МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к лабораторной работе

“Синхронизация потоков в параллельном программировании на языке C#”

по дисциплине “Технологии программирования”

для студентов специальностей

122 – Компьютерные науки и информационные технологии,

124 – Системный анализ, в том числе для иностранных студентов

Утверждено
Редакционно-издательским
советом университета,
протокол № 2 от 23.06.16 г

Харьков
НТУ ”ХПИ”
2019

Методические указания к лабораторным работам “Синхронизация потоков в параллельном программирования на языке C#” по дисциплине ”Технологии программирования” специальностей 122 – Компьютерные науки и информационные технологии, 124 – Системный анализ, в том числе для иностранных студентов /сост. Ю. Н. Кожин, О. Н. Малых, В. Ф. Прокопенков.– Харьков: НТУ “ХПИ”, 2019.– 96 с.– На рус.яз.

Составители: Ю. Н. Кожин,
О. Н. Малых,
В. Ф. Прокопенков,

Рецензент И. В. Кононенко

Кафедра системного анализа и информационно-аналитических технологий

ВСТУПЛЕНИЕ

Современные многоядерные ПЭВМ представляют собой мощные вычислительные средства, и они используются неэффективно, если программное обеспечение для них представляет собой обычные последовательные программы. Одним из способов организации параллельных вычислений являются многопоточные приложения.

Многопоточное выполнение подразумевает оформление каждой подзадачи в виде потока в рамках одного процесса. Для организации потоков приложения используется класс *Thread* пространства имён System.Threading.

Для взаимодействия потоков по данным не нужно применять какие-либо средства коммуникации. Потоки могут непосредственно обращаться к общим переменным, которые изменяют другие потоки.

Потоки являются простой и эффективной основой для разработки параллельных программ, но работа с общими переменными приводит к необходимости использования средств синхронизации, регулирующими порядок работы потоков с данными.

Предлагаемые методические указания помогут студентам познакомиться со средствами синхронизации потоков для разработки многопоточных приложений на языке C# для платформы .NET Framework и расширить свои умения в параллельном программировании.

Методические указания содержат необходимые теоретические сведения, а также рекомендации для выполнения лабораторных работ.

1. ВЫПОЛНЕНИЕ МЕТОДОВ В ПУЛЕ ПОТОКОВ

Кроме класса *Thread* в пространстве *System.Threading* существует еще один полезный статический класс для работы с потоками *ThreadPool*.

Для параллельного выполнения метода программист может явно не создавать вторичный поток, а поставить этот метод в очередь пула потоков. Выполнение метода начнётся, когда в пуле потоков появится свободный рабочий поток.

В таком способе организации приложения проблема формирования оптимального числа потоков в системе решается не программистом, а планировщиком задач. Планировщик задач контолирует и оптимизирует количество рабочих потоков в системе с учётом возможностей вычислительной системы, её фактической загруженностью и прогрессом выполнения задач. По завершении выполнения метода в рабочем потоке пула, рабочий поток не уничтожается и используется вдальнейшем.

Недостаток такой организации приложения – основной поток не может узнать о состоянии завершении потока, используя метод *Join()*. Использовать пул потоков целесообразно, если в приложении необходимо создавать множество потоков и нет необходимости тонко управлять их выполнением.

Необходимо помнить, что потоки из пула выполняются в фоновом режиме, а значит, их работа должна быть завершена до окончания основного потока. В противном случае завершение основного потока приведёт к завершинию потоков в пуле.

Следующие статические методы класса *ThreadPool*

bool QueueUserWorkItem(WaitCallback met)

bool QueueUserWorkItem(WaitCallback met,Object state)

необходимо использовать для постановки метода *met* в очередь пула потоков на выполнение. Он должен соответствовать типу делегата *WaitCallback:*

public delegate void WaitCallback(Object state).

Параметр *state* в вызове *QueueUserWorkItem()* определяет объект входных данных для метода *met*.

