CS 4348 Multi-threading DMV Simulation Project Summary - Ethan Fischer

How to Compile and Run

project2.c is the main source file. Compile the project by entering the following command in your terminal.

```
gcc -pthread project2.c -o project2 -std=c99
```

project2 will be the output file. Then enter this command to run the program and view the output.

```
./project2
```

You should see 'Done' at the end of the output.

Project Purpose

This project uses multiple threads to simulate a DMV where there are many interactions happening between customers and employees concurrently. Each thread is a part of the same process, and therefore share memory. If they were to write or read from the same piece of memory at the same time, complications would arise. This is why we used semaphores to control the order of execution of each thread. The system has 5 threads. There are 20 customer threads, 2 agent threads, 1 announcer thread, and 1 information desk thread, and the main thread itself which creates and joins the other threads. The semaphores ensure that as the customer enters the DMV, they first go to the info desk which assigns the customer a ticket number, then they enter the waiting room where the announcer will call their ticket number. The customer will finally go to the agent line where the next available agent has them take a photo and eye exam and gives them their license. The objective in building this system is to develop a complete understanding of how semaphores are used to allow (and prevent) concurrency in multi-threading and how to handle issues with concurrency like deadlock and starvation.

Implementation

Details on the design of the system can be found in design.pdf

Here is an explanation of how each thread communicated and how the customers were queued up.

A customer is a struct I created for each customer thread, that keeps track of the customer's thread id, its ticket number, and the thread id of the agent that serves it.

```
typedef struct Customer {
    int threadid;
    int ticket_num;
```

```
    int agent_num;
}customer;
```

These customer "objects" are added to the correct queue in the customer thread so that the appropriate worker thread can dequeue and process the customer.

Because C doesn't have a queue object, I jerry rigged my own.

```
typedef struct myQueue {
    int next_index;
    int last_index;
    customer* queuePtr;
}my_queue;
```

`next_index` is the index of the next customer in line and is incremented in `dequeue()`. `last_index` is the index of the last customer in line and is incremented in `enqueue()`. `queuePtr` is a customer pointer to the first object in the array that is set up in `main()`. When the queues are initialized in `main()`, I malloc enough memory for each queue to have an array of customers that is `NUM_CUSTOMERS` long. (`NUM_CUSTOMERS` is how many customers will enter the system and is defined at the top. Technically, the number of customers is 20 according to project specification, but I used this instead of hard coding the 20 customers because this allowed me to run the system with 1, 2, 3, or 6 customers so I can make sure the system is working at a simpler level before adding in more customers.)

I created my own `enqueue()` and `dequeue()` functions, and they work exactly like you'd expect them to. Because C doesn't allow member functions in a struct, I had each function take a queue pointer so it knew which queue to operate on. `enqueue()` also takes a customer pointer to add to the queue. Both `enqueue()` and `dequeue()` return a pointer to a customer.

The `mutex` semaphoes are used to make sure that only one thread can `enqueue()` and `dequeue()` from a queue at a time.

Because a pointer to the customer is being added to the queue and then dequeued in the other thread (info desk, announcer, or agent), when the thread that dequeued that customer makes changes to it (`ticket_num` or `agent_num`) the customer thread has access to those changes in its customer struct.

My Experience

The very first issue I ran into was named vs. unnamed POSIX semaphores in C. The sample files provided to us all used unnamed semaphores which use `sem_init()` to create the semaphore. However, in macOS, despite being Unix based and allowing POSIX threads and semaphores, `sem_init()` is depracated. I could have worked on the project entirely within the UTD CS1 and CS2 machines which run on Linux, but that would have been a pain and meant I wouldn't have been able to easily work on the project locally on my machine. After looking into how to use semaphores on macOS, I discovered that I could use named semaphores which use `sem_open()`. They function the same way as unnamed semaphores but they actually have a string name that you give it in the `sem_open()` function. `sem_open()` takes a few more arguments as well, like the oflag and permissions, which I really struggled to, and still don't, understand. I used the arguments that I saw in an example and they worked so I stuck with it.

One of the biggest things I struggled with for so long was successfully transmitting data between the customer and info desk, announcer, and agent threads. Printing out the address of the customers being passed around allowed me to see that the references to the customer were not the same. By return a pointer to the customer that enqueued into the queue in `enqueue()` I was able to maintain the reference to the same customer so that changes made to it would be visible in any thread. This project taught me a lot more about pointers, passing by reference, and how memory is managed in C.

Another problem I ran into was the photo and eye exam steps of the agent interaction happening out of order. For example the customer would complete the photo and eye exam before the agent had asked it to. I was certain that my semaphores were correctly placed so something else was going on. While testing for solutions, I added another `sem_wait()` and `sem_post()` in the same place that the `sem_wait(photo_and_eye_exam_request)` and `sem_post(photo_and_eye_exam_request)` were, and that fixed the order of those steps for each customer. I messed around with moving that semaphore, removing it, removing the photo and eye exam request and just using that extra semaphore, etc. but nothing worked unless I had both adjacent to each other even though theoretically they did the same thing. I'm not sure why an extra semaphore there fixed the problem or why it wasn't working in the first place, but I decided to leave that semaphore there and call it `coord` and describe it as a coordination semaphore used to ensure the correct order of events.

After that issue had been resolved, I started to notice that sometimes the customer would get their license and depart before the customer had even given them the license. Sometimes an agent would give the customer their license before the customer had even completed the photo and eye exam. Huh? That doesn't make sense. I think what might have been screwing with this was that each of my customer/agent semaphores were arrays so that the correct customer would wait for their agent to signal to them specifically. When testing for a solution to this problem, I decided to treat each of those semaphores as a single semaphore for all customers, and that seemed to work. I couldn't figure out exactly what the problem was, but I was getting correctly printed results now while threads were still working concurrently.

Hands down the biggest problem I had with this project and why I'm yet again submitting this on a Sunday, the day after it was due, is the sheer length of the output. With 20 customers, there is over 200 lines of output. Combine this with extra output used for debugging, and you had an enormous output to sift through to debug. Debugging was such tedious and difficult task because so many things were happening concurrently that it was very difficult to determine if everything was happening in the correct order and it was very easy to miss errors. Unfortunately, there was no solution for this problem, I just had to suck it up and deal with it.