CSE 344 SYSTEM PROGRAMMING

HOMEWORK #3 REPORT

BUKET GENÇER

210104004298

20.05.2024

Table of Contents

Introduction and Project Description

This report details the development and execution of a parking system simulation managed through multithreading and semaphore synchronization. The primary objective is to efficiently coordinate the entry and parking of automobiles and pickups using threads that represent vehicle owners and attendants, with the aid of semaphores to manage resource contention and ensure orderly access to parking spaces.

Code Description and Analysis

This section delves into the implementation of the parking system project, highlighting how threads and semaphores are utilized to efficiently manage the inflow of automobiles and pickups. It outlines the key functions and structures, their roles, and the synchronization mechanisms that ensure smooth operations and prevent conflicts within the system. This analysis aims to clarify the technical approaches used to optimize real-time vehicle parking

a. Signal Handling

To ensure that the parking system can handle unexpected events like program interruptions gracefully, the code incorporates comprehensive signal handling mechanisms. These mechanisms allow the system to terminate threads cleanly and release resources properly when signals, particularly SIGINT (Ctrl+C), are received.

```
void setupSignalHandling() { // Signal handling function
    struct sigaction sa; // Signal action struct to set up signal handling
    sa.sa_handler = signalHandler; // Set the signal handler function to be called when a signal is received
    sigemptyset(&sa.sa_mask); // Set the signal mask to empty to allow all signals to be received
    sa.sa_flags = 0; // Set the flags to 0
    sigaction(SIGINT, &sa, NULL); // Set up the signal handling for SIGINT (Ctrl+C) signal  // SIGINT is the signal for Ctrl+C
}
```

This setup initializes and configures the **sigaction** structure to manage SIGINT signals using the **signalHandler** function. This function is crucial for ensuring that the system can respond to the Ctrl+C interrupt signal, allowing for a graceful shutdown of the system by releasing any resources and stopping all threads safely.

```c
void signalHandler(int sig) { // Signal handler function
    printf("Received signal %d, stopping threads...\n", sig); // Print the signal number
    keepRunning = 0; // Set the flag to stop the threads

    // Wake up the threads that are waiting on semaphores
    sem_post(&newAutomobile);
    sem_post(&newPickup);
    sem_post(&inChargeforAutomobile);
    sem_post(&inChargeforPickup);
}
```

This function is triggered whenever a SIGINT is received. It sets a flag (**keepRunning**) to false, which is checked in the loops of the various threads, instructing them to cease operations. Additionally, it posts to the semaphores involved in the synchronization process, ensuring that any threads waiting on these semaphores are awakened and can terminate gracefully instead of remaining indefinitely blocked

Integration in Main Function:

```c
int main() {

    setupSignalHandling(); // Set up signal handling
```

Within the main function, **setupSignalHandling** is called to initialize the signal handling setup before the system becomes fully operational. This proactive setup ensures that the system is prepared to handle interrupts from the very start, aligning with best practices for robust program design.

b. Threads Used and Their Explanations

The parking system employs multiple threads to simulate the activities of vehicle owners and parking attendants, ensuring efficient management of vehicle arrivals, parking, and departures in a synchronized manner.

Vehicle Owner Threads

automobileOwner

```c
void* automobileOwner(void* arg) {
    struct timespec ts; // timespec struct to hold the time for the semaphore timeout
    while (keepRunning) {
        printf("                                                | o-o   : Automobile owner arrived.\n");
        clock_gettime(CLOCK_REALTIME, &ts);
        ts.tv_sec += 5; // Wait for 5 seconds for a spot to be available
       if (sem_timedwait(&newAutomobile, &ts) != 0) {
            continue;  // If semaphore wait times out, retry
        }
        if (mFree_automobile > 0) {
            printf("                                                | o-o   : Temporary spot available for automobile. Parking now.\n");
            mFree_automobile--; // Decrement the number of free spots for automobile
            //print the number of free spots
            printf("                                                | N     : Number of free spots for automobile: %d\n", mFree_automobile);
            sem_post(&inChargeforAutomobile); // Release the semaphore for the attendant
        } else {
            printf("                                                | X     : No temporary spot available for automobile. Exiting.\n");
        }
        sem_post(&newAutomobile); // Release the semaphore for the next car
        sleep(5); // Simulate time taken for the next car to arrive
    }
    pthread_exit(NULL); // Exit the thread
}
```

