Concurrent Programming

C#

Wait Handles

- Win32 Api provides three classes
  - `EventWaitHandle`
  - `Mutex`
  - `Semaphore`
- All 3 are based on the abstract `WaitHandle` classs
- `EventWaitHandle` has two subclasses
  - `AutoResetEvent`
  - `ManualResetEvent`
- They only differ in the way the constructor is invoked.
- `WaitHandles` allow objects to be named and used between separate processes

Wait Handles

`AutoResetEvent`

- It can be compared to a gate that only passes one process at the push of a button.
- When the gateway is open, the process or thread that calls the `WaitOne()` method passes through the gateway and closes it `WaitOne()`
- The process is queued when the gateway is closed
- Any other unblocked process can unblock the gate by calling method `Set()`
- One call to `Set()` will only admit one process..
- When there are no processes in the queue, `Set()` will open the gateOnce the gate is open, any subsequent `Set()` are ignored

Wait Handles

AutoResetEvent

```
EventWaitHandle waitingGate = new EventWaitHandle (false,

                              EventResetMode.Auto);

EventWaitHandle waitingGate = new AutoResetEvent (false);
```

- The above two calls are equivalent.
- The first parameter determines whether the gate should be opened during creation.

Wait Handles

`EventWaitHandle` - interprocess

```
EventWaitHandle waitingGate = new EventWaitHandle (false,
            EventResetMode.Auto,"Name of our waitingGate");
```

- The third parameter can be the name seen by all other processes in the system.
- If during creation it turns out that the object with the given name exists, we will only get a reference and the fourth parameter will be false;

```
EventWaitHandle (false, EventResetMode.Auto,"Name of our
waitingGate", out isNew);
```

Ready Go

Suppose we have such a scenario

- The main process has new tasks to complete every moment
- These tasks are to be done by thread
- A new thread is started each time
- The job is handed over
- After the work is done, the thread is terminated

To reduce the load resulting from creating threads (or even other processes), we can follow the following algorithm:

- The main process creates the thread
- The thread is waiting for the task
- Does the job
- It goes into a state of waiting for the next task

Ready Go

The simplest version of the producer and the consumer

```
static EventWaitHandle ready = new AutoResetEvent(false);
static EventWaitHandle go = new AutoResetEvent(false);
static volatile string job;
static void Main(string[] args)
{
    new Thread(Consumer).Start();
    for (int i = 1; i <= 5; i++) //order job 5 times
        {
        ready.WaitOne(); // Waiting for consumer ready
        job = "a".PadRight(i, 'a'); // prepare job
        go.Set(); // signal job are ready to read
        }
    ready.WaitOne(); job = null; go.Set(); // it is signal to end
    Console.ReadKey();
    }
static void Consumer()
    {
    while (true)
        {
        ready.Set(); // Inform producer we are ready
        go.WaitOne(); // and waiting for a job
        if (job == null) return; // when we get null we ends
            Console.WriteLine(job);
        }
    }
```


example ReadyGo

Producer consumer - queue

- The producer queues the items
- The consumer dequeues the items
- We use named WaitHandle

```
static void Main(string[] args)
        {
            Queue<string> queue = new Queue<string>();
            Thread producerThread = Producer.StartProduction(queue);
            Thread consumerThread = Consumer.StartConsumption(queue);
            producerThread.Join();
            consumerThread.Join();
        }
```

Producer consumer - queue

```
public static Thread StartProduction(Queue<string> queue)
    {
    if (producerThread == null)
        {
        producerThread = new Thread((q) =>
            {
            Random r = new Random();
            for (int i = 0; i < 100; i++)
                {
                Thread.Sleep(r.Next(0, 30));
                lock (q)
                    {
                    (q as Queue<string>).Enqueue("This is a product no " + i);
                    Console.WriteLine("P: I'v put to the queue");
                    }
                 wh.Set();
                 }
             lock (q)
                {
                (q as Queue<string>).Enqueue(null);
                }
         });
         producerThread.Start(queue);
        }
    else
        {
        Console.WriteLine("There is already one thread");
        }
    return producerThread;
    }
```

