1117. Building H2O

Medium Concurrency

Leetcode Link

Problem Description

The problem provides a scenario that simulates the formation of water molecules (H2O) using threads. These threads represent the elements hydrogen (H) and oxygen (O). The objective is to ensure that two hydrogen threads (H) and one oxygen thread (O) can proceed to form a water molecule by passing a barrier, but they must do so in groups of three (two H and one O). The problem statement specifies the following constraints:

- An oxygen thread has to wait at the barrier until two hydrogen threads are available so they can all proceed together.
- Similarly, a hydrogen thread must wait for another hydrogen thread and an oxygen thread before proceeding.

The threads themselves do not need to be aware of their specific pairings; the key is to allow them to pass the barrier in complete sets that form water molecules. The code needs to enforce these synchronization constraints to ensure that no thread is allowed through the barrier until its corresponding set is complete.

Intuition

The solution uses the concept of semaphores which are synchronization tools to control access to a common resource by multiple threads in a concurrent system. In the context of this problem, semaphores are used to manage the passage of hydrogen and oxygen threads through the barrier.

Here is the intuition behind the solution:

- **Semaphore for Hydrogen (`self.h`)**: This semaphore starts with a count of 2, allowing two hydrogen threads to `acquire` it and, once acquired, proceed to release a hydrogen molecule by calling `releaseHydrogen()`. The semaphore's count decrements with each acquire. When it reaches 0, it means two hydrogen threads have passed, and an oxygen thread can now be allowed to pass by calling `self.o.release()`.

- **Semaphore for Oxygen (`self.o`)**: This semaphore starts with a count of 0, ensuring that no oxygen thread can proceed until it's allowed by hydrogen threads. When the semaphore for hydrogen reaches a count of 0 (meaning two hydrogens have been acquired), then `self.o.release()` is called to increase the count to 1, allowing one oxygen thread to `acquire` it and proceed to release an oxygen molecule by calling `releaseOxygen()`. After this, it resets the count of the hydrogen semaphore to 2 by calling `self.h.release(2)`, allowing the next group of hydrogen threads to acquire and start the process again.

Through this mechanism, we ensure that the barrier passes threads in sets of three with the correct combination to form water molecules, complying with the problem's constraints.

Solution Approach

The implementation of the solution revolves around the use of semaphores, which are counted locks that indicate the status of a resource. In the context of the problem, they are used to control the number of hydrogen and oxygen threads that can pass the barrier to form a water molecule.

Here's the breakdown of the solution approach step by step:

- **Initialization of Semaphores**:
  - `self.h`: A semaphore representing the hydrogen atoms. It is initialized with a count of 2, indicating that at the start, two hydrogen threads can proceed.
  - `self.o`: A semaphore representing the oxygen atom. It is initialized with a count of 0 since we need two hydrogen atoms to be ready before an oxygen atom can proceed.
- **Hydrogen Method**:
  - `self.h.acquire()`: Each hydrogen thread calls this method to decrement the semaphore's count. If two hydrogen threads call it, the count will become 0 and no more hydrogen threads can proceed until it's reset.
  - `releaseHydrogen()`: A callback function provided to the hydrogen method which, when invoked, signifies the passing of a hydrogen thread.
  - After releasing hydrogen, the code checks if the value of the semaphore `self.h` is 0, which means two hydrogen threads have acquired the lock and passed. It then calls `self.o.release()` to increment the oxygen semaphore, signaling that an oxygen thread can proceed.
- **Oxygen Method**:
  - `self.o.acquire()`: The oxygen thread calls this to wait until its count is greater than 0. This can only happen after two hydrogen threads have passed and called `self.o.release()`.
  - `releaseOxygen()`: A callback function provided to the oxygen method which, when invoked, signifies the passing of an oxygen thread.
  - `self.h.release(2)`: After an oxygen thread has passed, the hydrogen semaphore is reset back to a count of 2 by releasing it twice. This allows the next pair of hydrogen threads to proceed and restart the cycle.

By coordinating the actions via semaphores, the solution can ensure that at any point in time, if three threads pass through the barrier, they will always be in the `H-H-O` combination, hence forming a water molecule. It establishes a pattern where exactly two hydrogen threads must pass before an oxygen thread can, and this cycle repeats for each water molecule produced. The algorithm guarantees that these conditions are met, allowing the correct formation of water molecules without any explicit matching between threads.

The use of semaphores is a classic synchronization mechanism for enforcing limits on access to resources in concurrent programming, and this solution uses them effectively to solve the given problem.

