Summary Anqi Chen

In this project, we learned how to synchronize multiple threads using semaphores and we simulated a post office using several semaphores.

First, we created 3 threads to simulate 3 post workers and also created 50 threads to simulate 50 customers. Since the process of creating threads does not involve any competition of resources, no semaphore was used in this case and there is no mutual exclusion required. We use a struct customer\_node to store the information for each customer such as customer number and random task assigned as well as a semaphore finished which is used to signal each customer whether the task has been finished by the post worker.

Within each customer threads, first of all, we need to check if there is any vacancy in the post office since only 10 customers are allowed to be in the post office at one time. Therefore, we use the semaphore max\_capacity to get mutual exclusion in this case. Once the customer is signaled by the max\_capacity semaphore, he can enter the post office. Then he will be put in line to be served by the post workers. Here, we added the customer in line using the enqueue function to add each customer\_node to a linkedlist. Since each time only one customer can be added in line (or only one post worker can remove a customer in line), we used another semaphore mutex to limit the number of access to the queue. In this way, only one thread at a time can modify the queue. After the customer was put in line, he will signal the post worker that he is ready to be served. Then the customer will wait until the post worker signaled him that the task has been finished. This is realized by the semaphore stored in each customer\_node which has been talked about before. After that, the customer will leave the office and signal max\_capacity so that the next customer waiting outside the post office can come in.

For each post worker thread, they have to wait for the customer to signal ready first. If no customer signals ready, that means there is no customer waiting in line so the post worker cannot remove a customer from an empty line. After the post worked is signaled ready, he will take the first customer waiting in line. Here we used the function dequeue to remove the customer from line and get the customer’s information. Since the customer\_node has all the information of the customer, the worker will know which customer he is serving and what task he is supposed to do. In this way, we would be able to know which post worker serves which customer. Again, the semaphore mutex is used to protect the dequeue function so that only one thread has access to the queue at one time. Then the post worker begins to serve the customer. If the task is to mail a letter or buy stamps, no mutual exclusion will be used since there is no resource competition between different post worker threads. However, if the task is to mail a package, then we need to use the semaphore scale to limit the number of access to the scale since only one post worker can use the scale at one time. After the task is finished, the post worker will signal the finished semaphore in the customer\_node. And also, the post worker will count how many customers have been served so far. Each time, only one post worker can increment the variable count, so we need another semaphore mutex2 to limit the number of access to the variable count. If the post worker found that the count reaches 50, which means he is the worker serving the last customer, he will need to signal other post workers that all customers have been served before exiting the thread. All other post workers at this time will be waiting for the customer\_ready semaphore, which is 0 because no customer are waiting in line and the post workers in this case will keep waiting. So the post worker that finishes serving the last customer will tell other workers by signaling the customer\_ready semaphore 2 times (number of post worker – 1). The other two workers after signaled by customer\_ready semaphore will check to see if count has reached 50. If true, they will exit the thread.

When I was working on this project, I found that the most difficult part is to signal the post workers when there is no customer waiting in line. Since only the post worker serving the last customer can find out count reaches 50, others at the same moment are already waiting for the customer\_ready, we have to think about a way to notify the other workers to exit the thread. If the workers serving the last customer exit directly without signaling the customer\_ready semaphore, the other workers will be still waiting for the customer\_ready semaphore and since no customer is waiting in line, they will wait forever.

Other things that take me a long time are the enqueue and dequeue functions. Since all threads in this project are accessing the same queue, we have to pass the head pointer and tail pointer of the queue by reference into the enqueue and dequeue function. At the beginning, I passed the head and tail pointer by value into the enqueue function. This does not change the object that these two pointers point to and after return from the function they are still pointing to the NULL. As a result, even though it calls the enqueue function several times, the queue is always empty.

From this project I learned a lot of multithreading and differences between multithreading and multiprocessing. Most importantly, I got to learn how multithreads share the same resources and how to use semaphore to synchronize these multiple threads and to realize mutual exclusion.