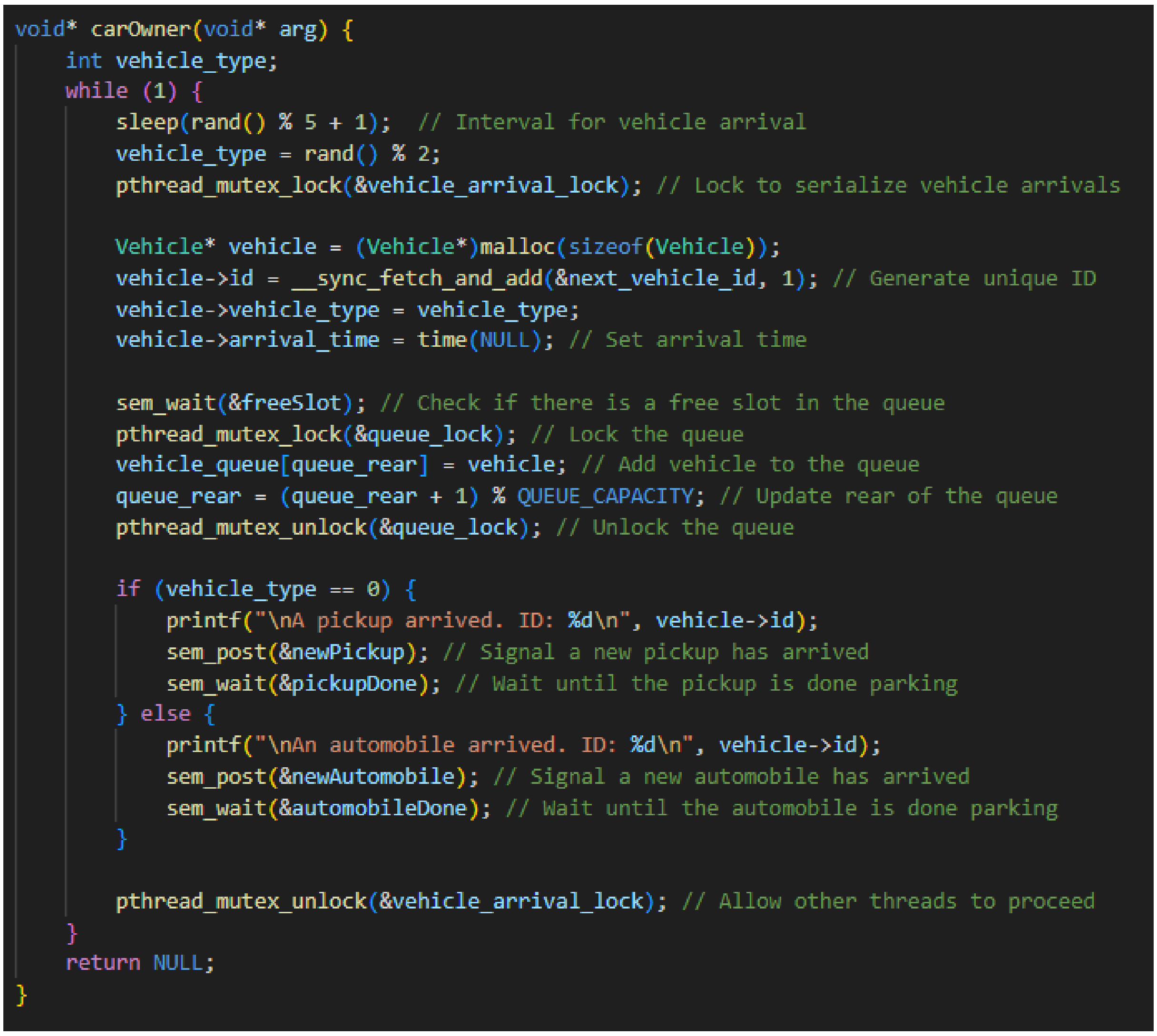
Makefile:

Make all => compile and run.

Make clean => deletes exe program.