Продемонстрируем использование рассмотренных методов на примере:

```
struct Arguments
   {
     public int val;          // число
     public int count;        // количество
     public Arguments(int v, int c) { val = v; count = c; }
   }

   public class Printer
   {
      public void PrintNumbers(object par)
      {
         if (par != null)
         {
            Arguments args = (Arguments)par;
            int v = args.val;
            int num = args.count;

            for (int i = 0; i < num; i++)
            {
               Console.Write("{0} ", v);
               Thread.Sleep(0);
            }
            Console.WriteLine();
         }
         else
           Console.WriteLine("Ид.номер потока: {0} IsThread-
PoolThread={1} не определены входные параметры",
               Thread.CurrentThread.ManagedThreadId,
               Thread.CurrentThread.IsThreadPoolThread.ToString());
      }
   }

private void button1_Click(object sender, EventArgs e)
      {
         Console.WriteLine("\nMain(): Ид.номер потока: {0}",
```

```
                                Thread.CurrentThread.ManagedThreadId);
        Console.Write("\nСколько задач ставить в очередь? :");
        int iCount=Int32.Parse(Console.ReadLine());
        Console.WriteLine("\nСтавим в очередь на исполнение {0}
задач\n", iCount);

        Printer printer = new Printer();
        WaitCallback workTask = new
                                WaitCallback(printer.PrintNumbers);
        for (int i = 1; i <= iCount; i++)
           ThreadPool.QueueUserWorkItem(workTask);
     }

private void button2_Click(object sender, EventArgs e)
     {
        Console.WriteLine("\nMain(): Ид.номер потока: {0}",
                        Thread.CurrentThread.ManagedThreadId);

        Console.Write("\nСколько задач ставить в очередь? :");
        int iCount = Int32.Parse(Console.ReadLine());

        Console.WriteLine("\nСтавим в очередь на исполнение {0}
задач\n", iCount);

        Printer printer = new Printer();
        WaitCallback workTask = new WaitCallback(
                                              printer.PrintNumbers);

        for (int i = 1; i <= iCount; i++)
           ThreadPool.QueueUserWorkItem(workTask,
                                                 new Arguments(i,i*10));
     }
```

В первом случае (*button1_Click()*) в очередь пула ставится пять задач (метод *PrintNumbers()* класса *Printer*) без параметров (результат выполнения на рис.1.1). Как видно из окна вывода для их выполнения использовано три рабочих потока пула (номера: 3, 4, 6).

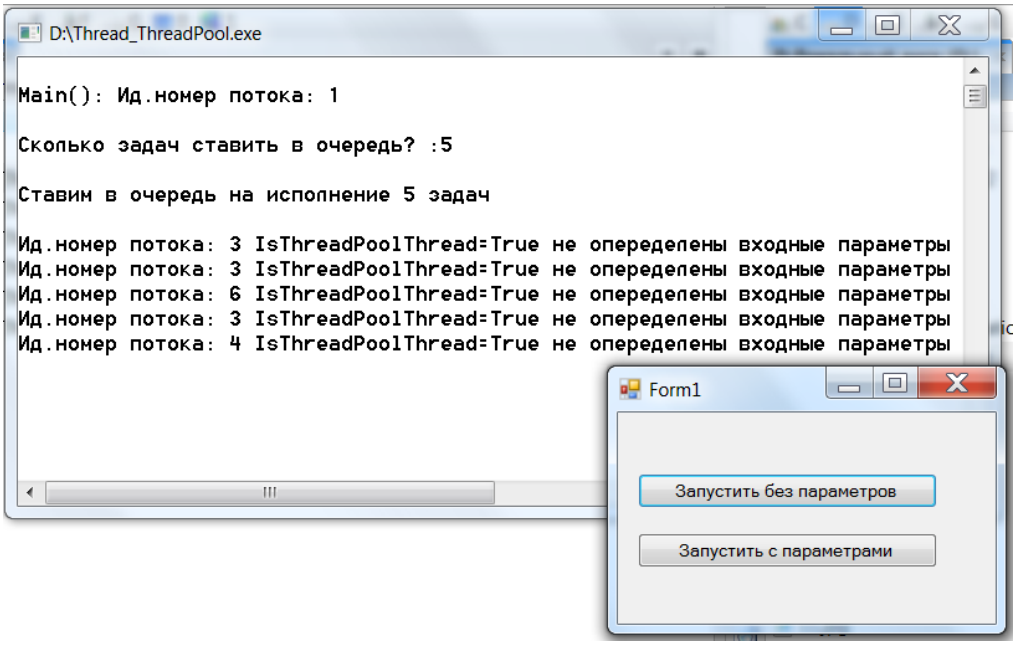


Рис. 1.1. Запуск потоков в пуле без параметров

Во втором случае (*button2_Click()*) в очередь пула ставится пять задач (метод *PrintNumbers()* класса *Printer*) с параметрами (результат выполнения на рис.1.2).

