lock; and an array of RW locks. We will then benchmark the procedures.

2. Methodologies:

a. Server design:



All the activities on the left side are handled by the main thread (T0). When the server successfully accepts a client request, it creates a new thread and assign it a lambda.

After a certain number of connection requests, the main thread detach the all the child threads in the queue. A thread that is detached will automatically destroy itself upon finishing the lambda. This achieves 2 aspects: The main thread does not have to wait for the child threads to return (as opposed to join); it will keep creating new threads without delay. The child threads will terminate automatically and never exceed the thread limit given by the OS.

All the activities on the right are handled by a new separate child thread (Tn). The thread runs its assigned lambda. When the lambda returns the main thread collect it and add to total access time. When the thread count equals COM_NUM_REQUEST. The main thread performs an average calculation, log to file, and reset.

b. Lock design:

All lock and unlock subroutine are mutex to each other. For single and array mutex, we use an array of built-in std::mutex, with size 0 and data-pool-size respectively.

Read-write (RW) lock implementation details can be found at src/SharedPoolAccess.cpp. Same strategies as above for single vs array lock.

c. Lambda design:

The main thread accepts whatever lambda is given to it and assign to a child thread (polymorphism of OOP design). The lambda implementations are in src/DataCommManager.cpp with their details.

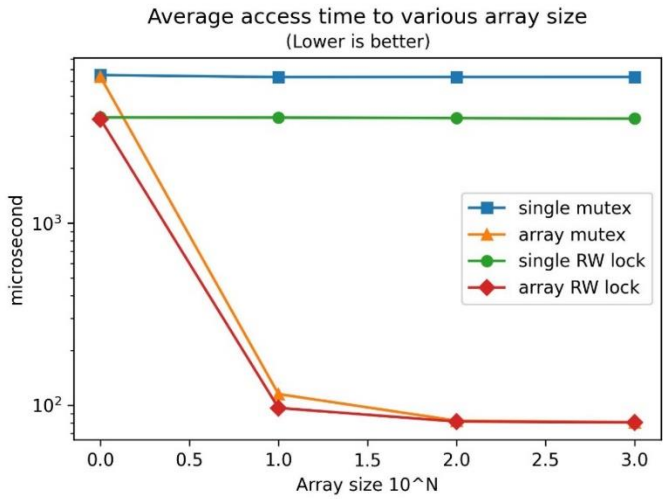
d. Testing methodologies:

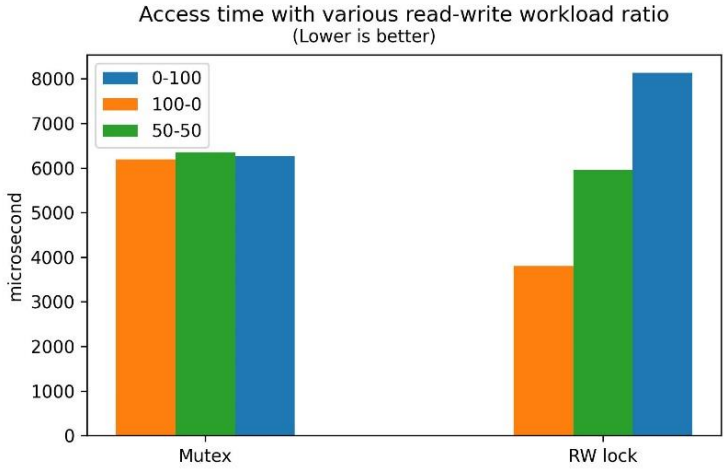
We test each mutex strategy by starting mainN and run client 100 times. We test for 3 data pool size: 1, 10, 100, 1000. We take the mean of the time and plot the

data. (Data transparency can be found at Report.ipynb, requires python3.9 and Jupyter notebook). We

Performance Discussion

3. Results:





4. Discussions:

- We see that a RW lock will be faster than a (normal) mutex since it doesn't block if there are multiple readers.

- Single locks (both kind) don't improve with data pool size since it will always block entire array.

- An array of size 1 is equivalent to single lock because whenever blocking is issue it happens to all resources.

- As the array size increase, array locks will get better simply because the chance of be blocked is reduced. (To be blocked, the slot must be “occupied” for mutex-array; and must be “write-occupied” for RW-array).

- As data pool size increase, array-locks converge and converge to no lock implementation (not shown but will be equals to overhead only).

- WR lock is better the higher the read-to-write workload ratio increases.

5. Conclusion:

From the 4 mutex-strategies we discussed in this project, we conclude that an array lock is best for a moderate refinement of data pool (>1 <1000 slots). With zero refinement (size 1) the strategies are equivalent single lock method. With high refinement, the strategies maxed-out their effectiveness to a no-lock approach (however, incorrect).

Between the needs of mutex vs RW lock: In a workload that is higher in read-write-ratio, the RW lock becomes the clear better choice. Its effectiveness maxed-out at 100% ratio to a no-lock approach (incorrect). At its minimum effectiveness (which is 0% ratio), it can be worse than a mutex due to implementation complexity.

In this project, we find that RW lock array with refinement of 10 yields the best access time among all approached. Its advantage, however, becomes insignificant to a normal mutex array at higher refinement.

For further dive in this topic, we can find that a RW lock causes write starvation the higher the read-write ratio. To address this issue, a timed-write-release-RW lock can be implemented for a periodic update; or a priority-based-write-release-RW-lock can ensure a proper write-read access ratio is achieved. These are ideas for future developments.