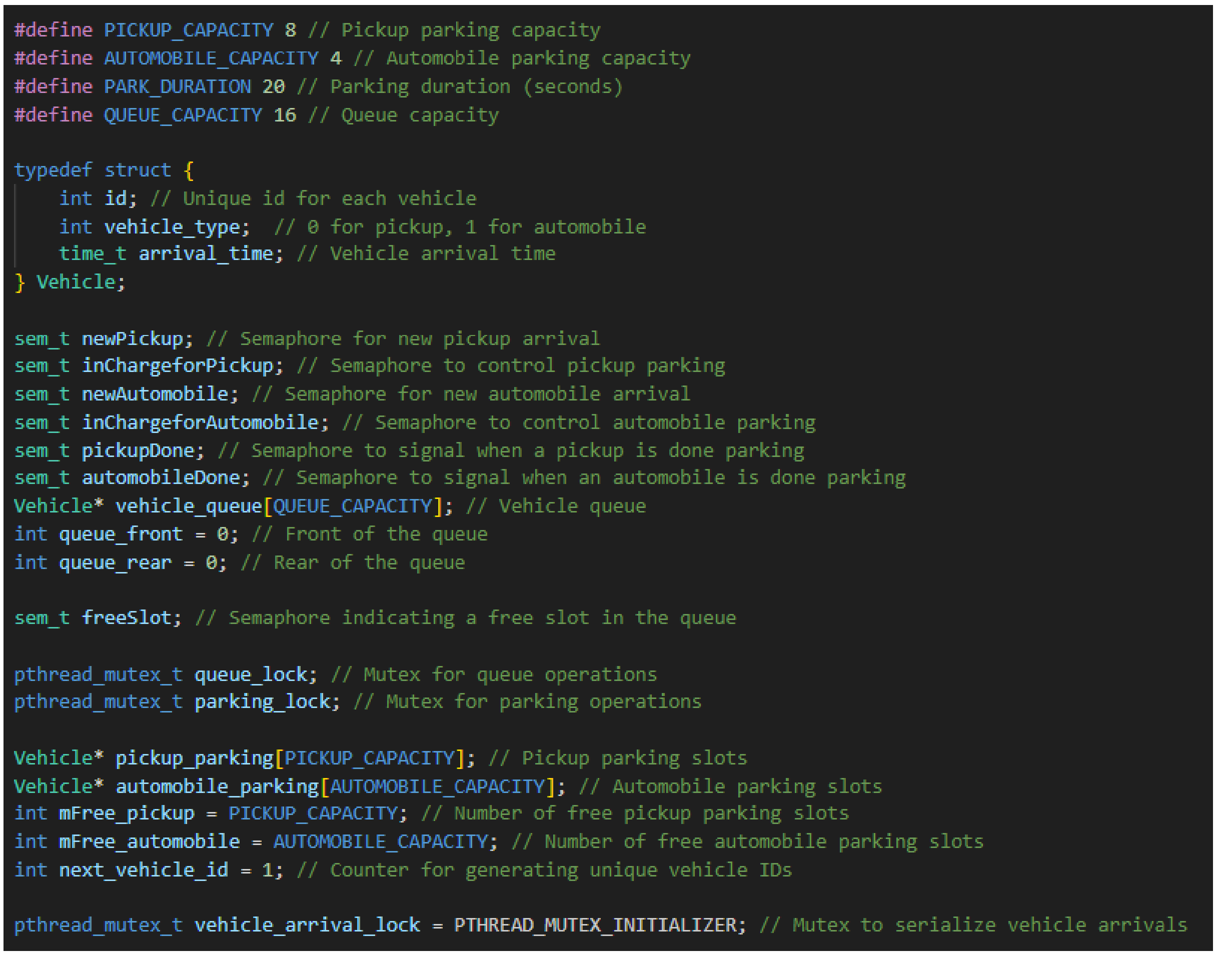
- CarOwner 2 Thread:



```c
void* carOwner(void* arg) {
    int vehicle_type;
    while (1) {
        sleep(rand() % 5 + 1);  // Interval for vehicle arrival
        vehicle_type = rand() % 2;
        pthread_mutex_lock(&vehicle_arrival_lock); // Lock to serialize vehicle arrivals

        Vehicle* vehicle = (Vehicle*)malloc(sizeof(Vehicle));
        vehicle->id = __sync_fetch_and_add(&next_vehicle_id, 1); // Generate unique ID
        vehicle->vehicle_type = vehicle_type;
        vehicle->arrival_time = time(NULL); // Set arrival time

        sem_wait(&freeSlot); // Check if there is a free slot in the queue
        pthread_mutex_lock(&queue_lock); // Lock the queue
        vehicle_queue[queue_rear] = vehicle; // Add vehicle to the queue
        queue_rear = (queue_rear + 1) % QUEUE_CAPACITY; // Update rear of the queue
        pthread_mutex_unlock(&queue_lock); // Unlock the queue

        if (vehicle_type == 0) {
            printf("\nA pickup arrived. ID: %d\n", vehicle->id);
            sem_post(&newPickup); // Signal a new pickup has arrived
            sem_wait(&pickupDone); // Wait until the pickup is done parking
        } else {
            printf("\nAn automobile arrived. ID: %d\n", vehicle->id);
            sem_post(&newAutomobile); // Signal a new automobile has arrived
            sem_wait(&automobileDone); // Wait until the automobile is done parking
        }

        pthread_mutex_unlock(&vehicle_arrival_lock); // Allow other threads to proceed
    }
    return NULL;
}
```

Here comes 2 thread vehicle owners. Vehicle types are selected randomly. Since one vehicle must enter, the vehicle_arrival_lock mutex prevents the other from entering. Vehicle type is created. I created a unique id for each vehicle. And I kept arrival_time for the vehicles to leave the parking lot. I set it as 20 seconds. Cars that exceed 20 seconds are removed from the parking lot by the attendants. And the arriving vehicles are waiting in a temporary queue. If there is a vehicle in the queue, the parking process starts for that vehicle. The queue is locked with mutex. Then that vehicle is removed from the queue. We used sem_post() to make the attendants work. vehicle_arrival_lock is unlocked when the last vehicle enters and the process is finished