This function manages the arrival and parking attempts of automobile owners. It uses a semaphore to ensure that only one automobile owner can attempt to park at any given time. The thread waits for up to 5 seconds to get a semaphore, ensuring that it does not block indefinitely if the parking lot is full.

pickupOwner

```c
void* pickupOwner(void* arg) {
    struct timespec ts; // timespec struct to hold the time for the semaphore timeout
    while (keepRunning) { // Keep running until the signal is received
        printf("o-o^  : Pickup owner arrived.                          |\n"); // Print the message for the pickup owner
         clock_gettime(CLOCK_REALTIME, &ts); // Get the current time
        ts.tv_sec += 5; // Wait for 5 seconds for a spot to be available
        if (sem_timedwait(&newPickup, &ts) != 0) {
            continue; // If semaphore wait times out, retry
        }
        if (mFree_pickup > 0) {
            printf("o-o^  : Temporary spot available for pickup. Parking now.|\n");
            mFree_pickup--; // Decrement the number of free spots for pickup
            //print the number of free spots
            printf("N^    : Number of free spots for pickup: %d              |\n", mFree_pickup);
            sem_post(&inChargeforPickup); // Release the semaphore for the attendant
        } else {
            printf("X^    : No temporary spot available for pickup. Exiting. |\n");
        }
        sem_post(&newPickup); // Release the semaphore for the next pickup
    }
    pthread_exit(NULL); // Exit the thread
}
```

This function mirrors the operation of the automobileOwner thread but is specialized for handling pickup vehicles. It ensures synchronized access to the temporary parking

spots designated for pickups, managing the flow and parking operations based on parking lot availability.

Parking Attendant Threads

automobileAttendant

```
void* automobileAttendant(void* arg) { // Automobile attendant function
    struct timespec ts; // timespec struct to hold the time for the semaphore timeout
    while (keepRunning) {
        printf("                                                    | :) R  : Automobile attendant ready.\n");
        clock_gettime(CLOCK_REALTIME, &ts);
        ts.tv_sec += 5; // Wait for 5 seconds for a car to be parked
        if (sem_timedwait(&inChargeforAutomobile, &ts) != 0) {
            continue; // If semaphore wait times out, retry
        }
        printf("                                                    | :)    : Automobile attendant parks the automobile.\n");
        mFree_automobile++; // Increment the number of free spots
        sem_post(&newAutomobile); // Release the semaphore for the next car
        sleep(1); // Simulate time taken to park a car
    }
}
```

This thread function synchronizes with the automobileOwner thread via semaphores. It ensures that each vehicle is parked correctly and manages the parking slots by updating the availability status.

pickupAttendant

```
void* pickupAttendant(void* arg) { // Pickup attendant function
    struct timespec ts; // timespec struct to hold the time for the semaphore timeout
    while (keepRunning) { // Keep running until the signal is received
        printf("^:) R : Pickup attendant ready.                        |\n");
        clock_gettime(CLOCK_REALTIME, &ts); // Get the current time
        ts.tv_sec += 5; // Wait for 5 seconds for a car to be parked
        if (sem_timedwait(&inChargeforPickup, &ts) != 0) { // Wait for the semaphore with a timeout
            continue; // If semaphore wait times out, retry
        }

        printf("^:)   : Pickup attendant parks the pickup.             |\n");
        mFree_pickup++; // Increment the number of free spots
        sem_post(&newPickup); // Release the semaphore for the next pickup
        sleep(1); // Simulate time taken to park a car
    }
}
```

This thread function ensures the orderly parking of pickups by coordinating with the pickupOwner threads through semaphores. It manages the slots specifically set aside for pickups, maintaining a smooth operational flow within the parking lot, and updating the availability of spots as pickups are parked and exit the temporary parking area.

