a. Kevin Nguyen
b. List of what I done
    i. Create the listener function, test the semaphore sem_empty by giving the program a lot of requests and spawning a few workers. This will make sure that the queue is full which will let me check whether the listener is waiting for an open space.
    ii. Create the worker function, test the semaphore sem_full by not sending any request in the first few seconds to see if the worker is doing work when there is no request. Afterward, I would send a few requests and make sure that the workers are then doing the work. Additionally, I tested the mutex lock to make sure only one thread accesses the queue at a time so no concurrency problems occur by sending a lot of requests at once.
    iii. Implemented mult_threaded_server function, test that the listener and worker thread all spawn. Also test that when a thread fails, a new thread spawn by giving the program a 30% failure rate.
c. The design of my program consists of three components: the main thread, the listener thread, and the worker thread pool. The main thread first spawn off the listener thread and n worker thread. It will then infinitely loop through the worker threads and check if the thread terminates, if so, it will spawn a new worker thread. Both listener and worker thread access a shared data structure called the request queue, so I use semaphores and mutex to prevent concurrency problems. In the listener thread, it will check with a semaphore called sem_empty which will tell the listener thread whether there is space in the request queue. If there is space, it will place the request in the queue using a head variable and update the variable afterward. If there is no space, it will wait until a worker thread grabs the request and opens space by calling sem_post(&sem_empty). After putting a request in the queue, the listener will post to the sem_full semaphore. On the other hand, the worker thread pool is more complex since there are multiple threads. First, it will check with a semaphore called sem_full that tells the worker thread if there are any requests in the queue. If not, the worker thread will wait until there is a request. If there are requests, the correct number of worker threads will go through and attempt to retrieve a mutex lock to actually access the request queue. Only one thread at a time can take a request out of the queue and update the tail variable. Afterwards, it will release the mutex lock and post to the sem_empty semaphore. To test against race conditions and deadlocks, I overwhelm the server with multiple clients sending requests at once.