CSE 425 Studio: Concurrency III

These studio exercises are intended to expand further your 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, 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 links on the main course web page 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. 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 in the last optional enrichment exercise, especially for invoking the **`g++`** compiler with the necessary **`-std=c++0x`** and **`-pthread`** switches.

Please record your answers as you work through the following exercises. After you have finished please send e-mail with your answers to the required exercises, and to any of the enrichment exercises you completed, to the cse425@seas.wustl.edu course e-mail account, with subject line "Concurrency Studio III". 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_III`**) and modify the main function's signature to be the standard portable one we have used in previous studios.

Implement a struct for a bounded stack, which has a pointer to integer (for the values in the stack) and an unsigned integer (for the maximum size of the stack) as member variables, a public **`max`** method that returns the value of the unsigned integer member variable, a destructor that calls **`delete`** on the pointer member variable, and a constructor that: (1) takes a single unsigned integer parameter, (2) in the base/member initialization list uses the passed parameter value to initialize the unsigned integer member variable and initializes the pointer member variable to 0, and (3) in the body of the constructor, if (and only if) the unsigned integer member variable's value is greater than zero, uses the **`new`** operator to allocate (dynamically) a buffer of integers of the size given by the value of that unsigned integer member variable, and stores the memory address returned by the **`new`** operator in the pointer member variable.

In your main function declare objects of the struct type you just declared and defined, passing different maximum size values to their constructors. Call the **`max`** methods of those objects and print out the values they return. Also print out the addresses stored in the pointer member variables of those objects. Build and run your program, and as the answer to this exercise please show your code and the output your program produced.

3. Add another unsigned integer member variable to the class, which the constructor should initialize to 0, which will count how many values are on the stack. Add a **`push`** method that takes a single integer value as a parameter and first checks whether the count is less than the maximum size of the stack, and if not throws an exception. Otherwise (if the count is less than the maximum size of the stack) the **`push`** method should assign the passed value to the position in the (dynamically allocated) buffer of integer values (pointed to by its pointer member variable) that is indexed by the count member variable, and then increment (add 1 to) the value of the count member variable.

Modify your main function from the previous exercise so that after it constructs each stack object it calls the **`push`** method of that object repeatedly from within a try/catch block with different integer values, until it catches an exception and then should print out all of the values that are stored in the stack object's buffer. Build and run your program, and as the answer to this exercise please show your code and the output your program produced.

4. Add a **`pop`** method that takes no arguments and returns a single integer value. The **`pop`** method should first check whether the count member variable is greater than 0 and if not should throw an exception. Otherwise (if the stack is not empty) the **`pop`** method should decrement (subtract 1 from) the value of the count member variable and return the value found at the position in the (dynamically allocated) buffer of integer values (pointed to by its pointer member variable) that is now indexed by the value of the count member variable (i.e., indexing should be done using the value ***after*** it was decremented).

Modify your main function from the previous exercise so that after it catches an exception from calling the **`push`** method repeatedly for each object, it then (again within a try/catch block) repeatedly calls the object's **`pop`** method and prints out the value that was returned by that method, until it catches an exception that indicates the stack object is empty. Build and run your program, and as the answer to this exercise please show your code and the output your program produced.

5. Modify your code from the previous exercise so that different sequences of calls to the **`push`** and **`pop`** methods are made on the same stack object from within different threads (please apply the techniques from the previous studio exercises to do that using C++11 features) and use **`cout`** to print the values that are pushed and popped. Build and run your program, and observe how the program behaves.

Then, add a single global mutex to the program and use it to synchronize all uses of **`cout`**, and again observe how the program behaves. Finally, add a mutex member variable to your stack struct and use it to synchronize the **`push`** and **`pop`** methods (please apply the techniques from the previous studio exercises to do that using C++11 features), and again observe how the program behaves. As the answer to this exercise, please describe (and show output traces illustrating) how the behavior changed from (1) when the program was single threaded, to (2) when the program ran with multiple threads but was unsynchronized, to (3) when the program was multi-threaded but used a global mutex to synchronize only the uses of **`cout`**, to (4) when the multi-threaded program also used a mutex member variable in each stack object to synchronize accesses to its **`push`** and **`pop`** methods.

6. Add a condition variable as a member of the stack struct, and modify the push and pop methods so that instead of throwing exceptions they should check whether the queue is full (in the case of the push method) or empty (in the case of the pop method) and if so they should ***repeatedly*** (i.e., in a loop) atomically release their mutex lock and wait on the condition variable (both are achieved atomically by calling the **`wait`** method on the condition variable) until the queue is no longer full or empty, respectively. Otherwise (or once the queue is no longer fully or empty, respectively) the method should complete the appropriate code to push or pop a value, and then (as the very last thing they do before unlocking the mutex and exiting) make a call to the **`notify_all`** method on the condition variable.

Build and run your program, and as the answer to this exercise please show your code and describe how the behavior of the program changed, in comparison to the behavior in the previous exercise.

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

7. Using the code from the previous exercise, add an additional unsigned integer parameter (which is defaulted to 0) to the stack object's constructor, for an "offset" that corresponds to how far from full (or empty) the stack should be when the program waits in the **`push`** (or **`pop`**) method.

If the passed offset value is less than the stack's size, the **`push`** method should subtract it from the stack's size to calculate a "high water mark". If a valid high water mark was calculated then ***after*** the **`push`** method first detects that the stack is full and waits on the condition variable for the first time, it should use the high water mark instead of the queue's size in the condition expression that determines whether it will ***continue*** to wait on the condition variable. That is, once the stack is full and the **`push`** method is waiting, it should wait until the queue drains all the way down to the high water mark, before continuing.

Similarly, if the passed offset value is less than the queue's size, the **`pop`** method should use it (as a "low water mark") instead of 0 in the condition expression that determines whether or not it will continue to wait on the condition variable. That is, once it discovers the stack is empty, the pop method will wait until the stack count reaches the low water mark, before continuing.

Construct stack objects of different sizes and with different offset values, and run multiple threads pushing and popping values into and out of the stack object at once, again using the global mutex to synchronize printing out the values pushed and popped in each thread, as in the previous exercises. Build and run your program, and as the answer to this exercise please describe how the program's behavior changed with different sizes and offset values, along with any other observations you made.

8. Repeat either or both of the previous exercises, with the **`push`** and **`pop`** methods calling **`notify_one`** instead of **`notify_all`**, and as the answer to this exercise describe whether you saw any differences in the program's behavior as a result of doing that (and if you did, what differed).

9. Repeat any of the previous exercises on shell.cec.wustl.edu using g++ and the Makefile that was provided for doing that in the previous studios. As the answer to this exercise please describe whether any changes were needed (and if so which ones) in order to port your code to g++ on Linux, and whether you observed any differences (and if so what they were) between the platforms.