Cab Facility Management System

Introduction:

This project is a Cab Facility Management System designed to handle the booking, allocation, and management of cabs. The system includes functionalities such as creating, updating, deleting, and searching for cabs, managing cab requests, and allocating available cabs to users. It is built using Flask (for the API), SQLite (for database management), and threading/multiprocessing (for simulating real-time cab booking).

Key Features:

- Add new vehicles to the system
- Retrieve all available vehicles
- Search vehicles by type or vehicle number
- Update vehicle details
- Delete vehicles from the system
- Manage cab requests in a multithreading/multiprocessing environment
- Simulate user requests and allocate cabs accordingly
- Generate a JSON file for bookings and send booking confirmation via email

System Requirements and Setup:

System Requirements:

- Python Version: 3.x
- Libraries: `Flask`, `sqlite3`, `requests`, `BeautifulSoup`, `smtplib`, `json`, `threading`, `multiprocessing`

Setup Instructions:

1. **Install Python**: Ensure Python 3.x is installed on your system.
2. **Install Required Libraries**: Use the following command to install the required libraries:
Pip install Flask requests beautifulsoup4
3. **Run the Application**: Execute the script using the following command.
   Python3 cab_facility_management.py

Code Structure and Functions:

Code Structure

- **Database Functions**:Interact with SQLite database to store and retrieve vehicle data.
- **Flask Routes**: Define API endpoints to handle HTTP requests (POST, GET, PUT, DELETE).
- **Multithreading/Multiprocessing:** Simulate multiple users requesting cabs simultaneously.
- **Email Notification:** Send booking confirmations to users via email with JSON attachment.

Functions Overview:

1. **createVehicle(vehicle)**:
   - Purpose: Adds a new vehicle to the database.
   - Parameters: `vehicle` (Vehicle object with type, number, price per hour).
   - Returns: ID of the newly created vehicle.
2. **readVehicleById(id):**
   - Purpose: Retrieves a vehicle's details by its ID.
   - Parameters: `id` (Vehicle ID).
   - Returns: A vehicle object.
3. **readAllVehicles():**
   - Purpose: Retrieves a list of all vehicles in the system.
   - Returns: A list of vehicle objects.
4. **updateVehicle(vehicle)**:
   - Purpose: Updates a vehicle's details.
   - Parameters: `vehicle` (Vehicle object).
   - Returns: None.

5. **deleteVehicle(id)**:
   - Purpose: Deletes a vehicle by ID.
   - Parameters: `id` (Vehicle ID).
   - Returns: None.
6. **searchVehicle(vehicle_type, vehicle_number)**:
   - Purpose: Searches for vehicles based on type or vehicle number.
   - Parameters: `vehicle_type` (type of vehicle), `vehicle_number` (vehicle registration number).
   - Returns: List of vehicles matching the search criteria.

Multithreading Functions:

1. **request_cab(user_id)**:
   - Purpose: Allocates a cab to a user if available.
   - Parameters: `user_id` (ID of the user requesting the cab).
   - Returns: None (Prints cab allocation status).
2. **simulate_user_requests(system, num_users)**:
   - Purpose: Simulates multiple users requesting cabs at the same time.
   - Parameters: `system` (CabManagementSystem object), `num_users` (Number of users).
   - Returns: None.

Email Functionality:

1. **write_product_json(id)**:
   - Purpose: Writes booking details of a cab to a JSON file.
   - Parameters: `id` (Booking ID).
   - Returns: Filepath to the JSON file.
2. **send_mail(product_json, recipient_email):**
   - Purpose: Sends an email with booking details attached in a JSON file.
   - Parameters: `product_json` (path to the JSON file), `recipient_email` (email of the recipient).
   - Returns: None (Sends email and prints success/failure message).

User Interface and Interaction:

The system provides a menu-driven interface where users can interact with the system through the API endpoints and can manage cab booking operations via multithreading or multiprocessing.

API Endpoints:

1. POST /vehicle: Add a new vehicle to the system.
2. GET /vehicles/<id>: Retrieve vehicle details by ID.
3. GET /vehicles: Retrieve all vehicles.
4. PUT /vehicles/<id>: Update vehicle details by ID.
5. DELETE /vehicles/<id>: Delete a vehicle by ID.
6. POST /vehicles_search: Search for vehicles by type or number.

