d) none of the mentioned

4. What are the two atomic operations permissible on semaphores?

a) wait

b) stop

c) hold

d) none of the mentioned

5. When several processes access the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called _________

a) dynamic condition

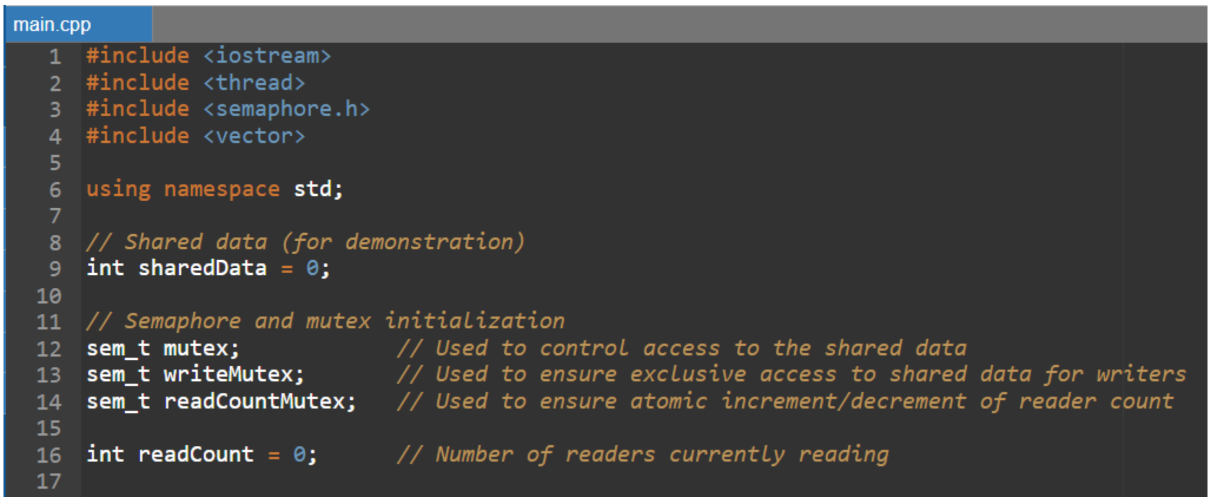
b) race condition

c) essential condition

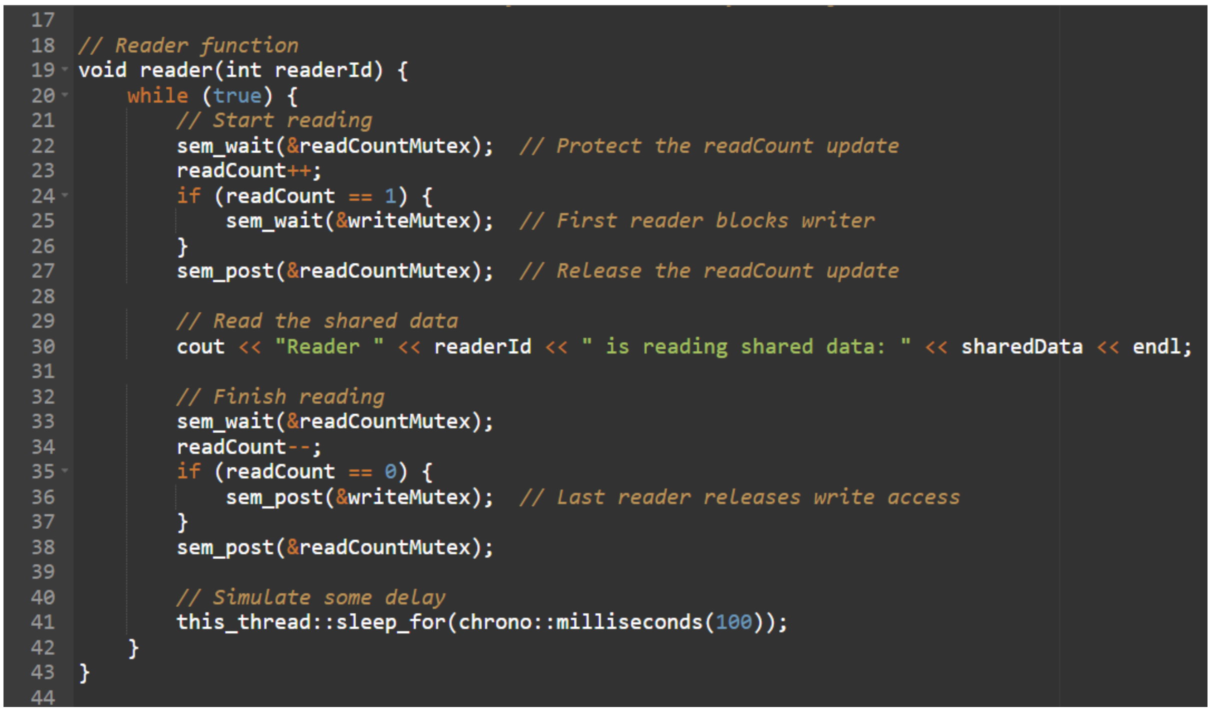
d) critical condition

Student Work Area

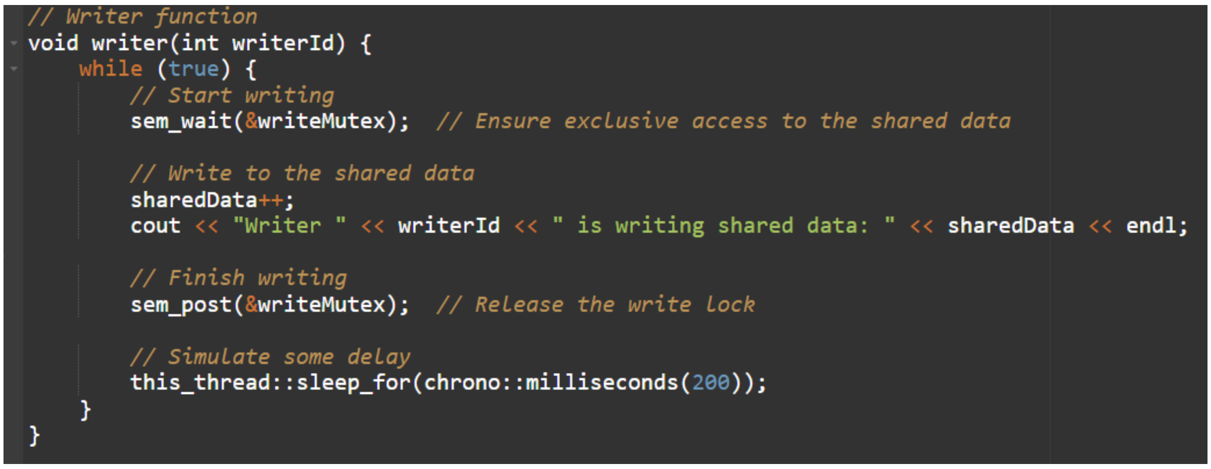
Algorithm/Flowchart/Code/Sample Outputs


main.cpp