Dynamic Thread Creation

createVehicle

```
void* createVehicle(void* arg) { // Vehicle creation function

    while (keepRunning) {
        int vehicleType = rand() % 100; // randomly determine the vehicle type
        pthread_t thread; // Thread variable to hold the thread ID
        if (vehicleType % 2 == 0) {
            pthread_create(&thread, NULL, pickupOwner, NULL); // Create a new thread for the pickup owner
        } else {
            pthread_create(&thread, NULL, automobileOwner, NULL); // Create a new thread for the automobile owner
        }
        pthread_detach(thread); // Detach the thread to allow it to run independently
        sleep(1); // Sleep for 1 second to create the next vehicle
    }
}
```

This function continuously generates threads for either pickup or automobile owners based on random selection. This simulates a realistic scenario where vehicles arrive at different times and require immediate attention from the parking system.

c. Threads Management

Thread management in this parking system is pivotal to maintaining orderly operations, allowing vehicles to be parked and managed without collisions or deadlocks. This section discusses how threads are created and managed within the system, emphasizing the distinction between thread creation and ongoing thread management.

Thread Creation

Threads are initialized in the **main** function where specific functions are assigned to handle different roles within the parking system. Here's a look at how threads are created:

```
pthread_t autoAttendantThread, pickupAttendantThread, vehicleCreatorThread; // Thread variables to hold the thread IDs
pthread_create(&autoAttendantThread, NULL, automobileAttendant, NULL); // Create a new thread for the automobile attendant
pthread_create(&pickupAttendantThread, NULL, pickupAttendant, NULL); // Create a new thread for the pickup attendant
pthread_create(&vehicleCreatorThread, NULL, createVehicle, NULL);  // Create a new thread for the vehicle creator
```

pthread_create is used to initiate threads that handle different tasks—automobileAttendant, pickupAttendant, and createVehicle. Each thread runs a function that either manages vehicle parking or generates new vehicles.

Continuous Operation: The createVehicle thread continuously creates new threads for vehicle owners, reflecting a dynamic system where new vehicles arrive intermittently.

- **autoAttendantThread**: Manages the parking of automobiles by coordinating with the automobileOwner threads.
- **pickupAttendantThread**: Manages the parking of pickups by coordinating with the pickupOwner threads.
- **vehicleCreatorThread**: Continuously generates new vehicle threads, simulating the arrival of vehicles at the parking lot.

These threads are vital for the dynamic and responsive operation of the parking system, enabling simultaneous and independent management of multiple vehicles.

Thread Synchronization

```
// pthread_join is used to wait for the threads to finish before exiting the program
 pthread_join(autoAttendantThread, NULL); // Wait for the automobile attendant thread to finish
 pthread_join(pickupAttendantThread, NULL); // Wait for the pickup attendant thread to finish
 pthread_join(vehicleCreatorThread, NULL);  // Wait for the vehicle creator thread to finish
```

Once threads are created, they are allowed to run concurrently, handling different tasks within the parking system. The main thread waits for all other threads to complete, ensuring that the program can gracefully exit after all operations have been handled.

- **pthread_join** is called for each thread, which blocks the main thread until the specified threads have terminated. In a real-world application, mechanisms such as these ensure that no threads are left running unintentionally, which could lead to resource leaks or unfinished business when the application is intended to shut down.

d. Semaphore Usage and Management

Semaphores play a crucial role in this parking system, facilitating synchronization between various threads to ensure safe and efficient parking management. This section explores how semaphores are utilized to control access to parking spaces and synchronize the interaction between vehicle owners and parking attendants.

