GTU Department of Computer Engineering

CSE 344 – Spring 2024

Homework 3 Report

Barış Batuhan Bolat

210104004029

Usage

- Makefile contains clean, compile and run .

```
baris@Baris:/mnt/c/Users/baris/Desktop/GTU/3.Sınıf/CSE344/HW3$ make
```

- User must enter the number of regular parking spaces at the start of the program.

```
Enter max automobile parking spot: 10
Enter max pickup parking spot: 6
```

Constants and Shared Memory

- **NUM_TEMP_PICKUP_SPOTS:** Constant number of temporary pickup spots (default 4).
- **NUM_TEMP_AUTOMOBILE_SPOTS:** Constant number of temporary automobile spots (default 8).
- **mFree_automobile:** Tracks free temporary automobile spots.
- **mFree_pickup:** Tracks free temporary pickup spots.
- **normal_Free_automobile:** Tracks free normal automobile spots. Received by user at begining of program.
- **normal_Free_pickup:** Tracks free normal pickup spots. Received by user at begining of program.

Functions

printEmptySpaces()

- Prints the number of free temporary and normal parking spots for automobiles and pickups.

```c
void printEmptySpaces()
{
    char buffer[100];
    int len = snprintf(buffer, sizeof(buffer), "Empty temporary spaces - Automobiles: %d, Pickups: %d\n", mFree_automobile, mFree_pickup);
    write(STDOUT_FILENO, buffer, len);
    len = snprintf(buffer, sizeof(buffer), "Empty normal spaces - Automobiles: %d, Pickups: %d\n", normal_Free_automobile, normal_Free_pickup);
    write(STDOUT_FILENO, buffer, len);
}
```

carOwner()

- This thread function simulates car arrivals (automobile or pickup) and assigns them temporary parking spots.
- It takes a void pointer argument, which is currently unused.
- It enters a loop that continues until the program exits.
- Inside the loop:
    - It randomly chooses the vehicle type (automobile or pickup).
    - It checks if a temporary spot is available for the chosen vehicle type.
    - If available:
        - It acquires the corresponding attendant's semaphore (**inChargeforPickup** or **inChargeforAutomobile**).
        - It decrements the free temporary parking spot counter for the chosen vehicle type.
        - It signals a new car arrival with the **newPickup** or **newAutomobile** semaphore.
        - It releases the attendant's semaphore.
        - It prints a message indicating the car's entry into the temporary parking area.

- If not available:
  - It prints a message indicating no space is available for the arriving car.
  - It calls **printEmptySpaces** to display the updated parking space information.
  - It checks if all parking spots (temporary and normal) are full. If so, it exits the program.
  - It sleeps for a random amount of time before simulating the next car arrival.

```
void *carOwner(void *arg)
{
    int vehicleType;
    srand(time(NULL));
    while (1)
    {
        vehicleType = rand() % 2 == 0 ? 0 : 1;

        if (vehicleType == 0 && mFree_automobile > 0)
        {
            int attendant = rand() % 2;
            sem_wait(&inChargeforAutomobile[attendant]);
            mFree_automobile--;
            sem_post(&newAutomobile);
            sem_post(&inChargeforAutomobile[attendant]);
            write(STDOUT_FILENO, "-> An automobile has entered the temporary parking area.\n", strlen("-> An automobile has entered the temporary parking area.\n"));
        }
        else if (vehicleType == 1 && mFree_pickup > 0)
        {
            int attendant = rand() % 2;
            sem_wait(&inChargeforPickup[attendant]);
            mFree_pickup--;
            sem_post(&newPickup);
            sem_post(&inChargeforPickup[attendant]);
            write(STDOUT_FILENO, "-> A pickup has entered the temporary parking area.\n", strlen("-> A pickup has entered the temporary parking area.\n"));
        }
```

```
        else if (vehicleType == 0 && mFree_automobile <= 0)
        {
            write(STDOUT_FILENO, "\033[0;31m", sizeof("\033[0;31m") - 1);
            write(STDOUT_FILENO, "-> No available temporary parking spot for the arriving automobile.\n", strlen("-> No available temporary parking spot for the arriving automobile.\n"));
            write(STDOUT_FILENO, "\033[0m", sizeof("\033[0m") - 1);
        }
        else if (vehicleType == 1 && mFree_pickup <= 0)
        {
            write(STDOUT_FILENO, "\033[0;31m", sizeof("\033[0;31m") - 1);
            write(STDOUT_FILENO, "-> No available temporary parking spot for the arriving pickup.\n", strlen("-> No available temporary parking spot for the arriving pickup.\n"));
            write(STDOUT_FILENO, "\033[0m", sizeof("\033[0m") - 1);
        }
        printEmptySpaces();
        sleep(1);
        if (mFree_automobile == 0 && normal_Free_automobile == 0 && mFree_pickup == 0 && normal_Free_pickup == 0)
        {
            write(STDOUT_FILENO, "\033[0;31m", sizeof("\033[0;31m") - 1);
            write(STDOUT_FILENO, "-> All temporary and normal parking spots are full. Exiting...\n", strlen("-> All temporary and normal parking spots are full. Exiting...\n"));
            write(STDOUT_FILENO, "\033[0m", sizeof("\033[0m") - 1);
            exit(0);
        }
    }
    return NULL;
}
```

