Multithreaded Code
Cpt S 422 Homework Assignment
by Evan Olds

Assignment Instructions:

Read all instructions *carefully* before you write any code.

In this assignment you will build a C# class library (DLL) containing several classes that meet various requirements. Although most of the general requirements are described later, there are a few things to note right off the bat:

1. For each class that you add to your DLL, you will be told the name of the namespace and class. These must match EXACTLY. If you are told to create a class named MyClass, then creating a class named myClass, myclass, my_class, My_Class, or any other slight name variant will be marked as incorrect.
2. The target platform for testing is Linux. Although C# is generally pretty good with respect to consistent functionality across different platforms, you must test on Linux before submitting. If your app doesn't work on Linux during grading, you will not be given credit, even if the app worked fine on Windows or Mac OS. Ideally you'd test your code on all 3 platforms to ensure that you can write good, platform-independent code. But in the interest of keeping things simple, you'll be graded only on Linux.
3. Following from the previous point, you will need to install Mono in Linux. If your Linux distribution supports apt-get for package management, then the following terminal command will usually suffice for installing Mono:
   sudo apt-get install mono-complete
   - See this page for official and more detailed installation instructions: http://www.mono-project.com/docs/getting-started/install/linux/
4. You will submit only code files (.CS) for the DLL project. You can test your DLL in whatever way you see fit, such as writing a standalone application that loads the DLL or using the unit testing framework in some IDE. Just make sure that the test code is not submitted (unless it is specifically requested in the instructions). Submit ONLY the code files needed to compile the DLL.

Part 1: Implement the **PCQueue** class (in PCQueue.cs)

Create a class named **PCQueue** in the CS422 namespace. Give it its own .cs file named PCQueue.cs. Match the file and class names exactly. It should not inherit from anything. This is the producer/consumer class that will have been discussed in lecture. Your task is to implement it without using any thread-specific functionality and merely relying on the fact that assignment of references is atomic in C#.

The queue will store data as a singly-linked list. For simplicity, the class enqueues and dequeues integer values, so each node only needs to have an integer for the data value and a reference to the next node in the list.

The queue must be safe for use on 2 threads at once, with the guarantee that only one thread will enqueue and the other will only dequeue. Recall that you cannot use locks, nor anything that's designed for thread-safe operations. A mutex, wait handle, critical section, atomic integer increment/decrement, or any other similar thing cannot be used. You should be able to implement the queue using only the fact that reference (and integer) assignment is atomic. We will have discussed the idea behind the solution in class.

Implement only the PCQueue class (and Node for the PCQueue) in the PCQueue.cs file. Do not put any other classes in this code file. You should not need ANYTHING that's not in the System namespace. Therefore, at the top of your code file, other than some comments, you should only need "using System;" and nothing else. Match the function signatures for the 2 functions listed below and have a parameter-less constructor (that initializes an empty queue).

```
public bool Dequeue(ref int out_value)
public void Enqueue(int dataValue)
```

Recall that Dequeue returns false immediately if the queue is empty. Otherwise it dequeues a value, puts it in out_value, and returns true. The Enqueue function should always succeed, provided there is enough memory to allocate the node, hence the void return type. In addition to this, make sure the following criteria are met:

- Must not deadlock
- Must not just keep all values in memory and never remove them. Every dequeue that removes an item should advance a reference and leave a node ready to be freed by the garbage collector.
- Must be O(1) for enqueue and dequeue
- Must not have busy-waiting (using 100% of CPU in a loop while waiting for another thread to complete some action)

Part 2: Implement the **ThreadPoolSleepSorter** class (in ThreadPoolSleepSorter.cs)

Create a class named **ThreadPoolSleepSorter** in the CS422 namespace. Give it its own .cs file named ThreadPoolSleepSorter.cs. Match the file and class names exactly. It should not inherit from anything. This is the "sleep sort" implementation that we will have discussed in class. Recall that this algorithm displays a collection of numbers in sorted order by spawning a "task" (thread) for each number X, having the task sleep for X seconds, then displaying X.

As we discussed in class, to ensure that the time taken to create multiple threads doesn't ruin the algorithm, we will have a collection of worker threads already created before the Sort function is called. Add a constructor with the following signature:

```
public ThreadPoolSleepSorter(TextWriter output, ushort threadCount)
```

- The first parameter is the TextWriter used to display the numbers after sleeping. Have the constructor store a reference to it in a member variable. You may assume it has a thread-safe implementation and therefore a bunch of different threads call all call its WriteLine function safely.
- The **ushort** parameter indicates the number of threads to create. The constructor will create that many threads and put them all in a waiting state so that they can be quickly utilized when needed later. If the parameter value is 0, then default to 64 threads, otherwise create the exact number of threads requested.

Add a public member function with the signature below that actually does the sort. Remember that the sort doesn't modify the array of values, it simply prints them out in sorted order. For each number X in the array, get a thread from the thread pool, have it sleep for X seconds and then write X to the TextWriter using WriteLine. Make sure you use WriteLine and not Write for each number. So the result should be a sorted version of the list with one number per line.

```
public void Sort(byte[] values)
```

You may assume that the number of threads passed to the constructor of your class is greater than or equal to the size of any array you'll be asked to sort. In other words, there will be at least 1 thread per value. Also, make sure that your threads from the thread pool stay alive after doing a task. You must not dispose a thread and create a new one afterwards. Each thread should must live on to potentially complete additional tasks. Therefore, you should be able to call Sort several times on one instance of a ThreadPoolSleepSorter and have it work correctly each time.

Final Notes:

- Although C# console/terminal applications run pretty consistently across different platforms, test your code in Linux, since this is the OS that will be used for grading.
- Make sure that your classes are public, so that they can be seen and accessed by applications that reference your DLL.
- Recall that you must only put .CS files in your .zip and you must be able to compile your code from the command line with: **mcs –target:library *.cs**
- Your zip file must not contain any nested folders. Zip from the command line to ensure this.
- Submit to the online system well before the due date, record the submission ID, and look up the feedback. Deal with issues and resubmit if need be before the due date/time.