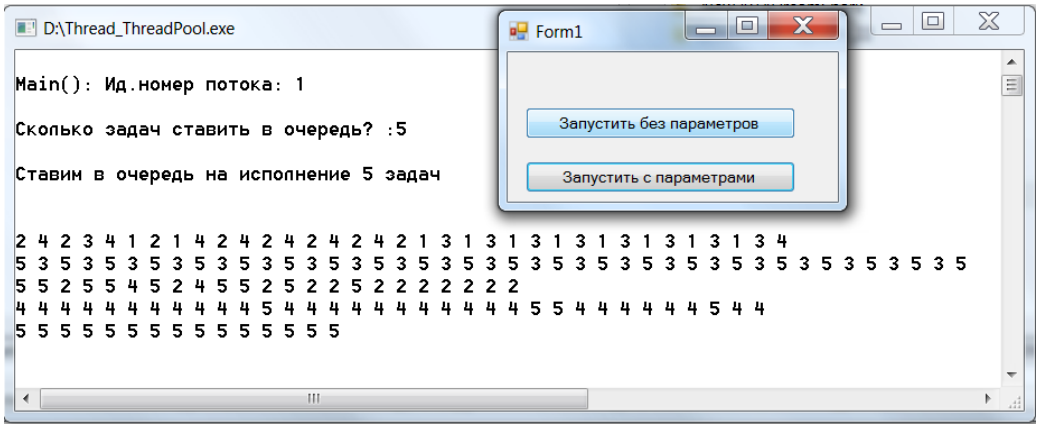


Рис. 1.2. Запуск потоков в пуле с параметрами

Организация параллельных вычислений средствами пула потоков позволяет программисту сконцетрироваться на решение задачи приложения, а не на потоковой инфраструктуре приложения. Но, в некоторых случаях «ручное» управление потоками приложения оказывается более предпочтительным, но требует осуществления управления потоками.

2. ПРОБЛЕМЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

2.1. Способы распараллеливания

В традиционном последовательном программировании предполагается, что приложение выполняется покомандно на одном процессоре. Для того, чтобы разработать параллельное приложение необходимо так построить решение, чтобы одновременно было возможно выполнять отдельные подзадачи в рамках общей решаемой приложением задачи.

Если удаётся общую задачу разбить на подзадачи, которые можно выполнять параллельно на разных процессорах, то говоря о расспараллеливании по задачам.

Расспараллеливание по задачам не всегда оказывается возможным. Но не все плохо, если решение задачи предполагает однотипную обработку массива однотипных данных с помощью определенного единого метода. В этом случае массив данных можно разбить на k подмассивов, для каждого из которых организовать свой поток обработки с одним и тем же методом. Такой способ называют расспараллеливанием по данным.

2.2. Потокобезопасность

Параллельно работающие потоки выполняют методы, работающие с входными и выходными данными. Когда у каждого потока свой выполняемый метод, свои входные данные и свои выходные данные, т.е. потоки полностью независимые это идеальная ситуация, при которой программисту не нужно заботиться о потоках. Отсутствие взаимодействия между потоками облегчает организацию работы многопоточного приложения. Такое возможно, но чаще приходится иметь дело с взаимодействующими потоками.

Одна из простейших форм взаимодействия - это ожидание. Выполнив свою часть работы поток должен приостановить выполнение, дожидаясь окончания работы другого потока или потоков, поскольку дальнейшие вычисления требуют данных, от других потоков. И для этого в классе *Thread* есть необходимые механизмы.

Другая, более сложная, форма взаимодействия потоков связана с использованием общих данных. Для обеспечения корректной работы взаимодействующих потоков в таких ситуациях приходится применять различные средства синхронизации, блокируя действия потока.

Для гарантирования корректной работы метода, который может быть вызван в разных потоках, на него накладываются такие ограничения – метод должен быть реентерабельным и потоко-безопасным модулем.

Метод называется ***реентерабельным*** (повторно вызываемым), если при вызове его в разных потоках он будет корректно работать при условии, что каждый вызов использует свои собственные данные.

Для обеспечения реентерабельности необходимо выполнение нескольких условий:

- никакая часть кода метода не должна модифицироваться;
- если метод сохраняет или изменяет какие-либо данные, то они должны быть уникальными для каждого потока;
- метод не должен возвращать ссылки на объекты, общие для разных пользователей.

Таким образом, метод является реентерабельным, если методы, которые он вызывает, являются реентерабельными и не имеет разделяемых данных при параллельных вызовах.

