CSE344 Homework 3: Semaphore and Shared Memory

Tugay Talha İçen

May, 2024

Chapter 1

Introduction

This report presents the implementation of a parking lot simulation program written in C. The program models the arrival of cars and trucks, their temporary waiting, and their eventual parking in designated spots by attendants. This project emphasizes the use of semaphores and shared memory for synchronization and resource management, aligning with the requirements of the CSE344 homework assignment.

Chapter 2

Problem-Solving Approach

This chapter dives into the design decisions and the approach taken to solve the parking lot simulation problem.

2.1 Design Decisions

2.1.1 Choice of Data Structures

Integers were chosen to represent the parking spots because they are a simple and efficient way to track the availability of spots. Semaphores were used for synchronization because they provide a straightforward mechanism for threads to wait for a resource to become available and signal when a resource is released. A mutex lock was used to protect shared variables from race conditions, ensuring that changes to these variables are atomic and consistent.

2.1.2 Thread Coordination

The program utilizes three types of threads:

* **Car Owner Threads:** These threads simulate the arrival of vehicles. They wait for a random amount of time before attempting to park. * **Car Attendant Automobile Threads:** These threads are responsible for parking automobiles in real parking spots. * **Car Attendant Pickup Threads:** These threads are responsible for parking trucks in real parking spots.

These threads interact with each other using semaphores. For instance, a car owner thread will increment the `inChargeforPickup` semaphore when it occupies a temporary truck spot, signaling the attendant thread to park the truck. The attendant thread then decrements this semaphore after successfully parking the truck.

2.1.3 Synchronization Strategy

The primary goal of the synchronization strategy is to prevent car owners from entering the parking lot if there are no temporary spots available and to ensure that attendants do not park vehicles in real spots that are already occupied.

This is achieved through the following mechanisms:

* **Entrance Semaphore:** The `entrance` semaphore limits the number of vehicles entering the parking lot to one at a time. This prevents a rush of vehicles from overwhelming the temporary parking spaces. * **Temporary Parking Semaphores:** The `newPickup` and `newAutomobile` semaphores are used to manage the availability of temporary parking spots. Car owners wait on these semaphores before occupying a temporary spot, preventing multiple cars from taking the same spot. * **Attendant Semaphores:** The `inChargeforPickup` and `inChargeforAutomobile` semaphores signal the attendant threads to handle vehicles waiting in temporary spots.

2.2 Code Utilization

The following code snippets illustrate key aspects of the synchronization strategy:

2.3 Car Owner Thread (Key Snippet)

```
sem_wait(&entrance);

pthread_mutex_lock(&mutex);
totalFreeSpots--;
if (carType == 0 && tempFree_automobile > 0) {

    sem_post(&inChargeforAutomobile);
    sem_wait(&newAutomobile); // Wait for parking confirmation
} else if (carType == 1 && tempFree_pickup > 0) {
    // ... (Occupy temporary spot)
    sem_post(&inChargeforPickup);
    sem_wait(&newPickup); // Wait for parking confirmation
} else {
    // ... (No temporary spot available, leave)
}
pthread_mutex_unlock(&mutex);
```

```
17 sem_post(&entrance); // Release entrance
       semaphore
```

Listing 2.1: Car Owner Thread - Key Snippet

2.4 Car Attendant Automobile Thread (Key Snippet)

```
1  sem_wait(&inChargeforAutomobile);
2  printf(red"redAttendantred redparkingred
       redanred redautomobilered.\rednred");
3  WAIT_ATTENDANT;
4
5  if (mFree_automobile > 0) {
6      // ... (Park vehicle in a real spot)
7      sem_post(&newAutomobile); // Signal
           owner thread
8  } else {
9      // ... (No real spot available, wait)
10     sem_post(&inChargeforAutomobile);
11 }
```

Listing 2.2: Car Attendant Automobile Thread - Key Snippet

This snippet shows the attendant waiting for a signal from an owner thread through the `inChargeforAutomobile` semaphore. After parking, the attendant signals the owner thread using the `newAutomobile` semaphore.

Chapter 3

Implementation Details

This chapter provides a detailed description of the shared resources and synchronization mechanisms used in the parking lot simulation.

3.1 Shared Resources

The parking lot simulation utilizes several shared resources that are accessed and modified by multiple threads. These resources must be managed carefully to prevent race conditions and ensure data consistency.

