CSE 344 – SYSTEM PROGRAMMING

HOMEWORK 3 REPORT

SEMAPHORE AND SHARED MEMORY

Emre YAVUZ

200104004003

Introduction

The objective of this homework is to simulate a parking lot system using semaphores and shared memory to manage parking spots for different types of vehicles, specifically automobiles and pickups. The simulation involves creating threads to represent car owners and parking attendants, ensuring proper synchronization to handle vehicle parking. This project aims to provide practical experience with thread synchronization and semaphores, essential concepts in systems programming. Through this exercise, we will understand how to manage concurrent processes and ensure data integrity in a multithreaded environment.

Problem Statement

This homework requires simulating a parking lot system with specific constraints and synchronization requirements.

Requirements:

- **Vehicle Types: The simulation involves two types of vehicles: automobiles and pickups.**
- **Parking Spots: The parking lot has separate spots for each vehicle type: 8 spots for automobiles and 4 spots for pickups.**
- **Synchronization: Use semaphores to synchronize the actions of car owners and attendants.**
- **Threads: Implement two main functions, carOwner and carAttendant, to simulate the behavior of vehicle owners and parking attendants.**

Constraints:

- **Ensure that only one vehicle can enter the parking system at a time.**

- **Use semaphores named newPickup, inChargeforPickup, newAutomobile, and inChargeforAutomobile.**
- **The simulation should include mechanisms to confirm parking availability and manage the temporary parking spots correctly.**

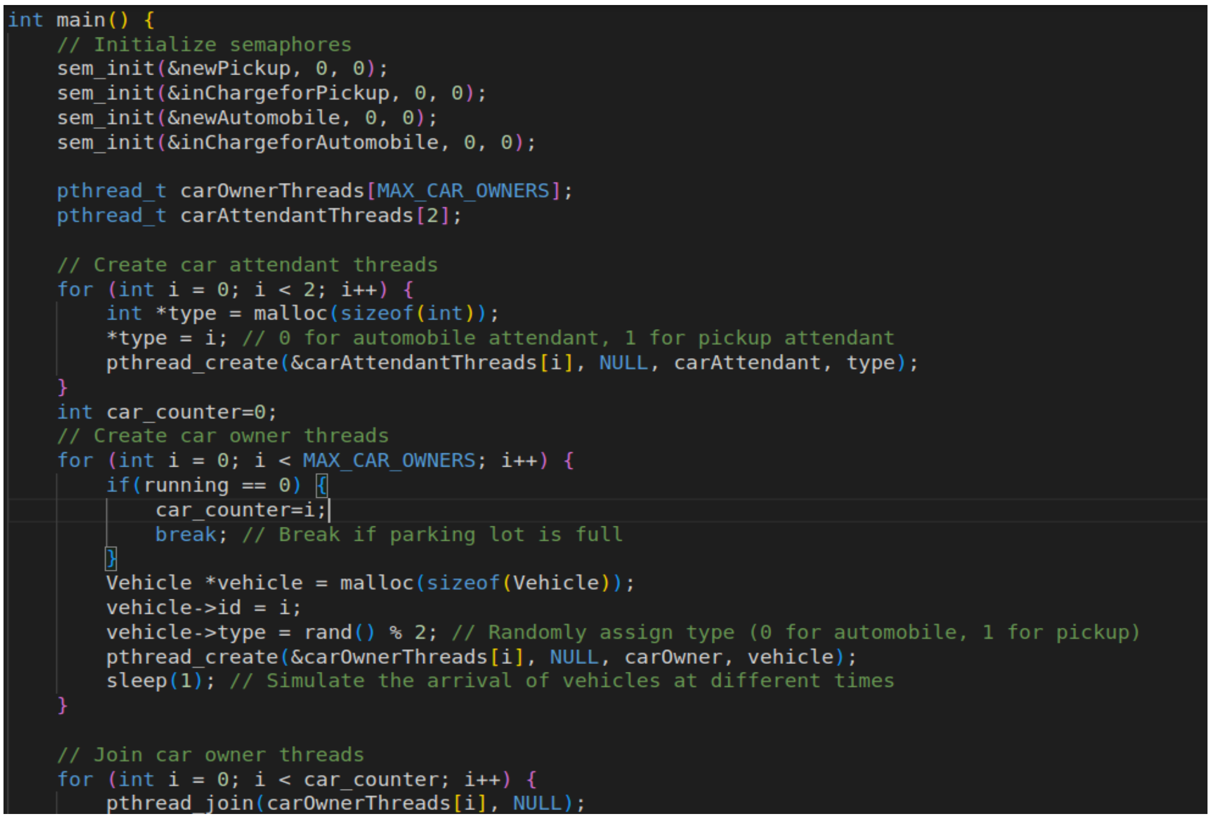
Design and Implementation

The design includes multiple threads for car owners, representing incoming vehicles (pickup or automobile), and separate threads for automobile and pickup attendants. Shared variables manage the available parking spots, and a terminate program variable checks if all the parking spaces are full. Semaphores ensure only one vehicle can enter the parking area to be parked by an attendant at a time.

