Overview:

This C program simulates the controlled movement of cars across a bridge from both the left and right sides. The primary goal is to manage traffic flow and avoid concurrency issues using pthreads and semaphores.

Code Structure

Variables

• **RightCars:** Number of cars coming from the right side.

• **LeftCars:** Number of cars coming from the left side.

• **f:** Sleep duration (in seconds).

• **leftMessage and rightMessage:** Arrays containing messages for cars from the left and right sides.

Semaphores

• **mutex**: Semaphore for mutual exclusion, ensuring synchronized access to shared variables.

• **leftSem and rightSem:** Semaphores for synchronization between left and right threads.

Thread Functions

1. left Function:

• Represents cars coming from the left side.

• Uses semaphores for synchronization.

• Simulates cars passing through, printing messages for each car.

2. right Function:

• Represents cars coming from the right side.

• Uses semaphores for synchronization.

• Simulates cars passing through, printing messages for each car.

3. monitorCars Function:

• Monitoring thread checking the state of cars on both sides.

• Releases the semaphore for the other side if there are no cars on one side.

• Uses usleep to avoid busy-waiting

Main Function

• Takes user input for the number of cars on the left and right sides.

• Initializes semaphores.

• Creates threads for left, right, and monitoring functions.

• Waits for all threads to finish using pthread_join.

• Destroys semaphores.

Running the Program

1. Compile the program using a C compiler (e.g., gcc q3.c -o q3).

2. Run the executable (./q3).

3. Enter the number of cars on the left and right sides when prompted.

Avoiding Concurrency Bugs

1. Semaphore Usage:

• sem_wait and sem_post functions control access to the bridge, preventing multiple threads from accessing it simultaneously and avoiding race conditions.

2. Thread Synchronization:

• The use of pthread_join ensures that the main function waits for both threads to complete before proceeding, preventing premature termination of the program.

3. Atomic Operations:

• Operations on shared variables (RightCars and LeftCars) are simple and atomic, reducing the chances of data corruption.

4. Sequential Movement:

• Cars from the left side move first, followed by cars from the right side. This is achieved through proper signaling between threads.