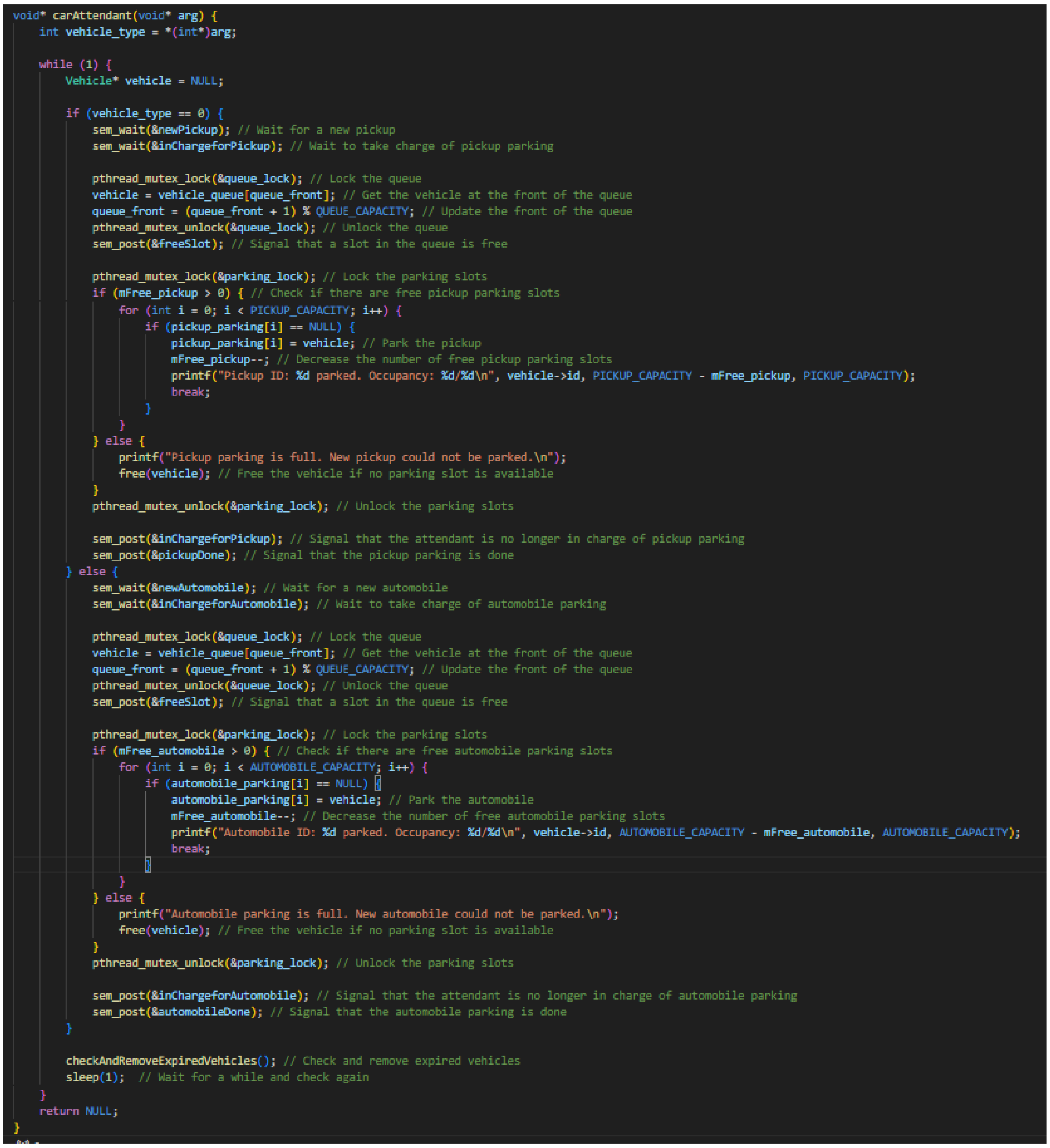
- Required Variables:



```c
#define PICKUP_CAPACITY 8 // Pickup parking capacity
#define AUTOMOBILE_CAPACITY 4 // Automobile parking capacity
#define PARK_DURATION 20 // Parking duration (seconds)
#define QUEUE_CAPACITY 16 // Queue capacity

typedef struct {
    int id; // Unique id for each vehicle
    int vehicle_type;  // 0 for pickup, 1 for automobile
    time_t arrival_time; // Vehicle arrival time
} Vehicle;

sem_t newPickup; // Semaphore for new pickup arrival
sem_t inChargeforPickup; // Semaphore to control pickup parking
sem_t newAutomobile; // Semaphore for new automobile arrival
sem_t inChargeforAutomobile; // Semaphore to control automobile parking
sem_t pickupDone; // Semaphore to signal when a pickup is done parking
sem_t automobileDone; // Semaphore to signal when an automobile is done parking
Vehicle* vehicle_queue[QUEUE_CAPACITY]; // Vehicle queue
int queue_front = 0; // Front of the queue
int queue_rear = 0; // Rear of the queue

sem_t freeSlot; // Semaphore indicating a free slot in the queue

pthread_mutex_t queue_lock; // Mutex for queue operations
pthread_mutex_t parking_lock; // Mutex for parking operations

Vehicle* pickup_parking[PICKUP_CAPACITY]; // Pickup parking slots
Vehicle* automobile_parking[AUTOMOBILE_CAPACITY]; // Automobile parking slots
int mFree_pickup = PICKUP_CAPACITY; // Number of free pickup parking slots
int mFree_automobile = AUTOMOBILE_CAPACITY; // Number of free automobile parking slots
int next_vehicle_id = 1; // Counter for generating unique vehicle IDs

pthread_mutex_t vehicle_arrival_lock = PTHREAD_MUTEX_INITIALIZER; // Mutex to serialize vehicle arrivals
```