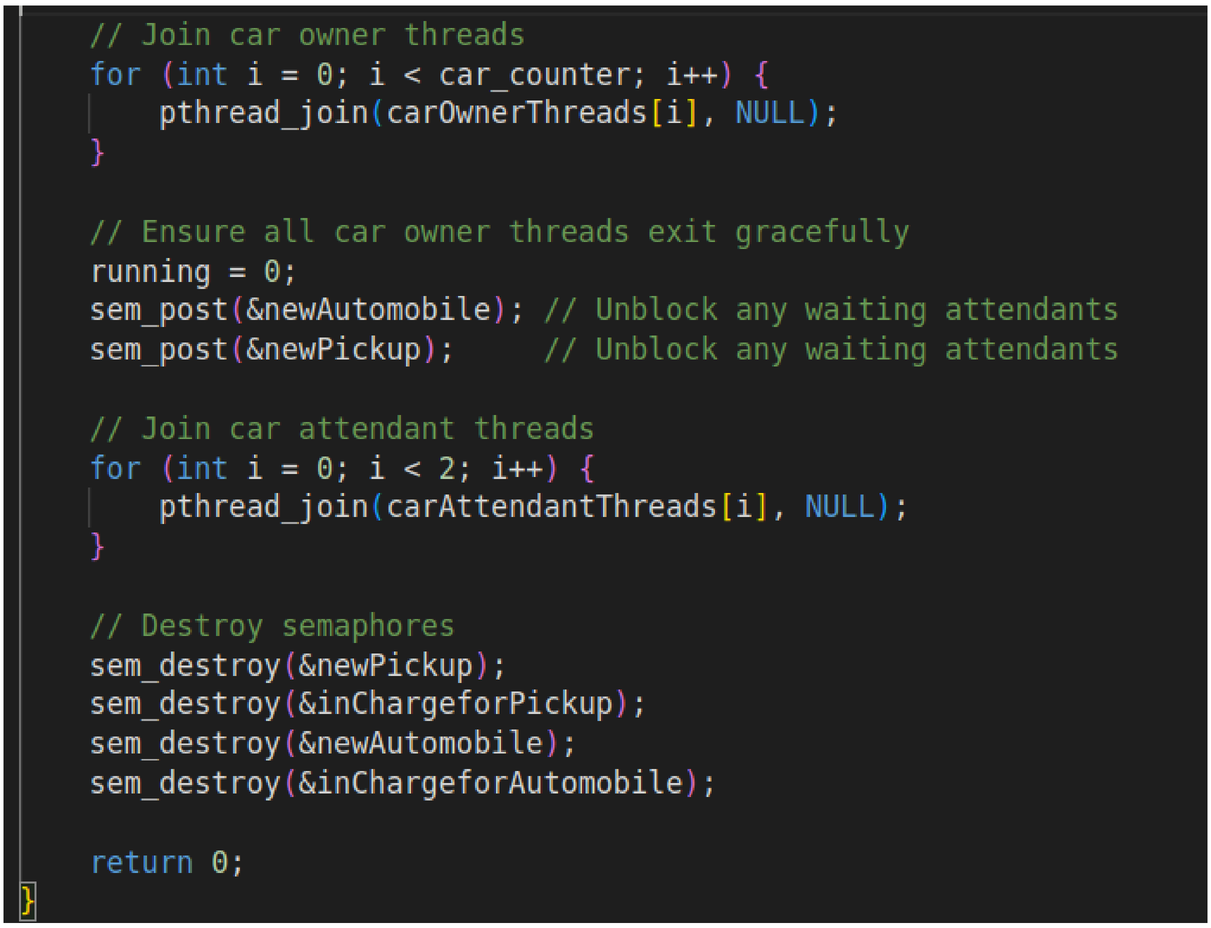
Initialization

```c
// Constants for the capacity of the parking lot
#define PICKUP_CAPACITY 4
#define AUTOMOBILE_CAPACITY 8

#define MAX_CAR_OWNERS 40

// Semaphores for synchronization
sem_t newPickup;
sem_t inChargeforPickup;
sem_t newAutomobile;
sem_t inChargeforAutomobile;

// Counter variables for free parking spots
int mFree_pickup = PICKUP_CAPACITY;
int mFree_automobile = AUTOMOBILE_CAPACITY;

// Counter variables for total parked vehicles
int total_pickup = 0;
int total_automobile = 0;

// Mutexes to protect the counter variables
pthread_mutex_t pickupMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t automobileMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t print_mutex = PTHREAD_MUTEX_INITIALIZER;

volatile int running = 1; // Global flag to control running state of attendants
```

