Description: Using Threads to Calculate Sets of Prime Numbers

Setup

Open the Lab3 solution provided.

The solution contains the Program.cs and IOFunctions.cs files. Program will be where you create most of the thread processing code. IOFunctions contains useful methods for querying data from the user via the Console.

You will add three more files in the following steps.

Definitions

Prime Number – A number, greater than 1, that it divisible only by itself and 1

Examples: 2, 3, 5, 7, 11

Factor – A number that is evenly divisible into another number

Example: 3 and 2 are factors of 6

MathStuff

Create a new class called *MathStuff*. This class is **public** and **static**.

IsPrime

The *MathStuff* class contains two methods. The first one is a **public static** method, *IsPrime*, that accepts a **ulong** called *number*, and returns a **bool**.

The first step of *IsPrime* is to eliminate any simple values and calculate the simple cases.

If *number* is less than or equal to 1, return **false**.

If *number* is 2, 3 or 5, return **true**.

Next, return **false** for any even numbers. Since we already looked at 2, any even number at this point is not prime. You could use a modulus operator to test for evenness. However, a more efficient way is to look at the first bit of the number. All odd numbers must have the first bit set as it is the only bit that is odd for integers. All other bits are evenly divisible by 2. Using the **&** operator, we need to check the first bit (2^0) which happens to equal 1.

```
if ((number & 1) == 0) return false;  // First bit not set
```

We can also do a quick check for values that are multiples of 5.

```
// Check for numbers divisible by 5
if (number % 5 == 0)
{
   // Any multiple of 5 (except 5) is not prime
   return false;
}
```

Now, due to the nature of prime factors, we can use the following theorem to our advantage:

If n is a positive composite integer, then n has a prime divisor less than or equal to sqrt(n)

What this means is that in our calculations to test for prime we only have to check for factors up to the square root of our target, *number*. Let's create a **ulong** called *max* and set it equal to Math.Sqrt(*number*). You will have to cast the result to **uint**.

Next, create an **int** called *counter* and set it to 3.

Another final optimization is within our loop we can eliminate any number that ends with 5 since those values are always divisible by 5. We took care of the value 5 in the above check so we don't have to worry about that special case. So, all prime numbers must end with 1, 3, 7 or 9. We will use the *counter* field, defined above, to make sure we don't test values ending with 5.

Create a **for** loop, using an **int** *i* as the looping value, that starts with 3, test for $i \leq max$, incrementing by 2.

```
for (ulong i = 3; i <= max; i += 2)
```

The reason we are starting with 3 is because 1 is not a valid factor to test as all prime numbers are divisible by 1. Also, evens are never prime except for 2. We are using *counter* to monitor when we are testing for multiples of 5. When it is equal to 4, reset it to 0 and skip to the next iteration. Within the **for** loop, add the following test:

```
if (counter == 4)
{
   counter = 0;
   continue;
}
counter++;
```

The next step within the **for** loop is to do the actual test for prime. We can use the modulus operator for this task. If the remainder of *number* divided by *i* is not 0, then *number* cannot be prime as it has a whole factor:

```
if (number % i == 0)
{
   // Found a factor, not prime
   return false;
}
```

If the **for** loop completes without finding a factor, then we can return **true** as *number* is prime.

ApproximateNumberOfPrimes

The next method of MathStuff will approximate the number of primes that will be found between 2 and a given number. There are many techniques for arriving at this number, but we'll use a simple one that is not super accurate but good enough for our purposes. The formula, sometimes called pi(x), equals:

$$x / (\ln x - 1)$$

We are going to modify this formula to add an additional "fudge-factor" of 5% to the result.

```
public static ulong ApproximateNumberOfPrimes(ulong x)
{
   double num = ((double)x / (Math.Log((double)x, Math.E) - 1)) * 1.05;
   return (ulong)num;
}
```

SafeQueue<T>

SafeQueue<T> is a generic queue that uses a **ReaderWriterLockSlim** to allow for safe multi-threaded access. This class emulates System.Collections.Concurrent.ConcurrentQueue<T> which would normally be the class of choice for this task. However, we are going to create our custom version to practice synchronization.

Fields

Define the class *SafeQueue*<T> as public. Within the class define two fields:

```
private ReaderWriterLockSlim m_Lock = new ReaderWriterLockSlim();
private Queue<T> m_Items = null;
```

The first field, *m_Lock*, will be the synchronization construct we'll use to prevent simultaneous writes into the second field, *m_Items*.

