Project 2 Summary

Daniel Crawford([dsc160130@utdallas.edu](mailto:dsc160130@utdallas.edu))

10-18-2018

Beginning this simulation, I had a very short and simple pseudo code for how the code would work using semaphores. Each semaphore was singular which proved to be a problem later on. There was a significant issue not accounted for in the pseudo code and that was how the data should be exchanged. I planned on having several global variables for communication, but it turned out to be very difficult and ruined mutual exclusivity because two threads communicating required that they were the only ones running, without risk of the data being overridden. This flaw is dealt with by using arrays and then simply communicating with each thread the customer number so that they can use that as an index to the array. Another issue came with using singular semaphores as well. Because each semaphore can trigger any thread, sometimes the customer thread would go through earlier than expected because of some other customer that triggered the front desk thread and then another customer got the semaphore from the front desk. This issue became too difficult to deal with singular semaphores, and required too much of them. To fix this, an array of semaphores can be used which causes threads to send signals for specific other threads, so that other threads do not interpret the signals as their own. For this project, all the difficulties came from really not knowing what is available to use within the pthreads library. At one point, mutexes were used to communicate data between threads but not only was that against the project requirements, but it did not work like expected. After using mutexes, I realized that they function almost exactly alike as semaphores, but did not help solve the problem. To be specific, the hardest part of the project was setting up the semaphores so that they were ordered correctly, and then finding a way so that the threads can act independently of each other. The easiest way to get more independence for each thread is to use more memory to create more flags, which is a great trade off because it provides less mutual exclusivity, meaning better performance. Once I realized all these techniques I could use for the project, I created a new design using more semaphores by creating arrays of how many customers there are. This overall made the code look much less complex, and when first implemented into C++ it worked on the first try.

Testing with multiple runs, and multiple inputs at first provided very interesting and strange results. On the first implementation of the project, the program seemed to work fine when the semaphores were singular, when the customer size was equal to 3. However, whenever the size was increased, the program came into a deadlock. This deadlock was hard to diagnose, but definitely had to do with how the data was being exchanged. A conjecture on what this was, is that at one point in the bellhop thread the semaphore stopped and did not notify the other thread of the end of the exchange of data. This resulted in one thread stuck waiting to be able to exchange data, and the other thread stuck within the exchange process. Overall, this is was a result of the code being too complex, or too hard to follow, so this is one problem that was fixed in the re-implementation of the semaphores. With the new implementation, the project started working as expected on the first try, so with larger numbers of inputs it still behaved around the save. However, it is easy to give a number of customers that over allocates the amount of memory available. The program will seem to end on its own without any errors but it will obviously not finish the program. This is only for large numbers of customers though, for example one million. A typical computer would probably run out of memory for operations that large, so this is likely the case for this program.

The rest of the project besides the semaphores was very straight forward and did not take too much time to figure out. This involves everything that has to do with printing and setting up the semaphores and threads. Getting the threads to all start running is easy, however some threads such as the bellhop thread and the front desk thread are in an infinite loop. At the end of the main program, the program initially tried to join all those threads into the main process, but it would never finish because the infinite loop threads go out of scope. In the end, the loop was set up to only wait for the customer threads to finish running, and then the program worked terminated safely.