Метод называется ***потоко-безопасным***, если при вызове его в разных потоках он будет работать как реентерабельный (т.е. корректно) и в случае разделяемых данных. Согласно определению реентерабельный модуль является потоко-безопасным.

Создавая многопоточное приложение, программист обязан обеспечить потоковую безопасность разрабатываемых им методов, т.е. гарантировать корректность работы методов, вызываемых в разных потоках.

Если метод использует разделяемые (глобальные или статические) переменные, необходимо обеспечить, чтобы каждый поток хранил свою локальную копию этих переменных.

2.3. Проблемы параллельных вычислений

Создание многопоточных приложений сопряжено с двумя видами трудностей:

- наличие накладных расходов на создание и обслуживание потоков в операционной системе (требуется память и время на создание соответствующих структур данных, поддержка их работы);
- реализация многопоточного приложения требует усложнения обычного последовательного алгоритма (или даже разработки нового). Алгоритм, реализующий распараллеливание (параллельный алгоритм), бывает сложнее последовательного алгоритма, решающего одну и ту же задачу;
- необходимость решения проблем, связанных параллельной работой потоков.

3. СИНХРОНИЗАЦИЯ ВЫПОЛНЕНИЯ ПОТОКОВ

К проблемам параллельной потоковой обработки данных относятся:

- необходимость синхронизации потоков;
- ситуация гонки данных;
- ситуация клинча.

Синхронизация необходима для координации выполнения потоков. Такая координация необходима для согласования порядка выполнения потоков или для согласования доступа потоков к разделяемым потоками ресурсам.

3.1. Понятие синхронизации

Когда несколько методов выполняются в разных потоках (на разных процессорах), возникает необходимость в синхронизации их выполнения.

Например, потоку A необходимы результаты потоков B и C, т.е. он не может начать своё выполнение раньше, чем закончат выполнение потоки

B и C. Значит, запуск потока A на выполнение должен быть согласован во времени с завершением работы потоков B и C.

Другой пример, при расспараллеливании по данным один и тот же метод может быть запущен на k процессорах и параллельно выполняться (в потоках A1, A2, …,Ak), обрабатывая разные подмножества данных X=(X1, X2, …, Xk). Результаты работы этих k процессоров должны быть объединены в работе другого метода (поток B), для которого они являются исходными данными. Очевидно, что выполнение потока B должно быть согласовано с потоками A1, A2, …,Ak.

Таким образом, параллельные вычисления требуют тщательной синхронизации работы потоков приложения, выполняемых разными процессорами.

Под синхронизацией потоков понимают такую организацию их выполнения во времени, которая обеспечивает требуемую логику обработки даных в рамках процесса решения задачи приложением, частью которого они являются.

Временными характеристиками исполнения потока являются:

- момент запуска потока на исполнение,
- длительность работы потока;
- момент завершения работы потока

или какой-либо другой момент времени, характеризующий работу потока, важный для их согласованной работы во времени (например, момент освобождения ресурса).

Пусть имеем методы M1 и M2 , которые по логике решения задачи должны исполняться последовательно в потоках A и B так, что метод M2 не может начать своё исполнение раньше, чем закончит исполнение метод M1. Тогда, если t_a – это момент начала исполнения потока A, t_b – это момент начала исполнения потока B и Δt_a – это длительность работы потока A, то моменты начала исполнения этих потоков должны быть согласованы так, чтобы выполнялось условие:

$$t_b \geq t_a + \Delta t_a.$$

Но, когда мы пишем программу в потоках, мы не знаем ни одной из приведенных характеристик, потому что все эти характеристики зависят от быстродействия обработки данных в системе и операционой системы, которая управляет исполнением потоков. На самом деле для решения проблемы синхронизации потоков не требуется знание этих характеристик.

3.2. Проблема гонки данных

Эта проблема связана с использованием разделяемых ресурсов. Поскольку потоки, выполняемые разными процессорами, работают параллельно, то в одни и те же моменты времени они могут получать доступ к одним и тем же данным, хранимым в общей памяти, как для чтения, так и для записи.

Если совместное чтение данных представляется возможным и допустимым, то одновременная запись двух разных значений в одну и ту же ячейку памяти не допустима.

Поскольку, запись в один и тот же регистр памяти разными потоками в любом случае должна идти по очереди, то возникает ситуация конкурирования (гонка за обладание) потоков за регистр. Интересно, что в этой гонке выигрывает не тот, кто пришел первый, а тот, кто пришел последний, поскольку именно его результаты будут сохранены в памяти. Это явление в параллельном программировании называется гонкой данных.