Semaphore Initialization and Thread Creation



```c
int main() {
    // Initialize semaphores
    sem_init(&newPickup, 0, 0);
    sem_init(&inChargeforPickup, 0, 0);
    sem_init(&newAutomobile, 0, 0);
    sem_init(&inChargeforAutomobile, 0, 0);

    pthread_t carOwnerThreads[MAX_CAR_OWNERS];
    pthread_t carAttendantThreads[2];

    // Create car attendant threads
    for (int i = 0; i < 2; i++) {
        int *type = malloc(sizeof(int));
        *type = i; // 0 for automobile attendant, 1 for pickup attendant
        pthread_create(&carAttendantThreads[i], NULL, carAttendant, type);
    }
    int car_counter=0;
    // Create car owner threads
    for (int i = 0; i < MAX_CAR_OWNERS; i++) {
        if(running == 0) {
            car_counter=i;
            break; // Break if parking lot is full
        }
        Vehicle *vehicle = malloc(sizeof(Vehicle));
        vehicle->id = i;
        vehicle->type = rand() % 2; // Randomly assign type (0 for automobile, 1 for pickup)
        pthread_create(&carOwnerThreads[i], NULL, carOwner, vehicle);
        sleep(1); // Simulate the arrival of vehicles at different times
    }

    // Join car owner threads
    for (int i = 0; i < car_counter; i++) {
        pthread_join(carOwnerThreads[i], NULL);
```

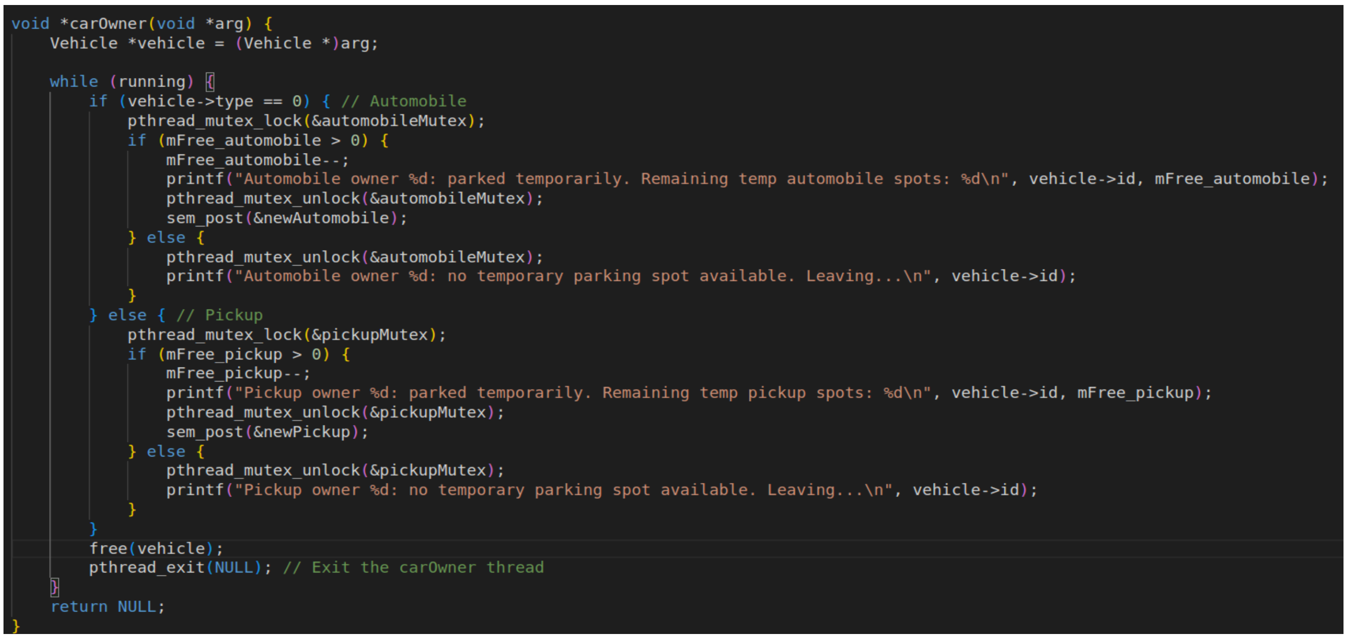


```c
    // Join car owner threads
    for (int i = 0; i < car_counter; i++) {
        pthread_join(carOwnerThreads[i], NULL);
    }

    // Ensure all car owner threads exit gracefully
    running = 0;
    sem_post(&newAutomobile); // Unblock any waiting attendants
    sem_post(&newPickup);     // Unblock any waiting attendants

    // Join car attendant threads
    for (int i = 0; i < 2; i++) {
        pthread_join(carAttendantThreads[i], NULL);
    }

    // Destroy semaphores
    sem_destroy(&newPickup);
    sem_destroy(&inChargeforPickup);
    sem_destroy(&newAutomobile);
    sem_destroy(&inChargeforAutomobile);

    return 0;
}
```

carOwner Function

The carOwner function is used to simulate the actions of car owners arriving at the parking lot. Each carOwner thread represents an individual car owner attempting to park their vehicle. Here's how it works:

- **Thread Creation and Arrival Simulation: The function simulates the arrival of a new car owner thread by creating a vehicle structure. The type of vehicle (automobile or pickup) is assigned randomly. This simulates the random arrival of different types of vehicles at the parking lot.**
- **Parking Attempt: The car owner checks for available temporary parking spots based on the type of vehicle.**
    - **If the vehicle is an automobile (vehicle->type == 0), the function locks the automobileMutex to safely access and modify the shared variable mFree_automobile.**
    - **If there is an available spot (mFree_automobile > 0), the car owner parks the vehicle temporarily by decrementing mFree_automobile and prints the remaining spots. It then signals the newAutomobile semaphore to notify the attendant that a vehicle is ready to be parked.**
    - **If there is no available spot, the car owner prints a message indicating that no temporary parking spot is available and leaves.**
- **Semaphore Usage: The function uses semaphores to synchronize the decrement of shared variables safely, ensuring that only one thread can modify the parking spot counters at a time.**
- **Termination Check: The function checks if the parking lot is full by calling the check_termination_condition function after each modification.**
- **Memory Management and Exit: The function frees the allocated memory for the vehicle structure and exits the thread using pthread_exit(NULL).**



```c
void *carOwner(void *arg) {
    Vehicle *vehicle = (Vehicle *)arg;

    while (running) {
        if (vehicle->type == 0) { // Automobile
            pthread_mutex_lock(&automobileMutex);
            if (mFree_automobile > 0) {
                mFree_automobile--;
                printf("Automobile owner %d: parked temporarily. Remaining temp automobile spots: %d\n", vehicle->id, mFree_automobile);
                pthread_mutex_unlock(&automobileMutex);
                sem_post(&newAutomobile);
            } else {
                pthread_mutex_unlock(&automobileMutex);
                printf("Automobile owner %d: no temporary parking spot available. Leaving...\n", vehicle->id);
            }
        } else { // Pickup
            pthread_mutex_lock(&pickupMutex);
            if (mFree_pickup > 0) {
                mFree_pickup--;
                printf("Pickup owner %d: parked temporarily. Remaining temp pickup spots: %d\n", vehicle->id, mFree_pickup);
                pthread_mutex_unlock(&pickupMutex);
                sem_post(&newPickup);
            } else {
                pthread_mutex_unlock(&pickupMutex);
                printf("Pickup owner %d: no temporary parking spot available. Leaving...\n", vehicle->id);
            }
        }
        free(vehicle);
        pthread_exit(NULL); // Exit the carOwner thread
    }
    return NULL;
}
```