I created a struct for the vehicle. I made a queue for the temporary parking lot. I used arrays of vehicle type for the parking lot of 8 - 4. When the Array is full, the vehicles are not parked. The stopping time of the vehicles in the parking lot is 20 seconds. The queue capacity consists of 16 vehicles.

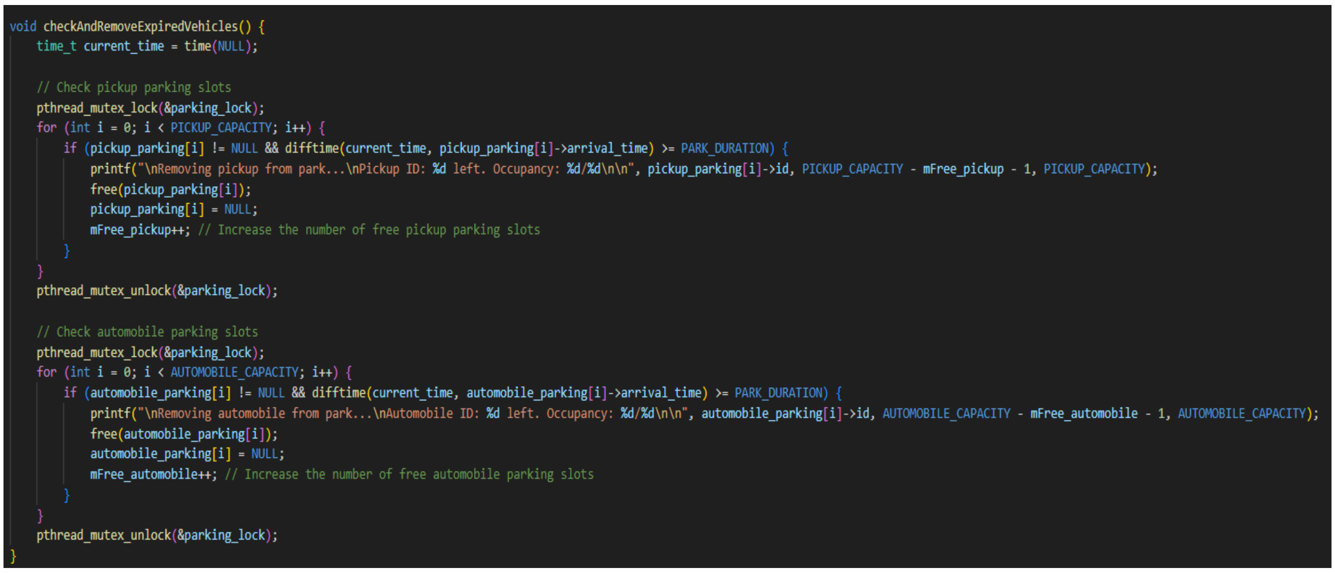
- CarAttendant 2 Thread:



```c
void* carAttendant(void* arg) {
    int vehicle_type = *(int*)arg;

    while (1) {
        Vehicle* vehicle = NULL;

        if (vehicle_type == 0) {
            sem_wait(&newPickup); // Wait for a new pickup
            sem_wait(&inChargeforPickup); // Wait to take charge of pickup parking

            pthread_mutex_lock(&queue_lock); // Lock the queue
            vehicle = vehicle_queue[queue_front]; // Get the vehicle at the front of the queue
            queue_front = (queue_front + 1) % QUEUE_CAPACITY; // Update the front of the queue
            pthread_mutex_unlock(&queue_lock); // Unlock the queue
            sem_post(&freeSlot); // Signal that a slot in the queue is free

            pthread_mutex_lock(&parking_lock); // Lock the parking slots
            if (mFree_pickup > 0) { // Check if there are free pickup parking slots
                for (int i = 0; i < PICKUP_CAPACITY; i++) {
                    if (pickup_parking[i] == NULL) {
                        pickup_parking[i] = vehicle; // Park the pickup
                        mFree_pickup--; // Decrease the number of free pickup parking slots
                        printf("Pickup ID: %d parked. Occupancy: %d/%d\n", vehicle->id, PICKUP_CAPACITY - mFree_pickup, PICKUP_CAPACITY);
                        break;
                    }
                }
            } else {
                printf("Pickup parking is full. New pickup could not be parked.\n");
                free(vehicle); // Free the vehicle if no parking slot is available
            }
            pthread_mutex_unlock(&parking_lock); // Unlock the parking slots

            sem_post(&inChargeforPickup); // Signal that the attendant is no longer in charge of pickup parking
            sem_post(&pickupDone); // Signal that the pickup parking is done
        } else {
            sem_wait(&newAutomobile); // Wait for a new automobile
            sem_wait(&inChargeforAutomobile); // Wait to take charge of automobile parking

            pthread_mutex_lock(&queue_lock); // Lock the queue
            vehicle = vehicle_queue[queue_front]; // Get the vehicle at the front of the queue
            queue_front = (queue_front + 1) % QUEUE_CAPACITY; // Update the front of the queue
            pthread_mutex_unlock(&queue_lock); // Unlock the queue
            sem_post(&freeSlot); // Signal that a slot in the queue is free

            pthread_mutex_lock(&parking_lock); // Lock the parking slots
            if (mFree_automobile > 0) { // Check if there are free automobile parking slots
                for (int i = 0; i < AUTOMOBILE_CAPACITY; i++) {
                    if (automobile_parking[i] == NULL) {
                        automobile_parking[i] = vehicle; // Park the automobile
                        mFree_automobile--; // Decrease the number of free automobile parking slots
                        printf("Automobile ID: %d parked. Occupancy: %d/%d\n", vehicle->id, AUTOMOBILE_CAPACITY - mFree_automobile, AUTOMOBILE_CAPACITY);
                        break;
                    }

                }
            } else {
                printf("Automobile parking is full. New automobile could not be parked.\n");
                free(vehicle); // Free the vehicle if no parking slot is available
            }
            pthread_mutex_unlock(&parking_lock); // Unlock the parking slots

            sem_post(&inChargeforAutomobile); // Signal that the attendant is no longer in charge of automobile parking
            sem_post(&automobileDone); // Signal that the automobile parking is done
        }

        checkAndRemoveExpiredVehicles(); // Check and remove expired vehicles
        sleep(1);  // Wait for a while and check again
    }
    return NULL;
}
```

