CSE 425 Studio: Concurrency II

These studio exercises, and the next ones, are intended to give you additional experience working with techniques and mechanisms for managing concurrency (in C++, a multi-paradigm language, using threading and synchronization 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 II". 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_II`**) and in the directory for the source and header files for that project, make a copy of your code from the previous studio exercises and add those files to your new project, since in these exercises you will continue to use and extend the concurrency features you have already implemented. In your main function for this studio, declare a single thread object that calls the output function you wrote in the previous studio exercises on concurrency. If you have not done so, modify the output function so that it uses a loop to print a message repeatedly (a fixed number of times), and modify your main function so that it joins with that thread object before returning. Build your program and observe its behavior (i.e., the messages it outputs).

Then, in your program include the **`<mutex>`** library, and declare a global (static) variable of type **`mutex`** (you will either need to open up the standard namespace or declare it using the scope **`std::`**). In the output function, add a statement within the loop (i.e., just before the statement that prints out the message) that calls the lock method of the global mutex variable (but does not call its unlock method). Build and run your program, and as the answer to this exercise please say (1) how the behavior of your program differed with and without the lock statement, and (2) why it differed.

3. Modify your code from the previous exercise so that instead of being of type **`mutex`** the global variable is of type **`recursive_mutex`**. Build and run your program, and as the answer to this exercise please say (1) how the program's behavior differed with a **`mutex`** versus with a **`recursive_mutex`**, and (2) what that tells you about the different locking semantics for **`mutex`** versus **`recursive_mutex`** variables in the C++11 concurrency libraries.

4. Modify your code from the previous exercise so that instead of creating and joining with a single thread the program creates and joins with multiple threads, each of which calls the output function. Modify the output function so that in addition to locking the **`recursive_mutex`** before executing the output statement, after the output statement it calls the **`unlock`** method on the **`recursive_mutex`**. Build and run your code with (1) different numbers of threads running the output function, (2) different numbers of messages being output by the output function, and (3) with and without (i.e., by commenting and un-commenting them) the statements that lock and unlock the **`recursive_mutex`** at each iteration of the loop. As the answer to this exercise, please describe (1) how the program's behavior differed with and without locking and unlocking of the **`recursive_mutex`**, and (2) why that difference occurred (i.e., what critical section of code was or was not being protected in each case)?

5. Modify your code from the previous exercise so that it declares two global (static) variables of type **`recursive_mutex`**. Modify the output function so that it (1) locks one of the **`recursive_mutex`** variables, (2) prints out a message, (3) locks the other **`recursive_mutex`** variable, (4) prints out another message, and then (5) unlocks both variables. Then, declare and define a second output method that is identical to the other output method, except that it locks and unlocks the **`recursive_mutex`** variables in the opposite order as the first output method (i.e., it locks the second variable first and the first variable second). In your main function, give some of the threads the first output function to run, and the other threads the second output function to run. Build and run your program, and as the answer to this exercise please indicate (1) whether you were able to see the threads deadlocking (if not, try with different numbers of threads and/or numbers of times the output functions loop), and (2) if so what output demonstrates that there was a deadlock.

6. Modify your code from the previous exercise so that both functions acquire and release the locks in exactly the same order. Build and run your program, and as the answer to this exercise please then indicate (1) whether or not a deadlock occurred, and (2) why it did or why it did not.

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

7. Repeat exercise 5 with the output functions acquiring and releasing the locks in the opposite order, but modified so that instead of grabbing each lock directly they repeatedly use **`try_lock`** until they acquire the lock (alternatively, you could use a **`timed_mutex`** for the lock type, and have the output functions try repeatedly to lock in a similar manner but with each unsuccessful attempt timing out). Build and run your program, and as the answer to this exercise please say (1) whether or not that approach (or each of the approaches if you tried both of them) was able to prevent deadlock, and if so (2) how it did that.

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_II`**) , and change to that directory (e.g., **`cd concurrency_II`**). 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 the **`Makefile`** (which you can download from the course web site if you have not done that already). 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.