Homework 4: Multithreaded Programming

Due: 11:59 PM, October 5, 2021

This homework assignment will require you to create a series of header files that I will be able to link to in order to run your code against test cases that I have designed. You will end up implementing header files for each of the included exercises (`ex_1.h`, `ex_2.h`, and `ex_3.h`). In each of these header files you will implement a provided function that is specific to the exercise at hand. You may also create and use other functions within that header file to decompose your problem and make it easier for you to manage. In fact, this will become necessary as you launch threads because you will need to specify new functions for those threads to begin executing.

On the CS-Data server, I have provided you with template files to work with. These files include `ex_1.h`, `ex_2.h`, and `ex_3.h` as well as `hw_4.c` which is used as a driver program to run and test the code you implement in the header files. Within `hw_4.c`, I've provided you with some testing code so that you can see how your implemented functions handle various input. My code provides sample input, checks it with the correct output, and prints whether your code would have passed or failed that test. Feel free to use and augment this code to test additional boundary cases (I certainly will do so when grading to provide more exhaustive test cases). You can find all of this template code provided for you in:

```
\\CS-Data\Courses\cmsc432\homework\hw_4
```

Ex 1 (20 points): Matrix Multiplication

Your first task is to write a function that multiplies two matrices together. The prototype for the function you will implement is:

```
void m_mult(int *mat1, int mat1_m, int mat1_n,
            int *mat2, int mat2_m, int mat2_n,
            int *mat3)
```

There are quite a few arguments (7) but they are not overly complicated. The first argument, `mat1`, represents an array of integer values which will be your first matrix. The values in the matrix are layed-out contiguously in memory in row-major order [1]. The dimensions of this matrix are `mat1_m`-by-`mat1_n`. Therefore there will be `mat1_m` rows and `mat1_n` columns. The fourth argument, `mat2`, represents the second matrix (also in row-major order) and its dimensions are `mat2_m`-by-`mat2_n`. The final argument, `mat3` is the matrix where you should store the result of the multiplication. As specified by the rules of matrix multiplication, its dimensions will be `mat1_m`-by-`mat2_n`.

Implement this function so that the results of the multiplication are stored in the array `mat3`. All arrays will have been previously `malloc`'ed so there is no need for you to perform your own memory management.

[1] See: `http://en.wikipedia.org/wiki/Row-major_order`

Ex 2 (40 points): Multithreaded Matrix Multiplication

Your goal for this exercise is to write a new function that will perform matrix multiplication. However, this time you will utilize pthreads to distribute the work that needs to be done. The prototype of the function is similar to the one in the previous exercise:

```
void m_mult_threaded(int *mat1, int mat1_m, int mat1_n,
                     int *mat2, int mat2_m, int mat2_n,
                     int *mat3)
```

Only the function name differs – the input arguments are the same as in the previous exercise. Ultimately, the result should also be the same. All that differs between these two functions will be the implementation.

Within this function, you are to launch **four** threads to help you perform the work of matrix multiplication. It is entirely up to you as to how you choose to divide the work, but you should make the workload division fairly equitable. You can have each thread producing results for different quadrants of the resulting matrix or even have them working on different rows, columns, alternating cells – your choice. Not all of these methods are as efficient as the others at dividing up the workload, so think about the strategy you choose. Part of your score will depend on how evenly the work is divided.

Remember that the end-result should be the same as the previous exercise, all that differs is the implementation.

Ex 3 (40 points): The Dining Philosophers

You can thank Edsger Dijkstra again for coming up with interesting examples – this one comes from him.

And the story goes: there were a number of philosophers sitting around a circular table. They were attending for dinner and each had their own plate of noodles. It just so happened that all of these philosophers required two forks to eat their noodles and, coincidentally, there were only just as many forks as philosophers present. These philosophers (who didn't mind the germs of their neighbors and are now quite dead) decided to share forks and had one fork placed between each adjacent pair of philosophers all the way around the table. The philosophers were not mindless eaters; they also took time to think. Therefore, if either of the forks to the left or right of a philosopher were currently occupied, a philosopher could be content to sit and think a while if unable to use both forks to eat. Certainly, however, everyone hoped to finish their meals eventually as the evening progressed.

Your goal is to simulate these dining philosophers. The prototype of the function you will implement is:

```
float dine_philosophers_dine(int numPhilosophers, int mealSize);
```

As you can see from the prototype, I will provide you with the number of philosophers seated at the table (which also represents the number of forks at the table) and the initial meal size for each philosopher. Each time a philosopher chooses to eat from their meal, you may decrease that philosopher's meal size by one. After taking a bite to eat, a philosopher will put down their forks and ponder life before reaching for another bite. The simulation finishes once all philosopher have had a chance to eat `mealSize` times. The return value is a float which will contain the ratio of the number of times the philosophers collectively attempted to eat to the total size of their collective meals. Note that this return value is not extremely important but more a statistic of interest. In fact, there are different (and perfectly acceptable) implementations of the simulation which allow for numbers equal to and greater than 1. No particular number is better than any other – it's just an interesting statistic that may relate to thread contention or contention from other external processes over the processors.

Submission: Turning in the assignment

To turn in your assignment, place your work on the CS-Data server (\\`CS-Data`) under the following directory within your student folder:

```
\\CS-Data\Students\your_user_name\cmsc432\hw_4\
```

Make sure your folder structure is identified **exactly** as it is outlined here or you will lose 5 points for not following directions and hindering my scripts which run and test your code.

After completing all of the exercises in this homework assignment, you should have (at a minimum) the following files/folders in your network drive:

```
\\CS-Data\Students\your_user_name\cmsc432\hw_4\ex_1.h
\\CS-Data\Students\your_user_name\cmsc432\hw_4\ex_2.h
\\CS-Data\Students\your_user_name\cmsc432\hw_4\ex_3.h
```

Again, points will be deducted if your files are not stored in exactly these locations. Make sure that you test your functions to ensure that they work in a variety of cases (including using test cases that I didn't provide to you in the example portion of each exercise).