This program uses threads and semaphore to coordinate a doctor’s office. The program contains 4 thread types: patient, receptionist, nurse, and doctor. The patient thread has a maximum of 15 possible threads it can run. The nurse and doctor threads have a maximum of 3 possible threads it can run. The threads used semaphores to communicate with each other and to control the execution order of each thread so that the output made sense when reading it. 5 semaphores had mutex behaviors mainly for controlling the id of the threads and the access if the nurse queue. One of the mutex-like semaphores was to control the patient-receptionist interaction so that the receptionist registers the patient one at a time as would be expected by a normal receptionist. A random number is used to assign the patient to the respective nurse and doctor. The receptionist will use this assignment to add the patient to the queue of its respective nurse and doctor. The receptionist will signal the exact nurse that the patient is ready and the nurse will come and remove the patient from the queue to take it to the doctor. The queue is a simple First-In-First-Out implementation to allow for the nurse to have an ordering of patients to process. The remainder of the simulation uses semaphores to communicate between the doctor and nurse specifically to communicate the patient ID. The rest of it is semaphore to communicate between the patient and doctor. Some semaphores are array semaphores which use the id of the patient or doctor in order to correctly signal the right thread to continue. This achieves the concurrency in that multiple doctor, nurse, and patient threads can run independently, and multiple examinations can occur.

The hardest part of this project was figuring out the logic behind terminating the threads. Specifically, the doctor and nurse thread terminations were very difficult. The solution I got was to use an integer flag which the receptionist would set to 1 as soon as he finished registering all the patients. This flag would allow nurses who didn’t have any patients in their queue to exit. For the doctor threads, the doctor was informed by the nurse thread’s exit that the doctor can exit after treating its last patient.

I learned how to properly implement concurrency with multiple threads. Specifically, I learned how to ensure effective communication between the threads. It wasn’t too difficult to figure out the basic semaphore setup, but the foresight of knowing that a certain semaphore was needed was key in making this project go smoother.

The results of the program speak for itself. The program can take in an input of the number of doctor and patient threads to run. And it will show the simulation and run successfully. The printed statements flow well when reading through them which is a result of the semaphore implementation.