Movie Ticket Booking System Documentation

This document provides a comprehensive explanation of the **Movie Ticket Booking System** implementation, including detailed descriptions of the functions, variables, and parameters. The system includes functionality for managing movies, booking and locking seats, handling administrative tasks, and providing a graphical user interface (GUI) for both users and administrators.

1. Database Initialization

Code:

```
import sqlite3

def create_db():
    conn = sqlite3.connect('Movie_Ticket_Booking_System.db',
check_same_thread=False)
    c = conn.cursor()

    c.execute('''CREATE TABLE IF NOT EXISTS movies (
                    id INTEGER PRIMARY KEY,
                    name TEXT,
                    hall TEXT,
                    time TEXT,
                    is_active BOOLEAN DEFAULT 1)''')

    c.execute('''CREATE TABLE IF NOT EXISTS seats (
                    id INTEGER PRIMARY KEY,
                    seat_number TEXT,
                    booked BOOLEAN,
                    user_name TEXT,
                    locked BOOLEAN,
                    movie_id INTEGER,
                    FOREIGN KEY (movie_id) REFERENCES movies
(id))''')

    c.execute('''CREATE TABLE IF NOT EXISTS logs (
                    id INTEGER PRIMARY KEY,
                    action TEXT,
                    seat_number TEXT,
                    user_name TEXT,
                    movie_id INTEGER,
                    timestamp DATETIME DEFAULT CURRENT_TIMEST
AMP)''')
    conn.commit()
    conn.close()
```

Explanation:

- `create_db` **Function**:
  - Establishes a connection to the SQLite database `Movie_Ticket_Booking_System.db`.
  - Creates three tables:
    1. `movies`: Stores information about movies (ID, name, hall, showtime, and active status).
    2. `seats`: Tracks seat booking status (seat number, booking status, user details, and movie association).
    3. `logs`: Maintains a log of all actions (e.g., booking, locking) with timestamps.

Key Variables:

- `conn`: Connection object for the database.
- `c`: Cursor object for executing SQL commands.

2. Movie and Seat Initialization

Code:

```
def initialize_movies_and_seats():
    conn = sqlite3.connect('Movie_Ticket_Booking_System.db',
check_same_thread=False)
    c = conn.cursor()

    movies = [
        ("The Amazing Spiderman", "Hall 1", "10:00 AM"),
        ("The Godfather", "Hall 2", "01:00 PM"),
        ("The Dark Knight", "Hall 3", "04:00 PM"),
        ("Inception", "Hall 1", "04:00 PM"),
        ("Oppenheimer", "Hall 2", "10:00 AM")
    ]
    c.executemany('''INSERT OR IGNORE INTO movies (name, hal
l, time, is_active) VALUES (?, ?, ?, 1)''', movies)

    c.execute('SELECT id FROM movies')
    movie_ids = [row[0] for row in c.fetchall()]
    for movie_id in movie_ids:
        for i in range(1, 21):
            c.execute('''
                INSERT OR IGNORE INTO seats (seat_number, boo
ked, user_name, locked, movie_id)
                VALUES (?, ?, ?, ?, ?)
            ''', (f'S{i}', False, None, False, movie_id))

    conn.commit()
    conn.close()
```

Explanation:

- `initialize_movies_and_seats` **Function**:

    - Populates the `movies` table with predefined movies and their details (e.g., hall and showtime).
    - For each movie, initializes 20 seats (`S1` to `S20`) in the `seats` table.

Key Variables:

- **`movies`**: A list of tuples containing movie details.
- **`movie_ids`**: List of IDs fetched from the `movies` table for assigning seats.
- **`S{i}`**: Seat identifiers (e.g., `S1`, `S2`).

---

3. Thread Synchronization

Code:

```
import threading
lock = threading.Lock()
```

Explanation:

- **`lock`**: A threading lock used to ensure thread-safe operations when booking or locking seats.