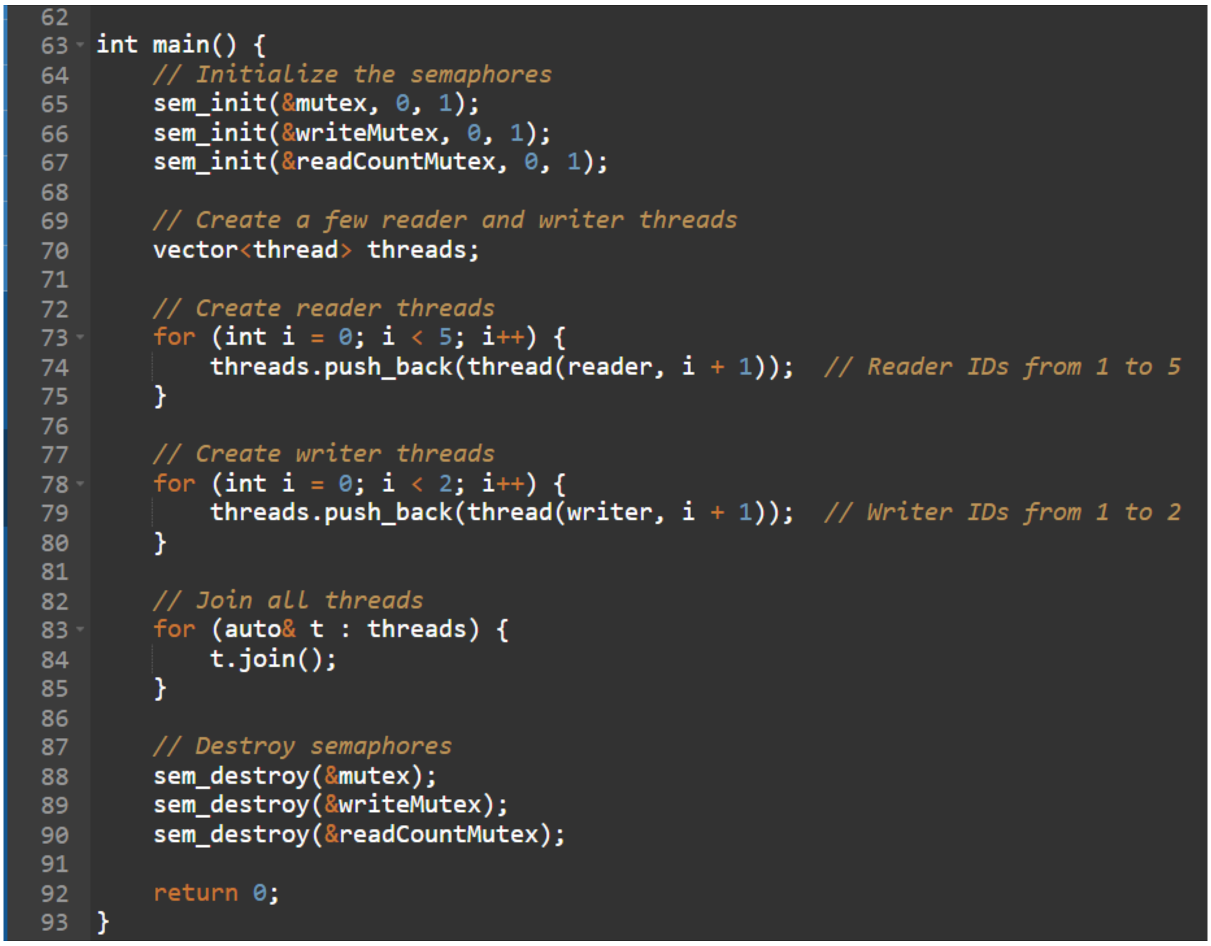
```cpp
#include <iostream>
#include <thread>
#include <semaphore.h>
#include <vector>

using namespace std;

// Shared data (for demonstration)
int sharedData = 0;

// Semaphore and mutex initialization
sem_t mutex;            // Used to control access to the shared data
sem_t writeMutex;       // Used to ensure exclusive access to shared data for writers
sem_t readCountMutex;   // Used to ensure atomic increment/decrement of reader count

int readCount = 0;      // Number of readers currently reading