Semaphore Initialization

At the beginning of the program, semaphores are initialized to manage the entry and parking processes for automobiles and pickups. Here is how they are set up in the main function:

```c
int main() {

    setupSignalHandling(); // Set up signal handling
    srand(time(NULL));

    // Initialize the semaphores for threads to wait on and signal each other when ready to proceed
    sem_init(&newAutomobile, 0, 1);
    sem_init(&inChargeforAutomobile, 0, 0);
    sem_init(&newPickup, 0, 1);
    sem_init(&inChargeforPickup, 0, 0);
```

newAutomobile and newPickup: These semaphores control the entry of new vehicles into the system. They are initialized to 1, meaning that one vehicle of each type can enter the parking lot or wait for a parking spot at any given time.

inChargeforAutomobile and inChargeforPickup: These semaphores are used to signal the respective parking attendants when a vehicle is ready to be parked. They start at 0 because parking can only proceed when a vehicle owner has secured a temporary spot and signaled the attendant.

Semaphore Usage in Thread Operations

Each thread that represents a vehicle owner or attendant uses these semaphores to synchronize operations as shown in the example below:

```c
void* automobileOwner(void* arg) {
    struct timespec ts; // timespec struct to hold the time for the semaphore timeout
    while (keepRunning) {
        printf("                                                    | o-o   : Automobile owner arrived.\n");
        clock_gettime(CLOCK_REALTIME, &ts);
        ts.tv_sec += 5; // Wait for 5 seconds for a spot to be available
       if (sem_timedwait(&newAutomobile, &ts) != 0) {
            continue;  // If semaphore wait times out, retry
        }
        if (mFree_automobile > 0) {
            printf("                                                    | o-o   : Temporary spot available for automobile. Parking now.\n");
            mFree_automobile--; // Decrement the number of free spots for automobile
            //print the number of free spots
            printf("                                                    | N     : Number of free spots for automobile: %d\n", mFree_automobile);
            sem_post(&inChargeforAutomobile); // Release the semaphore for the attendant
        } else {
            printf("                                                    | X     : No temporary spot available for automobile. Exiting.\n");
        }
        sem_post(&newAutomobile); // Release the semaphore for the next car
        sleep(5); // Simulate time taken for the next car to arrive
    }
    pthread_exit(NULL); // Exit the thread
}
```

Waiting for Entry: sem_timedwait(&newAutomobile, &ts) is used to manage access to the parking lot, ensuring that only one automobile at a time attempts to find a parking space.

Signaling Attendants: After securing a temporary spot, the vehicle owner uses **sem_post(&inChargeforAutomobile)** to notify the attendant that the vehicle is ready to be parked.

Handling Termination and Deadlocks

The signal handler ensures that all semaphores are posted when the program is interrupted, preventing deadlocks and ensuring that all threads can terminate gracefully.

```
void signalHandler(int sig) { // Signal handler function
    printf("Received signal %d, stopping threads...\n", sig); // Print the signal number
    keepRunning = 0; // Set the flag to stop the threads

    // Wake up the threads that are waiting on semaphores
    sem_post(&newAutomobile);
    sem_post(&newPickup);
    sem_post(&inChargeforAutomobile);
    sem_post(&inChargeforPickup);
}
```

Releasing Blocked Threads: Posting all semaphores during a shutdown ensures that no threads are indefinitely blocked waiting for a semaphore, allowing them to proceed to termination checks and exit cleanly.

Testing

In order to clearly differentiate between the two types of vehicles in the system, I arranged the output so that pickups appear on the left side of the screen and automobiles on the right. This setup makes it easier to track and understand the activities related to each type of vehicle separately during testing.

The system effectively handles the dynamic arrival of vehicles, manages parking availability, and ensures attendants are synchronized to service the vehicles promptly.

Examples of Outputs:

The lines in the middle represent the flow of time, with information about pickups displayed on the left and details about automobiles on the right. This layout helps to visually distinguish and follow the sequences of events for each vehicle type.

