MCS4663 Operating Systems – Summer 2022 – Leetcode Concurrency Coding Challenges

Devson Butani

07/16/2022

Introduction:

This is a report for the Leetcode Concurrency Coding Challenges project. This project gives insight into thread management and using lock primitives such as mutex to solve concurrency and synchronization in multithreading.

Design Diagrams, Structure and Algorithm:

Using mutex with multiple threads is very straightforward because locking and unlocking at the process level is handled by the concurrency support library for C++. For challenge 1 - #1117, two concurrent threads Hydrogen and Oxygen needed a barrier system such that they synchronize to form a water molecule. Each thread released one atom every cycle therefore a set of 2 hydrogen atoms and 1 oxygen atom was required to complete the molecule and move on to the next one (if any). Aside from counting the number of atoms in every thread, a mutex was required such that the counter can be edited by only one thread at a time.

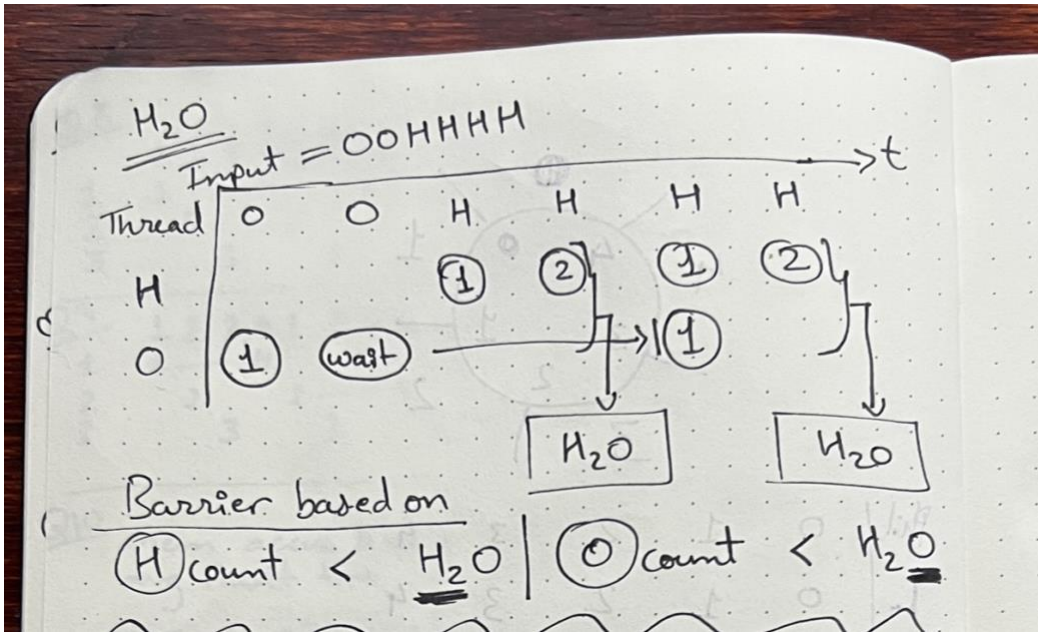

Figure 1: Design process for synchronization of H/0 atoms to form H2O

C++11 offers a solution condition_variable class that has a wait() function based on the unique_lock class for mutex lock management. Together these allow for the logic of wait until condition is met to acquire the lock. In this challenge, the condition was based on the count of the atoms required in the molecule. Since the input was constrained to be perfect number of atoms to make the molecules, simply waiting for the molecule to be complete prevented a deadlock as well as synchronized the molecule output.

For challenge 2 - #1226, five philosophers sitting at a round table had to use both forks on their left and right to eat spaghetti. Since this case is resource limited, given that one philosopher needs two forks to eat, the best-case scenario is that out of five forks, four forks can be used at the same time. However, the forks need to be on the left and right side of the philosopher eating.