---

4. Seat Booking

Code:

```
def book_seat(seat_number, user_name, movie_id):
    with lock:
        conn = sqlite3.connect('Movie_Ticket_Booking_System.d
b', check_same_thread=False)
        cursor = conn.cursor()

        cursor.execute('SELECT booked, locked FROM seats WHER
E seat_number = ? AND movie_id = ?', (seat_number, movie_id))
```

```
        seat = cursor.fetchone()

        if not seat:
            conn.close()
            return f"Seat {seat_number} not found for movie I
D {movie_id}."

        if seat[1]:
            conn.close()
            return f"Seat {seat_number} is locked and cannot
be booked!"
        elif seat[0]:
            conn.close()
            return f"Seat {seat_number} is already booked!"
        else:
            cursor.execute('UPDATE seats SET booked = ?, user
_name = ? WHERE seat_number = ? AND movie_id = ?',
                           (True, user_name, seat_number, mov
ie_id))
            cursor.execute('INSERT INTO logs (action, seat_nu
mber, user_name, movie_id) VALUES (?, ?, ?, ?)',
                           ('Booked', seat_number, user_name,
movie_id))
            conn.commit()
        conn.close()
    return None
```

Explanation:

- `book_seat` **Function**:
  - **Parameters**:
    - `seat_number`: The seat identifier to be booked.
    - `user_name`: Name of the user booking the seat.
    - `movie_id`: ID of the movie associated with the seat.

- Checks if the seat exists and whether it’s locked or already booked.
- If available, marks the seat as booked and logs the action.

Key Variables:

- `seat`: A tuple containing booking and locking status of the seat.

---

5. Seat Locking

Code:

```
def toggle_lock_seat(seat_number, movie_id):
    with lock:
        conn = sqlite3.connect('Movie_Ticket_Booking_System.d
b', check_same_thread=False)
        cursor = conn.cursor()

        cursor.execute('''SELECT locked
                          FROM seats
                          WHERE seat_number = ? AND movie_id
= ?''',
                       (seat_number, movie_id))
        result = cursor.fetchone()

        if not result:
            conn.close()
            return f"Seat {seat_number} not found for movie I
D {movie_id}."

        current_lock = result[0]
        new_lock = not current_lock

        cursor.execute('''UPDATE seats
                          SET locked = ?
                          WHERE seat_number = ? AND movie_id
= ?''',
                         (new_lock, seat_number, movie_id))

        action = 'Locked' if new_lock else 'Unlocked'
        cursor.execute('''INSERT INTO logs
                            (action, seat_number, movie_id)
                            VALUES (?, ?, ?)''',
                         (action, seat_number, movie_id))

        conn.commit()
        conn.close()
    return new_lock
```

Explanation:

- `toggle_lock_seat` **Function**:
  - Toggles the lock status of a seat.
  - Logs the action as either 'Locked' or 'Unlocked'.

Key Variables:

- `current_lock`: The current lock status of the seat.
- `new_lock`: The updated lock status.

---

6. Reset Single Seat

Code:

```
def reset_single_seat(self):
    if not self.selected_movie_id:
        messagebox.showwarning("Warning", "Please select a mo
vie first.")
        return

    seat_to_reset = simpledialog.askstring("Reset Single Sea
t", "Enter seat number to reset:")
    if seat_to_reset in self.buttons:
        conn = sqlite3.connect('Movie_Ticket_Booking_System.d
b')
        cursor = conn.cursor()

        cursor.execute('''UPDATE seats
                          SET booked = ?, user_name = ?, lock
ed = ?
                          WHERE seat_number = ? AND movie_id
= ?''',
                       (False, None, False, seat_to_reset, se
lf.selected_movie_id))

        cursor.execute('''INSERT INTO logs (action, seat_numb
er, movie_id)
                          VALUES (?, ?, ?)''',
                       ('Reset', seat_to_reset, self.selected
_movie_id))

        conn.commit()
        conn.close()

        messagebox.showinfo("Admin Action", f"Seat {seat_to_r
eset} has been reset.")
    else:
        messagebox.showerror("Error", "Invalid seat number.")
```

Explanation:

This function allows an admin to reset the state of a specific seat for a selected movie. It ensures the seat is unbooked, unlocked, and cleared of user information.

Process:

1. Checks if a movie is selected; displays a warning if not.

2. Prompts the admin to input a seat number to reset.
3. Validates the seat exists in the seat grid.
4. Updates the database to:
    - Mark the seat as unbooked (`booked = False`).
    - Remove the associated user (`user_name = None`).
    - Unlock the seat (`locked = False`).
5. Logs the action with details in the `logs` table.
6. Displays a success or error message.

Key Variables:

- **`self.selected_movie_id`**: The currently selected movie's ID.
- **`seat_to_reset`**: The seat number entered by the admin for resetting.