Producer consumer - queue

```
public static Thread StartConsumption(Queue<string> queue)
     {
         if (consumerThread == null)
         {
             consumerThread = new Thread((q) =>
             {
                 Random r = new Random();
                 while (true)
                 {
                     string mesg = null;
                     lock (q)
                         if ((q as Queue<string>).Count > 0)
                         {
                             mesg = (q as Queue<string>).Dequeue();
                             if (mesg == null) return;
                         }
                     if (mesg != null)
                     {
                         Console.WriteLine("C: I'v consumed: " + mesg);
                         Thread.Sleep(r.Next(0, 20));
                     }
                     else
                     {
                         //sometimes it happens twice
                         Console.WriteLine("C: So I'm waiting...");
                         wh.WaitOne();
                     }
                 }
             });
             consumerThread.Start(queue);
         }
         else
         {
             Console.WriteLine("There is already one thread");
         }
         return consumerThread;
 }
```

Wait Handles

`ManualResetEvent`

```
EventWaitHandle waitingGate = new EventWaitHandle (false,
  EventResetMode.Manual);
```

```
EventWaitHandle waitingGate = new ManualResetEvent (false);
```

- The above two calls are equivalent.
- The first parameter determines whether the gate should be opened during creation.
- The `Set` method lets all callers WaitOne in, until it close by `Reset`.

Wait Handles

`Mutex`

- It works the same as `lock` except that it can be used between processes and is about 100 times slower (assuming you are not blocking)
- As well as `lock`, it provides exclusive access to the program block between the call to WaitOne and ReleaseMutex
- Locking and unlocking must be invoked from the same thread.
- The advantage is the automatic release of the mutex even when the application exits without calling `ReleaseMutex`

Wait Handles

Mutex

```
static Mutex mutex = new Mutex(false, "tu.kielce.pl mutex");
static void ThreadWithMutex(object o)
    {
    //it is very slow, there is 100 times less iterations then in other 
examples
    for (int ii = 0; ii < 10000; ii++)
        {
        mutex.WaitOne();
        counter++;
        mutex.ReleaseMutex();
        }
    }
```

Example: Synchronization

Wait Handles

`Semaphore`

- A semaphore is like a counter that can never be less than 0.
- The **`WaitOne`** operation decreases this counter by 1, if it is 0, the given thread waits for another thread to increase it with **`Release`**.
- In the case of a semaphore, it can be released by any other thread, not just the one that call WaitOne, as is the case with lock or Mutex.
- Semaphore is similarly fast as than Mutex.

Wait Handles

Semaphore

```
static Semaphore sem = new Semaphore(1, 1);
        static void ThreadWithSemaphore(object o)
        {
            //it is very slow, there is 100 times less iterations
            for (int ii = 0; ii < 10000; ii++)
            {
                sem.WaitOne();
                counter++;
                sem.Release();
            }
        }
```

Wait Handles

Wait, wait, wait...

WaitHandle.SignalAndWait – Simultaneous signal sending and waiting. For example, meetings can be organized in this way.

```
private static EventWaitHandle wh1 =
    new
  EventWaitHandle(false,EventResetMode.AutoReset);
  private static EventWaitHandle wh2 =
    new
  EventWaitHandle(false,EventResetMode.AutoReset);
```

One of the threads calls:

```
WaitHandle.SignalAndWait(wh1, wh2);
```

The second thread calls:

```
WaitHandle.SignalAndWait(wh2, wh1);
```

Wait Handles

Wait, wait, wait...