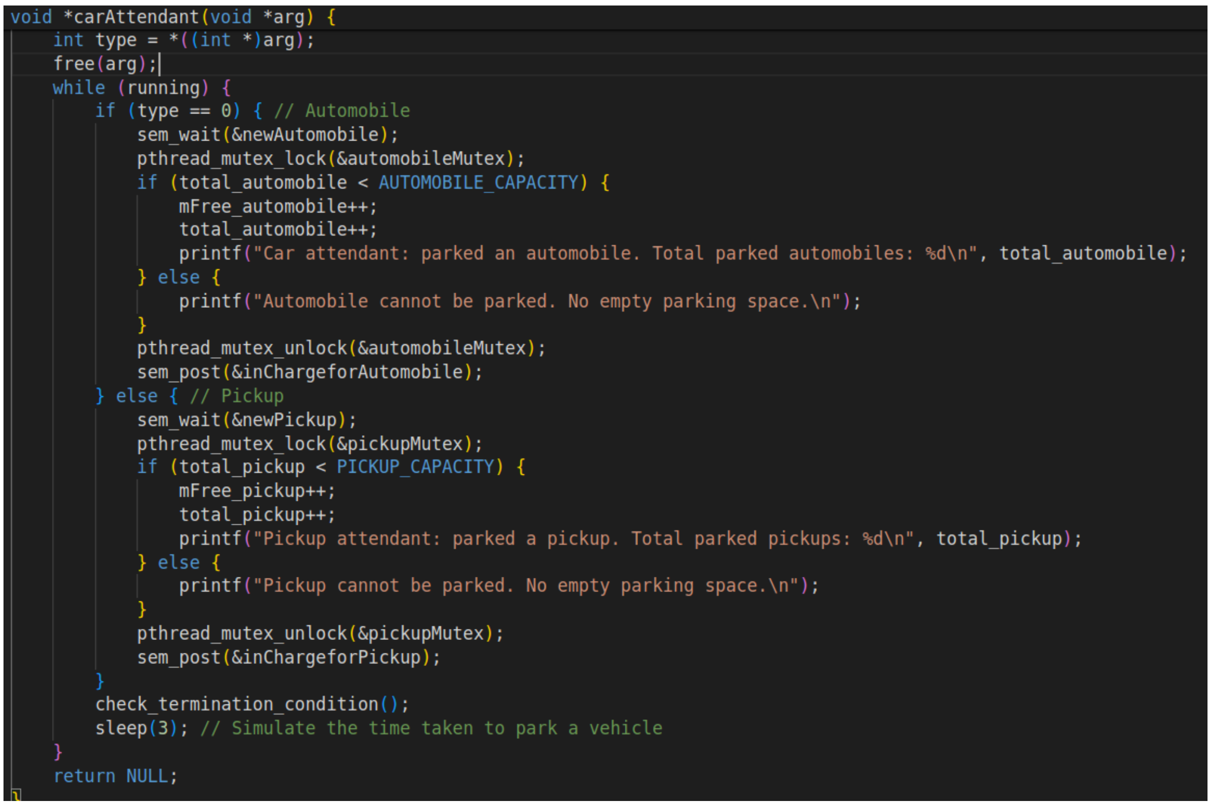
carAttendant Function

The **carAttendant** function is used to simulate the actions of parking attendants. There are separate threads for automobile attendants and pickup attendants. The vehicle type is passed as an argument to distinguish between the two types of attendants. Here's how it works:

- **Vehicle Handling:** The function first determines the type of vehicle it is responsible for (automobile or pickup) based on the argument passed.
    - **Automobile Attendant:** If the vehicle type is automobile (**type == 0**), the function waits on the **newAutomobile** semaphore to be signaled by a car owner thread. Once signaled, it locks the **automobileMutex** to safely access and modify the shared variables **mFree_automobile** and **total_automobile**.
        - If there is space available (**total_automobile < AUTOMOBILE_CAPACITY**), the attendant parks the vehicle by incrementing **mFree_automobile** and **total_automobile**, and prints the total parked automobiles.
        - If there is no space available, the attendant prints a message indicating that no empty parking space is available.
        - The function then signals the **inChargeforAutomobile** semaphore.
    - **Pickup Attendant:** If the vehicle type is pickup (**type == 1**), the function follows a similar process using the **newPickup** semaphore and **pickupMutex** to handle pickup vehicles.
        - If there is space available (**total_pickup < PICKUP_CAPACITY**), the attendant parks the vehicle by incrementing **mFree_pickup** and **total_pickup**, and prints the total parked pickups.

        - If there is no space available, the attendant prints a message indicating that no empty parking space is available.
        - The function then signals the **inChargeforPickup** semaphore.

- **Termination Check:** After parking a vehicle, the function calls the **check_termination_condition** function to check if the parking lot is full.
- **Sleep Simulation:** The function uses **sleep(3)** to simulate the time taken to park a vehicle.



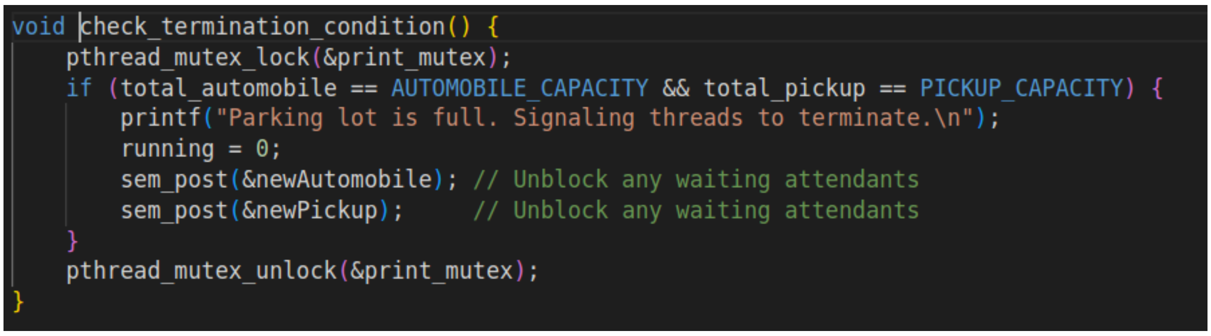
```
void *carAttendant(void *arg) {
    int type = *((int *)arg);
    free(arg);
    while (running) {
        if (type == 0) { // Automobile
            sem_wait(&newAutomobile);
            pthread_mutex_lock(&automobileMutex);
            if (total_automobile < AUTOMOBILE_CAPACITY) {
                mFree_automobile++;
                total_automobile++;
                printf("Car attendant: parked an automobile. Total parked automobiles: %d\n", total_automobile);
            } else {
                printf("Automobile cannot be parked. No empty parking space.\n");
            }
            pthread_mutex_unlock(&automobileMutex);
            sem_post(&inChargeforAutomobile);
        } else { // Pickup
            sem_wait(&newPickup);
            pthread_mutex_lock(&pickupMutex);
            if (total_pickup < PICKUP_CAPACITY) {
                mFree_pickup++;
                total_pickup++;
                printf("Pickup attendant: parked a pickup. Total parked pickups: %d\n", total_pickup);
            } else {
                printf("Pickup cannot be parked. No empty parking space.\n");
            }
            pthread_mutex_unlock(&pickupMutex);
            sem_post(&inChargeforPickup);
        }
        check_termination_condition();
        sleep(3); // Simulate the time taken to park a vehicle
    }
    return NULL;
}
```

check_termination_condition Function

The **check_termination_condition** function checks if all the parking spots are full and signals the program to terminate if so. Here's how it works:

- **Mutex Locking:** The function locks the **print_mutex** to ensure that the check and the print statement are executed atomically, preventing race conditions.
- **Full Parking Lot Check:** The function checks if the total number of parked automobiles has reached the automobile capacity (**total_automobile == AUTOMOBILE_CAPACITY**) and the total number of parked pickups has reached the pickup capacity (**total_pickup == PICKUP_CAPACITY**).