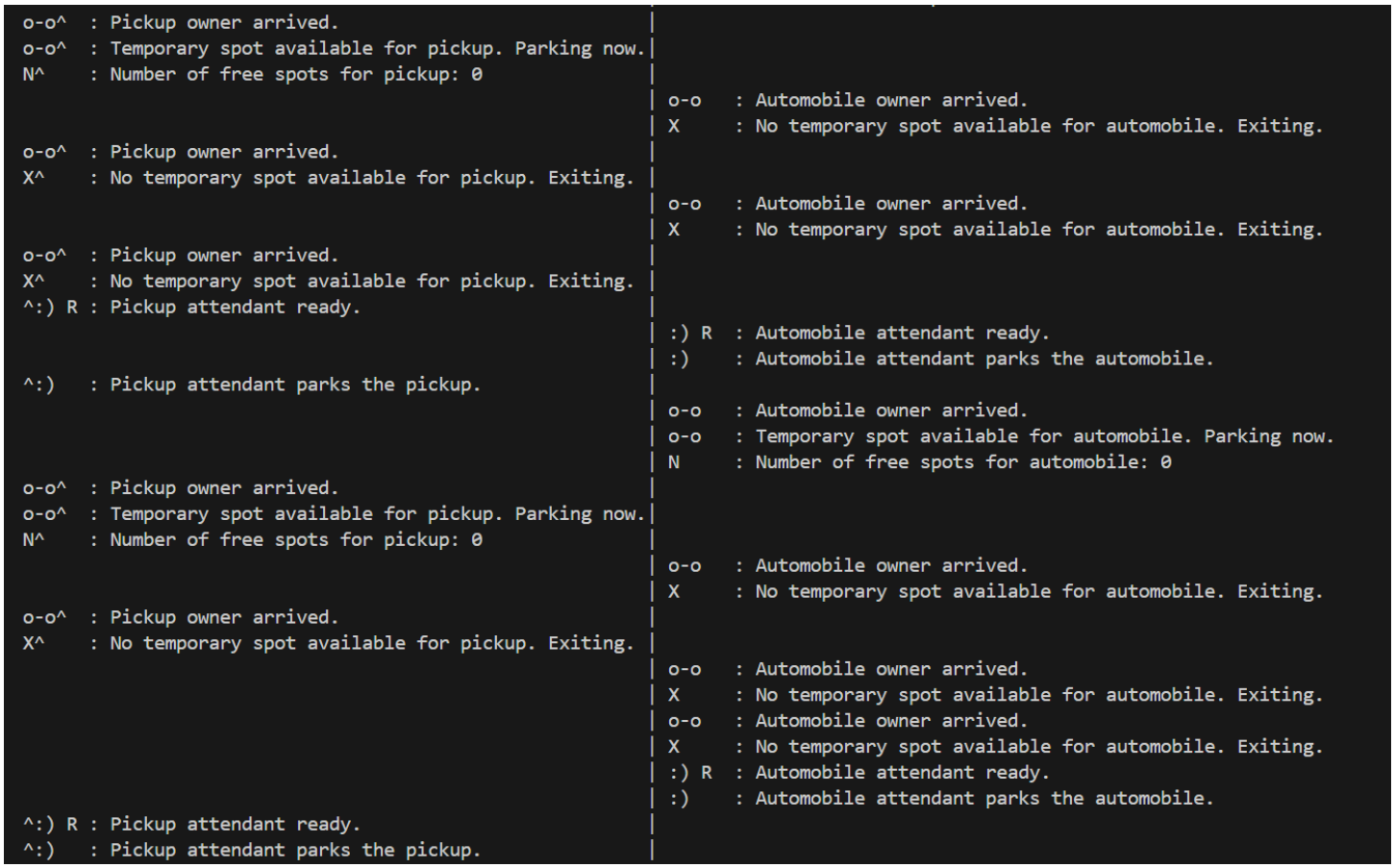
Symbols and their explanations used in the program's output:

- ***o-o: Represents an automobile arriving.***
- ***o-o^: Represents a pickup arriving.***
- ***N: Shows the number of available temporary parking spots for automobiles.***

- ***N^: Shows the number of available temporary parking spots for pickups.***
- ***:): Indicates the automobile attendant.***
- ***^:): Indicates the pickup attendant.***
- ***:) R and ^:) R: Indicate that the automobile and pickup attendants are ready, respectively.***
- ***X and X^: Indicate that there is no temporary parking spot available for automobiles and pickups, respectively.***

These symbols help track the flow of vehicles and attendants' activities, providing a clear and organized view of the parking lot operations.