`WaitHandle.WaitAll(WaitHandle[] waitHandles)`

- Wait for permission from all of the waitHandles

`WaitHandle.WaitAny(WaitHandle[] waitHandles)`

- Wait for permission from any of the waitHandles

Barrier

The barrier is used to synchronize the work of the threads in certain stages. For example, in genetic algorithms where we wait for all threads to finish working in a given iteration. Below, several threads work unsynchronized

```
const int threads = 10;
static void printString(string inputstring)
{
    ThreadStart thread = () =>
    {
        char[] inputArray = inputstring.ToArray();
        for (int i = 0; i < inputArray.Length; i++)
        {
            Console.Write(inputArray[i]);
        }
    };
    Thread[] watki = new Thread[threads];
    for (int i = 0; i < threads; ++i)
    {
        watki[i] = new Thread(thread);
        watki[i].Start();
    }
    //Here we wait until all threads running in this method are finished
    for (int i = 0; i < threads; ++i)
    {
        watki[i].Join();
    }
}
```

Example: CollectionsAndBariers/Barrier

Barrier

Attempting to apply the barrier with the Monitor

```
static void printStringWaitPulse(string inputstring)
       {
           object o = new object();
           int callCounter = 0;
           ThreadStart thread = () =>
           {
               char[] inputArray = inputstring.ToArray();
               for (int i = 0; i < inputArray.Length; i++)
               {
                   lock (o)
                   {
                       Console.Write(inputArray[i]);
                       callCounter++;
                       if (callCounter < threads)
                       {   //if not all threads finished we wait
                           //and simultanously free the lock
                           Monitor.Wait(o);
                       }
                       else
                       {   //if we are the last thread we pulse others
                           Monitor.PulseAll(o);
                           callCounter = 0;
                       }
                   }
               }
           };
          ...
```

Example: CollectionsAndBariers/Barrier

Barrier

Using *Barrier* and *CountdownEvent*

```
static System.Threading.Barrier barrier =
        new System.Threading.Barrier(threads, (b) =>
        { Console.WriteLine(" Barrier in a phase: {0}", b.CurrentPhaseNumber); });
static void printStringBarrier(string inputstring)
    {
    ThreadStart thread = () =>
        {
        for (int i = 0; i < inputstring.Length; i++)
            {
            Console.Write(inputstring[i]);
            barrier.SignalAndWait();
            }
        ce.Signal(); //when the task is done we report it to decrease the counter
        };
    Thread[] threadsArray = new Thread[threads];
    for (int i = 0; i < threads; ++i)
        {
        threadsArray[i] = new Thread(thread);
        threadsArray[i].Start();
        }
    //Here we wait until the counter reaches 0 (all threads calling ce.Signal ())
    ce.Wait();
    Console.WriteLine("Done. InitialCount={0}, CurrentCount={1}, IsSet={2}",
                        ce.InitialCount, ce.CurrentCount, ce.IsSet);
    }
```

Example: CollectionsAndBariers/Barrier

Collections

Adding to a list by multiple threads

```
static void Main(string[] args)
        {
            List<Thread> threads = new List<Thread>();
            List<int> numbers = new List<int>(10000);
            Random rand = new Random();
            for (int i = 0; i < 100; i++)
            {
                var thread = new Thread(() =>
                {
                    for (int l = 0; l < 100; l++)
                        numbers.Add(rand.Next());
                });
                threads.Add(thread);
                thread.Start();
            }
            foreach (var watek in threads)
            {
                watek.Join();
            }
            Console.WriteLine($"The number of items in a regular list: 
{numbers.Count}");
            Console.ReadLine();
        }
```


Example: CollectionsAndBariers/UnsafeCollection

Collections

Adding to a list by multiple threads with a lock

```
static void Main(string[] args)
    {
    List<Thread> threads = new List<Thread>();
    List<int> numbers = new List<int>(10000000);
    Stopwatch sw = new Stopwatch();
    sw.Start();
    Random rand = new Random();
    for (int i = 0; i < 100; i++)
        {
        var thread = new Thread(() =>
            {
            for (int l = 0; l < 100000; l++)
            lock (numbers) //comment this and mesure time
                {
                numbers.Add(rand.Next());
                }
            });
        threads.Add(thread);
        thread.Start();
        }
    foreach (var th in threads)
        th.Join();
    sw.Stop();
    Console.WriteLine($"The number of items in a regular list: {numbers.Count}");
    Console.WriteLine($"Elapsed {sw.ElapsedMilliseconds} ms");
    Console.ReadLine();
    }
```