- **Signaling Termination:** If both conditions are met, the function prints a message indicating that the parking lot is full and sets the global **running** flag to **0**. It then signals the **newAutomobile** and **newPickup** semaphores to unblock any waiting attendants, allowing them to exit gracefully.
- **Mutex Unlocking:** Finally, the function unlocks the **print_mutex**.



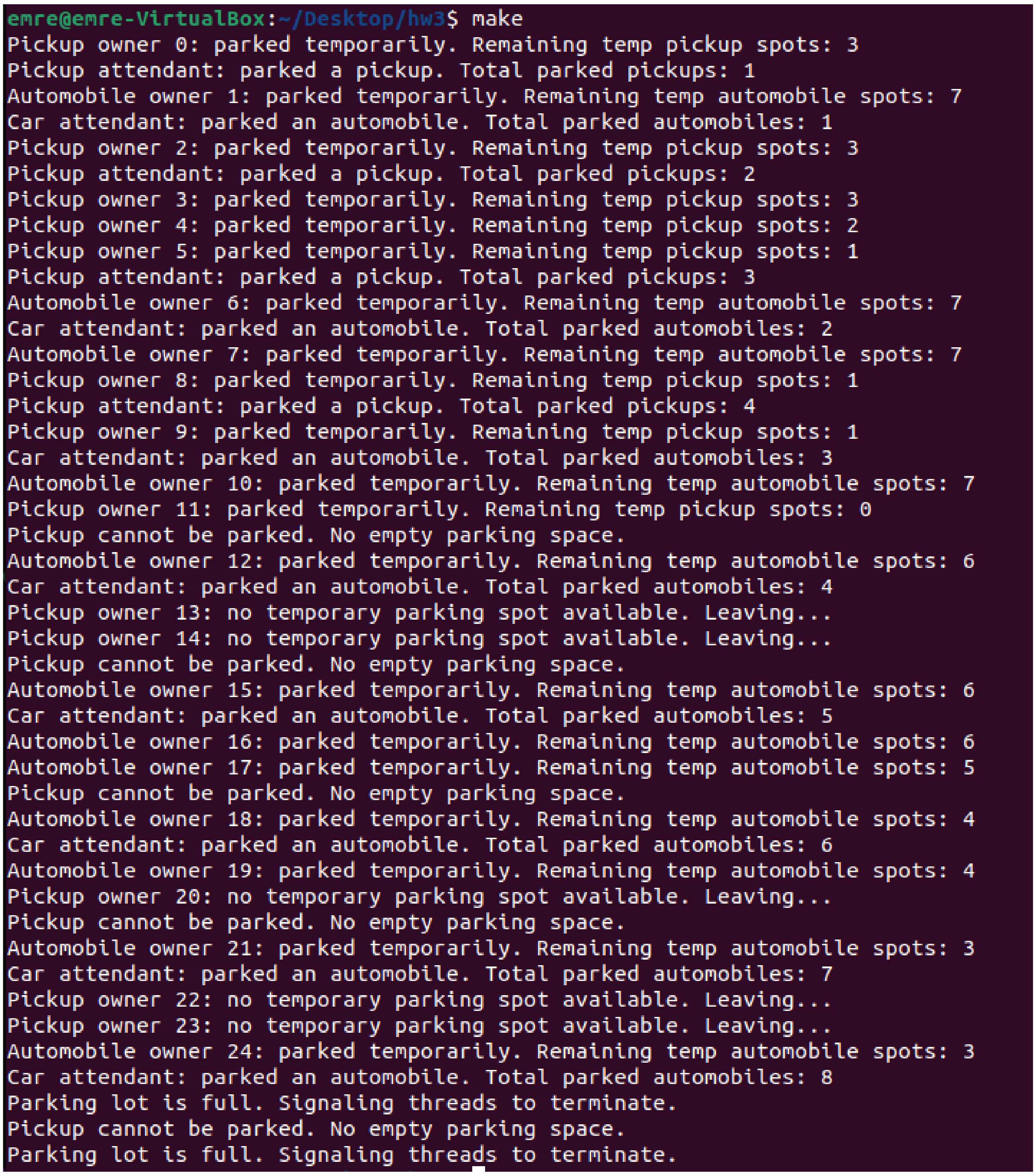
```c
void check_termination_condition() {
    pthread_mutex_lock(&print_mutex);
    if (total_automobile == AUTOMOBILE_CAPACITY && total_pickup == PICKUP_CAPACITY) {
        printf("Parking lot is full. Signaling threads to terminate.\n");
        running = 0;
        sem_post(&newAutomobile); // Unblock any waiting attendants
        sem_post(&newPickup);     // Unblock any waiting attendants
    }
    pthread_mutex_unlock(&print_mutex);
}
```

Main Flow of the Program

The program keeps accepting new vehicles and parks the vehicles until all the space is occupied. The attendants park the vehicles until there are cars to park and the parking space for that vehicle is not full. When all the park space is full, the program terminates.

