CSE 425 Studio: Concurrency I

These studio exercises, and the next ones, are intended to give you experience working with techniques and mechanisms for managing concurrency (in C++, a multi-paradigm language, using threading features introduced by the C++11 standard).

In this studio you will again work in self-selected groups of ~3 people, and will report the results of your work on the studio exercises on the course message board. Students who are more familiar with the material are encouraged to partner with and help those who are less familiar with it, and asking questions of your classmates and professor during the studio sessions is highly encouraged as well. There are some links on the main course web page (under Textbooks and Other Resources) to helpful tutorial web sites on the concurrency (also called multithreading) and synchronization features that have been added in the C++ 11 standard, which you are also encouraged to use as resources. For invoking **`g++`** (on Linux) with the necessary **`-std=c++0x`** and **`-pthread`** switches in the optional enrichment exercises, the sample **`Makefile`** (available from the course web page, or directly at **`http://www.cse.wustl.edu/~cdgill/courses/cse425/Makefile`**) also may be useful.

Please record your answers as you work through the following exercises. After you have finished please e-mail your answers to the required exercises, and to any of the enrichment exercises you completed, to the cse425@seas.wustl.edu course account with subject line "Concurrency Studio I". The enrichment exercises are a good way to dig into the material a little deeper, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people of your choice, and write down the names of the team members as the answer to this first exercise.

2. Open up Visual Studio 2013 and create a new C++ Win32 Console project (e.g., named **`concurrency_I`**) Edit the program's main function so that it has the standard portable C++ main function signature (e.g., **`int main (int, char*[])`**) and outside the main function declare and define an output function that takes no arguments, returns no value, and prints out a message to the standard output stream. Call that function from the main function of your program. Build and run your program, and as the answer to this exercise, show the output your program produced.

3. Modify your code from the previous exercise so that instead of calling the output function directly, the main function declares a **`thread`** object (defined in the **`std`** namespace in the **`<thread>`** library that is provided as part of the C++11 feature set) and passes the name of the output function to the thread object's constructor. The main function then should call the thread object's **`join()`** method to wait for the thread to complete, before executing its own return statement. Build and run your program, and as the answer to this exercise please show your code and the output your program produced.

4. Modify your code from the previous exercise, so that instead of printing only a single line of output your output function iterates several times, printing out a number representing which iteration it is and then the line of text that it had output before. Instead of spawning (and joining with) a single additional thread, your main function should then declare a vector (or other container) of thread objects and push several thread objects back into it (with the name of the output function passed into the thread object's constructor each time). The main function should then iterate through the vector of thread object and call each one's **`join()`** method to wait for it to complete, before executing its own return statement. Build and run your program, and as the answer to this exercise (1) show your code, (2) show the output your program produced, and (3) describe whether or not the output you saw indicates contention for the standard output stream (i.e., one thread's output intermixing with another thread's output) and specifically why you think the threads did or didn't contend, and if they did how they were in contention.

5. Comment out the code from the previous exercise, and inside your main function declare three two-dimensional arrays of integers, of sizes **`p`** by **`q`**, **`q`** by **`r`**, and **`p`** by **`r`**, respectively (for your choice of sizes for those dimensions). Initialize the first two arrays using a reasonably arbitrary sequence of integer values, and then write loops that perform matrix multiplication of the first array by the second array and store the result in the third array. Print out the resulting values of the third array (in a legible format, i.e., organized in row, column order). Build and run your program, and as the answer to this exercise show your code, and the output the program produced.

6. Modify your code from the previous exercise so that instead of a single thread performing all the work, the work is divided up among multiple threads that then write to independent locations within the third array (thus avoiding any race conditions between the threads since their only interactions are then read-only). For example, you might want to give each thread its own sub-range of the destination array.

In order to do that you should write a separate function that takes the addresses of the three arrays and one or more other values (e.g., row and/or column index values) indicating the input and/or output ranges it should operate over. Then, in your main function, when you spawn the different threads you should pass the threads' constructors not only the name of that function but also the parameters with which that thread should call it (giving each different thread a different range so that each necessary operation is performed by some thread but no output location in the third array is written to by more than one of the threads.

Build and run your program, and as the answer to this exercise show your code and the output the program produced, and say whether or not the same output was produced compared to the previous exercise (and why or why not).

PART II: ENRICHMENT EXERCISES (optional, feel free to skip or to try variations that interest you)

7. Repeat exercises 5 and 6 with different sizes of arrays. You may want to measure (and print out) how long it takes each version to run using statements within the programs themselves, or you can see whether or not you notice a significant difference in performance between the single-threaded and multi-threaded versions of the program. Also repeat the same set of trials with arrays of double precision floating point numbers, instead of integers. As the answer to this exercise please indicate whether or not you noticed any differences in how long the programs took (relative to the sizes of the arrays, the types of values being multiplied, and for each size/type combination relative to the other implementation) and why you think what you observed happened.

8. On the Start menu in Windows, open up All Programs→Internet→SSH Secure Shell and then click on Secure Shell Client. In the window that appears, click on Quick Connect to bring up a connection dialog window. Fill in **`shell.cec.wustl.edu`** as the host name, fill in your CEC login id below that, and click on the Connect button. Fill in your CEC password in the prompt that appears, and click OK. At the Linux shell prompt in the window that appears, create a new directory for your code for this lab (e.g., **`mkdir concurrency_I`**) , and change to that directory (e.g., **`cd concurrency_I`**). Transfer the code from any of the previous exercises into that directory, and as needed open up an editor of your choice (e.g., **`emacs`**, **`vi`**, or **`pico`**) and use it to edit the code (and depending on the files you created, possibly also the **`Makefile`** that you can download from the course web site). Build your program using **`g++`** (e.g., by updating the **`Makefile`** and then issuing the **`make`** command in that directory) and run it, and as the answer to this exercise show the output your program produced and discuss whether or not (and if so how) the program's behavior on Linux differed from its behavior on Windows.