---

7. Reset All Seats

Code:

```
def reset_seats(movie_id):
    conn = sqlite3.connect('Movie_Ticket_Booking_System.db', 
check_same_thread=False)
    c = conn.cursor()
    c.execute('''UPDATE seats SET booked = ?, user_name = ?, 
locked = ? WHERE movie_id = ?''', (False, None, False, movie_
id))
    c.execute('''DELETE FROM logs WHERE movie_id = ?''', (mov
ie_id,))
    conn.commit()
    conn.close()
```

Explanation:

This function resets all seats associated with a given movie by clearing their state and deleting related log entries.

Process:

1. Connects to the database.
2. Updates all seats for the given `movie_id` to:
    - Set `booked = False` (unbooked).
    - Set `user_name = None` (no user associated).
    - Set `locked = False` (unlocked).
3. Deletes all log entries related to the given `movie_id`.
4. Commits the changes and closes the database connection.

Key Variables:

- **`movie_id`**: The ID of the movie whose seats are being reset.

---

8. Get Seat Data

Code:

```
def get_seat_data(movie_id):
    conn = sqlite3.connect('Movie_Ticket_Booking_System.db',
check_same_thread=False)
    c = conn.cursor()

    c.execute('''SELECT seat_number, booked, user_name, locke
d
                 FROM seats
                 WHERE movie_id = ?''', (movie_id,))
    seats = c.fetchall()
    conn.close()
    return seats
```

Explanation:

This function retrieves detailed information about all seats associated with a specific movie.

Process:

1. Connects to the database.
2. Executes a query to fetch the seat data for the provided `movie_id`, retrieving:
    - **`seat_number`**: Identifier for the seat (e.g., `S1`).
    - **`booked`**: Whether the seat is booked (`True` or `False`).
    - **`user_name`**: The name of the user who booked the seat (if any).
    - **`locked`**: Whether the seat is locked (`True` or `False`).
3. Fetches all matching rows as a list of tuples.
4. Closes the database connection.
5. Returns the list of seat data.

Key Variables:

- **`movie_id`**: The ID of the movie for which seat data is being retrieved.
- **`seats`**: A list of tuples containing seat details (e.g., `[('S1', False, None, False), ...]`).

---

9. Get Movie List

Code:

```
def get_movie_list():
    conn = sqlite3.connect('Movie_Ticket_Booking_System.db', check_same_thread=False)
    c = conn.cursor()

    c.execute('SELECT id, name, hall, time FROM movies WHERE is_active = 1 OR is_active IS NULL')
    movies = c.fetchall()
    conn.close()
    return movies
```

Explanation:

This function retrieves the list of all active movies currently available in the system.

Process:

1. Connects to the database.
2. Executes a query to select details of movies that are active (`is_active = 1` or `is_active IS NULL`):
   - `id`: Unique movie ID.
   - `name`: Movie title.
   - `hall`: Screening hall.
   - `time`: Show time.
3. Fetches all matching rows as a list of tuples.
4. Closes the database connection.
5. Returns the list of active movies.

Key Variables:

- `movies`: A list of tuples containing movie details (e.g., `[(1, 'The Godfather', 'Hall 2', '01:00 PM'), ...]`).

---

10. Add Movie to List

Code:

```
def add_movie_to_list(name, hall, time):
    try:
        conn = sqlite3.connect('Movie_Ticket_Booking_System.d
b')
        cursor = conn.cursor()

        cursor.execute("SELECT COUNT(*) FROM movies WHERE nam
e = ? AND hall = ? AND time = ?", (name, hall, time))
        exists = cursor.fetchone()[0]

        if exists:
            conn.close()
            return False, f"Movie '{name}' in hall '{hall}' a
t time '{time}' already exists. Ignored."

        cursor.execute("INSERT INTO movies (name, hall, time)
VALUES (?, ?, ?)", (name, hall, time))
        conn.commit()

        cursor.execute('SELECT id FROM movies WHERE name = ?
AND hall = ? AND time = ?', (name, hall, time))
        new_movie_id = cursor.fetchone()[0]

        for i in range(1, 21):
            cursor.execute('''
                INSERT OR IGNORE INTO seats (seat_number, boo
ked, user_name, locked, movie_id)
                VALUES (?, ?, ?, ?, ?)
            ''', (f'S{i}', False, None, False, new_movie_id))

        conn.commit()
        conn.close()

        return True, f"Movie '{name}' added successfully."
    except Exception as e:
        return False, f"Error adding movie: {e}"
```

Explanation:

This function adds a new movie to the database along with its associated seats. Each new movie is assigned 20 seats automatically.

