CSE344-Systems Programming

HW3 REPORT

Adem Murat ÖZDEMİR-200104004110

Compiling and Running

Compile the code : “make”

Run the program : “make run”

Clean the object files : “make clean”

There are total 50 vehicles. The program will end after 50 vehicles arrive.

Problem Overview

This project involves creating a parking system using semaphores and shared memory to manage a parking lot for automobiles and pickups. The system handles the arrival of vehicles, their temporary parking, and their transfer to a main parking lot, ensuring proper synchronization and resource management. Two types of attendants manage the parking operations, and semaphores ensure orderly and conflict-free processing of vehicles.

Key Features

- **Vehicle Arrival and Handling**:
    - Vehicles arrive randomly and are designated as either automobiles or pickups.
    - Only one vehicle is processed at a time to maintain order.
- **Parking Management**:
    - The system has eight spots for automobiles and four spots for pickups in a temporary parking lot.
    - Each vehicle type has a dedicated attendant responsible for parking.
- **Synchronization Using Semaphores**:
    - Semaphores ensure that vehicle owners can only park if there are available spots.
    - Attendants are synchronized to manage the transfer of vehicles from the temporary lot to the main parking lot.
- **Shared Resource Protection**:
    - Shared variables track the availability of parking spots and must be protected to prevent race conditions.
    - Semaphores manage the queue of vehicles, the count of processed vehicles, and single vehicle entry at a time.

- **Concurrency Management**:
    - Threads representing vehicle owners and attendants run concurrently, with semaphores ensuring proper synchronization and data integrity.

How to Solve

To implement the parking system, the following steps and considerations were taken:

- **Initialization**:
    - **Semaphores**: Initialize semaphores to manage parking spot availability and ensure synchronization between vehicle owners and attendants. This includes semaphores for new arrivals, control of spot assignment, and overall vehicle queue management.
    - **Shared Variables**: Initialize shared variables to track the availability of parking spots and manage the vehicle queue.
- **Vehicle Queue Setup**:
    - Populate the vehicle queue with random vehicle types, simulating the random arrival of automobiles and pickups.
- **Thread Creation**:
    - A total of 4 threads are created. and these threads work with producer-consumer logic.
    - Vehicle owners represent the vehicles arriving at the parking lot, while attendants manage parking operations.
- **Vehicle Owner Logic**:
    - Each vehicle owner checks for available spots in the temporary parking lot. If a spot is available, the vehicle is parked; otherwise, the owner leaves. Semaphores ensure that only one vehicle is processed at a time and that vehicle queue access is synchronized.
- **Parking Attendant Logic**:
    - Attendants wait for signals indicating a new vehicle has arrived in the temporary lot. Vehicles are then moved from the temporary lot to the main parking lot, and the availability of spots is updated accordingly. Each attendant is responsible for a specific type of vehicle (automobile or pickup).
- **Thread Synchronization and Completion**:
    - Join threads to ensure the main program waits for all threads to complete their execution. Destroy semaphores to release resources after the simulation ends.

By managing thread synchronization with semaphores and protecting shared resources, the system ensures efficient and conflict-free handling of vehicle parking operations.

Semaphores that are used :

```
// Semaphores
sem_t newAutomobile; // Semaphore for new automobile
sem_t inChargeforAutomobile;  // Semaphore for automobile attendant
sem_t newPickup; // Semaphore for new pickup
sem_t inChargeforPickup; // Semaphore for pickup attendant
sem_t automobileSpots; // Semaphore for automobile spots
sem_t pickupSpots; // Semaphore for pickup spots
sem_t vehicleQueueSemaphore; // Semaphore for vehicle queue to take one car at a time
sem_t processedSemaphore; // Semaphore for processed vehicle count
sem_t singleVehicleEntry; // Semaphore to allow only one vehicle to be processed at a time
```

Shared Variables that are used :

```
// Shared variables
int mFree_automobile = MAX_AUTOMOBILE;
int mFree_pickup = MAX_PICKUP;
int vehicleQueue[TOTAL_VEHICLES]; // Queue for vehicles
int vehicleIndex = 0; // Index for vehicle queue
int vehiclesProcessed = 0; // Count of processed vehicles
```

Function Explanation: carOwner()

The carOwner function simulates the behavior of vehicle owners arriving at the parking lot. It is designed to run continuously and handle each vehicle's attempt to park in the temporary parking lot. Here's a detailed explanation of the function:

- **Single Vehicle Entry Control**:
    - The function starts by ensuring that only one vehicle can be processed at a time. This is achieved using the singleVehicleEntry semaphore, which acts as a mutex to serialize access to the parking lot entry.
- **Critical Section for Vehicle Queue**:
    - The function then enters a critical section to fetch the next vehicle type from the queue. This is managed using the vehicleQueueSemaphore to ensure that the vehicle queue is accessed in a thread-safe manner.
    - If all vehicles have been processed (vehicleIndex exceeds TOTAL_VEHICLES), the semaphores are released, and the function exits.
- **Vehicle Type Handling**:
    - The function checks the type of the vehicle (automobile or pickup).
    - **Automobiles**:
        - It acquires the inChargeforAutomobile semaphore to ensure exclusive access to the automobile parking spots.