Here's a breakdown of the shared resources:

* **Real Parking Spots:** * `mFree_automobile` (integer): Represents the number of available parking spots for automobiles. This is initialized to 8. * `mFree_pickup` (integer): Represents the number of available parking spots for trucks (pickups). This is initialized to 4. * **Temporary Parking Spots:** * `tempFree_automobile` (integer): Represents the number of available temporary parking spots for automobiles. This is initialized to 8. * `tempFree_pickup` (integer): Represents the number of available temporary spots for trucks. This is initialized to 4. * **Total Free Spots:** * `totalFreeSpots` (integer): Represents the total number of free parking spots. This is calculated by summing the number of real and temporary parking spots for both automobiles and trucks.

3.2 Synchronization Mechanisms

The following synchronization mechanisms are implemented to manage access to the shared resources and ensure that threads operate correctly and without interfering with each other:

* **Semaphores:** Semaphores are integer variables that act as signals. They allow threads to wait for a resource to become available and signal when a resource is released. * `newPickup`: This semaphore is initialized to the number of temporary parking spots available for trucks. It signals when a temporary parking spot for a truck becomes available. A car owner thread will decrement

this semaphore when it occupies a temporary spot, and increment it when the spot is vacated. * `inChargeforPickup`: This semaphore is initialized to 0. It is used to signal an attendant thread that a truck is waiting in a temporary spot and ready to be moved to a real parking spot. A car owner thread will increment this semaphore after occupying a temporary spot, and an attendant thread will decrement it before parking the truck. * `newAutomobile`: This semaphore functions like `newPickup`, but controls access to temporary parking spots for automobiles. * `inChargeforAutomobile`: This semaphore functions like `inChargeforPickup`, but controls the parking process for automobiles. * `entrance`: This semaphore is initialized to 1. It ensures that only one vehicle enters the parking lot at a time. A car owner thread will decrement this semaphore before entering the parking lot and increment it after leaving. * **Mutex Lock:** A mutex lock (`mutex`) is used to protect the shared variables (`mFree_automobile`, `mFree_pickup`, `tempFree_automobile`, `tempFree_pickup`, and `totalFreeSpots`) from race conditions. Only one thread can hold the mutex lock at any time, ensuring that changes to these variables are atomic and consistent.

Chapter 4

Thread Functions

The simulation involves three different types of threads:

4.1 carOwner(void *arg):

This function simulates the behavior of a car owner. It takes the car type as an argument (0 for an automobile, 1 for a truck).

* **Arrival and Temporary Parking:** The owner thread waits for a random amount of time, simulating the arrival of a vehicle. It then acquires the `entrance` semaphore to ensure exclusive access to the parking lot. The thread then checks if a temporary parking spot is available for its vehicle type. If a temporary spot is available, the owner occupies it, decrements the corresponding temporary spot counter, and signals the appropriate attendant thread by incrementing the `inChargefor...` semaphore. * **Waiting and Departure:** The owner thread then waits for the corresponding `new...` semaphore to be signaled, indicating that the attendant has successfully parked the vehicle. Once the semaphore is signaled, the owner thread releases the `entrance` semaphore, allowing other vehicles to enter the parking lot, and then exits. If no temporary parking spot is available, the owner thread leaves the parking lot without parking and releases the `entrance` semaphore.

4.2 carAttendantAutomobile(void *arg):

This function simulates the actions of an attendant responsible for parking automobiles.

* **Waiting for Signal:** The attendant thread waits for a signal from an owner thread through the `inChargeforAutomobile` semaphore. This signal indicates that an automobile is waiting in a temporary parking spot. * **Parking Procedure:** Upon receiving the signal, the attendant checks if a real parking spot is available for automobiles. If a spot is found, the attendant decrements the `mFree_automobile` counter, increments the `tempFree_automobile` counter, and

signals the waiting owner thread by incrementing the `newAutomobile` semaphore. If no real parking spot is available, the attendant waits for a short period of time and then releases the `inChargeforAutomobile` semaphore, allowing the owner thread to continue waiting for a spot to become available. * **Thread Termination:** The `carAttendantAutomobile` thread continues this process until all temporary and real parking spots for automobiles are occupied. At this point, the thread prints a message indicating its termination and exits.

4.3 carAttendantPickup(void *arg):

This function operates similarly to `carAttendantAutomobile`, but it handles trucks instead of automobiles.

Chapter 5

Main Function

The `main` function orchestrates the entire simulation process.

5.1 Initialization:

The `main` function initializes the semaphores with their initial values, sets up the shared variables, calculates the `totalFreeSpots`, and creates the `carAttendantAutomobile` and `carAttendantPickup` threads.