Process:

1. **Validate Movie**:
    - Checks if a movie with the same name, hall, and time already exists.
    - If it exists, returns a failure message and skips the addition.
2. **Insert Movie**:
    - Adds the new movie to the `movies` table.
    - Fetches the `id` of the newly added movie.
3. **Initialize Seats**:
    - Creates 20 seats (`S1` to `S20`) for the new movie and inserts them into the `seats` table.
4. **Commit Changes**:
    - Saves all updates to the database and closes the connection.
5. **Return Status**:
    - Returns a success message if the movie is added successfully or an error message in case of an exception.

Key Variables:

- **`name`**: The name of the movie to be added.
- **`hall`**: The hall in which the movie is being shown.
- **`time`**: The showtime of the movie.
- **`new_movie_id`**: The unique ID assigned to the newly added movie in the database.

---

11. Remove Movie from List

Code:

```python
def remove_movie_from_list(movie_id):
    try:
        conn = sqlite3.connect('Movie_Ticket_Booking_System.db')
        cursor = conn.cursor()

        cursor.execute("SELECT name FROM movies WHERE id = ?", (movie_id,))
        movie_result = cursor.fetchone()

        if not movie_result:
            conn.close()
            return False, f"Movie with ID {movie_id} does not exist."

        movie_name = movie_result[0]

        cursor.execute("UPDATE movies SET is_active = 0 WHERE id = ?", (movie_id,))

        cursor.execute('''INSERT INTO logs (action, seat_number, movie_id)
                          VALUES (?, ?, ?)''',
                       ('Movie Removed', movie_name, movie_id))

        conn.commit()
        conn.close()
        return True, f"Movie '{movie_name}' marked as inactive."

    except Exception as e:
        return False, f"Error removing movie: {e}"
```

Explanation:

This function removes a movie from the active list by marking it as inactive and logs the removal action.

Process:

1. **Check Movie Existence**:
   - Verifies if the movie with the given `movie_id` exists.
   - If not, returns a failure message.
2. **Mark Movie as Inactive**:
   - Updates the movie's `is_active` status to `0` to mark it as inactive.
3. **Log Action**:
   - Logs the removal action in the `logs` table with the movie's name and ID.
4. **Commit Changes**:
   - Commits the changes to the database and closes the connection.
5. **Return Status**:
   - Returns a success message if the movie is removed or an error message in case of an exception.

Key Variables:

- **`movie_id`**: The ID of the movie to be removed from the active list.
- **`movie_name`**: The name of the movie being removed, fetched from the database.

---

12. Generate Logs

Code:

```
def generate_logs():
    conn = sqlite3.connect('Movie_Ticket_Booking_System.db',
check_same_thread=False)
    c = conn.cursor()
    c.execute('SELECT action, seat_number, user_name, timesta
mp FROM logs ORDER BY timestamp')
    logs = c.fetchall()

    print("\nSession Logs Summary:")
    if not logs:
        print("No actions were performed during this sessio
n.")
    else:
        for log in logs:
            action, seat_number, user_name, timestamp = log
            if user_name:
                print(f"[{timestamp}] {user_name} performed
'{action}' on {seat_number}.")
            else:
                print(f"[{timestamp}] Admin performed '{actio
n}' on {seat_number}.")

    c.execute('SELECT seat_number, booked, user_name, locked
FROM seats')
    seats = c.fetchall()
    print("\nFinal Seat States:")
    for seat in seats:
        seat_number, booked, user_name, locked = seat
        status = (
            f"Locked" if locked else
            f"Booked by {user_name}" if booked else
            "Available"
        )
        print(f"{seat_number}: {status}")

    conn.close()
```

Explanation:

This function generates and prints a summary of session logs and the final state of all seats in the system.

Process:

1. **Fetch Logs**:
    - Retrieves all logs from the `logs` table, ordered by timestamp.
    - Prints each log entry, specifying the action performed, the seat number, the user (if applicable), and the timestamp.
2. **Fetch Seat States**:
    - Retrieves the final state of all seats from the `seats` table.
    - For each seat, prints whether it is **locked**, **booked**, or **available**.
3. **Close Connection**:
    - Closes the database connection after retrieving and printing the information.