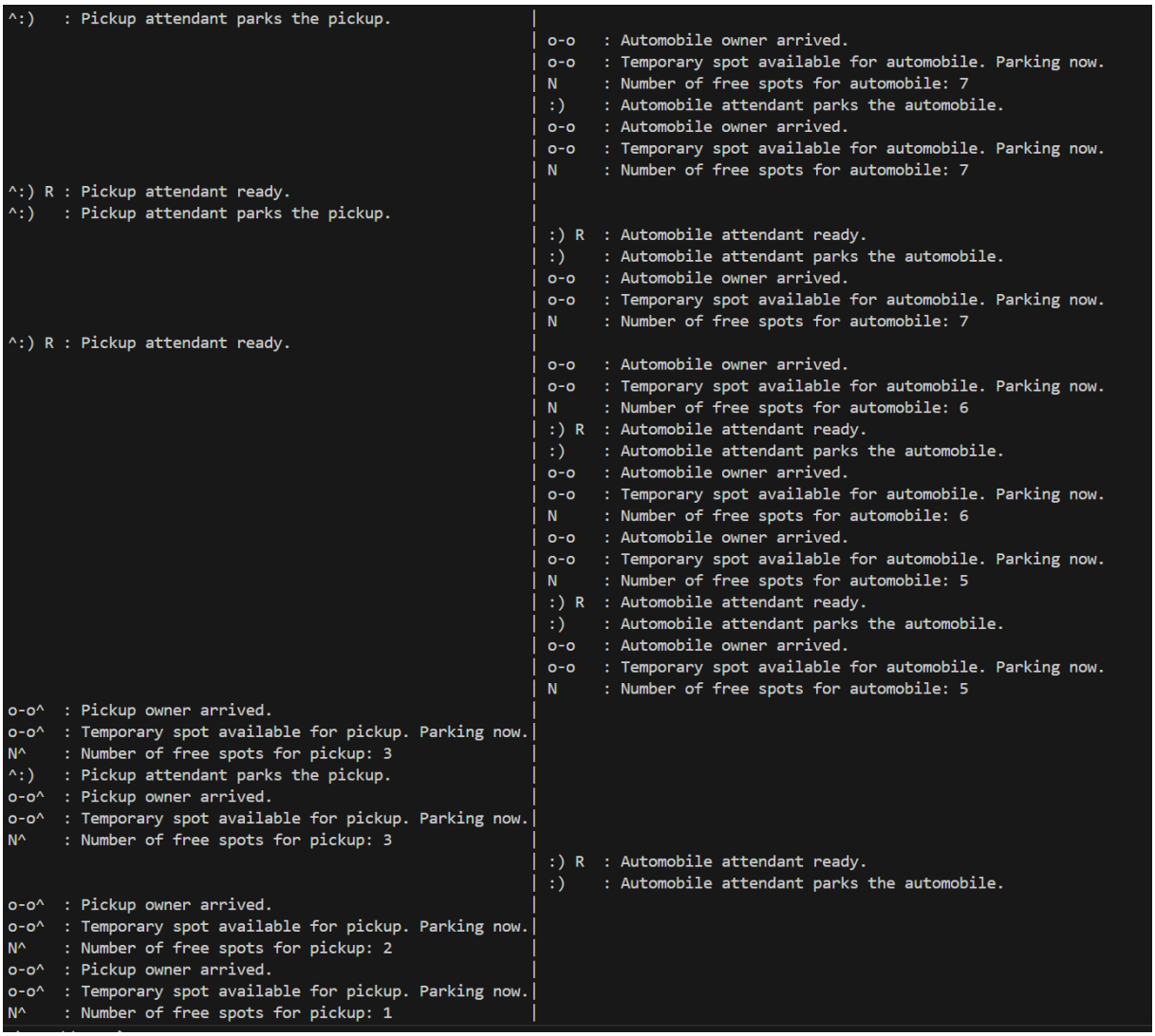
```
bktgncr@DESKTOP-AI758A5:/mnt/c/Users/HUAWEI/OneDrive/Masaüstü/CSE344HW3$ ./parking_system
                                                    | :) R  : Automobile attendant ready.
^:) R : Pickup attendant ready.                     |
                                                    | o-o   : Automobile owner arrived.
                                                    | o-o   : Temporary spot available for automobile. Parking now.
                                                    | N     : Number of free spots for automobile: 7
                                                    | :)    : Automobile attendant parks the automobile.
                                                    | :) R  : Automobile attendant ready.
o-o^  : Pickup owner arrived.                       |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 3          |
^:)   : Pickup attendant parks the pickup.          |
^:) R : Pickup attendant ready.                     |
                                                    | o-o   : Automobile owner arrived.
                                                    | o-o   : Temporary spot available for automobile. Parking now.
                                                    | N     : Number of free spots for automobile: 7
                                                    | :)    : Automobile attendant parks the automobile.
                                                    | o-o   : Automobile owner arrived.
                                                    | o-o   : Temporary spot available for automobile. Parking now.
                                                    | N     : Number of free spots for automobile: 7
                                                    | :) R  : Automobile attendant ready.
                                                    | :)    : Automobile attendant parks the automobile.
o-o^  : Pickup owner arrived.                       |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 3          |
                                                    | :) R  : Automobile attendant ready.
^:)   : Pickup attendant parks the pickup.          |
                                                    | o-o   : Automobile owner arrived.
                                                    | o-o   : Temporary spot available for automobile. Parking now.
                                                    | N     : Number of free spots for automobile: 7
                                                    | :)    : Automobile attendant parks the automobile.
^:) R : Pickup attendant ready.                     |
o-o^  : Pickup owner arrived.                       |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 3          |
^:)   : Pickup attendant parks the pickup.          |
^CReceived signal 2, stopping threads...
Cleaned up and exiting.
bktgncr@DESKTOP-AI758A5:/mnt/c/Users/HUAWEI/OneDrive/Masaüstü/CSE344HW3$
```