Properties

Next, we'll add an **int** property called *Count* that returns the number of items in the queue. Since we only need read access into the queue, we will use a read lock while getting the item count.

```
public int Count
{
   get
   {
      m_Lock.EnterReadLock();
      try
      {
         return m_Items.Count;
      }
      finally
      {
         m_Lock.ExitReadLock();
      }
   }
}
```

Notice that the read lock is acquired prior to the **try** block and released within the **finally** block. This way when we acquire the lock we are guaranteed it will be released.

Constructor

The constructor for this class accepts no parameters and simply initializes *m_Items* as a new **Queue**<T>:

```
m_Items = new Queue<T>();
```

Methods

The first method we'll add will simply clear the queue of all contents. This method will look very much like the *Count* property in that we will acquire the lock before the **try** block and release it in the **finally** block. The biggest differences are that we need a write lock and we have no return value:

```
public void Clear()
{
   m_Lock.EnterWriteLock();
   try
   {
      m_Items.Clear();
   }
   finally
   {
      m_Lock.ExitWriteLock();
   }
}
```

Next, we need a method to add an item into the queue. Like the **Queue**<T> class, we'll call the method *Enqueue* which will accept an item of type T. Again, we'll need a write lock as we are changing the contents of the queue.

```
public void Enqueue(T item)
{
   m_Lock.EnterWriteLock();
   try
   {
      m_Items.Enqueue(item);
```

```
        }
        finally
        {
            m_Lock.ExitWriteLock();
        }
    }
```

Since we also need the ability to remove items from the queue, we will add a method called *TryDequeue*. We are calling this method *TryDequeue* because like the **TryParse** function of standard types we want to fail gracefully if no items exist. As a result, the return value will be a **bool**, not T. Instead we will have an **out** parameter to set the value if successful.

```
    public bool TryDequeue(out T value)
    {
        value = default(T);
        m_Lock.EnterWriteLock();
        try
        {
            if (m_Items.Count > 0)
            {
                value = m_Items.Dequeue();
                return true;
            }
        }
        finally
        {
            m_Lock.ExitWriteLock();
        }
        return false;
    }
```

Finally, add a method called *TryPeek*. This method will be identical to *TryDequeue* except instead of calling *Dequeue* on *m_Items*, call *Peek*.

Program

Program is where we will do the majority of the threading. We are going to use the producer/consumer model for sharing resources. Basically, one set of threads will be responsible for creating data that will be consumed by the consumer threads. In our application the producers will generate the numbers to be tested for prime. The consumers will read the numbers and check them for prime.

First, let's discuss what is already in the *Program* class to start with. The following fields are defined to assist with our task:

```
Dictionary<int, List<ulong>> m_Primes
```

This **Dictionary** will contain a set of lists of the found primes. Each consumer thread will work with its own list. When all threads are done, these lists will be sorted and combined.

```
SafeQueue<ulong> m_Numbers
```

This instance of the *SafeQueue* will hold all of the numbers produced by the producer threads.

```
ManualResetEvent m_ProducersComplete
```

We will use this event to signal the consumers when the producers have finished their work.

```
object m_ConsoleLock
```

This object will be used to prevent multiple threads from accessing the Console at the same time.

```
Timer m_Timer
```

We will use a timer to periodically update the number of items left to process in the Console window.

The following methods have been provided as well to assist mostly with the Console display:

`ShowStats()`

When the calculations are complete the user has the option to view statistics related to the number of primes found in total and per thread.

`ShowPrimes()`

This method will show all of the primes found.

`CountTimerCallback()`

This is the timer event that fires periodically to show the number of numbers left to process.

`OutputHeader()`

Outputs a grid and its header to the Console.

`UpdateThreadStatus()`

This method is called by the producer and consumer threads to show their current state.

The final three methods, *Main*, *ProducerThread* and *ConsumerThread*, will be added next.