carAttendant()

- This thread function simulates a parking attendant who moves cars from **temporary** to **normal** parking spots when available.
- It takes a void pointer argument that actually holds the attendant's ID (**0 or 1**).
- It enters a loop that continues until the program exits.
  - Inside the loop:
    - It checks the attendant's responsibility (pickup or automobile) based on the ID.
    - It waits on the corresponding semaphore (**newPickup** or **newAutomobile**) for a new car arrival.
    - It acquires the attendant's semaphore (**inChargeforPickup** or **inChargeforAutomobile**).
    - If a normal parking spot is available for the corresponding vehicle type:
      - It "parks" the car in a normal spot (updates counters).
      - It prints a message indicating the attendant is parking a car.
    - If no normal spot is available:
      - It prints a message indicating the car remains in the temporary spot.
    - It releases the attendant's semaphore.
    - It calls printEmptySpaces to display the updated parking space information.
    - It checks if all parking spots (**temporary** and **normal**) are full. If so, it exits the program.

```c
void *carAttendant(void *arg)
{
    int attendant_id = *(int *)arg;
    while (1)
    {
        if (attendant_id == 0)
        {
            sem_wait(&newAutomobile);
            sem_wait(&inChargeforAutomobile[attendant_id]);
            if (normal_Free_automobile > 0)
            {
                write(STDOUT_FILENO, "-> Attendant (1) is parking an automobile.\n", strlen("-> Attendant (1) is parking an automobile.\n"));
                mFree_automobile++;
                normal_Free_automobile--;
            }
            else
            {
                write(STDOUT_FILENO, "\033[0;31m", sizeof("\033[0;31m") - 1);
                write(STDOUT_FILENO, "-> No available parking spot for the automobile. Staying on temporary.\n", strlen("-> No available parking spot for the automobile.  Staying on temporary.\n"));
                write(STDOUT_FILENO, "\033[0m", sizeof("\033[0m") - 1);
            }
            sem_post(&inChargeforAutomobile[attendant_id]);
        }
        else
        {
            sem_wait(&newPickup);
            sem_wait(&inChargeforPickup[attendant_id]);
            if (normal_Free_pickup > 0)
            {
                write(STDOUT_FILENO, "-> Attendant (2) is parking a pickup.\n", strlen("-> Attendant (2) is parking a pickup.\n"));
                mFree_pickup++;
                normal_Free_pickup--;
            }
            else
            {
                write(STDOUT_FILENO, "\033[0;31m", sizeof("\033[0;31m") - 1);
                write(STDOUT_FILENO, "-> No available parking spot for the pickup. Staying on temporary.\n", strlen("-> No available parking spot for the pickup. Staying on temporary.\n"));
                write(STDOUT_FILENO, "\033[0m", sizeof("\033[0m") - 1);
            }
            sem_post(&inChargeforPickup[attendant_id]);
        }
        printEmptySpaces();
```

```c
        if (mFree_automobile == 0 && normal_Free_automobile == 0 && mFree_pickup == 0 && normal_Free_pickup == 0)
        {
            write(STDOUT_FILENO, "\033[0;31m", sizeof("\033[0;31m") - 1);
            write(STDOUT_FILENO, "-> All temporary and normal parking spots are full. Exiting...\n", strlen("-> All temporary and normal parking spots are full. Exiting...\n"));
            write(STDOUT_FILENO, "\033[0m", sizeof("\033[0m") - 1);
            exit(0);
        }
    }
    return NULL;
}
```