Example: CollectionsAndBariers/LockCollection

Collections

Using ConcurrentBag

```
static void Main(string[] args)
        {
            ConcurrentBag<int> bag = new ConcurrentBag<int>();
            List<Thread> threads = new List<Thread>();
            Stopwatch sw = new Stopwatch();
            sw.Start();
            Random rand = new Random();
            for (int i = 0; i < 100; i++)
            {
                var thread = new Thread(() =>
                {
                    for (int l = 0; l < 100000; l++)
                        bag.Add(rand.Next());
                });
                threads.Add(thread);
                thread.Start();
            }
            foreach (var th in threads)
            {
                th.Join();
            }
            sw.Stop();
            Console.WriteLine($"The number of items in the concurrent bag: 
{bag.Count}");
            Console.WriteLine($"Elapsed {sw.ElapsedMilliseconds} ms");
            Console.ReadLine();
        }
```

Example: CollectionsAndBariers/ConcurrentBagCollection

Collections

The problem with unique values

```
for (int i = 0; i < 100; i++)
            {
                var watek = new Thread(() =>
                {

                    for (int l = 0; l < 100; l++)
                    {
                        int number = rand.Next(10001);
                        lock (numbers) //we use this lock only for that
there are no errors like "collection was modified"
                            while (numbers.Any(x => x == number))
                        {
                            number = rand.Next(10001);
                        };
                        lock (numbers) //we use this lock only for that
there are no errors like "collection was modified"
                            numbers.Add(number);
                    }
                });
                threads.Add(watek);
                watek.Start();
            }
```


Example: CollectionsAndBariers/NonUniqueCollection

Collections

Using Lock to meke collection unique

```
for (int i = 0; i < 100; i++)
            {
                var watek = new Thread(() =>
                {

                    for (int l = 0; l < 100; l++)
                        lock (numbers)
                        {
                            int number = rand.Next(10001);
                            while (numbers.Any(x => x == number))
                            {
                                number = rand.Next(10001);
                            };
                            numbers.Add(number);
                        }
                });
                threads.Add(watek);
                watek.Start();
            }
```


Example: CollectionsAndBariers/UniqueCollection

Collections

The same when we want concurent bag with unique numbers

```
ConcurrentBag<int> numbers = new ConcurrentBag<int>();
List<Thread> threads = new List<Thread>();

for (int i = 0; i < 100; i++)
    {
     var thread = new Thread(() =>
         {
         for (int l = 0; l < 100; l++)
         lock (numbers) //without this, cb has non unique numbers
             {
             int number = rand.Next(10001);
             while (numbers.Any(x => x == number))
                 {
                 number = rand.Next(10001);
                 };
             numbers.Add(number);
             }
        });
     threads.Add(thread);
     thread.Start();
     };
```

Example: CollectionsAndBariers/NonUniqueBag

Collections

Using ConcurrentDictionary

Keys in dictionary must be unique so we use kes like value and value set null.

```
List<Thread> threads = new List<Thread>();
ConcurrentDictionary<int, object> numbers = new
ConcurrentDictionary<int, object>();
for (int i = 0; i < 100; i++)
    {
     var thread = new Thread(() =>
        {
        Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
        for (int l = 0; l < 100; l++)
            {
            int number = rand.Next(10001);
            while (!numbers.TryAdd(number, null)) //we use key like a
value and value set null
                {
                number = rand.Next(10001);
                };
            }
        });
    threads.Add(thread);
    thread.Start();
    }
```

Example: CollectionsAndBariers/UniqueDictionary

Collections

Using `BlockingCollection in producer consumer`

```
var blockingCollection = new BlockingCollection<int>(10);

var producer = new Thread(() =>
    {
     for (int l = 0; l < 100; l++)
        {
        Console.WriteLine($"Put  {l}");
        blockingCollection.Add(l);
        Thread.Sleep(rand.Next(500));
        }
    blockingCollection.CompleteAdding();
     });
producer.Start();

var consumer = new Thread(() =>
    {
    for (int l = 0; l < 10; l++)
        {
        var result2 = blockingCollection.Take();
        Console.WriteLine($"Take {result2} ");
        Thread.Sleep(rand.Next(5000));
        }
    });
```