Sent the tool types parameter for Threads in Main.Attendants wait for the newPickup and newAutomobile semaphores in carOwner to be opened. When the car arrives, it is put in free space in the array. mFree variable is decremented. Queue capacity increases and front value changes. When one attendant is working, it locks with parking_lock and the other one is not working. The last two carAttendants check the vehicles to be removed from the parking lot. The CheckAndRemoveExpiredVehicles function is used to remove vehicles over 20 seconds.

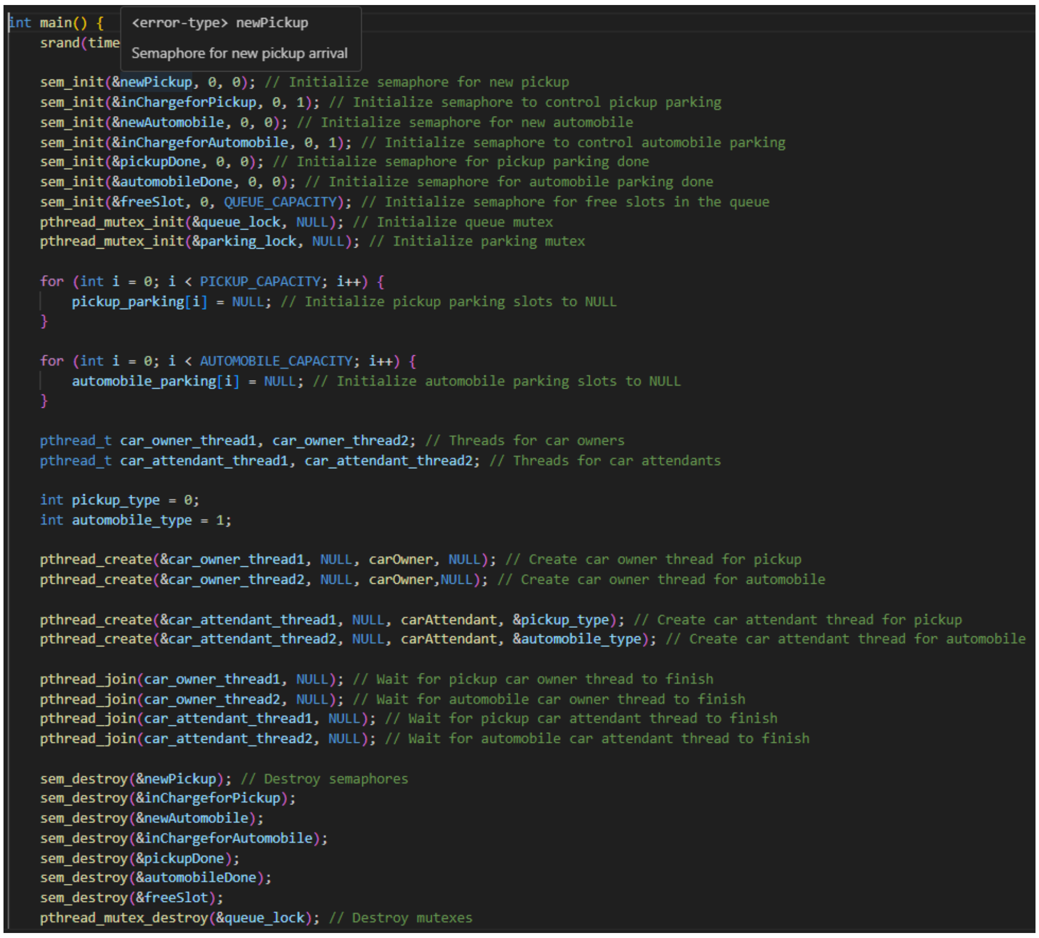
- Taking out of Park:



```c
void checkAndRemoveExpiredVehicles() {
    time_t current_time = time(NULL);

    // Check pickup parking slots
    pthread_mutex_lock(&parking_lock);
    for (int i = 0; i < PICKUP_CAPACITY; i++) {
        if (pickup_parking[i] != NULL && difftime(current_time, pickup_parking[i]->arrival_time) >= PARK_DURATION) {
            printf("\nRemoving pickup from park...\nPickup ID: %d left. Occupancy: %d/%d\n\n", pickup_parking[i]->id, PICKUP_CAPACITY - mFree_pickup - 1, PICKUP_CAPACITY);
            free(pickup_parking[i]);
            pickup_parking[i] = NULL;
            mFree_pickup++; // Increase the number of free pickup parking slots
        }
    }
    pthread_mutex_unlock(&parking_lock);

    // Check automobile parking slots
    pthread_mutex_lock(&parking_lock);
    for (int i = 0; i < AUTOMOBILE_CAPACITY; i++) {
        if (automobile_parking[i] != NULL && difftime(current_time, automobile_parking[i]->arrival_time) >= PARK_DURATION) {
            printf("\nRemoving automobile from park...\nAutomobile ID: %d left. Occupancy: %d/%d\n\n", automobile_parking[i]->id, AUTOMOBILE_CAPACITY - mFree_automobile - 1, AUTOMOBILE_CAPACITY);
            free(automobile_parking[i]);
            automobile_parking[i] = NULL;
            mFree_automobile++; // Increase the number of free automobile parking slots
        }
    }
    pthread_mutex_unlock(&parking_lock);
}
```

