Project #2: Threads Summary

The DMV simulation was created to show how semaphores can be used to create mutually-exclusive sections of code and to communicate between processes using shared semaphores as flags. My initial design was correct for most of the semaphore actions, but did not account for some of the things that I learned during the project like getting the thread actors to reach a pthread\_exit() call. Originally I had defined the actors using simple while(true) {} loops to have them run until the process terminated and destroyed its associated threads. After running my program through valgrind, however, I realized that this was possibly causing memory leakage. I first joined my desk and announcer threads, and then checked my results in valgrind to see if my hunch was correct. The output showed I had two less possible memory leaks so I knew that I needed to join the other two agent threads as well to prevent any errors.

The desk, announcer, and customer threads were relatively easy to bring to an exit state - desk and announcer run exactly MAX\_CUSTOMERS times on input size = MAX\_CUSTOMERS, a predefined value. Customers exit when they leave the DMV. But agents must exit only when all of the customers have been processed, meaning that they must communicate with each other and with the customers. This resulted in a large section of code within the agent thread that must remain mutually exclusive to prevent errors in communication where one agent will still be waiting when the other agent has finished the last customer. This took me some time to realize – I first attempted to use multiple calls to the same semaphore within the agent thread to create several sections of mutual exclusivity, but this approach did not work as I still ran into concurrency issues. Then I enclosed the entire agent thread in a mutually exclusive semaphore, ran the program, and got 0 errors in valgrind. I knew that I was close but I was not happy with the amount of code in the critical section so I tried to remove as much as possible.

After getting the code to compile and valgrind without error on my machine, I transferred the project2.cpp file to cs1.utdallas.edu for testing. I compiled without error, ran the program a few times without a problem, and then set up a valgrind test. To my surprise, valgrind hung every time on a random customer, requiring me to input ^c (ctrl-c) before giving me output with references to pthread\_create() in the stack trace. Initially I thought there may have been an issue because the thread library referenced in the cs1 machine’s output was a 64-bit library and my machine uses a 32-bit library. After talking with you after class, it because clear that the real issue was that the number of posix threads allowed per process is implementation-specific, and is limited on the cs1 server. By lowering the number of MAX\_CUSTOMERS to 50, I was able to ensure successful compilation, execution, and memory leakage testing on the cs1 server. I was curious about where exactly the limit was being derived from, so I ran ulimit –a, and saw that my stack size is limited to 10 MB. This seems rather low, so I decided to check how much stack space each of my threads was using. I threw a call to pthread\_attr\_getstacksize() in my code and found the stack is size 10485760. That’s slightly larger than the 10MB stack limit, so I tried using a call to pthread\_attr\_setstacksize() to set the thread attribute object to use a 1MB stack. The code ran fine without any stack overflow, but valgrind randomly hangs like before with 100 customers.