Key Variables:

- **`logs`**: A list of tuples containing log details (`action`, `seat_number`, `user_name`, `timestamp`).
- **`seats`**: A list of tuples containing seat details (`seat_number`, `booked`, `user_name`, `locked`).

---

13. Book Seat (Threaded)

Code:

```
def book_seat_thread(seat_number, user_name, movie_id):
    def task():
        error = book_seat(seat_number, user_name, movie_id)
        if error:
            messagebox.showerror("Error", error)
    threading.Thread(target=task).start()
```

Explanation:

This function allows booking a seat asynchronously using threads, preventing the main UI from freezing while performing the seat booking operation.

Process:

1. **Thread Creation**:
    - A new thread is created and started to run the `task()` function.
2. **Task Execution**:
    - Inside the thread, it calls the `book_seat()` function to book the seat.
    - If an error occurs (e.g., seat is already booked or locked), it shows an error message using `messagebox.showerror()`.

Key Variables:

- **`seat_number`**: The seat identifier (e.g., `S1`).
- **`user_name`**: The name of the user attempting to book the seat.
- **`movie_id`**: The ID of the movie for which the seat is being booked.

---

14. Toggle Lock Seat (Threaded)

Code:

```
def toggle_lock_seat_thread(seat_number, movie_id):
    def task():
        result = toggle_lock_seat(seat_number, movie_id)
        if isinstance(result, str):
            messagebox.showerror("Error", result)
        else:
            state = "locked" if result else "unlocked"
            messagebox.showinfo("Admin Action", f"Seat {seat_
number} is now {state}.")
    threading.Thread(target=task).start()
```

Explanation:

This function toggles the lock status of a seat asynchronously, ensuring that the UI remains responsive during the process.

Process:

1. **Thread Creation**:
    - A new thread is created to execute the `task()` function.
2. **Task Execution**:
    - The `task()` function calls `toggle_lock_seat()` to toggle the lock status of the seat.
    - If an error occurs (e.g., invalid seat), an error message is shown.
    - If the seat is successfully locked or unlocked, a confirmation message is displayed to the admin.

Key Variables:

- **`seat_number`**: The seat identifier (e.g., `S1`).
- **`movie_id`**: The ID of the movie to which the seat belongs.

---

15. Booking GUI

Code:

```
class BookingGUI:
    def __init__(self, root, user_type, user_name=None):
        self.root = root
        self.user_type = user_type
        self.user_name = user_name
        self.buttons = {}
        self.selected_movie_id = None
        self.root.title(f"{user_type} Window")
        self.root.configure(bg="#2c3e50")
        if user_type in ["User", "Admin"]:
            self.create_movie_selector()
        self.create_grid()
        if self.user_type == "Admin":
            self.create_movie_management_section()
        if self.user_type == "User":
            user_windows.append(self)
```

Explanation:

The `BookingGUI` class provides a user interface for booking movie seats, with distinct functionalities for "User" and "Admin" roles. The constructor (`__init__`) sets up the window, creates the movie selector dropdown, seat grid, and other UI components depending on the user type.

Process:

1. **Initialize GUI**:
    - Sets the root window (`self.root`) and customizes the window's appearance.
    - **`user_type`** is used to distinguish between regular users and admins, and different UI components are rendered accordingly.
    - **`self.selected_movie_id`** initializes to `None`, as no movie is selected by default.
2. **Movie Selector**:
    - Calls `create_movie_selector()` for both users and admins to allow selecting a movie.
3. **Seat Grid**:
    - Calls `create_grid()` to display a grid of 20 seat buttons (`S1`, `S2`, ...). Each button is linked to a specific seat.
    - If the user is an admin, the grid includes options for resetting seats (`reset_all_seats`, `reset_single_seat`).
4. **Movie Management Section**:

  - Admin users get additional controls for adding or removing movies via `create_movie_management_section()`.

5. **User Management**:
   - If the user type is "User", the window is added to the `user_windows` list, allowing for multiple user sessions.

