Design - Project Two

Kaushik Nadimpalli
CS4348.001

Project Purpose: This project will study the coordination of multiple threads. The purpose is to simulate a hotel by using threads to model customer and employee behavior.

Implementation in: Java

Part 1 - List of Semaphores, purpose of each semaphore, and initial value of each semaphore

Semaphore Name	Purpose of Semaphore	Initial Value(initial number of permits available and fairness setting)
availableCustomer	This is a binary semaphore(no threads can hold it simultaneously). The purpose of this semaphore is to ensure that a guest is available to come to the front desk. The employee thread waits until the guest signals him. Similarly, it can only be released by the guest when an employee has served the guest.	(0, true) - Initial value is 0. (the thread waits until someone signals it on the basis of condition)
customerService	This is a binary semaphore. The purpose of this semaphore is to make sure that the guest is taken care by the bellhop. Guest has the semaphore and the bellhop has to wait until the guest signals for help, and then acquires it. Guest releases this semaphore after they acquire the bellhop semaphore.	(0, true) - Initial value is 0. (the thread wait until someone signals it on the basis of condition)

Semaphore Name	Purpose of Semaphore	Initial Value(initial number of permits available and fairness setting)
employee	This is a counting semaphore. The two front desk employee threads will be able to access this semaphore simultaneously. The purpose of this semaphore is to ensure that there are two employees serving one guest at a time. The guest waits on the employee. The employee signals this semaphore and releases it once an employee is free. Note that this can occur in two different buffers since its a counting semaphore and there are 2 front desk employees that can be serving different guests each at a time.	(2, true) - 2 employees will be able to access this semaphore simultaneously
servedEmployee	This is a binary semaphore. The purpose of this semaphore is ensure that guests are served. The employee thread acquires the semaphore after he or she has checked in the guest. After the guest is checked in and is about to leave the front desk, he/she signal the semaphore.	(0, true) - Initial value is 0. (the thread wait until someone signals it on the basis of condition)
allocatedRoom	This is a binary semaphore. The purpose of this semaphore is check for an available room, and allocate a room to a guest. Front desk employee signals the semaphore and it is acquired by the guest.	(0, true) - Initial value is 0. (the thread wait until someone signals it on the basis of condition)

Semaphore Name	Purpose of Semaphore	Initial Value(initial number of permits available and fairness setting)
bellHop	This is a counting semaphore. Since there are 2 bellhops that can simultaneously use this semaphore. It serves to make sure that guests get bellhops if they need them. When a bellhop is available, he or she signals/releases the semaphore and the guest acquires it.	(2, true) - 2 bellhops can access the semaphore simultaneously
baggage	This is a binary semaphore. The purpose of this semaphore is to ensure that the actual baggage has been delivered to the guest by the bellhop. Bellhop signals the baggage and the guest will acquire it.	(0, true) - Initial value is 0. (the thread wait until someone signals it on the basis of condition)
baggageService	This is a binary semaphore. The purpose is to ensure that the baggage service has been allocated to the guest. The bellhop signals the service and the guest acquires it(so they can give their bags to the bellhop).	(0, true) - Initial value is 0. (the thread wait until someone signals it on the basis of condition)
mutexOne	This is a mutex, a specific type of semaphore, where one thread has exclusive access to the semaphore. The purpose of this mutex is to ensure that guest has the semaphore to be part of the guest queue. After they are enrolled in the queue, the guest himself who held that semaphore must release it.	(1, true) - Initial value is 1. Only one thread is allowed to access the semaphore(mutex) exclusively

Semaphore Name	Purpose of Semaphore	Initial Value(initial number of permits available and fairness setting)
mutexTwo	This is a mutex, a specific type of semaphore as before. The purpose of this semaphore is for the employee to make sure that they allocate a room to the guest. The mutex is released by the same employee after room is allocated to the guest. Mutex is released employee checks that guest is removed from guest queue.	(1, true) - Initial value is 1. Only one thread is allowed to access the semaphore(mutex) exckusively
mutexThree	This is a mutex, a special type of semaphore. The purpose of this semaphore is for the bellhop to make sure that guest's baggage troubles are taken care of. After the bellhop makes sure of it, he then releases this mutex(must be same bellhop who waited for it)	(1, true) - Initial value is 1. Only one thread is allowed to access the semaphore(mutex) exclusively
serviceDone[] (Semaphore array)	This is a semaphore array. The purpose of this semaphore array is to keep track of how many guests were served. After a guest has been served, the employee releases the semaphore and another guest may acquire it.	[25] - Initial value is 0. It increments depending on how many guests are served.
roomOccupied[] (Semaphore array)	This is a semaphore array. The purpose of this semaphore array is to keep track of how many rooms have been occupied. The guest will signal this semaphore when a room has been occupied and the bellhop needs to wait for it so that he can deliver the baggage of that guest.	[25] - Initial value is 0. It increments depending on how many rooms have been occupied.

Fairness setting is the second parameter of initializing a semaphore(in java). The semaphore guarantees that threads invoking any of the acquire methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in-first-out; FIFO).

Part 2 - Pseudocode for each function with wait and signal calls

In the following Project, I have created four different classes. I have utilized the Runnable interface defines a single method, run, meant to contain the code executed in the thread.

Classes in program:

- **ProjectTwo** (which contains the main function for the simulation to run)
- **FrontDeskEmployee** (in charge of creating the two threads for employee and managing their functionality, printing each thread's actions)
- **Bellhop** (in charge of creating the two threads for bellhop and managing their functionality, printing each thread's actions)
- **Guest** (in charge of creating the 25 guest threads in the simulation and managing their functionality, printing each thread's actions)

Below are a list of all the functions between the different classes. Specifically, those that have the wait and signal calls are bolded and the pseudocode for those functions(similar to the Barbershop example) is provided below with the wait and signal calls.