5.2 Car Owner Simulation:

The `main` function enters a loop that continues as long as there are free parking spots available (`totalFreeSpots` ¿ 0). In each iteration, the loop randomly generates a car type (0 for an automobile, 1 for a truck) and creates a new `carOwner` thread to simulate the arrival of a vehicle of that type. The loop waits for each `carOwner` thread to complete before proceeding to the next iteration.

5.3 Attendant Thread Termination and Cleanup:

Once all parking spots are occupied, the loop terminates. The `main` function then waits for the `carAttendantAutomobile` and `carAttendantPickup` threads to complete, indicating that the parking lot is closed. Finally, the function destroys all semaphores and the mutex lock to clean up resources.

Chapter 6

Simulation Logic and Control Flow

The simulation follows this general flow:

6.1 Initialization:

Initialize semaphores, shared variables, and create attendant threads.

6.2 Car Arrival:

Create a new `carOwner` thread for each arriving vehicle. The vehicle type is randomly assigned.

6.3 Temporary Parking:

The `carOwner` thread checks for an available temporary parking spot for its vehicle type and occupies a spot if available. The owner signals the appropriate attendant thread to initiate the parking process.

6.4 Attendant Action:

The corresponding attendant thread waits for a signal from a `carOwner` thread. It then attempts to park the vehicle in a real parking spot if available, updating the shared counters accordingly.

6.5 Owner Departure:

The `carOwner` thread is notified when its vehicle is parked and subsequently exits. If no temporary parking was available, the owner leaves without parking.

6.6 Parking Lot Closure:

The simulation continues until all parking spots are filled. Attendant threads exit, and the parking lot is closed.

Chapter 7

Running the Code

This chapter provides instructions on how to compile and run the parking lot simulation code.

7.1 Compilation

To compile the code, you will need a C compiler (such as GCC) and the necessary libraries for thread management and shared memory. The exact compilation command will depend on your system and compiler. However, a common command should like in makefile:

`make`

`This` command assumes your code is in a file named ‘main.c‘. It links the necessary libraries (‘-pthread‘ for threads) and creates an executable named ‘main‘.

7.2 Execution

Once compiled, you can run the simulation with the following commands:

`./main` or `make run`

The program will execute, simulating the arrival and parking of vehicles. The output will display status messages indicating the progress of the simulation.

7.3 Notes

* The code may require adjustments based on your specific operating system and compiler. * You may need to modify the code to increase or decrease the number of parking spots, adjust arrival rates, or modify other simulation parameters.

Chapter 8

Conclusion

This implementation effectively simulates the operation of a parking lot with limited capacity, utilizing semaphores and a mutex lock for synchronization and shared memory for resource management. The detailed explanations provided offer a comprehensive understanding of the program's logic, thread interactions, and overall functionality.

Chapter 9

Expected Output and Testing

The program outputs status messages throughout its execution, indicating the arrival of vehicles, parking actions, and the availability of parking spots. The expected output should demonstrate:

9.1 Proper Synchronization of Threads:

No race conditions or inconsistencies in shared variables.

9.2 Accurate Tracking of Available Parking Spots:

Correct updates to the `mFree_automobile`, `mFree_pickup`, `tempFree_automobile`, `tempFree_pickup`, and `totalFreeSpots` variables.

9.3 Correct Use of Semaphores and Mutex Locks:

Threads wait for and signal each other correctly, ensuring coordinated access to shared resources.