Main

- This is the main function where the program execution starts.
- It handles user input for the maximum number of normal parking spots for automobiles and pickups, ensuring valid positive integers are entered.
- It initializes all semaphores used for synchronization.
- It creates threads:
    - **ownerThread**: Runs the **carOwner** function.
    - **attendantThread1** & **attendantThread2**: Each runs the **carAttendant** function with a different attendant ID (**0 and 1**).
- It waits for all threads to finish execution using **pthread_join**.
- It destroys all semaphores.
- It exits the program.

```c
int main()
{
    char buffer[10];
    while (1)
    {
        write(STDOUT_FILENO, "Enter max automobile parking spot: ", strlen("Enter max automobile parking spot: "));
        int bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer) - 1);
        if (bytes_read == -1)
        {
            perror("Error reading from stdin");
            return 1;
        }

        buffer[bytes_read] = '\0';
        int is_valid = 1;
        for (int i = 0; i < bytes_read - 1; i++)
        {
            if (!isdigit(buffer[i]))
            {
                is_valid = 0;
                break;
            }
        }

        if (is_valid)
        {
            normal_Free_automobile = atoi(buffer);
            if (normal_Free_automobile > 0)
            {
                break;
            }
            else
            {
                write(STDOUT_FILENO, "Please enter a positive integer.\n", strlen("Please enter a positive integer.\n"));
            }
        }
        else
        {
            write(STDOUT_FILENO, "Invalid input. Please enter an integer.\n", strlen("Invalid input. Please enter an integer.\n"));
        }
    }
```

```c
    while (1)
    {
        write(STDOUT_FILENO, "Enter max pickup parking spot: ", strlen("Enter max pickup parking spot: "));
        int bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer) - 1);
        if (bytes_read == -1)
        {
            perror("Error reading from stdin");
            return 1;
        }
        buffer[bytes_read] = '\0';
        int is_valid = 1;
        for (int i = 0; i < bytes_read - 1; i++)
        {
            if (!isdigit(buffer[i]))
            {
                is_valid = 0;
                break;
            }
        }

        if (is_valid)
        {
            normal_Free_pickup = atoi(buffer);
            if (normal_Free_pickup > 0)
            {
                break;
            }
            else
            {
                write(STDOUT_FILENO, "Please enter a positive integer.\n", strlen("Please enter a positive integer.\n"));
            }
        }
        else
        {
            write(STDOUT_FILENO, "Invalid input. Please enter an integer.\n", strlen("Invalid input. Please enter an integer.\n"));
        }
    }
```

```c
    printEmptySpaces();

    sem_init(&newPickup, 0, 0);
    for (int i = 0; i < 2; i++)
    {
        sem_init(&inChargeforPickup[i], 0, 1);
    }
    sem_init(&newAutomobile, 0, 0);
    for (int i = 0; i < 2; i++)
    {
        sem_init(&inChargeforAutomobile[i], 0, 1);
    }

    pthread_t ownerThread, attendantThread1, attendantThread2;
    pthread_create(&ownerThread, NULL, carOwner, NULL);
    int attendant1_id = 0;
    pthread_create(&attendantThread1, NULL, carAttendant, &attendant1_id);
    int attendant2_id = 1;
    pthread_create(&attendantThread2, NULL, carAttendant, &attendant2_id);

    pthread_join(ownerThread, NULL);
    pthread_join(attendantThread1, NULL);
    pthread_join(attendantThread2, NULL);

    sem_destroy(&newPickup);
    for (int i = 0; i < 2; i++)
    {
        sem_destroy(&inChargeforPickup[i]);
    }
    sem_destroy(&newAutomobile);
    for (int i = 0; i < 2; i++)
    {
        sem_destroy(&inChargeforAutomobile[i]);
    }

    return 0;
}
```

Program Flow

- The program initializes the number of temporary parking spots and prompts the user for the number of normal parking spots.
- It creates semaphores for controlling access to shared resources (parking spot counters).
- Two threads are created:

- **carOwner** thread continuously loops, simulating car arrivals.
    - It decides on the vehicle type (automobile or pickup) randomly.
    - It tries to acquire a free temporary spot for the chosen vehicle type.
    - If no temporary spot is available (both temporary and normal), it exits, indicating a full parking lot.
    - If all normal spots are full it exists.
- **carAttendant** thread waits for signals from carOwner indicating a new car arrival.
    - It tries to park the car in a free normal spot.
    - If no normal spot is available, it indicates no space for the car.
    - It repeats the same process for pickups arriving at the parking lot.
    - Similar to **carOwner**, it exits if all spots are full.
- The main thread waits for both carOwner and carAttendant threads to finish.
- Finally, the program destroys the semaphores and exits.