Mutex

Класс Mutex (mutual exclusion — взаимное исключение или мьютекс) является одним из классов в .NET Framework, позволяющих обеспечить синхронизацию среди множества процессов. Он очень похож на класс Monitor тем, что тоже допускает наличие только одного владельца. Только один поток может получить блокировку и иметь доступ к защищаемым мьютексом синхронизированным областям кода.

В конструкторе класса Mutex указывается, должен ли мьютексом изначально владеть вызывающий поток, и его имя. Кроме того, конструктор позволяет получить информацию о том, существует ли уже такой класс.

Мьютекс представляет собой взаимно исключающий синхронизирующий объект. Это означает, что он может быть получен потоком только по очереди. Мьютекс предназначен для тех ситуаций, в которых общий ресурс может быть одновременно использован только в одном потоке. Допустим, что системный журнал совместно используется в нескольких процессах, но только в одном из них данные могут записываться в файл этого журнала в любой момент времени. Для синхронизации процессов в данной ситуации идеально подходит мьютекс.

У Mutex имеется несколько конструкторов. Ниже приведены два наиболее употребительных конструктора:

public Mutex()
public Mutex(bool initiallyOwned)

В первой форме конструктора создается мьютекс, которым первоначально никто не владеет. А во второй форме исходным состоянием мьютекса завладевает вызывающий поток, если параметр initiallyOwned имеет логическое значение true. В противном случае мьютексом никто не владеет.

Для того чтобы получить мьютекс, в коде программы следует вызвать метод **WaitOne()** для этого мьютекса. Метод WaitOne() наследуется классом Mutex от класса Thread.WaitHandle. Метод WaitOne() ожидает до тех пор, пока не будет получен мьютекс, для которого он был вызван. Следовательно, этот метод блокирует выполнение вызывающего потока до тех пор, пока не станет доступным указанный мьютекс. Он всегда возвращает логическое значение true.

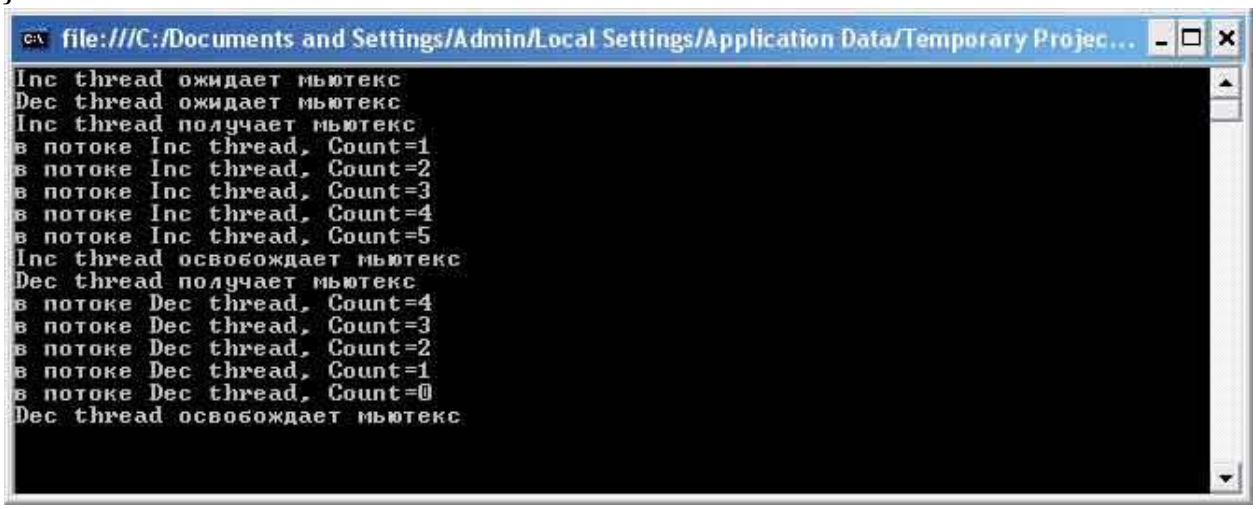
Когда же в коде больше не требуется владеть мьютексом, он освобождается посредством вызова метода **ReleaseMutex()**.