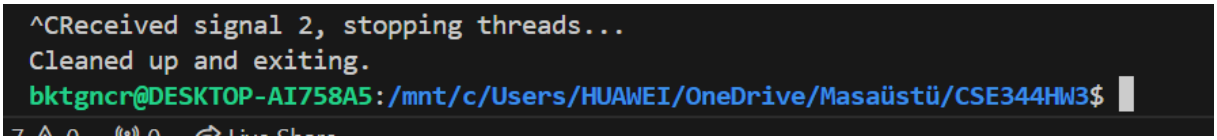


In this screenshot, you can see an example of how vehicles exit when there are no available temporary parking spots left after the program has been running for a while. This demonstrates the system's handling of full parking conditions efficiently.

More examples of outputs:

```
bktgncr@DESKTOP-AI758A5:/mnt/c/Users/HUAWEI/OneDrive/Masaüstü/CSE344HW3$ ./parking_system
                                                  | :) R  : Automobile attendant ready.
^:) R : Pickup attendant ready.                   |
o-o^  : Pickup owner arrived.                     |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 3        |
^:)   : Pickup attendant parks the pickup.        |
o-o^  : Pickup owner arrived.                     |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 3        |
^:) R : Pickup attendant ready.                   |
^:)   : Pickup attendant parks the pickup.        |
^:) R : Pickup attendant ready.                   |
                                                  | o-o   : Automobile owner arrived.
                                                  | o-o   : Temporary spot available for automobile. Parking now.
                                                  | N     : Number of free spots for automobile: 7
                                                  | :)    : Automobile attendant parks the automobile.
                                                  | o-o   : Automobile owner arrived.
                                                  | o-o   : Temporary spot available for automobile. Parking now.
                                                  | N     : Number of free spots for automobile: 7
                                                  | :) R  : Automobile attendant ready.
                                                  | :)    : Automobile attendant parks the automobile.
                                                  | :) R  : Automobile attendant ready.
                                                  | o-o   : Automobile owner arrived.
                                                  | o-o   : Temporary spot available for automobile. Parking now.
                                                  | N     : Number of free spots for automobile: 7
                                                  | :)    : Automobile attendant parks the automobile.
o-o^  : Pickup owner arrived.                     |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 3        |
^:)   : Pickup attendant parks the pickup.        |
o-o^  : Pickup owner arrived.                     |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 3        |
                                                  | :) R  : Automobile attendant ready.
o-o^  : Pickup owner arrived.                     |
o-o^  : Temporary spot available for pickup. Parking now.|
N^    : Number of free spots for pickup: 2        |
^:) R : Pickup attendant ready.                   |
^:)   : Pickup attendant parks the pickup.        |
                                                  | o-o   : Automobile owner arrived.
                                                  | o-o   : Temporary spot available for automobile. Parking now.
                                                  | N     : Number of free spots for automobile: 7
                                                  | :)    : Automobile attendant parks the automobile.
                                                  | o-o   : Automobile owner arrived.
                                                  | o-o   : Temporary spot available for automobile. Parking now.
```





