The largest part of my lab that helped make everything thread safe was modulation. In lab1 I had one massive class, Server, that did basically everything. Knowing the volatile nature of threads I decided to break everything up to consolidate what threads needed to touch.

Originally I was storing everything in a map. I moved the map and every method that interacted with the map to a data storage class. That class had a single semaphore that governed the access to the map. The vast majority of the algorithms associated with my server were moved to a ServerFacade. Every method from handle(), to send_response() was put into the ServerFacade, handle() was modified to handle only a single request rather than loop.

When a thread was created, it created a ServerFacade object. When a thread was given a client to serve, it called: facade.handle(client); the facade took care of the rest, referencing the data storage structure with a pointer.

The server class was reduced to only accepting clients, creating threads, and passing clients into a queue for the threads to systematically handle.

Bellow I have included the thread involved segments of Server. At run(), server creates 10 threads which then wait in handle_que() for clients to be added to the queue. The server then sits permanently in serve(), accepting clients and adding them to the queue.

```
struct handle_
{
   queue<int>* client_que;
   Database* data;
   sem_t* que_lock;
   sem_t* que_notEmpty;
   bool debug;
};

// thread function
void *
handle_que(void *ptr)
{
   // disassemble struct package
   struct handle_* package;
   package = (struct handle_*) ptr;
   queue<int>* client_que = package->client_que;
   Database* data = package->data;
   sem_t* que_lock = package->que_lock;
   sem_t* que_notEmpty = package->que_notEmpty;
   bool debug = package->debug;

   ServerFacade facade = ServerFacade(data, debug);

   while(1)
   {
      sem_wait(que_notEmpty);
      sem_wait(que_lock);
```

```
        int client = client_que->front();
        client_que->pop();
        sem_post(que_lock);
        bool success = facade.handle(client);
        if(success)
        {
            sem_wait(que_lock);
            client_que->push(client);
            sem_post(que_lock);
            sem_post(que_notEmpty);
        }
    }
}

void
Server::run() {
    // create and run the server
    create();

    // create and run 10 threads
    for(unsigned int i = 0; i < 10; i++)
    {
        struct handle_ package;
        package.client_que = &client_que;
        package.data = &data;
        package.que_lock = &que_lock;
        package.que_notEmpty = &que_notEmpty;
        package.debug = debug;

        pthread_t thread;
        pthread_create(&thread, NULL, &handle_que, &package);
    }
    serve();
}

void
Server::serve()
{
    // setup client
    int client;
    struct sockaddr_in client_addr;
    socklen_t clientlen = sizeof(client_addr);

      // accept clients, add to que, and post to semaphore
    while ((client = accept(server_,(struct sockaddr *)&client_addr,&clientlen)) > 0)
    {
        if(debug)
            cout << "SERVER:: serve()" << endl;
        sem_wait(&que_lock);
        client_que.push(client);
        sem_post(&que_lock);
        sem_post(&que_notEmpty);
    }
    close_socket();
}
```