Allison Nguyen
CS 4348.501
Professor Ozbirn

Project 2 Summary

My program will simulate a Hotel using 2 Employee threads, 2 Bellhop threads, and 25 Guest threads. To start, the Project2 class will hold the main class. This will create threads and print that the simulation starts, ends, and that the guests have joined. The Hotel class will initialize and declare all the necessary semaphores in order for this simulation to run smoothly. The Hotel constructor will start the Hotel thread, which will then loop to create the rest of the threads in the main method.

Starting with the Employee thread, once created, the Employee constructor will print that it was created, then start the thread. The employee thread starts by waiting for the *guestsWaitingForCheckIn* semaphore and *employeeMutex* semaphore. This guides the check in process as the employees cannot check in guests if there are no guests waiting. The employee mutex helps ensure that only 2 employees are able to operate at a time. Once these 2 semaphores are acquired, the employee class will dequeue from the *queueForFrontDesk,* and increment the roomNumber. It sets the roomNumber attribute to the Guest instance and releases the *employeeMutex* and *employeeChecksGuestIn* as check in is complete and the employee can work with another Guest and the guest can go to their assigned room. The employee then waits for *frontDeskAvailable,* as the employees do not check in another guest until the front desk is free. Then after, the *employee* semaphore will be released to signal that there is an available employee.

The Bellhop thread is only in use when a guest has more than 2 bags and needs assistance. The Bellhop constructor will print that it was created and start the bellhop thread. The Bellhop thread waits for the *requestForBellhop* semaphore that is signaled by the Guest if needed. The Bellhop then waits for the *bellhopMutex* to be available to ensure that only one bellhop attempts to deliver bags to the same guest at the same time by dequeuing from the queue. Once this is done, the *bellhopMutex* is released so that other bellhops can access the queue. The Bellhop prints that it has received bags and releases the *bagsReceived* so that the guest can go to their room. The Bellhop waits for *roomReadyForGuest* to ensure that the Guest has arrived in their room before delivering their bags. After the bellhop then prints that the bags have been delivered and releases the *bagsDelivered* semaphore. The *bellhop* semaphore is then released so that the bellhops can now help other guests.

The Guest thread constructor prints when it was created, randomly generates a numBag value (0-5) and starts the thread. When the Guest thread runs, it prints that the guest has entered the Hotel and acquires the *guestMutex* to ensure that only one guest is trying to access *queueForFrontDesk* at a time. Once the guest joins the queue, it releases the *guestMutex* and waits for the *employee* semaphore to become available. The guest will signal that it is in queue and ready for check in. The guest will wait for the employee to check them in and print that they have received their key once done. The employee will release that the front desk is now available.

Now if the Guest has more than 2 bags, the guest will wait for the bellhop semaphore to be available, join the *queueForBellhop* and then request help. The Guest waits for bags to be received from the bellhop and continues to go to their room. Once in their room, the Guest will signal *guestEnteredRoom* so that the bellhop can deliver their bags. The Guest will wait for *bagsDelivered* and print that they have received their bags and tipped the bellhop. If the Guest does not have more than 2 bags, they will continue to enter their rooms.

Allison Nguyen
CS 4348.501
Professor Ozbirn

After the guests enter their rooms and receive their bags if they requested the bellhop, they will then retire.

One of the main difficulties I encountered while implementing this project was debugging the deadlocks and when my project would hang. I learned that it was better to implement small pieces at a time and checking that they worked accordingly rather than implementing large portions not knowing where the bugs would be. Another issue I had was being able to access the IDs in other classes for the print statements (example "Guest 2 receives bags from bellhop 1 and gives tip" required me to have access to the bellhopID so that the Guest class could print the attribute.). I addressed this by introducing two global arrays, *frontDeskIDs[]* and *bellhopIDs[],* which store the respective IDs for each guest index, so we can access them in the Guest class. One niche issue I had was that the "simulation ends" line would print before some "Guest joins" lines, indicating that the value of the *joinedGuests* that variable was 25 before the thread even printed that it joined. To fix this, I created functions for all print statements and had the *joinedGuests* variable increments after printing it had joined so that the "simulation ends" line prints at the very end. I also initially had my Guests join in the Guest class instead of the main method. To fix this, I created an array of Guests in my hotel class and when I created my Guest instances. In the constructor, I put each created guest thread in the guest array. In the main method in my Project2.java class, I used a for loop to join all the guests. I believe I was able to fully implement the project with all requirements. Overall, this project was a very insightful project to expose me to the miniscule details of threads and mutual exclusion. It was interesting how each output was different based on how the threads would run and which would access the semaphores and mutexes before the others.