For vehicles exceeding 20 seconds, the array is traversed and mFree variables are incremented.

- Main:



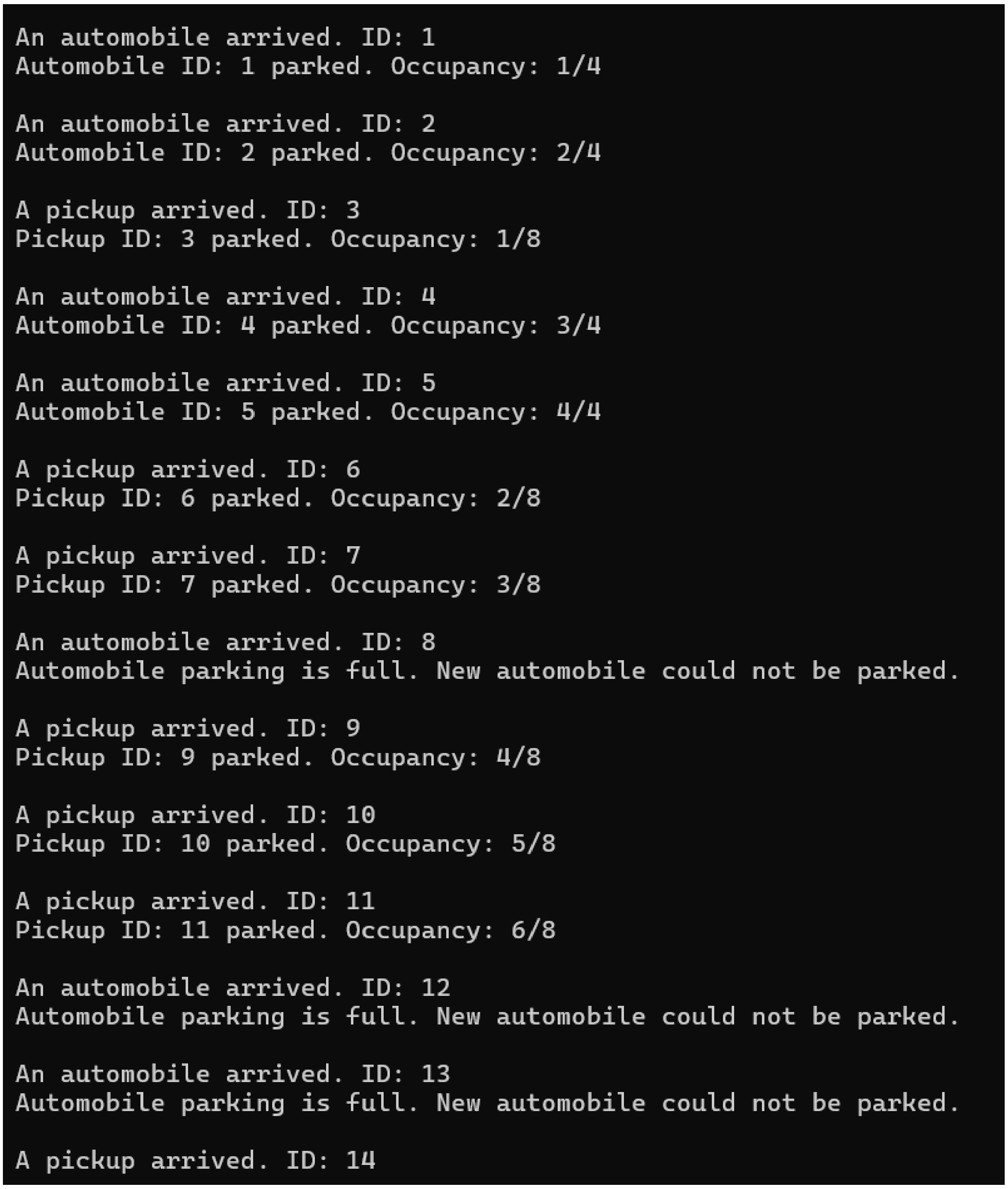
<error-type> newPickup

Semaphore for new pickup arrival

```c
int main() {
    srand(time

    sem_init(&newPickup, 0, 0); // Initialize semaphore for new pickup
    sem_init(&inChargeforPickup, 0, 1); // Initialize semaphore to control pickup parking
    sem_init(&newAutomobile, 0, 0); // Initialize semaphore for new automobile
    sem_init(&inChargeforAutomobile, 0, 1); // Initialize semaphore to control automobile parking
    sem_init(&pickupDone, 0, 0); // Initialize semaphore for pickup parking done
    sem_init(&automobileDone, 0, 0); // Initialize semaphore for automobile parking done
    sem_init(&freeSlot, 0, QUEUE_CAPACITY); // Initialize semaphore for free slots in the queue
    pthread_mutex_init(&queue_lock, NULL); // Initialize queue mutex
    pthread_mutex_init(&parking_lock, NULL); // Initialize parking mutex

    for (int i = 0; i < PICKUP_CAPACITY; i++) {
        pickup_parking[i] = NULL; // Initialize pickup parking slots to NULL
    }

    for (int i = 0; i < AUTOMOBILE_CAPACITY; i++) {
        automobile_parking[i] = NULL; // Initialize automobile parking slots to NULL
    }

    pthread_t car_owner_thread1, car_owner_thread2; // Threads for car owners
    pthread_t car_attendant_thread1, car_attendant_thread2; // Threads for car attendants