9.4 Result Images

You can see images of the sample running program in following pages:

```
rm -f *.o main
gcc -pthread -Wall -c main.c
gcc -pthread -Wall -o main main.o
(base) root@DESKTOP-D58326S:/home/c# valgrind ./main
==2267== Memcheck, a memory error detector
==2267== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2267== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2267== Command: ./main
==2267==
Automobile owner arrived. Remaining temporary automobile spots: 7
Attendant parking an automobile.
Pickup owner arrived. Remaining temporary pickup spots: 3
Attendant parking a pickup.
Automobile parked. Remaining real automobile spots: 7, Remaining temporary automobile spots: 8
Automobile owner arrived. Remaining temporary automobile spots: 7
Attendant parking an automobile.
Pickup parked. Remaining real pickup spots: 3, Remaining temporary pickup spots: 4
Automobile parked. Remaining real automobile spots: 6, Remaining temporary automobile spots: 8
Automobile owner arrived. Remaining temporary automobile spots: 7
Attendant parking an automobile.
Automobile parked. Remaining real automobile spots: 5, Remaining temporary automobile spots: 8
Pickup owner arrived. Remaining temporary pickup spots: 3
Attendant parking a pickup.
Automobile owner arrived. Remaining temporary automobile spots: 7
Attendant parking an automobile.
Pickup parked. Remaining real pickup spots: 2, Remaining temporary pickup spots: 4
Automobile parked. Remaining real automobile spots: 4, Remaining temporary automobile spots: 8
Automobile owner arrived. Remaining temporary automobile spots: 7
Attendant parking an automobile.
Automobile parked. Remaining real automobile spots: 3, Remaining temporary automobile spots: 8
Automobile owner arrived. Remaining temporary automobile spots: 7
Attendant parking an automobile.
Automobile owner arrived. Remaining temporary automobile spots: 6
Automobile parked. Remaining real automobile spots: 2, Remaining temporary automobile spots: 7
Attendant parking an automobile.
Automobile owner arrived. Remaining temporary automobile spots: 6
Automobile parked. Remaining real automobile spots: 1, Remaining temporary automobile spots: 7
Attendant parking an automobile.
Automobile owner arrived. Remaining temporary automobile spots: 6
Automobile parked. Remaining real automobile spots: 0, Remaining temporary automobile spots: 7
Attendant parking an automobile.
Pickup owner arrived. Remaining temporary pickup spots: 3
Attendant parking a pickup.
No place to park the automobile. Car attendant is waiting a bit for some automobile to leave.
Pickup owner arrived. Remaining temporary pickup spots: 2
Pickup parked. Remaining real pickup spots: 1, Remaining temporary pickup spots: 3
Attendant parking a pickup.
Pickup owner arrived. Remaining temporary pickup spots: 2
Pickup parked. Remaining real pickup spots: 0, Remaining temporary pickup spots: 3
Attendant parking a pickup.
Automobile owner arrived. Remaining temporary automobile spots: 6
```

```
Attendant parking an automobile.
Automobile owner arrived. Remaining temporary automobile spots: 6
Automobile parked. Remaining real automobile spots: 1, Remaining temporary automobile spots: 7
Attendant parking an automobile.
Automobile owner arrived. Remaining temporary automobile spots: 6
Automobile parked. Remaining real automobile spots: 0, Remaining temporary automobile spots: 7
Attendant parking an automobile.
Pickup owner arrived. Remaining temporary pickup spots: 3
Attendant parking a pickup.
No place to park the automobile. Car attendant is waiting a bit for some automobile to leave.
Pickup owner arrived. Remaining temporary pickup spots: 2
Pickup parked. Remaining real pickup spots: 1, Remaining temporary pickup spots: 3
Attendant parking a pickup.
Pickup owner arrived. Remaining temporary pickup spots: 2
Pickup parked. Remaining real pickup spots: 0, Remaining temporary pickup spots: 3
Attendant parking a pickup.
Automobile owner arrived. Remaining temporary automobile spots: 6
No place to park the pickup. Car attendant is waiting some a bit to some pickup leave.
Automobile owner arrived. Remaining temporary automobile spots: 5
Attendant parking an automobile.
No place to park the automobile. Car attendant is waiting a bit for some automobile to leave.
Pickup owner arrived. Remaining temporary pickup spots: 2
Automobile owner arrived. Remaining temporary automobile spots: 4
Attendant parking a pickup.
Automobile owner arrived. Remaining temporary automobile spots: 3
Pickup owner arrived. Remaining temporary pickup spots: 1
No place to park the pickup. Car attendant is waiting some a bit to some pickup leave.
Attendant parking an automobile.
Automobile owner arrived. Remaining temporary automobile spots: 2
No place to park the automobile. Car attendant is waiting a bit for some automobile to leave.
Pickup owner arrived. Remaining temporary pickup spots: 0
Automobile owner arrived. Remaining temporary automobile spots: 1
All temp and real parking spots for pickups are full. Car attendant Pickup thread is ending.
No temporary spot available for pickup. Owner leaves.
Attendant parking an automobile.
No temporary spot available for pickup. Owner leaves.
No place to park the automobile. Car attendant is waiting a bit for some automobile to leave.
No temporary spot available for pickup. Owner leaves.
Automobile owner arrived. Remaining temporary automobile spots: 0
All temp and real parking spots for automobiles are full. Car attendant Automobile thread is ending.
All parking spots are full. The parking system is closing.
==2267==
==2267== HEAP SUMMARY:
==2267==     in use at exit: 0 bytes in 0 blocks
==2267==   total heap usage: 11 allocs, 11 frees, 5,942 bytes allocated
==2267==
==2267== All heap blocks were freed -- no leaks are possible
==2267==
==2267== For lists of detected and suppressed errors, rerun with: -s
==2267== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
(base) root@DESKTOP-D58326S:/home/c#
```