```



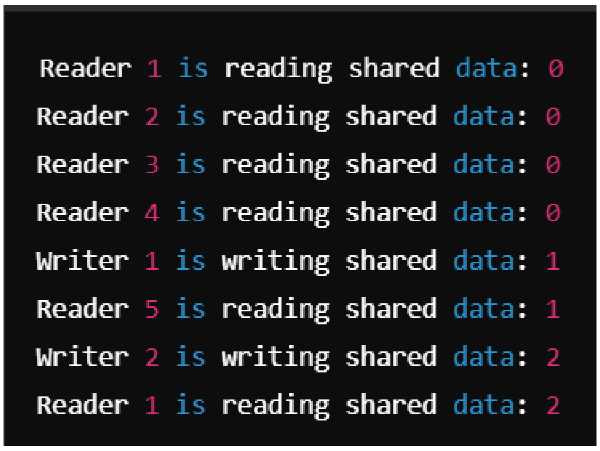
```cpp

// Reader function
void reader(int readerId) {
    while (true) {
        // Start reading
        sem_wait(&readCountMutex);  // Protect the readCount update
        readCount++;
        if (readCount == 1) {
            sem_wait(&writeMutex);  // First reader blocks writer
        }
        sem_post(&readCountMutex);  // Release the readCount update

        // Read the shared data
        cout << "Reader " << readerId << " is reading shared data: " << sharedData << endl;

        // Finish reading
        sem_wait(&readCountMutex);
        readCount--;
        if (readCount == 0) {
            sem_post(&writeMutex);  // Last reader releases write access
        }
        sem_post(&readCountMutex);

        // Simulate some delay
        this_thread::sleep_for(chrono::milliseconds(100));
    }
}

```



```cpp
// Writer function
void writer(int writerId) {
    while (true) {
        // Start writing
        sem_wait(&writeMutex);  // Ensure exclusive access to the shared data

        // Write to the shared data
        sharedData++;
        cout << "Writer " << writerId << " is writing shared data: " << sharedData << endl;

        // Finish writing
        sem_post(&writeMutex);  // Release the write lock

        // Simulate some delay
        this_thread::sleep_for(chrono::milliseconds(200));
    }
}
```



```
int main() {
    // Initialize the semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&writeMutex, 0, 1);
    sem_init(&readCountMutex, 0, 1);

    // Create a few reader and writer threads
    vector<thread> threads;

    // Create reader threads
    for (int i = 0; i < 5; i++) {
        threads.push_back(thread(reader, i + 1));  // Reader IDs from 1 to 5
    }

    // Create writer threads
    for (int i = 0; i < 2; i++) {
        threads.push_back(thread(writer, i + 1));  // Writer IDs from 1 to 2
    }

    // Join all threads
    for (auto& t : threads) {
        t.join();
    }

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&writeMutex);
    sem_destroy(&readCountMutex);

    return 0;
}
```



```
Reader 1 is reading shared data: 0
Reader 2 is reading shared data: 0
Reader 3 is reading shared data: 0
Reader 4 is reading shared data: 0
Writer 1 is writing shared data: 1
Reader 5 is reading shared data: 1
Writer 2 is writing shared data: 2
Reader 1 is reading shared data: 2
```