Example Walkthrough

Let's step through a small example to illustrate the solution approach using semaphores to enforce the synchronization constraints required for the formation of water molecules:

Assume we have a sequence of threads representing hydrogen and oxygen atoms ready to form water molecules, like so: `HHOHHO`.

- **Step 1:** Initialize semaphores `self.h` (count = 2) and `self.o` (count = 0).
- **Step 2:** The first hydrogen thread `H1` arrives and calls `self.h.acquire()`, decrementing the semaphore `self.h` count to 1.
- **Step 3:** The second hydrogen thread `H2` arrives and also calls `self.h.acquire()`, decrementing `self.h`'s count to 0. Now, with 2 hydrogens ready, the code allows for an oxygen thread to proceed by calling `self.o.release()`, setting the semaphore `self.o` count to 1.
- **Step 4:** The oxygen thread `O1` arrives and calls `self.o.acquire()`, decrementing `self.o`'s count back to 0. Now, `H1` and `H2` release their hydrogen atoms by invoking `releaseHydrogen()`, and `O1` releases its oxygen atom by invoking `releaseOxygen()`.
- **Step 5:** After `O1` releases the oxygen atom, `self.h.release(2)` is called, which resets `self.h`'s count back to 2, allowing the next two hydrogen threads to proceed.
- **Step 6:** Now the `self.h` count is back at 2, another pair of hydrogen threads `H3` and `H4` arrive and each calls `self.h.acquire()`, reducing the semaphore `self.h` count back to 0.
- **Step 7:** Similar to step 3, since `self.h`'s count is 0, `self.o.release()` is called again, incrementing `self.o`'s count to 1 and allowing the next oxygen thread `O2` to proceed.
- **Step 8:** `O2` then calls `self.o.acquire()`, decreasing its count to 0, and this enables `H3`, `H4`, and `O2` to release their atoms by invoking `releaseHydrogen()` and `releaseOxygen()`, respectively.
- **Step 9:** The cycle repeats with `self.h.release(2)` resetting the hydrogen semaphore count after each oxygen atom is released, ensuring the proper ratio for water molecule formation.

Throughout this example, the semaphore mechanisms enforce the correct arrival order and combination for the threads, closely emulating the constraints of water molecule formation and demonstrating the effectiveness of semaphores for synchronization in concurrent programming.

Python Solution

```python
from threading import Semaphore
from typing import Callable

class H2O:
    def __init__(self):
        # Initialize two semaphores for hydrogen and oxygen.
        # Semaphore for hydrogen starts at 2 since we need two hydrogen atoms.
        self.sem_hydrogen = Semaphore(2)
        # Semaphore for oxygen starts at 0 - it is released when we have two hydrogen atoms.
        self.sem_oxygen = Semaphore(0)

    def hydrogen(self, releaseHydrogen: Callable[[], None]) -> None:
        # Acquire the hydrogen semaphore.
        self.sem_hydrogen.acquire()

        # When this function is called, we output the hydrogen atom.
        # This simulates the release of a hydrogen atom.
        releaseHydrogen()  # Outputs "H"

        # If there are no more hydrogen permits available, it means we have two hydrogens.
        # Now we can release oxygen to balance and make an H2O molecule.
        if self.sem_hydrogen._value == 0:
            self.sem_oxygen.release()

    def oxygen(self, releaseOxygen: Callable[[], None]) -> None:
        # Acquire the oxygen semaphore.
        self.sem_oxygen.acquire()

        # When this function is called, we output the oxygen atom.
        # This simulates the release of an oxygen atom.
        releaseOxygen()  # Outputs "O"

        # After releasing an oxygen, we reset the hydrogen semaphore to 2.
        # This allows two new hydrogen atoms to be processed, starting the cycle over.
        self.sem_hydrogen.release(2)
```

Java Solution

```java
class H2O {

    // Semaphores to control the release of hydrogen and oxygen atoms.
    private Semaphore semaphoreHydrogen = new Semaphore(2); // hydrogen semaphore initialized to 2 permits, since we need 2 H for eve
    private Semaphore semaphoreOxygen = new Semaphore(0); // oxygen semaphore initialized with 0 permits, will be released by hydroge

    public H2O() {
        // Constructor for H2O, nothing needed here since semaphores are initialized above
    }

    public void hydrogen(Runnable releaseHydrogen) throws InterruptedException {
        // Acquire a permit for releasing a hydrogen atom
        semaphoreHydrogen.acquire();
        // releaseHydrogen.run() outputs "H"
        releaseHydrogen.run();
        // Release a permit for oxygen, signaling that one H has been released
        semaphoreOxygen.release();
    }

    public void oxygen(Runnable releaseOxygen) throws InterruptedException {
        // Acquire two permits for releasing an oxygen atom as we need two hydrogen atoms before releasing one oxygen atom
        semaphoreOxygen.acquire(2);
        // releaseOxygen.run() outputs "O"
        releaseOxygen.run();
        // Release two permits for hydrogen, allowing the release of two hydrogen atoms
        semaphoreHydrogen.release(2);
    }
}
```