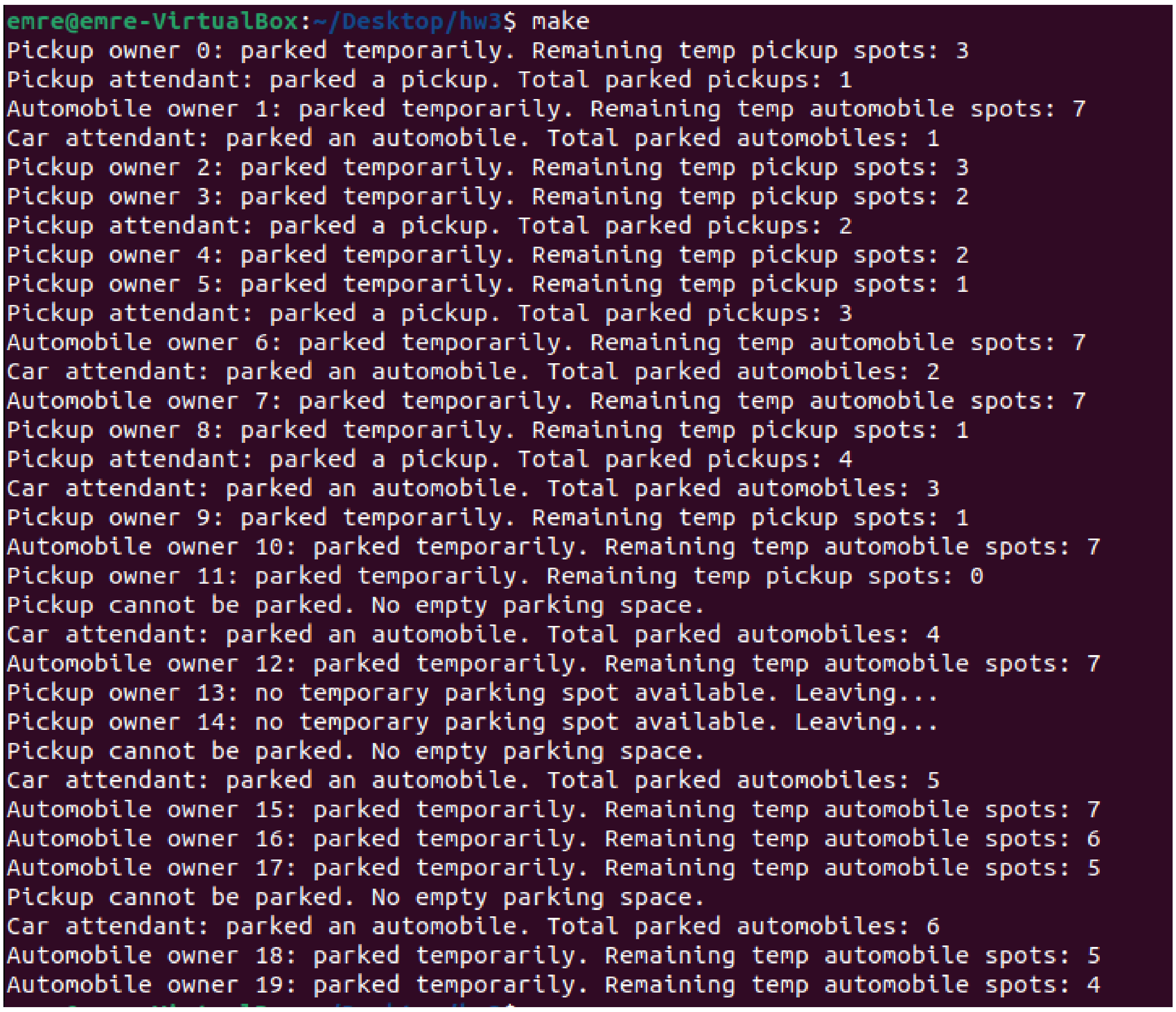
SAMPLE CASES :

