Hotel Booking Assignment

Concurrent & Distributed Systems

Seán Fahey – K00257361

Introduction

The purpose of this assignment was to create a Hotel Booking System in Java, with the focus being on the program's capabilities with regard to concurrency and multi-thread handling. The system I have implemented utilizes the Java *Semaphore* class, along with *ConcurrentHashMaps*, to provide individual access to the Hotel's booking resource; this specific Semaphore adaptation is a *binary semaphore*. I will now break down how this system works.

The Hotel Constructor

The Hotel Class contains only the binary semaphore and a ConcurrentHashMap (CHM) as local variables; The CHM contains Integers as its *key* and an object "Bookings" as its *values*. More specifically, the CHM maps the *room numbers* to bookings. This 'Bookings' object is a class that contains its own nested CHM, which maps the booking references (strings) to the days for which the room is booked.

```
2 usages
public class Hotel {
    4 usages
    private ConcurrentHashMap<Integer, Bookings> roomBookings; // links roomNumbers to Bookings (Booking References with Days Booked)
    3 usages
    private Semaphore s = new Semaphore( permits: 1); // binary semaphore - for concurrency, one single resource

    1 usage
    public Hotel(int[] roomNums){
        // create blank list of bookings
        roomBookings = new ConcurrentHashMap<>();
        for(int roomNumber : roomNums)
        {
            roomBookings.put(roomNumber, new Bookings()); // place blank bookings in each room number
        }
    }
```

Figure 1 - Hotel Constructor

Checking If A Room Is Booked

To check if a room is booked, with the setup I have created, all we must do is:

- Acquire the semaphore.

```
23          // acquire semaphore
24          s.acquire();
```

Figure 2 - Acquiring Java Semaphore

- Check if the room actually exists (by roomNum).

```
25          // ensure room number exists
26          if (!RoomExists(roomNum))
27          {
28              // the room number does not exist - return false
29              System.out.println("Room " + roomNum + " does not exist!");
30              return false;
31          }
```

Figure 3 - Ensure Room Exists

- Loop through the bookings and check for a match with the roomNumber on the specified day(s).

```
33          // check if the room is booked
34          for(int day : days)
35          {
36              // does the room number have the associated day in "days"? if so, the room is booked
37              if(roomBookings.get(roomNum).bookings.containsKey(day))
38              {
39                  return true;
40              }
41          }
```

Figure 4 - Checking If Room is Booked

- Finally release the semaphore again so other threads can access the bookings.

```
47          finally{
48              // release semaphore again
49              s.release();
50          }
```

Figure 5 - Releasing Java Semaphore

Booking A Room

Booking a room (or rooms) follows much of the same procedure as the previous step; acquire the semaphore, execute the required code to book the room, release the semaphore. I will quickly break down how the actual booking happens; the process involves two steps:

Step one: Check if the room has been booked, using the "roomBooked" method from before, returning false if so.

```
68          // check if any room has been booked already - return false if so
69          if(roomBooked(days, roomNum))
70          {
71              System.out.println("Room already booked.");
72              return false;
73          }
```

Figure 6 - Calling roomBooked function

Step two: Place the booking for the specified days inside the *bookings* map, with the associated booking reference.

```
74          else{
75              for(int day : days){
76                  // book the room for the days
77                  roomBookings.get(roomNum).bookings.put(bookingRef, day);
78              }
79              return true;
80          }
```

Figure 7 - Placing a Booking on a Room

Cancelling and Updating Bookings

To cancel bookings, we simply looking for the booking under the provided reference ID, then removing it if it is found. To update a booking, I opted to simply cancel the booking, then immediately create the new one with the new information.

```
120         boolean bookingFound = false;
121         // loop through bookings to find the specified reference ID
122         for (ConcurrentHashMap.Entry<Integer, Bookings> entry : roomBookings.entrySet())
123         {
124             Bookings bookings = entry.getValue();
125             // check if the booking reference exists
126             for (int day : bookings.bookings.keySet())
127             {
128                 if (bookings.bookings.get(day).equals(bookingRef)) {
129                     bookings.bookings.remove(day);
130                     bookingFound = true;
131                 }
132             }
133         }
```

Figure 8 - Cancelling a Booking by bookingRef

Handling Multiple Room Bookings

As part of the *extra credit* section of the assignment, I used many of the aforementioned functions to facilitate the booking of multiple rooms simultaneously; the key to implementing this effectively (which I learned the hard way through trial and error) was to ensure the deadlock did not occur between different threads due to sequential calls to booking functions (like bookRooms, roomsBooked, etc) being made immediately in the same function. This resulted in one function acquiring the semaphore, then calling another function, which attempted to acquire the semaphore before the initial function released it; both awaited the semaphore to become available, which would, of course, never happen. Once I discovered the root of the issue, I ensured that no attempts to acquire he semaphore were made unless the following actions were *atomic* i.e. they could execute without a call to any other function. Once this was done, the booking to multiple rooms worked perfectly.