Практическое занятие

Оператор lock

Оператор lock предназначен для того, чтобы одному потоку не дать войти в важный раздел кода в тот момент, когда в нем находится другой *поток*. При попытке входа другого потока в заблокированный код потребуется дождаться снятия блокировки объекта. Этот оператор оформляется следующим образом:

```
Object thisLock = new Object();
lock (thisLock)
{
   // Критический фрагмент код
}
```

где thisLock - обозначает ссылку на синхронизируемый *объект*, который гарантирует, что фрагмент кода, защищенный блокировкой для данного объекта, будет использоваться только в потоке, получающем эту блокировку, а все остальные потоки блокируются до тех пор, пока *блокировка* не будет снята. *Блокировка* снимается по завершении защищаемого ею фрагмента кода.

Так же для блокировки объектов можно применять конструкция lock (this). Но данную конструкцию следует применять в том случае, если this является ссылкой на закрытый *объект*.

Пример работы распараллеленного приложения без синхронизации потоков:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace OperatorLock
{
   class Work
   {
      int i, j;
      public void ThreadStart()
      {

            if (i != j)
               Console.WriteLine("Ошибка");
            i++;

            Thread.Sleep(200);
            j++;
            Console.WriteLine("{0},{1}", i, j);
             }
   }
```

```
class Program
{
   static void Main(string[] args)
   {
      Work store = new Work();

