CSCI 4730 – Project 2

Gregory Woolsey

This project involves creating a multithreaded web server that uses threads to divide the workload between a number of clients connecting to the server. Using the given files, I created a producer and consumer method to keep track of the tasks, or clients, that were being sent to the socket in the req_handler() method. Then in the main method, I first created the listener thread that called the req_handler method which in turn called the producer class, filling an int array made for storing the tasks to later be divided between the worker threads. After all the tasks were stored in this buffer array, a loop is created to make the appropriate amount of worker threads, as determined by user input, making sure to reuse threads that became available after finishing their task. This was all tested by running the webserver file on a localhost server and port with different amounts of threads, and then in a separate terminal, running the given client file connecting to that localhost server and port as well as specifying the number of connecting clients to simulate. This updated multithreaded model showed a significant speedup compared to the original code.

In regard to the structure and synchronization of the threads, as stated above, within the loop of creating threads, each thread is reused once the loop reaches a point in which the maximum integer parameter becomes larger than the number of worker threads. Through the use of the pthread_detach function, each thread is able to run simultaneously without having to worry about waiting on any other worker thread, allowing the entire program to run with maximum efficiency as the thread creation does not have to wait for any of the worker threads to finish before creating or reallocating the next one. In addition, this also alleviates any concerns about race conditions or deadlocks, as the resources are evenly and consistently divided between the worker threads created or to-be-created. This is also shown through crash testing, as whenever a thread is forced to exit prematurely, a new thread is simply created in its place, taking up the same task as the previously exited thread.