Example: CollectionsAndBariers/ProducerConsumerBC

Collections

Using `BlockingCollection in producer consumer`

```
var blockingCollection = new BlockingCollection<int>(10);

var producer = new Thread(() =>
    {
     for (int l = 0; l < 100; l++)
        {
        Console.WriteLine($"Put  {l}");
        blockingCollection.Add(l);
        Thread.Sleep(rand.Next(500));
        }
    blockingCollection.CompleteAdding();
     });
producer.Start();

var consumer = new Thread(() =>
    {
    for (int l = 0; l < 10; l++)
        {
        var result2 = blockingCollection.Take();
        Console.WriteLine($"Take {result2} ");
        Thread.Sleep(rand.Next(5000));
        }
    });
```

Example: CollectionsAndBariers/ProducerConsumerBC

Collections

Using simple Queue `in producer consumer`

`Producer puts sequential numbers to the simple queue in random time up to 0,5s`

```
var cq = new Queue<int>();
List<Thread> consumers = new List<Thread>();
var producer = new Thread(() =>
    {
    Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
    for (int l = 0; l < 100; l++)
        {
        cq.Enqueue(l);
        Console.WriteLine($"Put  {l}");
        Thread.Sleep(rand.Next(500));
        }
    });
producer.Start();
```

Example: CollectionsAndBariers/ProducerConsumerQ1

Collections

Using simple Queue in producer consumer

```
for (int i = 0; i < 10; i++)
    {
    var consumer = new Thread(() =>
        {
        Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
        for (int l = 0; l < 10; l++)
            {
            try
                {
                int result2 = cq.Dequeue();
                Console.WriteLine($"Take {result2} ");
                }
            catch (Exception ex)
                {
                Console.WriteLine("Exception during taking: {0}", ex.Message);
                }
            Thread.Sleep(rand.Next(5000));
            }
        });
    consumers.Add(consumer);
    consumer.Start();
    }
```

Example: CollectionsAndBariers/ProducerConsumerQ1

Collections

Here, we have serious problem. Consumers read even the queue is empty.

```
Take 0
Put  0
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Exception during taking: Queue empty.
Put  1
Put  2
```

Example: CollectionsAndBariers/ProducerConsumerQ1

Collections

Using Concurrent Queue in producer consumer

```
//the producer puts 100 elements into the collection at random intervals
of up to 500ms
var producer = new Thread(() =>
    {
    Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
    for (int l = 0; l < 100; l++)
        {
        Console.WriteLine($"Put  {l}");
        cq.Enqueue(l);
        Thread.Sleep(rand.Next(500));
        }
    });
producer.Start();
```

Example: CollectionsAndBariers/ProducerConsumerQ2

Collections

Using Concurrent Queue in producer consumer

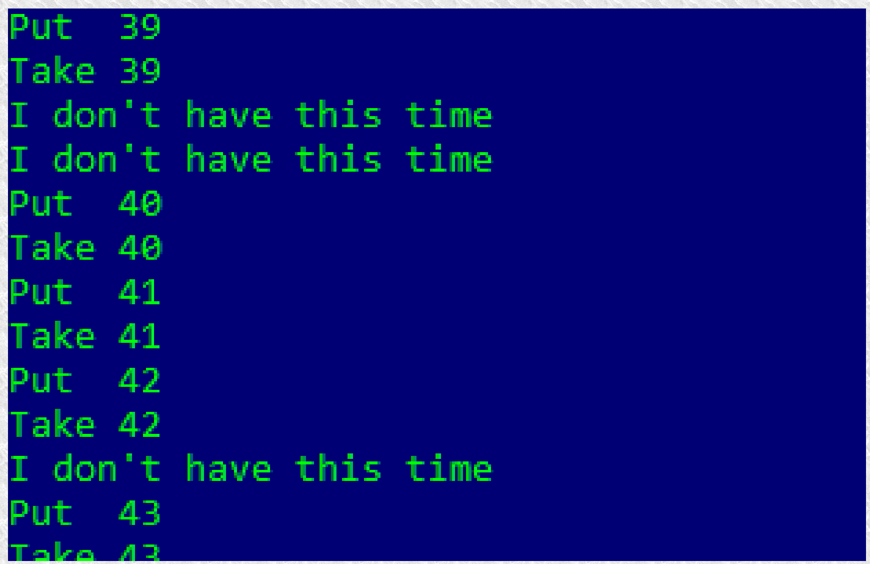
```
//Consumers take data from the collection at intervals of up to 5000 ms
for (int i = 0; i < 10; i++)
    {
    var consumer = new Thread(() =>
        {
        Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
        for (int l = 0; l < 10; l++)
            {
            Thread.Sleep(rand.Next(5000));
            int result;
            bool succeeded = cq.TryDequeue(out result);
            if (!succeeded)
                {
                l--;
                Console.WriteLine("I don't have this time");
                continue;   //continue for
                }
            Console.WriteLine($"Take {result} ");
            }
        });
    consumers.Add(consumer);
    consumer.Start();
    }
```