         for (int i = 0; i < 10; ++i)
         {
            new Thread(new ThreadStart(store.ThreadStart)).Start();
            Thread.Sleep(100);
          }
      Console.ReadLine();
   }
}
}
```

В методе Main() - метод ThreadStar() 10 раз, в отдельных потоках. Запустим программу. В случайном порядке выведется на экран цифры и надпись "ошибка" (Рис. 5.1).

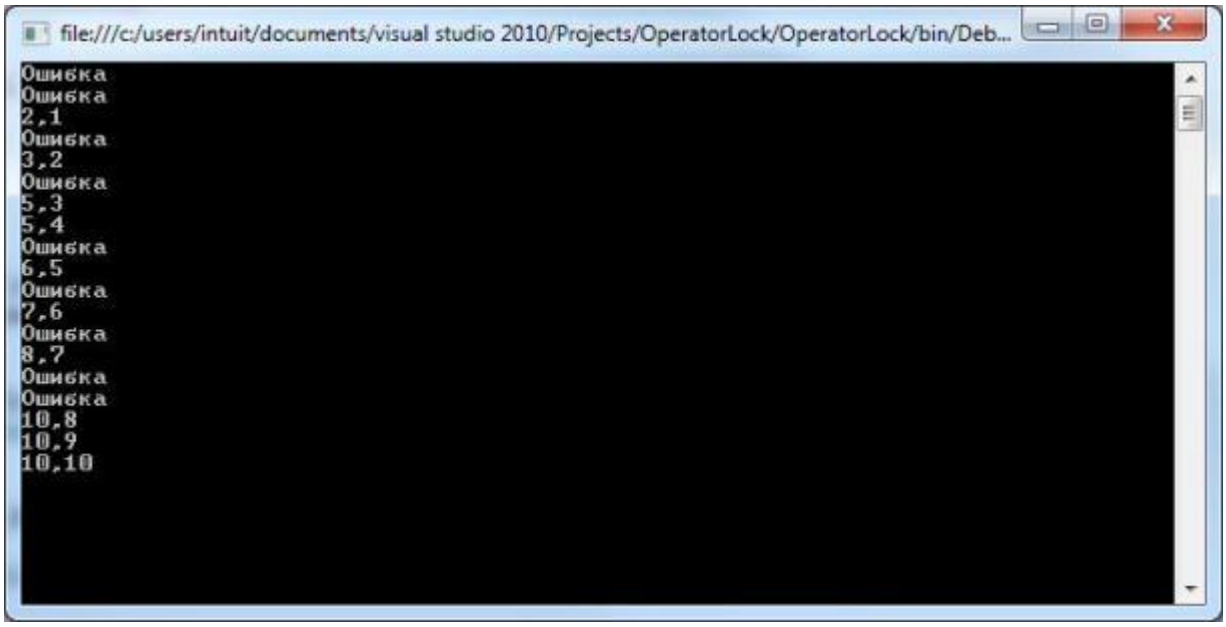

Рис. 5.1. Результат выполнения программы без оператора lock

Такой результат получается потому, что *приращение* значения переменной i и переменой j происходит не сразу, а с небольшой задержкой (в данном случае искусственной, полученной с помощью метода Sleep(), который усыпляет *поток* на 200 милисекунд). Именно на этом моменте происходит рассинхронизация потоков. Для синхронизации потоков используется оператор блокировки lock. Модифицируем пример, добавив в него синхронизацию:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace OperatorLock
{
   class Work
   {
      int i, j;
      object LockObj = new object();
      public void ThreadStart()
      {
         lock (LockObj)
         {
            if (i != j)
               Console.WriteLine("Ошибка");
            i++;
            Thread.Sleep(200);
            j++;
            Console.WriteLine("{0},{1}", i, j);
         }
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Work store = new Work();

            for (int i = 0; i < 10; ++i)
            {
               new Thread(new ThreadStart(store.ThreadStart)).Start();
               Thread.Sleep(100);
            }

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

Как только *поток* войдет в *контекст* lock, маркер блокировки (в данном случае - текущий *объект*) станет недоступным другим потокам до тех пор, пока *блокировка* не будет снята по выходе из контекста lock. Если теперь запустить *приложение*, можно увидеть, что каждый *поток* получил возможность выполнить свою работу до конца (Рис. 5.2).

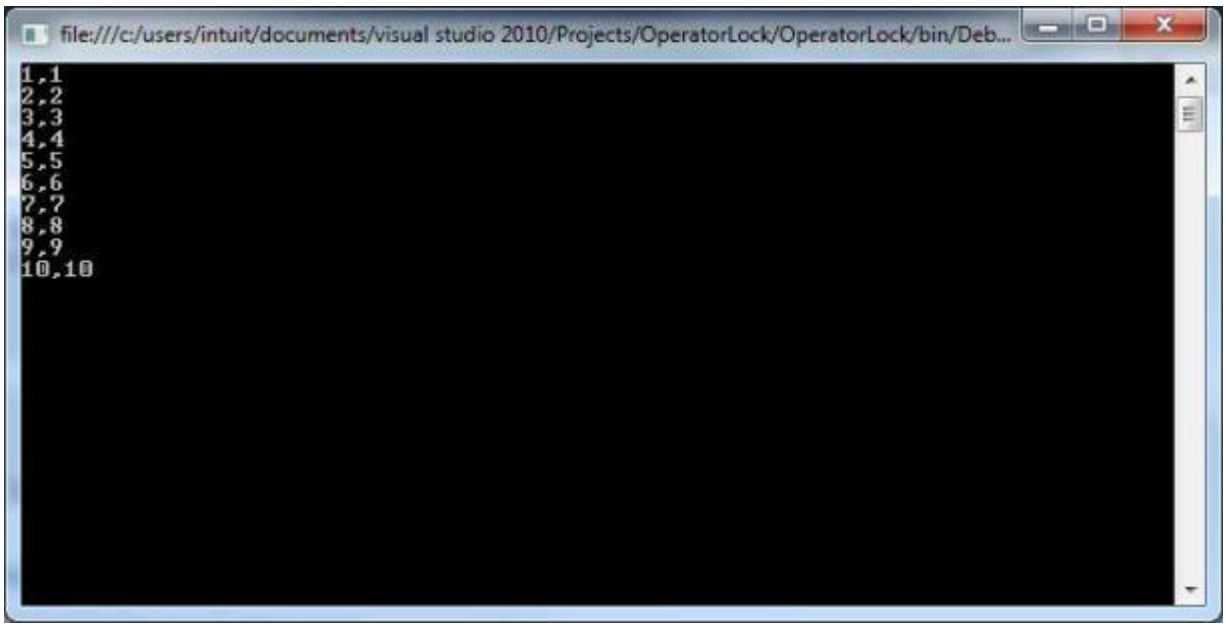

Рис. 5.2. Результат выполнения программы с оператором lock