Key Variables:

- **`root`**:
  - Type: `Tkinter.Tk`
  - Description: The main window where all GUI elements are placed.
- **`user_type`**:
  - Type: `str`
  - Description: Defines whether the user is an "Admin" or "User". This determines which GUI components are shown.
- **`user_name`**:
  - Type: `str` (optional)
  - Description: The name of the user, used for personalized bookings. If not provided, assumed to be a system-generated user.
- **`buttons`**:
  - Type: `dict`
  - Description: Stores the buttons representing each seat. Keys are seat numbers (`S1`, `S2`, ...) and values are the corresponding Tkinter button objects.
- **`selected_movie_id`**:
  - Type: `int` or `None`
  - Description: Stores the movie ID of the selected movie. Initially `None` until a movie is chosen by the user.
- **`movie_mapping`**:

- Type: `dict`
- Description: Maps movie names (formatted as `Movie Name (Hall - Time)`) to their respective `movie_id`. Used in the movie selection dropdown.

Method Breakdown:

`__init__(self, root, user_type, user_name=None)`

- **Purpose**: Initializes the booking GUI window and adjusts its components based on the user type.
- **Parameters**:
  - `root`: The main Tkinter window object.
  - `user_type`: Specifies the role of the user ("User" or "Admin").
  - `user_name`: (Optional) Name of the user.
- **Process**:
  - Sets up the window's title and background.
  - Calls `create_movie_selector()` if the user is either a User or Admin.
  - Calls `create_grid()` to create the seat grid.
  - If the user is an Admin, it calls `create_movie_management_section()` for movie management options.
  - Adds the window to `user_windows` if the user is a regular user.

Related Methods:

1. `create_movie_selector()`:

- Creates a dropdown list of available movies for the user to select.
- Populates the dropdown with movie names, halls, and times.
- Calls `on_movie_selected()` when a movie is chosen.

2. `create_grid()`:

- Creates a 5×4 grid of buttons representing seats.
- Each button corresponds to a seat (`S1`, `S2`, ...), which can be clicked to book or lock the seat.
- Admin users have additional buttons to reset seats.

3. `create_movie_management_section()`:

- Creates GUI components for Admin users to add or remove movies.
- Provides input fields for movie name, hall, and time, as well as buttons to add or remove movies.

4. `update_ui()`:

- Periodically updates the UI to reflect the current status of each seat (booked, available, locked).
- Adjusts seat button colors based on their state and whether the user is an Admin or not.

5. `on_movie_selected()`:

- Handles the selection of a movie from the dropdown.
- Updates the seat grid to show the availability of seats for the selected movie.

Key Functionalities:

- **Seat Grid**:
  - Displays seat availability and allows users to book or lock seats.
  - Admins can also reset individual or all seats.
- **Movie Management for Admin**:
  - Admin users can add and remove movies, updating both the movie list and seat availability.
- **Real-time UI Updates**:
  - Seat states (booked, locked, available) are refreshed periodically to reflect changes in the database.

16. Main Function

Code:

```
def main():
    root = tk.Tk()
    user1_name = simpledialog.askstring("Input", "Enter name for User 1:", parent=root)
    user2_name = simpledialog.askstring("Input", "Enter name for User 2:", parent=root)

    if not user1_name or not user2_name:
        messagebox.showerror("Error", "User names are required to proceed.")
        return

    admin_window = tk.Toplevel(root)
    admin_gui = BookingGUI(admin_window, user_type="Admin")

    def launch_user_gui(user_name):
        user_window = tk.Toplevel(root)
        gui = BookingGUI(user_window, user_type="User", user_name=user_name)
        user_window.protocol("WM_DELETE_WINDOW", lambda: user_window.destroy())

    user1_thread = threading.Thread(target=launch_user_gui, args=(user1_name,))
    user2_thread = threading.Thread(target=launch_user_gui, args=(user2_name,))
    user1_thread.start()
    user2_thread.start()

    root.mainloop()
    generate_logs()
```

Explanation:

The `main()` function initializes the Tkinter window, handles user input for two users, and starts their respective GUI sessions in separate threads. Additionally, it creates an admin GUI window and handles the closing of all user windows.

Process:

1. **Create Main Window**:
    - Initializes the main Tkinter window (`root`).
2. **User Name Input**:
    - Prompts the user for names of two users using simple dialogs.
    - If names are not provided, an error message is shown and the function exits.
3. **Admin GUI**:
    - Creates an admin GUI window (`admin_window`) and initializes the `BookingGUI` for the admin user.
4. **Launch User GUIs**:
    - Defines a helper function `launch_user_gui()` to create a window for each user.
    - Starts two separate threads (`user1_thread` and `user2_thread`) to launch both user sessions concurrently.
5. **Main Event Loop**:
    - Calls `root.mainloop()` to start the Tkinter event loop, which keeps the GUI responsive.
6. **Generate Logs**:
    - After the main loop ends (when the application is closed), `generate_logs()` is called to generate session logs for the actions performed during the GUI

session.

