Assignment 1 – Synchronization primitives in .NET

Exercise 1 – Multicore characteristics

Use the Performance tab in the Task Manager to answer the following questions.

- how many cores do you have?
- Does your machine support hyper-threading, the notion that each core is further divided into independent "sub-cores" (execution pipelines)?
- Finally, what are the cache characteristics of your machine? Level 1? Level 2? Level 3?

Exercise 2 – Lock vs. Mutex

In a program create two threads each incrementing a shared variable 100.000 times. The result should obviously be 200.000. Run the program. Does it count as expected?

Try to use a lock object to control exclusive updating of the shared variable. Is the counting done correctly now?

Now try to replace the lock object by a Mutex for synchronizing the writing to the shared variable.

What happens if the threads "forget" to release the Mutex?

How does the Mutex perform compared to using a lock object?

What would be most beneficial and safe to use in this case, lock or Mutex?

Exercise 3 – Semaphore

Create a program with two threads – a Producer thread and a Consumer thread. The Producer thread should each second add an increased value (starting at 1) to a shared array of size 10 (a FIFO queue implemented as a circular buffer). Add an EXIT value (say -1) as the last produced value.

The Consumer thread should each two seconds remove a value from the shared array and print out the value.

Try to use a Semaphore to control the insertion into the queue so the producer will block if the queue is full. Try in the same way to use a Semaphore to control the removal from the queue so the consumer will block if the queue is empty.

Exercise 4 – Semaphore

Add one more consumer thread to the program. The consumer may not print out the values sequentially, but each number should only be printed once. Is the program working correctly?

Can you cancel both Consumer threads when one has caught the EXIT value?

Exercise 5 – Thread-safety

Create a generic *ThreadSafeQueue<T>* class to encapsulate the circular buffer array. Add two methods to the class: An *Add* method for adding to the queue and a *Remove* method to remove and return the first value of the queue. Also move any needed Semaphores to this class, so the use of the *TreadSafeQueue* will be totally transparent to the callers.

Do you think this is a preferable and safer way of coding?

Exercise 6 – Thread-safe collections in .net

Try to find out if there exist such thread-safe collections in .net.
If so, replace your ThreadSafeQueue<T> by the found thread-safe collection.