MAX_CAR_OWNERS = 40



```
emre@emre-VirtualBox:~/Desktop/hw3$ make
Pickup owner 0: parked temporarily. Remaining temp pickup spots: 3
Pickup attendant: parked a pickup. Total parked pickups: 1
Automobile owner 1: parked temporarily. Remaining temp automobile spots: 7
Car attendant: parked an automobile. Total parked automobiles: 1
Pickup owner 2: parked temporarily. Remaining temp pickup spots: 3
Pickup attendant: parked a pickup. Total parked pickups: 2
Pickup owner 3: parked temporarily. Remaining temp pickup spots: 3
Pickup owner 4: parked temporarily. Remaining temp pickup spots: 2
Pickup owner 5: parked temporarily. Remaining temp pickup spots: 1
Pickup attendant: parked a pickup. Total parked pickups: 3
Automobile owner 6: parked temporarily. Remaining temp automobile spots: 7
Car attendant: parked an automobile. Total parked automobiles: 2
Automobile owner 7: parked temporarily. Remaining temp automobile spots: 7
Pickup owner 8: parked temporarily. Remaining temp pickup spots: 1
Pickup attendant: parked a pickup. Total parked pickups: 4
Pickup owner 9: parked temporarily. Remaining temp pickup spots: 1
Car attendant: parked an automobile. Total parked automobiles: 3
Automobile owner 10: parked temporarily. Remaining temp automobile spots: 7
Pickup owner 11: parked temporarily. Remaining temp pickup spots: 0
Pickup cannot be parked. No empty parking space.
Automobile owner 12: parked temporarily. Remaining temp automobile spots: 6
Car attendant: parked an automobile. Total parked automobiles: 4
Pickup owner 13: no temporary parking spot available. Leaving...
Pickup owner 14: no temporary parking spot available. Leaving...
Pickup cannot be parked. No empty parking space.
Automobile owner 15: parked temporarily. Remaining temp automobile spots: 6
Car attendant: parked an automobile. Total parked automobiles: 5
Automobile owner 16: parked temporarily. Remaining temp automobile spots: 6
Automobile owner 17: parked temporarily. Remaining temp automobile spots: 5
Pickup cannot be parked. No empty parking space.
Automobile owner 18: parked temporarily. Remaining temp automobile spots: 4
Car attendant: parked an automobile. Total parked automobiles: 6
Automobile owner 19: parked temporarily. Remaining temp automobile spots: 4
Pickup owner 20: no temporary parking spot available. Leaving...
Pickup cannot be parked. No empty parking space.
Automobile owner 21: parked temporarily. Remaining temp automobile spots: 3
Car attendant: parked an automobile. Total parked automobiles: 7
Pickup owner 22: no temporary parking spot available. Leaving...
Pickup owner 23: no temporary parking spot available. Leaving...
Automobile owner 24: parked temporarily. Remaining temp automobile spots: 3
Car attendant: parked an automobile. Total parked automobiles: 8
Parking lot is full. Signaling threads to terminate.
Pickup cannot be parked. No empty parking space.
Parking lot is full. Signaling threads to terminate.
```

MAX_CAR_OWNERS = 20



```
emre@emre-VirtualBox:~/Desktop/hw3$ make
Pickup owner 0: parked temporarily. Remaining temp pickup spots: 3
Pickup attendant: parked a pickup. Total parked pickups: 1
Automobile owner 1: parked temporarily. Remaining temp automobile spots: 7
Car attendant: parked an automobile. Total parked automobiles: 1
Pickup owner 2: parked temporarily. Remaining temp pickup spots: 3
Pickup owner 3: parked temporarily. Remaining temp pickup spots: 2
Pickup attendant: parked a pickup. Total parked pickups: 2
Pickup owner 4: parked temporarily. Remaining temp pickup spots: 2
Pickup owner 5: parked temporarily. Remaining temp pickup spots: 1
Pickup attendant: parked a pickup. Total parked pickups: 3
Automobile owner 6: parked temporarily. Remaining temp automobile spots: 7
Car attendant: parked an automobile. Total parked automobiles: 2
Automobile owner 7: parked temporarily. Remaining temp automobile spots: 7
Pickup owner 8: parked temporarily. Remaining temp pickup spots: 1
Pickup attendant: parked a pickup. Total parked pickups: 4
Car attendant: parked an automobile. Total parked automobiles: 3
Pickup owner 9: parked temporarily. Remaining temp pickup spots: 1
Automobile owner 10: parked temporarily. Remaining temp automobile spots: 7
Pickup owner 11: parked temporarily. Remaining temp pickup spots: 0
Pickup cannot be parked. No empty parking space.
Car attendant: parked an automobile. Total parked automobiles: 4
Automobile owner 12: parked temporarily. Remaining temp automobile spots: 7
Pickup owner 13: no temporary parking spot available. Leaving...
Pickup owner 14: no temporary parking spot available. Leaving...
Pickup cannot be parked. No empty parking space.
Car attendant: parked an automobile. Total parked automobiles: 5
Automobile owner 15: parked temporarily. Remaining temp automobile spots: 7
Automobile owner 16: parked temporarily. Remaining temp automobile spots: 6
Automobile owner 17: parked temporarily. Remaining temp automobile spots: 5
Pickup cannot be parked. No empty parking space.
Car attendant: parked an automobile. Total parked automobiles: 6
Automobile owner 18: parked temporarily. Remaining temp automobile spots: 5
Automobile owner 19: parked temporarily. Remaining temp automobile spots: 4
```