Ben Williams
Lab 4
CprE 308

The purpose of this lab was to learn about the creation of threads, managing those threads, and learning how to write programs that can use threads to increase efficiency. Before I could begin, I began by reviewing the pthread library functions on man pages and creating the suggested example by the lab report.

To create an additional thread, memory space for the thread must be allocated by initializing a pthread_t type variable. Then, the function pthread_create() should be called on the pointer to that variable to initialize and start the thread. In pthread_create(), attributes about the thread as well as a pointer to the start routine that the thread should run are all designated.

Checking the return value of pthread_create() can tell us more about what's happening to our program at run time. It will either return an error code (a non 0 value) or 0 to show that the thread is running successfully.

The expected output of running the previous program was:

I am thread 1
I am thread 2

The actual output of running the program was:

I am thread 1
I am thread 2

It could have very well switched around the order of printing the threads since there were no joins involved in the sample program, aka the threads executed asynchronously.

For the task at hand, we were given 2 input file names and a single output file name. The 2 input files contained data for 2d matrices and the program was expected to print out the multiplied matrix of the two to the 3rd file. Some assumptions had to be made by the programmer for the convention the data was presented in, firstly being that all of the files were ASCII .dat files. Secondly, the input file values consisted solely of integers between 0 and 9, inclusively. Third of all, the input convention should be program InputA InputB Output. Finally, the program is only capable of square matrix multiplication, aka only 2 square matrices of equal sizes are given as inputs.

To start, I looked up how you even did matrix multiplication. I was rusty on the topic, and believed after some research that the way to multiply matrix A by matrix B was to take a certain row from matrix A and sum the product of each corresponding element in a column from matrix B.

I began the programming by checking for correct input convention (4 arguments.) I then attempted to open the first input file. Using fgets() to get strings into my line buffer, I iterated through the first line of the file and grabbed the size (following the convention #N = X, where X is the size of the array.) I checked to make sure that this size was gotten by the program.

I initialized a double pointer for my 2D array and used malloc to dynamically allocated enough space for this array at runtime (I only allocated as much as needed using the size of the matrix.) I then continued on to parse through the file and grab values for my first matrix. I use a counter variable so that I know which line I am on, since I want to skip parsing the first line of text.

I then closed file A and proceeded to do the same process for all of file B (fopen(), fgets(), fclose()). I made sure that the array size specified in B was the same as the array size specified in A, otherwise I printed out an error message and terminated the program.

Once I had both of my input arrays fully populated with data from the input files, I followed the same memory allocation procedure to create my third matrix for the output file. I also created a struct called matrix_Data that would be used as an argument for my thread start routine, calculate_Cell().

My overall plan for multithreading was to use each thread to calculate a single cell in the output matrix, so each thread needed to know which cell it was calculating. For calculating the output, I gave the thread the address of Matrix A and Matrix B, the X and Y as integers, the size of both arrays, and an integer for storing the output.

I created an array of 1024 matrix_Data structs to store data for my threads, and also created an array of 1024 threads. Linux's implementation makes it so that each process can only hold 1024 threads at a time. I created a thread counter variable that would limit the number of threads that were running at a time.

I iterated over every single x and y in the output, initialized the matrix_Data, started the thread, and then checked if 1024 or the final number of threads were running. If so, I would join every single thread using the thread counter (thus waiting until every thread was finished running), and update the data inside Matrix_C with the output from the matrix_Data array. I would then continue running through every X and Y in the output.

Once I had successfully computed all of the values inside Matrix C, all that was left to do was print out the calculated values into the output file using the same convention as the input files. I used fprintf() to update the file with the size (same as the size of input matrix A), as well as each element delimited by spaces.

Once the entire file had been printed, I simply closed the output file and returned as the program was finished.

My favorite part was realizing how useful multi-threading can be a lot of fun. A serial calculation of

the entire matrix would have taken decently longer, depending on the hardware in place. I thought it was cool learning more about dynamic memory allocation and making array of structures to be used as inputs to each of the threads.