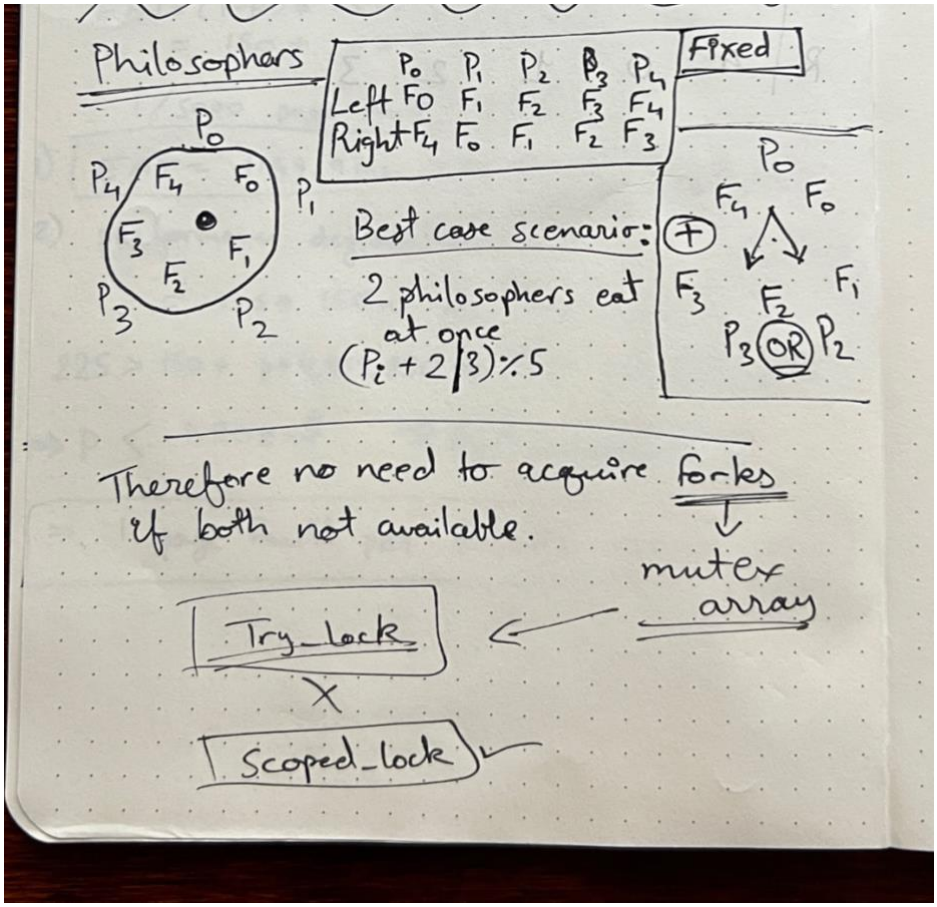

Figure 2: Design Process for Dining Philosophers

From figure 2, it is evident that only philosophers sitting opposite to each other (ie: Pi + (2 or 3) % 5) can eat at the same time. This allows simplification of the logic such that a philosopher only picks up a fork if both the forks on the left and right are available. Hence, the mutex lock required here is not singular like in the previous challenge. The forks are the limiting resource and therefore they are locked behind mutex access, in this case an array of mutexes. This allows independent philosopher threads to request and acquire forks given both are available.

C++17 offers a unique solution scoped_lock class that checks multiple mutexes and locks them in order and if any of them are not available, releases all of them. Using an array of mutexes in this way allows the best-case scenario while leaving philosophers to continue their eating process independently at the same time. For example, if philosopher 0 is eating very slowly, philosopher 2 and 3 who eat fast can keep alternating their eating process one after another.

Insights and Conclusion:

Concurrency and deadlock prevention is tricky to solve however, it is simplified when the problem gets broken down to the constraining resources. There are multiple ways to solve these challenges using different libraries and the result can be optimized by finding the right combination. Even though some classes may be faster or efficient, they sometimes require a lot more coding compared to slower classes designed for that exact use.

For example, scoped_lock class is designed particularly for locking multiple mutexes at once however, using unique_lock class with singular mutex and condition_variable class structure to check multiple values is faster. This is fast for single core concurrent processing however, in cases similar to where one philosopher takes too long to eat, scoped_lock class with multiple mutexes might be comparatively efficient for multi-core environments.