ProducerThread

This method encapsulates a thread that produces the numbers that will be tested by the consumer threads. The input of the method contains a three value tuple, two **int**s and one **ulong**. The first item in the tuple is the starting number to being adding to the *m_Numbers* queue. The second item is the number to skip per iteration while adding values to the queue. The final item is the highest number to add to the queue.

```
Tuple<int, int, ulong> parameters = state as Tuple< int, int, ulong>;
if (parameters == null) return;
int start = parameters.Item1;
int skip = parameters.Item2;
ulong upper = parameters.Item3;
```

Next, we'll update the status of the thread to the Console by calling *UpdateThreadStatus*. We will use a "+" character to indicate activity and a "-" character to indicate inactivity. We can use the

start field for the thread ID. This is a side-effect of the way we launch the thread from within *Main* - you will see this later.

```
UpdateThreadStatus(start, true, '+');
```

Next, we'll create the **for** loop that will add numbers to *m_Numbers*. In this loop we will also issue a sleep command for 1ms to slow down the process - this is just for display purposes.

```
for (ulong i = (ulong)start; i <= upper; i += (ulong)skip)
{
   m_Numbers.Enqueue(i);
   Thread.Sleep(1);
}
```

Finally, once all of the numbers have been enqueued, update the status in the Console.

```
UpdateThreadStatus(start, true, '-');
```

ConsumerThread

The consumer thread will also accept a parameter. This time it is just the thread ID defined in *Main*:

```
int id = (int)state;
```

Just as we did in *ProducerThread* we will update the thread status using the *UpdateThreadStatus* method:

```
UpdateThreadStatus(id, false, '+');
```

Next, we need a place to store the primes as they are determined. In *Main* we will initialize the *m_Primes* **Dictionary** where the key of each key/value pair is a **ulong** that corresponds to the thread IDs. The value of each pair will be a list of **ulong**s.

```
List<ulong> primes = m_Primes[id];
```

Now, we will start consuming the values being added to *m_Numbers*. However, we also need a mechanism to know when to quit. We have two pieces of information that will come in handy for this task: the *m_ProducersComplete* event and number of values in *m_Numbers*. So, while the

m_ProducersComplete event is not set or there are still numbers to process we will continue to retrieve values from the queue.

```
while (m_ProducersComplete.WaitOne(1) == false || m_Numbers.Count > 0)
```

Within the loop, fetch the next value from the queue and test if it is prime. Also, update the thread status to indicate that it is processing a value.

```
ulong number;
if (m_Numbers.TryDequeue(out number))
{
   UpdateThreadStatus(id, false, 'X');
   if (MathStuff.IsPrime(number))
   {
      primes.Add(number);
   }
   UpdateThreadStatus(id, false, '+');
}
```

When the loop is complete, update the thread's status to show it is idle.

```
UpdateThreadStatus(id, false, '-');
```

Main

Finally, let's look at *Main*. A large portion of this code is already complete. The first section sets up the Console and gets the program criteria from the user. Next, we calculate the approximate number of primes that will be calculated using *MathStuff.ApproximateNumberOfPrimes*. Then we create a list for the producer threads and a list for the consumer threads. The next step is to create and launch the producer threads. Note that we are going to use *i* as the threads' ID and the starting point for the producer:

```
for (int i = 0; i < producerThreadCount; i++)
{
   Thread producer = new Thread(ProducerThread);
   producer.Start(new Tuple<int, int, ulong>(i, producerThreadCount, highest));
   producerThreads.Add(producer);
}
```

Next, create and start the consumer threads, again using *i* as the threads' ID:

```
for (int i = 0; i < consumerThreadCount; i++)
{
   Thread consumer = new Thread(ConsumerThread);
   // Create the List of ulongs that the consumer will place its results into
   // Also, use the approx. # of primes to minimize list resizing
   List<ulong> primes =
      new List<ulong>((int)(approxNumPrimes / (uint)consumerThreadCount));
   // Add that list to m_Primes, using the thread's ID as its key
   m_Primes.Add(i, primes);
   consumer.Start((int)i);
   consumerThreads.Add(consumer);
}
```

Next, we will wait for all of the producer threads to finish. We can use each of the threads' *Join* method to wait for completion. Once all threads have joined, set the *m_ProducersComplete* event to let all consumers know that no more numbers are coming.

```
foreach (Thread thread in producerThreads)
{
   thread.Join();
}
// Signal to all consumers that no more values will be produced
m_ProducersComplete.Set();
```

Now, wait for all consumer threads:

```
foreach (Thread thread in consumerThreads)
{
   thread.Join();
}
```

All that is left is to show the results of our work. That code is already in *Main*.

Rather than give you the first *n* primes to test your output against, you can do a quick web search for those values.