C++ Solution

```cpp
#include <semaphore.h>
#include <functional>

class H2O {
private:
    sem_t sem; // Semaphore for hydrogen
    sem_t sem0; // Semaphore for oxygen
    int hydrogenCount; // Count the number of hydrogens produced

public:
    H2O() : hydrogenCount(0) {
        sem_init(&sem, 0, 2); // Initialize the semaphore for hydrogen to 2, allowing 2 hydrogens to be produced
        sem_init(&sem0, 0, 0); // Initialize the semaphore for oxygen to 0, blocking oxygen production until hydrogen is available
    }

    void hydrogen(std::function<void()> releaseHydrogen) {
        sem_wait(&sem); // Decrement the semaphore for hydrogen, blocking if unavailable
        // releaseHydrogen() outputs "H". Do not change or remove this line.
        releaseHydrogen();
        ++hydrogenCount; // Increment the count of produced hydrogen atoms
        if (hydrogenCount == 2) { // If 2 hydrogens are produced, an oxygen may be produced
            sem_post(&sem0); // Increment the semaphore for oxygen, allowing an oxygen to be produced
        }
    }

    void oxygen(std::function<void()> releaseOxygen) {
        sem_wait(&sem0); // Decrement the semaphore for oxygen, blocking if unavailable
        // releaseOxygen() outputs "O". Do not change or remove this line.
        releaseOxygen();
        hydrogenCount = 0; // Reset the count of produced hydrogen atoms
        sem_post(&sem); // Increment the semaphore for hydrogen, allowing another hydrogen to be produced
        sem_post(&sem); // Allow the second hydrogen to be produced
    }
};
```

Typescript Solution

```typescript
// Importing necessary libraries for semaphore functionality
const { Semaphore } = require('await-semaphore'); // Node.js library for semaphores, use 'npm install await-semaphore' to install

// Creating semaphores for hydrogen and oxygen
let sem0: Semaphore = new Semaphore(2); // Semaphore for hydrogen, starts at 2 to allow 2 hydrogens
let sem0: Semaphore = new Semaphore(0); // Semaphore for oxygen starts at 0
let hydrogenCount: number = 0; // Counter for the number of hydrogen atoms produced

// Function to simulate the release of a hydrogen atom
async function hydrogen(releaseHydrogen: () => void): Promise<void> {
    await sem0.acquire(); // Acquire a hydrogen semaphore permit (decrement)
    // releaseHydrogen() outputs "H". Do not change or remove this line.
    releaseHydrogen();
    hydrogenCount++; // Increment the hydrogen count

    if (hydrogenCount == 2) {
        // After 2 hydrogens, allow the release of an oxygen
        sem0.release(); // Release an oxygen semaphore permit (increment)
    }
}

// Function to simulate the release of an oxygen atom
async function oxygen(releaseOxygen: () => void): Promise<void> {
    await sem0.acquire(); // Acquire an oxygen semaphore permit (decrement)
    // releaseOxygen() outputs "O". Do not change or remove this line.
    releaseOxygen();
    hydrogenCount = 0; // Reset the hydrogen counter

    // Release two hydrogen semaphore permits (increment twice)
    sem0.release();
    sem0.release();
}

// Export the functions if this is being used as a module
module.exports = { hydrogen, oxygen };
```

Time and Space Complexity

Time Complexity

The time complexity of both `hydrogen` and `oxygen` methods in the `H2O` class can be described as O(1). These methods involve a fixed number of operations: a semaphore acquire operation and a semaphore release operation. The semaphore operations themselves have a constant time complexity since they are essentially atomic operations that manage an internal counter.

Space Complexity

The space complexity of the `H2O` class is O(1). This is because the class maintains a constant number of semaphores with fixed space usage regardless of the number of invocations of the `hydrogen` and `oxygen` methods. No additional space that scales with the input size or the number of method calls is utilized.