Key Variables:

- `root`:
    - Type: `Tkinter.Tk`
    - Description: The main window for the application.
- `user1_name` and `user2_name`:
    - Type: `str`
    - Description: The names of the two users, obtained through input dialogs.
- `admin_window`:
    - Type: `Tkinter.Toplevel`
    - Description: A separate window for the admin GUI.
- `user1_thread` and `user2_thread`:
    - Type: `threading.Thread`
    - Description: Threads that run the `launch_user_gui()` function for each user concurrently.
- `generate_logs()`:
    - Type: `function`
    - Description: Generates a log of all actions performed during the session.

Process Flow:

1. The program starts by creating the main window and asking for user names.
2. Two user sessions are launched in separate threads, along with the admin session.
3. The Tkinter event loop runs to handle user interactions.
4. When the window is closed, logs are generated and the program ends.

---

17. Performance Measurement

Code:

```
import time
import threading

def book_seat(seat_number, user_name):
    time.sleep(0.1)
    print(f"Seat {seat_number} booked by {user_name}.")

def book_seat_thread(seat_number, user_name):
    threading.Thread(target=book_seat, args=(seat_number, use
r_name)).start()

def measure_performance():
    start_time = time.time()
    for i in range(1, 21):
        book_seat(f"S{i}", "User")
    single_threaded_time = time.time() - start_time

    start_time = time.time()
    threads = []
    for i in range(1, 21):
        thread = threading.Thread(target=book_seat, args=(f"S
{i}", "User"))
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()

    multi_threaded_time = time.time() - start_time

    print(f"Single-threaded execution time: {single_threaded_
time:.4f} seconds")
    print(f"Multithreaded execution time: {multi_threaded_tim
e:.4f} seconds")
```

```
    if multi_threaded_time < single_threaded_time:
        print(f"Multithreading is {single_threaded_time / mul
ti_threaded_time:.2f} times faster.")
    else:
        print(f"Multithreading is {multi_threaded_time / sing
le_threaded_time:.2f} times slower.")

measure_performance()
```

Explanation:

This script measures the performance of seat booking using both **single-threaded** and **multi-threaded** approaches.

Process:

1. `book_seat(seat_number, user_name)`:
    - Simulates the process of booking a seat by sleeping for `0.1` seconds to mimic some processing time and then printing a confirmation message.
2. `book_seat_thread(seat_number, user_name)`:
    - Creates a new thread to run the `book_seat()` function for the given seat and user.
3. `measure_performance()`:
    - Measures the execution time for booking 20 seats using two methods:
        - **Single-threaded**: Books all 20 seats sequentially.
        - **Multi-threaded**: Creates 20 threads, one for each seat booking, and waits for all threads to complete.
    - It compares the time taken for both approaches and prints the results.
    - It also calculates how many times faster or slower the multi-threaded execution is compared to the single-threaded execution.

Key Variables:

- `seat_number`:
    - Type: `str`
    - Description: Represents the seat to be booked (e.g., `S1`, `S2`).
- `user_name`:
    - Type: `str`
    - Description: The name of the user booking the seat.
- `single_threaded_time`:
    - Type: `float`
    - Description: Time taken to book all seats in a single-threaded manner.
- `multi_threaded_time`:
    - Type: `float`
    - Description: Time taken to book all seats in a multi-threaded manner.
- `threads`:
    - Type: `list`
    - Description: A list of threads used in the multi-threaded booking process.

Process Flow:

1. **Single-Threaded Execution**:
    - Calls `book_seat()` for each seat sequentially from `S1` to `S20`.
    - Measures the total execution time.
2. **Multi-Threaded Execution**:
    - Creates 20 separate threads, each booking a seat concurrently.
    - Waits for all threads to complete using `thread.join()`.
3. **Performance Comparison**:
    - Prints the execution times for both methods.

- Compares how much faster or slower the multi-threaded approach is compared to the single-threaded one.

Output Example:

```
Single-threaded execution time: 2.0000 seconds
Multithreaded execution time: 0.5000 seconds
Multithreading is 4.00 times faster.
```