Example Usage:

- **Adding a Vehicle:**

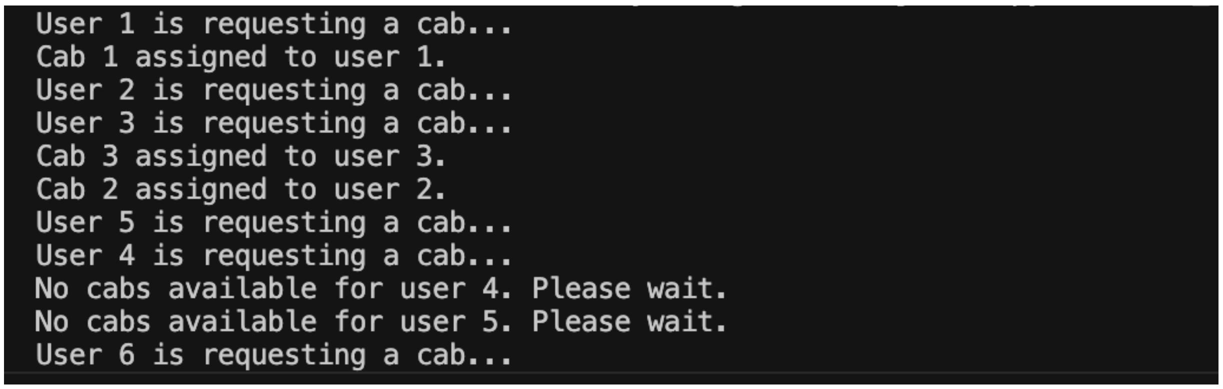
```bash
Curl -X POST -H "Content-Type: application/json" -d '{"vehicle_type": "SUV", "vehicle_number":
"CAB123", "price_per_hour": 50.0}' http://localhost:5000/vehicles
```

- **Retrieving All Vehicles:**

```bash
Curl http://localhost:5000/vehicles
```

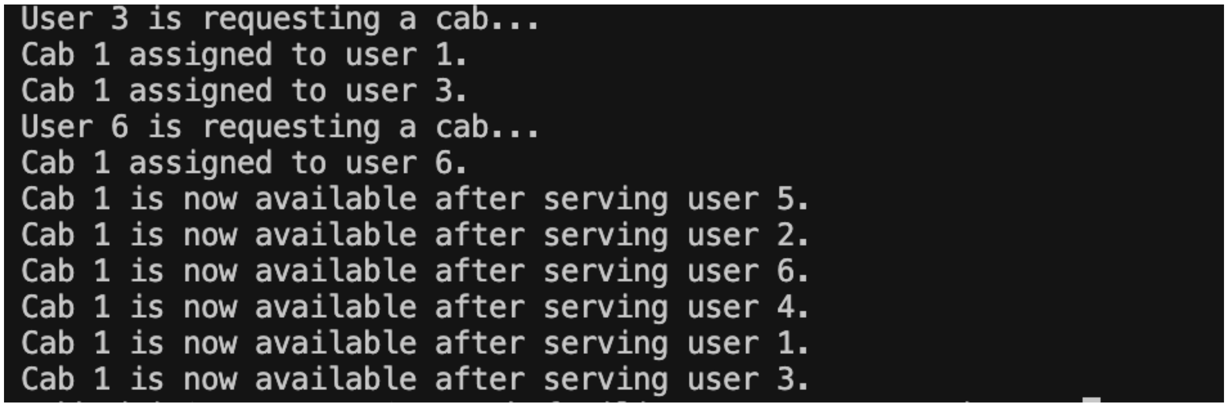
Multithreading / Multiprocessing Simulation:

- **Simulate user cab requests (multithreading):**



```
User 1 is requesting a cab...
Cab 1 assigned to user 1.
User 2 is requesting a cab...
User 3 is requesting a cab...
Cab 3 assigned to user 3.
Cab 2 assigned to user 2.
User 5 is requesting a cab...
User 4 is requesting a cab...
No cabs available for user 4. Please wait.
No cabs available for user 5. Please wait.
User 6 is requesting a cab...
```

- **Simulate user cab requests (multiprocessing):**



```
User 3 is requesting a cab...
Cab 1 assigned to user 1.
Cab 1 assigned to user 3.
User 6 is requesting a cab...
Cab 1 assigned to user 6.
Cab 1 is now available after serving user 5.
Cab 1 is now available after serving user 2.
Cab 1 is now available after serving user 6.
Cab 1 is now available after serving user 4.
Cab 1 is now available after serving user 1.
Cab 1 is now available after serving user 3.
```

Conclusion:

The Cab Facility Management System provides a streamlined solution for managing vehicle information and cab allocation in real-time. With Flask for API management, SQLite for data storage, and Python's threading/multiprocessing for simulating concurrent user requests, this project demonstrates an efficient way to handle large-scale cab operations.