All Functions/methods in each class:

- ProjectTwo
    - **main — entire simulation - in each class, each threads prints its own activities as the simulation goes on**
    - Constructor - public ProjectTwo
    - addedGuests()
    - run() - dummy class (overridden since we need the methods in the subclasses not the superclass ProjectTwo)
- FrontDeskEmployee
    - Constructor - public FrontDeskEmployee
    - **public void run()**
- Bellhop
    - Constructor - public Bellhop
    - **public void run()**
- Guest
    - Constructor - public Guest
    - **public void run()**

Pseudocode of functions with wait and signal calls

Note - In the actual code implemented in java, instead of wait and signal calls, there are acquire and release functions which are the java equivalent for semaphores. For the purposes of the pseudocode, we will list them as wait and signals. Furthermore, the following is the pseucodoe for the functions within the classes that facilitate the wait and signal calls for the semaphores. The pseudocode is similar to Barbershop example and hence the constructor method pseudocode is not included).

```
/* program Project2 (Hotel Simulation) */

semaphore availableCustomer = 0, customerService = 0;
semaphore employee = 2; //TOTALEMPLOYEES is a static integer equal to 2
semaphore servedEmployee = 0, allocatedRoom = 0;
semaphore bellHop = 2; //TOTALBELLHOPS is a static integer equal to 2
semaphore baggage = 0, baggageService = 0;
semaphore mutexOne = 1, mutexTwo = 1, mutexThree = 1;
semaphore serviceDone[25] = {0} //TOTALGUESTS is a static int equal to 25
semaphore roomsOccupied[25] = {0} //TOTALGUESTS is a static int equal to 25

public static int employeeCare[];
public static int bellHopCare[];
public static int newGuests;

public static Queue<Guest> gQueue;
public static Queue<Guest> bQueue;

public static final int TOTALEMPLOYEES = 2;
public static final int TOTALBELLHOPS = 2;
public static final int TOTALGUESTS = 25;
```

/* Class FrontDeskEmployee - We will see the wait and signal call behavior for the 2 front desk employee threads */

```
public Thread hotelemployee;
public int eNumber;
public static int ROOMNUMBER = 0;
void run()
{
        try {
                while (true)
                {
                        wait(availableCustomer);
                        waitUninterruptibly(mutexTwo);
                        hotelguest = ProjectTwo.gQueue.remove();
                        ROOMNUMBER++;
```

```
                        hotelguest.rNumber = ROOMNUMBER;
                        signal(mutexTwo);
                        employeeCare[hotelguest.gNumber] = eNumber;
                        signal(allocatedRoom);
                        System.out.println("desired formatted output");
                        signal(serviceDone[hotelguest.gNumber])
                        wait(servedEmployee);
                        signal(employee);
                }
        }
        catch(exception thrown)
        {
                //prints stack trace in program
        }

}
```

/* Class BellHop - We will see the wait and signal call behavior for the 2 Bell Hop threads */

```
public Thread bellhopboy;
public int bellHopNumber;
void run()
{
        try {
                while (true)
                {
                        wait(customerService);
                        waitUninterruptibly(mutexThree);
                        Guest hotelguest = ProjectTwo.bQueue.remove();
                        bellHopCare[hotelguest.gNumber] = bellHopNumber
                        signal(mutexThree)
                        System.out.println("desired formatted output");
                        signal(baggageService);
                        wait(roomOccupied[hotelguest.gNumber]);
                        System.out.println("desired formatted output");
                        signal(baggage);
                        signal(bellHop);
                }
        }
        catch(exception thrown)
        {
                //prints stack trace in program
        }

}
```

/* Class Guest - We will see the wait and signal call behavior for the 25 guest threads in this class */

```
public Thread hotelguest;
public static int gJoins = 0;
public ProjectTwo  hotelSimulation;
public int  gNumber; // guest number
public int  bNumber; // baggage number
public int  rNumber;
public static final int MAXBAGS = 5;
public static final int MINBAGS = 0;

void run()
{
        try {
                System.out.println("desired formatted output");
                wait(mutexOne);
                gQueue.add(this);
                signal(mutexOne);
                wait(employee);
                signal(availableCustomer);
                wait(serviceDone[gNumber]);
                wait(allocatedRoom);
                System.out.println("desired formatted output");
                signal(servedEmployee);

                if(bags > 2)
                {
                        wait(bellHop);
                        System.out.println("desired formatted output");
                        bQueue.add(this);
                        signal(customerService);
                        wait(baggageService);
                        System.out.println("desired formatted output");
                        roomOccupied[gNumber]
                        wait(baggage);
                        System.out.println("desired formatted output");
                }

                else
                {
                        signal(roomOccupied[gNumber]);
                        System.out.println("desired formatted output");

                }
```

```
        }
        catch(exception thrown)
        {
                //prints stack trace in program
        }
}
```

/* Main method - pseudocode for the entire simulation. Each threads will print its own activities when executed, but this is where they all come together. */

```
static void main(String[] args)
{
        hotelSimulation = new ProjectTwo();

        START SIMULATION

        for (int x = 0; x < TOTALEMPLOYEES; x++)
                new FrontDeskEmployee(x, hotelSimulation);

         for (int x = 0; x < TOTALBELLHOPS; x++)
                new Bellhop(x, hotelSimulation);

        for (int x = 0; x < TOTALGUESTS; x++)
                 new Guest(x, hotelSimulation);

      while (ProjectTwo.newGuests < TOTALGUESTS)
                System.out.print("");

      END SIMULATION
}
```