Bridge Crossing Simulation with Semaphores and Mutex

This code implements a simulation of cars crossing a bridge from both the left and right sides. It utilizes semaphores and a mutex to ensure synchronized and collision-free movement of vehicles.

Problem Description

The bridge crossing problem involves managing the movement of cars approaching a bridge from both the left and right sides to ensure they cross the bridge safely and without collisions.

Solution Overview

Semaphores are synchronization primitives that allow a limited number of threads to access a shared resource simultaneously. In the bridge crossing simulation, semaphores are used to control the number of cars from each direction that can be on the bridge at a given time.

- bridgeSem: This semaphore controls overall access to the bridge. Only one thread can hold the bridgeSem at a time, ensuring that only one car can be on the bridge at a time. This semaphore is initialized to a value of 5, allowing a maximum of 5 cars on the bridge at a given time.
- leftSem: This semaphore manages the number of cars from the left side that can be on the bridge at a given time. Only one thread from the left side can hold the leftSem at a time, ensuring that only one car from the left side can be on the bridge at a time. This semaphore is also initialized to a value of 5.
- rightSem: This semaphore manages the number of cars from the right side that can be on the bridge at a given time. Only one thread from the right side can hold the rightSem at a time, ensuring that only one car from the right side can be on the bridge at a time. This semaphore is also initialized to a value of 5.

Mutex

A mutex is a synchronization primitive that ensures that only one thread can access a shared resource at a time. In the bridge crossing simulation, a mutex is used to protect

the critical section of code where car counts are updated. This prevents data races, which can occur when multiple threads try to access and update the same shared resource at the same time.

The critical section of code is the part of the code where the `leftCount` and `rightCount` variables are updated. These variables track the number of cars from each direction that are currently on the bridge. The mutex is acquired before entering the critical section and released after exiting the critical section. This ensures that only one thread can update the car counts at a time, preventing data races.

Interaction between Semaphores and Mutex

The semaphores and mutex work together to ensure safe and synchronized access to the bridge. The semaphores control the overall number of cars that can be on the bridge at a time, while the mutex protects the critical section of code where car counts are updated. This ensures that cars from opposite directions do not simultaneously occupy the bridge, preventing collisions.

Here is a summary of how the semaphores and mutex are used in the bridge crossing simulation:

1. A car approaching the bridge from the left waits for a slot using the leftSem semaphore.
2. If a slot is available, the car acquires the mutex to protect the critical section of code.
3. The car updates the `leftCount` variable to reflect its presence on the bridge.
4. The car releases the mutex and waits for the bridge to be clear using the bridgeSem semaphore.
5. Once the bridge is clear, the car simulates crossing the bridge using the passing() function.
6. After crossing the bridge, the car releases the bridgeSem semaphore.
7. The car updates the `leftCount` variable to reflect its departure from the bridge.
8. The car signals an available slot using the leftSem semaphore.

The process for cars approaching from the right side is similar, using the rightSem semaphore and updating the `rightCount` variable.

This coordinated use of semaphores and a mutex ensures that cars from opposite directions do not simultaneously occupy the bridge, preventing collisions and ensuring

smooth traffic flow.This solution employs semaphores and a mutex to coordinate car crossings:

Semaphores:

- bridgeSem: Controls overall access to the bridge, allowing a maximum of 5 cars on the bridge at a time.
- leftSem: Manages the number of cars from the left side that can be on the bridge at a given time.
- rightSem: Manages the number of cars from the right side that can be on the bridge at a given time.

Mutex:

- mutex: Used to protect the critical section of code where car counts are updated, preventing data races and ensuring accurate car counts.

Code Structure

The code is organized into the following sections:

- Header Files: The code includes the necessary header files for input/output (`stdio.h`), standard library functions (`stdlib.h`), pthread operations (`pthread.h`), semaphores (`semaphore.h`), and Unix system calls (`unistd.h`).
- Macro Definitions: The code defines a macro for the maximum number of cars allowed on the bridge (`MAX_CARS`).
- Global Variables: The code declares global variables for the semaphores (`bridgeSem`, `leftSem`, `rightSem`), and a mutex (`mutex`).
- `passing()` Function: This function simulates a car crossing the bridge, taking the car's direction and number as parameters.
- `left()` and `right()` Thread Functions: These functions represent cars approaching the bridge from the left and right sides, respectively. They each:
    1. Wait for a slot using their respective semaphore (leftSem or rightSem).
    2. Acquire the mutex to protect the critical section.
    3. Wait for the bridge to be clear using bridgeSem.
    4. Simulate the car crossing the bridge using passing().
    5. Release bridgeSem and the mutex.
    6. Signal an available slot using their respective semaphore.
    7. Exit the thread.

- `main()` Function: This function controls the main flow of the program:
    1. Prompts the user to enter the number of cars from the left and right sides.
    2. Checks if the total number of cars exceeds the limit (`MAX_CARS`).
    3. Initializes semaphores and mutex.
    4. Creates threads for each car using `pthread_create()`.
    5. Waits for all threads to finish using `pthread_join()`.
    6. Destroys semaphores and mutex.
    7. Exits the program.

Testing Strategy

To test the program's effectiveness, run it with varying numbers of cars from the left and right sides. Verify that cars from opposite directions do not simultaneously occupy the bridge, ensuring collision-free traffic flow.

Conclusion

This project effectively demonstrates the use of semaphores and a mutex in synchronizing concurrent processes. Semaphores play a crucial role in preventing data races and ensuring safe and orderly execution of critical sections of code, particularly in scenarios involving shared resources. The mutex further enhances synchronization by protecting critical updates to car counts.