        - If there is an available spot (automobileSpots semaphore), the vehicle is parked, and the newAutomobile semaphore is posted to signal that an automobile is in the temporary parking lot.
        - The inChargeforAutomobile semaphore is released.
    - **Pickups**:
        - Similar to automobiles, the function acquires the inChargeforPickup semaphore to manage pickup parking spots.
        - If a spot is available (pickupSpots semaphore), the pickup is parked, and the newPickup semaphore is posted to signal that a pickup is in the temporary parking lot.
        - The inChargeforPickup semaphore is released.
    - If no spot is available for the vehicle type, a message is printed indicating that the owner left without parking.
- **Allow Next Vehicle Processing**:
    - The singleVehicleEntry semaphore is posted to allow the next vehicle to be processed.
- **Simulate Arrival Time**:
    - The function sleeps for a short period (usleep(500000)) to simulate the time between vehicle arrivals.
- **Check Processed Vehicles Count**:
    - The function checks if all vehicles have been processed using the processedSemaphore.
    - If the total number of processed vehicles reaches TOTAL_VEHICLES, the function exits.

By using semaphores to manage access to shared resources and ensure proper synchronization, the carOwner function effectively simulates vehicle owners attempting to park in the temporary parking lot in a controlled and orderly manner.

Function Explanation: carAttendant()

The carAttendant function simulates the behavior of parking attendants who are responsible for moving vehicles from the temporary parking lot to the main parking lot. It continuously checks for new vehicles and processes them based on the type of vehicle the attendant is responsible for. Here's a detailed explanation of the function:

1. **Attendant Type Determination**:
    - The function begins by determining the type of attendant (automobile or pickup) based on the argument passed to the thread.
2. **Continuous Operation**:
    - The function runs in an infinite loop, continuously checking for new vehicles to process.
3. **Automobile Attendant Logic**:
    - If the attendant is responsible for automobiles (attendantType == 1), the function waits for the newAutomobile semaphore, which signals that a new automobile has arrived in the temporary parking lot.
    - The function then simulates the time required to park the vehicle in the main lot by sleeping for 1 second.
    - The inChargeforAutomobile semaphore is acquired to ensure exclusive access to the automobile parking spots.
    - The count of free automobile spots (mFree_automobile) is incremented, indicating that an automobile has been moved to the main parking lot.
    - A message is printed to indicate the updated number of free automobile spots.
    - The automobileSpots semaphore is posted to signal that a spot is now available.
    - The inChargeforAutomobile semaphore is released.
4. **Pickup Attendant Logic**:
    - If the attendant is responsible for pickups (attendantType == 2), the function waits for the newPickup semaphore, which signals that a new pickup has arrived in the temporary parking lot.
    - The function simulates the time required to park the vehicle in the main lot by sleeping for a short period.
    - The inChargeforPickup semaphore is acquired to ensure exclusive access to the pickup parking spots.
    - The count of free pickup spots (mFree_pickup) is incremented, indicating that a pickup has been moved to the main parking lot.
    - A message is printed to indicate the updated number of free pickup spots.
    - The pickupSpots semaphore is posted to signal that a spot is now available.

- The inChargeforPickup semaphore is released.

5. **Increment Processed Vehicle Count**:
    - The function acquires the processedSemaphore to update the count of processed vehicles (vehiclesProcessed).
    - If the total number of processed vehicles reaches TOTAL_VEHICLES, the function exits.
    - The processedSemaphore is released.

By using semaphores to manage the synchronization of vehicle processing and shared resource access, the carAttendant function ensures that vehicles are efficiently moved from the temporary parking lot to the main parking lot, and the system operates smoothly and without conflicts.

Test Results

```
Pickup owner arrived. Free pickup spots before parking: 4
Pickup parked to the temporary park. Free pickup spots now: 3


Automobile owner arrived. Free automobile spots before parking: 8
Automobile parked to the temporary park. Free automobile spots now: 7


Pickup owner arrived. Free pickup spots before parking: 3
Pickup parked to the temporary park. Free pickup spots now: 2


Pickup owner arrived. Free pickup spots before parking: 2
Pickup parked to the temporary park. Free pickup spots now: 1


A pickup is taken from temporary park to the main parking lot. Free pickup spots now: 2

An automobile is taken from temporary park to the main parking lot. Free automobile spots now: 8

Pickup owner arrived. Free pickup spots before parking: 2
Pickup parked to the temporary park. Free pickup spots now: 1


Pickup owner arrived. Free pickup spots before parking: 1
Pickup parked to the temporary park. Free pickup spots now: 0
```

If there is no empty lot for parking , program will print this :

```
A pickup arrived but no free spots for pickups. Pickup owner left.
```

But when there will be an empty lot , next one will be able to park its vehicle :

```
A pickup is taken from temporary park to the main parking lot. Free pickup spots now: 1

Pickup owner arrived. Free pickup spots before parking: 1
Pickup parked to the temporary park. Free pickup spots now: 0
```

As I wrote at the beginning, the program will end after 50 vehicles arrive.

NOTE : When the program is run, in the terminal , it seems like two vehicles are coming at the same time, but I prevented this with the semaphores I used, as can be seen from the code. It looks like this because the threads start running at the same time. I thought about adding sleep(1) between the threads while creating them to suppress the prints better, but I did not do this because I thought it would not comply with the working logic of the program.