READ ME

OS ASSIGNMENT 4
VANSH YADAV (2022559)
SHAMIK SINHA (2022468)

OVERVIEW

This assignment delves into concurrency and synchronization primitives in C programming, encompassing three problems. Question 1 revisits the dining philosophers problem with the addition of bowls, necessitating two forks and a bowl for eating to avoid deadlocks. A solution was devised using an odd-even strategy for philosopher threads to alternately acquire forks, preventing deadlocks. Question 2, simulating a car ride scenario with limited capacity, is in progress. Question 3 focuses on cars crossing a narrow bridge without deadlocks or collisions, employing semaphores to control bridge access and ensure safe passage for a restricted number of cars.

QUESTIONS AND APPROACH

Question 1.

- The problem involves a modification of the classic dining philosophers problem by introducing bowls in addition to forks.

- Each philosopher requires two forks and a bowl for eating.

- Deadlock situations may arise if philosophers simultaneously try to acquire resources without a proper strategy.

<u>Code Explanation</u>

- **`philosopher()`** function: Represents the behavior of each philosopher thread. They alternate between thinking and eating states.

- **`thinking()`** and **`eating()`** functions: Simulate the thinking and eating states respectively using `sleep()`.

Approach

- The code implements an odd-even strategy for philosophers to avoid deadlocks.

- Philosophers with even IDs pick the left fork first, then the right fork, while odd IDs do the reverse.

- Mutex locks (`pthread_mutex_t`) and conditional variables (`pthread_cond_t`) are used for synchronization.

Fairness and Deadlock Avoidance

- The odd-even rule ensures that at most two philosophers eat simultaneously without sharing resources.

- By alternating the order of fork acquisition, deadlocks are prevented.

- Fairness: Each philosopher gets a chance to eat in a non-starvation manner.

-At max, two philosophers can eat at same time that too have too be not together like philosopher 1 can eat with philosopher 3 but not philosopher 2.

-Deadlock could happen if all the philosophers would pick their left fork then it would result in deadlock but it has been solved with our approach.

Question 2.

Car and Passenger Simulation This C program simulates a scenario where a car has limited capacity and multiple passengers eagerly await their turn to take a ride. The car and passenger interactions are modeled using threads and synchronized using semaphores. Code Logic: Car Thread (car function): Loading Passengers: The car loads passengers with the load function. Signaling Passengers to Board: Passengers are signaled to board using dispatch_semaphore_signal. Waiting for Passengers to Board: The car thread waits for passengers to board using a condition variable (pthread_cond_wait) and a mutex (pthread_mutex_lock). Running the Car: The car runs, simulating the duration of the ride with carruns. Unloading Passengers: The car unloads passengers with the unload function. Signaling Passengers to Offboard: Passengers are signaled to offboard using dispatch_semaphore_signal. Waiting for Passengers to Offboard: The car thread waits for passengers to offboard using a condition variable and a mutex. Passenger Thread (passenger function): Boarding the Car: Passengers board the car with the board function. Signaling Car Ready: Passengers signal that they have boarded using pthread_cond_broadcast. Waiting for Car to Run and Unload: Passengers wait for the car to run and unload using dispatch_semaphore_wait. Offboarding the Car: Passengers offboard the car with the offboard function. Signaling Car Empty: Passengers signal that they have offboarded using pthread_cond_broadcast. Synchronization: Mutex (car_mutex): Used to synchronize access to shared variables and for waiting on condition variables. Semaphore (passenger_sem): Controls the boarding and offboarding of passengers. Semaphore (mainSem): Ensures that passengers are created in a controlled manner. Concurrency Bug Prevention: Mutex and Semaphores: Proper use of mutex and semaphores ensures exclusive access to shared resources, preventing data corruption. Condition Variables: Used to synchronize the car and passenger threads, ensuring that actions are performed in the correct order. Sleep Function: Introduces delays to simulate time passing, allowing proper sequencing of actions.

Question 3.

- The problem involves cars crossing a narrow bridge from both sides without causing a deadlock.

- Constraints limit the number of cars on the bridge simultaneously.

Code Explanation

- **`passing()`** function: Represents the action of a car crossing the bridge. Simulated delay using `sleep()`.

Approach

- The program models cars as threads from both sides, allowing a specific number of cars to cross the bridge concurrently.

- Semaphores (`dispatch_semaphore_t`) are used to regulate the maximum number of cars on the bridge.

Code Logic and Concurrency Bug Avoidance

Concurrency Constraints: Cars from both sides aim to cross a narrow bridge with a maximum of 5 cars at a time without causing collisions.

We have created a semaphore having counter value initialized to (maximum number of cars that can travel at same time -1) ,i.e, 4 because this way only 5 threads can run concurrently and then in the loop , we keep on decrementing the value of semaphore by using semaphore_wait , and when the value becomes negative ,then that thread gets blocked until previous threads does their execution. This way concurrency is handled and at one go , at max 5 threads can execute.

Semaphore Implementation: Utilized semaphores to regulate access to the bridge, ensuring that the specified number of cars cross safely without causing deadlocks or collisions.

Concurrency Bug Avoidance: The use of semaphores prevents concurrent access violations, allowing safe passage for cars.

Also in our approach all the left cars cross bridge first then comes turn of right cars.