Example: CollectionsAndBariers/ProducerConsumerQ2

Collections

There is another problem. There is no blocking function *Dequeue* in ConcurrentQueue. (In BlockingCollection is *Take*)

There is only TryDeque and we use it. But sometimes we have empty queue and must TryDeque again, this is like active checking.



Example: CollectionsAndBariers/ProducerConsumerQ2

Collections

Using Concurrent Queue in producer consumer with semaphore

```
SemaphoreSlim elementCounter = new SemaphoreSlim(0, int.MaxValue);
var producer = new Thread(() =>
    {
     Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
     for (int l = 0; l < 100; l++)
        {
        Console.WriteLine($"Put  {l}");
        cq.Enqueue(l);
        elementCounter.Release();
        Thread.Sleep(rand.Next(500));
        }
    });
producer.Start();
```

Example: CollectionsAndBariers/ProducerConsumerQ3

Collections

Using Concurrent Queue in producer consumer with semaphore

```
for (int i = 0; i < 10; i++)
    {
     var consumer = new Thread(() =>
         {
         Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
         for (int l = 0; l < 10; l++)
             {
             Thread.Sleep(rand.Next(4000));
             int result;
             elementCounter.Wait();
             bool succeeded = cq.TryDequeue(out result);
             if (!succeeded)
                 {
                 throw new Exception("OMG we shouldn't be here");
                 }
             Console.WriteLine($"Take {result} ");
             }
         });
     consumers.Add(consumer);
     consumer.Start();
    }
```

Example: CollectionsAndBariers/ProducerConsumerQ3

Collections

Using Concurrent Queue in producer consumer with semaphores and upper limit

```
SemaphoreSlim elementCounter = new SemaphoreSlim(0, limit);
SemaphoreSlim upperLimit = new SemaphoreSlim(limit, limit);

var producer = new Thread(() =>
    {
     Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
     for (int l = 0; l < 100; l++)
        {
        upperLimit.Wait(); //check the queue not hit the upper limit
        Console.WriteLine($"Put  {l}");
        cq.Enqueue(l);
        elementCounter.Release(); //signal that item enqueued
        Thread.Sleep(rand.Next(500));
        }
    });
producer.Start();
```

Example: CollectionsAndBariers/ProducerConsumerQ4

Collections

Using Concurrent Queue in producer consumer with semaphores and upper limit

```
Thread.Sleep(5000); //wait for simulate hiting the limit
for (int i = 0; i < 10; i++)
    {
    var consumer = new Thread(() =>
        {
        Random rand = new Random(Thread.CurrentThread.ManagedThreadId);
        for (int l = 0; l < 10; l++)
            {
            Thread.Sleep(rand.Next(4000));
            int result;
            elementCounter.Wait();
            bool succeeded = cq.TryDequeue(out result);
            if (!succeeded)
                {
                throw new Exception("OMG we shouldn't be here");
                }
            upperLimit.Release();
            Console.WriteLine($"Take {result} ");
            }
        });
    consumers.Add(consumer);
    consumer.Start();
    }
```

Example: CollectionsAndBariers/ProducerConsumerQ4

Thank You