    int pickup_type = 0;
    int automobile_type = 1;

    pthread_create(&car_owner_thread1, NULL, carOwner, NULL); // Create car owner thread for pickup
    pthread_create(&car_owner_thread2, NULL, carOwner,NULL); // Create car owner thread for automobile

    pthread_create(&car_attendant_thread1, NULL, carAttendant, &pickup_type); // Create car attendant thread for pickup
    pthread_create(&car_attendant_thread2, NULL, carAttendant, &automobile_type); // Create car attendant thread for automobile

    pthread_join(car_owner_thread1, NULL); // Wait for pickup car owner thread to finish
    pthread_join(car_owner_thread2, NULL); // Wait for automobile car owner thread to finish
    pthread_join(car_attendant_thread1, NULL); // Wait for pickup car attendant thread to finish
    pthread_join(car_attendant_thread2, NULL); // Wait for automobile car attendant thread to finish

    sem_destroy(&newPickup); // Destroy semaphores
    sem_destroy(&inChargeforPickup);
    sem_destroy(&newAutomobile);
    sem_destroy(&inChargeforAutomobile);
    sem_destroy(&pickupDone);
    sem_destroy(&automobileDone);
    sem_destroy(&freeSlot);
    pthread_mutex_destroy(&queue_lock); // Destroy mutexes
```

Created 2 carOwner and 2 carAttendant threads. Array was made null at first. Semaphores are initialized. Finally semaphores are destroyed and threads are terminated. The program continues in an infinite loop. It ends with Ctrl+c.