When the program receives a signal, specifically Signal 2 (SIGINT, which is typically triggered by pressing Ctrl+C), it initiates the process of stopping all running threads. The output "Received signal 2, stopping threads..." confirms that the signal was detected and the system is proceeding to shut down the threads responsibly.

Following this, the message "Cleaned up and exiting." indicates that the program has successfully terminated all threads and cleaned up resources such as semaphores and memory allocations, ensuring that no resources are left allocated or locked inappropriately. This step is crucial to prevent any potential memory leaks or dangling resources, contributing to the robustness and stability of the software. This output demonstrates the effectiveness of the program's cleanup process in response to an interruption signal, ensuring a graceful and complete shutdown.

Performance Evaluation

Resource Utilization

Resource efficiency is critical, especially in how semaphores are managed to prevent resource locking and ensure smooth operation:

```
// Initialize the semaphores for threads
sem_init(&newAutomobile, 0, 1);
sem_init(&inChargeforAutomobile, 0, 0);
sem_init(&newPickup, 0, 1);
sem_init(&inChargeforPickup, 0, 0);
```

Initialization of semaphores with an initial value of 1 ensures that at any given time, only one vehicle owner can access a parking spot, effectively managing resource contention.

Concurrency Management

The system uses semaphores to manage access to shared resources, ensuring that operations such as parking are synchronized:

```
ts.tv_sec += 5; // Wait for 5 seconds for a spot to be ava
if (sem_timedwait(&newAutomobile, &ts) != 0) {
    continue;  // If semaphore wait times out, retry
}
if (mFree_automobile > 0) {
    printf("
    mFree_automobile--; // Decrement the number of free sp
    //print the number of free spots
    printf("
    sem_post(&inChargeforAutomobile); // Release the semap
} else {
    printf("
```

This mechanism ensures that when a spot becomes free, the semaphore is signaled (**sem_post**), allowing another vehicle owner to proceed, effectively managing concurrency without deadlocks.

Reliability and Error Handling

Upon receiving a termination signal (e.g., SIGINT), the system releases all semaphores and stops all threads, ensuring no resources are left hanging and that the system exits cleanly.

```c
void signalHandler(int sig) { // Signal handler function
    printf("Received signal %d, stopping threads...\n", sig); // Print the signal number
    keepRunning = 0; // Set the flag to stop the threads

    // Wake up the threads that are waiting on semaphores
    sem_post(&newAutomobile);
    sem_post(&newPickup);
    sem_post(&inChargeforAutomobile);
    sem_post(&inChargeforPickup);
}
```

Makefile

Makefile

```
CC=gcc
CFLAGS=-Wall
TARGET=parking_system

all: $(TARGET) run

$(TARGET): parking_system.c
    $(CC) $(CFLAGS) parking_system.c -o $(TARGET) -pthread

run:
    ./$(TARGET)

clean:
    rm -f $(TARGET)

```