В приведенном ниже примере программы описанный выше механизм синхронизации демонстрируется на практике. В этой программе создаются два потока в виде классов IncThread и DecThread, которым требуется доступ к общему ресурсу: переменной SharedRes.Count. В потоке IncThread переменная SharedRes.Count инкрементируется, а в потоке DecThread — декрементируется. Во избежание одновременного доступа обоих потоков к общему ресурсу SharedRes.Count этот доступ синхронизируется мьютексом Mtx, также являющимся членом класса SharedRes:

```
// C#

using System;
using System.Threading;

namespace ConsoleApplication1
{
   // В этом классе содержится общий ресурс в виде переменной Count,
   // а так же мьютекс mtx
   class SharedRes
   {
      public static int Count;
      public static Mutex mtx = new Mutex();
   }

   // В этом классе Count инкрементируется
   class IncThread
   {
      int num;
      public Thread Thrd;

      public IncThread(string name, int n)
      {
         Thrd = new Thread(this.Run);
         num = n;
         Thrd.Name = name;
         Thrd.Start();
}

      // Точка входа в поток
      void Run()
      {
         Console.WriteLine(Thrd.Name + " ожидает мьютекс");

         // Получить мьютекс
         SharedRes.mtx.WaitOne();
```

```
            Console.WriteLine(Thrd.Name + " получает мьютекс");

            do
            {
               Thread.Sleep(500);
               SharedRes.Count++;
               Console.WriteLine("в потоке {0},
Count={1}",Thrd.Name,SharedRes.Count);
               num--;
            } while (num > 0);

            Console.WriteLine(Thrd.Name + " освобождает мьютекс");

            SharedRes.mtx.ReleaseMutex();
         }
      }

      class DecThread
      {
         int num;
         public Thread Thrd;

         public DecThread(string name, int n)
         {
            Thrd = new Thread(new ThreadStart(this.Run));
            num = n;
            Thrd.Name = name;
            Thrd.Start();
         }

         // Точка входа в поток
         void Run()
         {
            Console.WriteLine(Thrd.Name + " ожидает мьютекс");

            // Получить мьютекс
            SharedRes.mtx.WaitOne();

            Console.WriteLine(Thrd.Name + " получает мьютекс");

            do
            {
               Thread.Sleep(500);
               SharedRes.Count--;
```

```
            Console.WriteLine("в потоке {0},
Count={1}",Thrd.Name,SharedRes.Count);
            num--;
        } while (num > 0);

        Console.WriteLine(Thrd.Name + " освобождает мьютекс");

        SharedRes.mtx.ReleaseMutex();
    }
  }

  class Program
  {
    static void Main()
    {
      IncThread mt1 = new IncThread("Inc thread", 5);

      // разрешить инкременирующему потоку начаться
      Thread.Sleep(1);

      DecThread mt2 = new DecThread("Dec thread", 5);

      mt1.Thrd.Join();
      mt2.Thrd.Join();

      Console.ReadLine();
    }
  }
}
```



Как следует из приведенного выше результата, доступ к общему ресурсу (переменной SharedRes.Count) синхронизирован, и поэтому значение данной переменной может быть одновременно изменено только в одном потоке.

Semaphore

Семафор подобен мьютексу, за исключением того, что он предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам. Поэтому семафор пригоден для синхронизации целого ряда ресурсов. Семафор управляет доступом к общему ресурсу, используя для этой цели счетчик. Если значение счетчика больше нуля, то доступ к ресурсу разрешен. А если это значение равно нулю, то доступ к ресурсу запрещен. С помощью счетчика ведется подсчет количества разрешений. Следовательно, для доступа к ресурсу поток должен получить разрешение от семафора.

Обычно поток, которому требуется доступ к общему ресурсу, пытается получить разрешение от семафора. Если значение счетчика семафора больше нуля, то поток получает разрешение, а счетчик семафора декрементируется. В противном случае поток блокируется до тех пор, пока не получит разрешение. Когда же потоку больше не требуется доступ к общему ресурсу, он высвобождает разрешение, а счетчик семафора инкрементируется. Если разрешения ожидает другой поток, то он получает его в этот момент. Количество одновременно разрешаемых доступов указывается при создании семафора. Так, если создать семафор, одновременно разрешающий только один доступ, то такой семафор будет действовать как мьютекс.

Семафоры особенно полезны в тех случаях, когда общий ресурс состоит из группы или пула ресурсов. Например, пул ресурсов может состоять из целого ряда сетевых соединений, каждое из которых служит для передачи данных. Поэтому потоку, которому требуется сетевое соединение, все равно, какое именно соединение он получит. В данном случае семафор обеспечивает удобный механизм управления доступом к сетевым соединениям.

Семафор реализуется в классе *System.Threading.Semaphore*, у которого имеется несколько конструкторов. Ниже приведена простейшая форма конструктора данного класса:

public Semaphore(int initialCount, int maximumCount)

где initialCount — это первоначальное значение для счетчика разрешений семафора, т.е. количество первоначально доступных разрешений; maximumCount — максимальное значение данного счетчика, т.е. максимальное количество разрешений, которые может дать семафор.

Семафор применяется таким же образом, как и описанный ранее мьютекс. В целях получения доступа к ресурсу в коде программы вызывается метод WaitOne() для семафора. Этот метод наследуется классом Semaphore от класса WaitHandle. Метод WaitOne() ожидает до тех пор, пока не будет получен семафор, для которого он вызывается. Таким образом, он блокирует

выполнение вызывающего потока до тех пор, пока указанный семафор не предоставит разрешение на доступ к ресурсу.

Если коду больше не требуется владеть семафором, он освобождает его, вызывая метод **Release()**. Ниже приведены две формы этого метода:

// C#

public int Release()
public int Release(int releaseCount)

В первой форме метод Release() высвобождает только одно разрешение, а во второй форме — количество разрешений, определяемых параметром releaseCount. В обеих формах данный метод возвращает подсчитанное количество разрешений, существовавших до высвобождения.

В .NET 4 предлагается два класса с функциональностью семафора: Semaphore и SemaphoreSlim. Класс Semaphore может быть именован, использовать ресурсы в масштабе всей системы и обеспечивать синхронизацию между различными процессами. Класс **SemaphoreSlim** представляет собой облегченную версию класса Semaphore, которая оптимизирована для обеспечения более короткого времени ожидания.