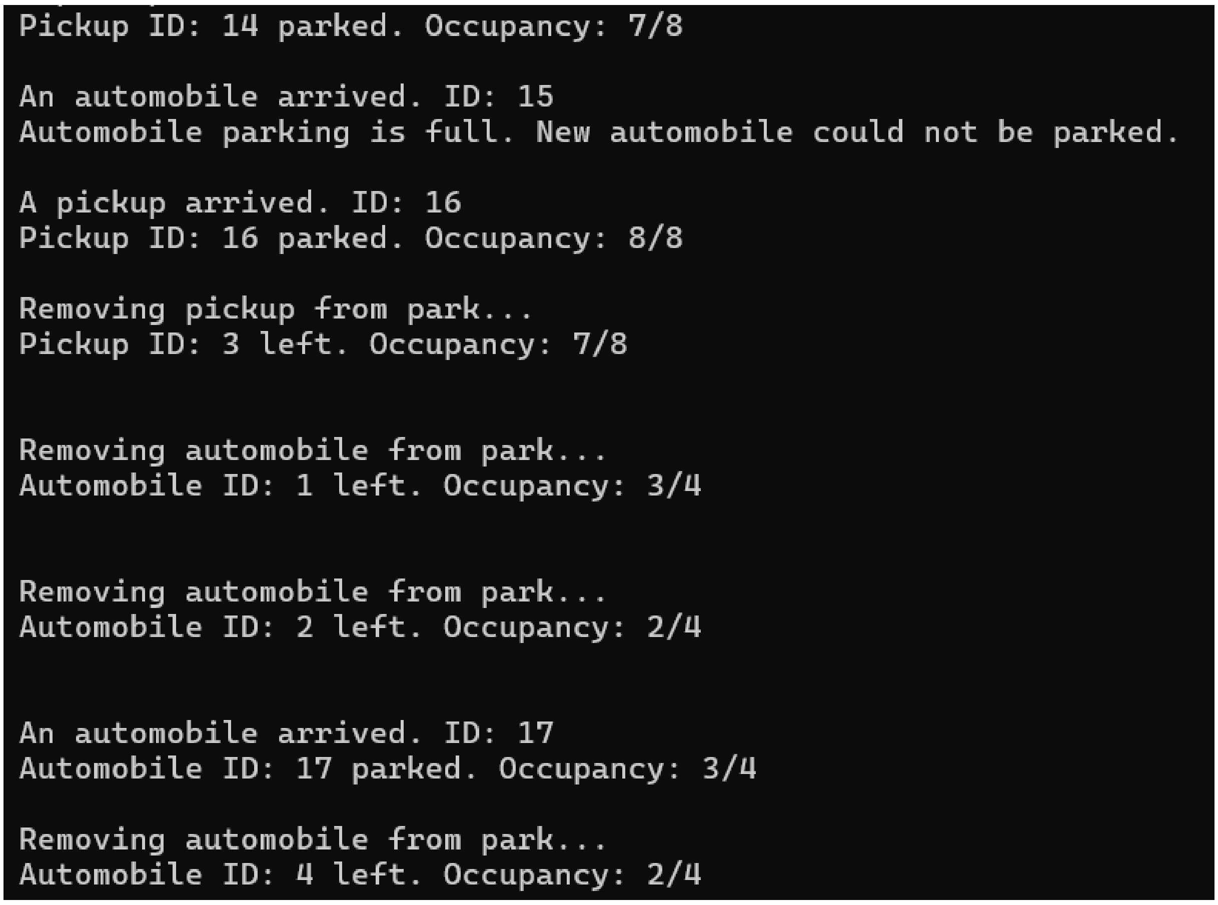
- Outputs:



```
An automobile arrived. ID: 1
Automobile ID: 1 parked. Occupancy: 1/4

An automobile arrived. ID: 2
Automobile ID: 2 parked. Occupancy: 2/4

A pickup arrived. ID: 3
Pickup ID: 3 parked. Occupancy: 1/8

An automobile arrived. ID: 4
Automobile ID: 4 parked. Occupancy: 3/4

An automobile arrived. ID: 5
Automobile ID: 5 parked. Occupancy: 4/4

A pickup arrived. ID: 6
Pickup ID: 6 parked. Occupancy: 2/8

A pickup arrived. ID: 7
Pickup ID: 7 parked. Occupancy: 3/8

An automobile arrived. ID: 8
Automobile parking is full. New automobile could not be parked.

A pickup arrived. ID: 9
Pickup ID: 9 parked. Occupancy: 4/8

A pickup arrived. ID: 10
Pickup ID: 10 parked. Occupancy: 5/8

A pickup arrived. ID: 11
Pickup ID: 11 parked. Occupancy: 6/8

An automobile arrived. ID: 12
Automobile parking is full. New automobile could not be parked.

An automobile arrived. ID: 13
Automobile parking is full. New automobile could not be parked.

A pickup arrived. ID: 14
```

In this section here, the vehicles have arrived. The occupancy rate is printed. If it was full, the vehicle could not be parked. No vehicle was removed from the parking lot.

Because no vehicle exceeded 20 seconds.



```
Pickup ID: 14 parked. Occupancy: 7/8

An automobile arrived. ID: 15
Automobile parking is full. New automobile could not be parked.

A pickup arrived. ID: 16
Pickup ID: 16 parked. Occupancy: 8/8

Removing pickup from park...
Pickup ID: 3 left. Occupancy: 7/8


Removing automobile from park...
Automobile ID: 1 left. Occupancy: 3/4


Removing automobile from park...
Automobile ID: 2 left. Occupancy: 2/4


An automobile arrived. ID: 17
Automobile ID: 17 parked. Occupancy: 3/4

Removing automobile from park...
Automobile ID: 4 left. Occupancy: 2/4
```

This section is filled with cars and pickups. Then the previous vehicles are removed from the parking lot. Thus, the new vehicle with 17 ids can be parked. Ids are kept in a unique way. There is a different id for each vehicle. Thus, it is easily determined which vehicle is being removed or parked.