For the second project, it was difficult, but not as difficult as the first project. For the first four days, I had great trouble figuring out how I wanted to make the Hotel Simulation program. Everything I tried either did not work at all or was filled with too many bugs. I was at a standstill until someone suggested to me that I use classes. Thus, I began creating the simulation with that in my mind.

The structure of my code is relatively simple. I have three classes in total, one for the guest, one for the front desk, and one for the bellhop. Each class contains any outputs to the screen needed for the simulation. I also have a separate function for the guest, front desk, and bellhop each. These three functions are what the threads run and where the simulation occurs. Finally, there is the main function. Inside of the main function, I create each of the threads and send their arguments to the function handed to the thread. I also join the guest threads here and indicate the start and end of the simulation. My semaphores and objects are created or initialized outside of any functions, making them global.

At first, this way of structuring my code didn’t work. I would get some sort of out of bounds error. My variables were also not being properly read nor was anything being outputted properly. At some point, I even started getting segmentation faults. I thought it was a problem with my semaphores at first. In the end, I completely ditched that code and started over from scratch. As I rewrote my code, I did something a bit different. In my main function, instead of creating the guest threads first, I created the front desk and bellhop threads first. Furthermore, I gave all my variables the volatile keyword. My changes worked, and I was finally making progress on the project.

The way my program outputs things is in no way confusing. The guest threads output that they have been created once they have entered the function specified. The first thing you’ll notice is that they are not seemingly created in numerical order. They are in order however. What is happening is whoever reaches the print statement first is the first to declare that they’re created. I use the speed of each thread to determine what order the guests should be processed in throughout the rest of the simulation. It’s also worth noting that for the interactions between the guest and the bellhop or front desk, those are the only times I use mutexes. I must force each thread to wait on each other there because the statements will be printed to screen out of order otherwise.

I learned a few interesting things while coding for this project. With the use of volatile, in some cases where I would’ve needed to use semaphores, I no longer needed to. For example, in the guest function, when I assign a random value to the randNum variable, the value would keep getting overwritten by other threads before I could get them into the localGuest object. Assigning the volatile keyword to randNum fixed that problem. So, by using volatile, I can cut down the number of semaphores I need to use. This didn’t always work however. During the guest’s entire interaction with the bellhop, volatile did not help prevent data races in the slightest. So, in cases like that, I had no choice but to use mutex-like semaphores. I also learned that who’s threads you start creating first matters quite a bit. It can determine whether your program will function properly or if it will fail to work and crash.

Overall, this was a very interesting project. Though it was very confusing at first, I learned a great deal from this. I feel like I have a very good grasp on semaphores now. In fact, this became more evident when my grade for exam two came back showing me that I only missed one semaphore question.