Гонка данных – это одна из самых серьезных проблем параллельного программирования, поскольку может приводить к ошибочным результатам работы приложения даже при нормальном его завершении.

3.3. Проблема клинча

Способом решения проблемы гонки данных является блокировка ресурса для использования его другими потоками.

Чтобы справиться с гонкой данных, одному потоку приходится осуществлять захват ресурса (блокировку), закрывая его для других потоков-конкурентов, которые должны прервать свою работу, ожидая момента освобождения ресурса.

Блокировка – полезный механизм, обойтись без которого в ряде ситуаций просто невозможно. Но неправильное использование блокировки может приводить к ситуации, называемой тупиком или клинчем (deadlock).

Например, потоку A и B для выполнения требуются регистры R1 и R2. Допустим, поток A захватил регистр R1, а B захватил регистр R2. Ни поток A ни поток B не могут завершить своё исполнение, поскольку A ожидает освобождения R2, а поток B – освобождения R1. Типичная ситуация клинча.

4. СРЕДСТВА СИНХРОНИЗАЦИИ

Синхронизация необходима для обеспечения согласованного выполнения потоков:

- для задания требуемого порядка выполнения потоков;
- для обеспечения правильного использования разделяемых ресурсов.

Среда .NET предоставляет широкий набор средств синхронизации, представленные в табл. 4.1.

4.1. Средства блокировки потока

В основе синхронизации лежит понятие блокировки – перевода потока в состояние ожидания, до момента совершения какого-либо события (выполнения условия). Таким событием может быть завершения работы некоторого потока или освобождения разделяемого ресурса.

Ожидание (блокировка) может быть активным или пассивным.

Таблица 4.1 – Средства потоковой синхронизации .Net

Группа средств	Классы группы
Блокировка	*Join, Sleep*
Взаимно-исключительный доступ	*Lock, Monitor, Mutex, SpinLock*
Сигнальные сообщения	*AutoResetEvent, ManualResetEvent, ManualResetEventSlim*
Семафоры	*Semaphore, SemaphoreSlim*
Атомарные операторы	*Interlocked*
Конкурентные коллекции	*ConcurrentBag, ConcurrentQueue, ConcurrentDictionary, Concurrent-Stack, BlockedCollection*
Блокировки чтения-записи	*ReaderWriterLock, ReaderWriterLockSlim*
Шаблоны синхронизации	*Barrier, CountdownEvent*

4.1.1.Активная блокировка

При ***активном ожидании*** поток циклически проверяет факт наступления ожидаемого события, т.е. фактически поток не прекращает своей работы и не освобождает процессорное время для других потоков. Активное ожидание эффективно только при незначительном времени ожидания. Перевод потока в активное ожидание называется ***активной блокировкой***.

Рассмотрим пример активной блокировки потока. Основной поток запускает поток *thr* и переходит в состояние активного ожидания его завершения.

```
using System;
using System.Threading;
namespace activeThreadLock
{
   class Program
```

```
    {
        static void SomeWork()
        {
            Console.WriteLine("Thread {0} : State: {1} -- doing some-
thing...", Thread.CurrentThread.Name,
              Thread.CurrentThread.ThreadState);
            Thread.Sleep(1);
        }

        static void Main(string[] args)
        {
            Thread.CurrentThread.Name = "MainThread";

            Thread thr = new Thread(SomeWork);
            thr.Name = "SecondThread";
            thr.Start();

            while (thr.IsAlive) // активная блокировка осн. потока
            {  // пока цикл – активное ожидание
               // действия активного ожидания
                Console.WriteLine("Thread: {0} State: {1} -- Активное
ожидание завершения потока... ", Thread.CurrentThread.Name,
                                          Thread.CurrentThread.ThreadState);
            }
            Console.WriteLine("Thread: {0} State: {1} -- завершил свою
работу.", thr.Name,
                thr.ThreadState);
        }
    }
}
```

Результат выполнения данного примера программы представлен на рис. 4.1.

Как видно из листинга, после запуска вторичного потока основной поток продолжает выполняться благодаря циклу ожидания. Как только вторичный поток завершает свою работу основной поток выходит из активного ожидания. Для анализа состояния вторичного потока мы использовали объектное свойство IsAlive, значение true которого говорит, что поток не завершил свою работу.

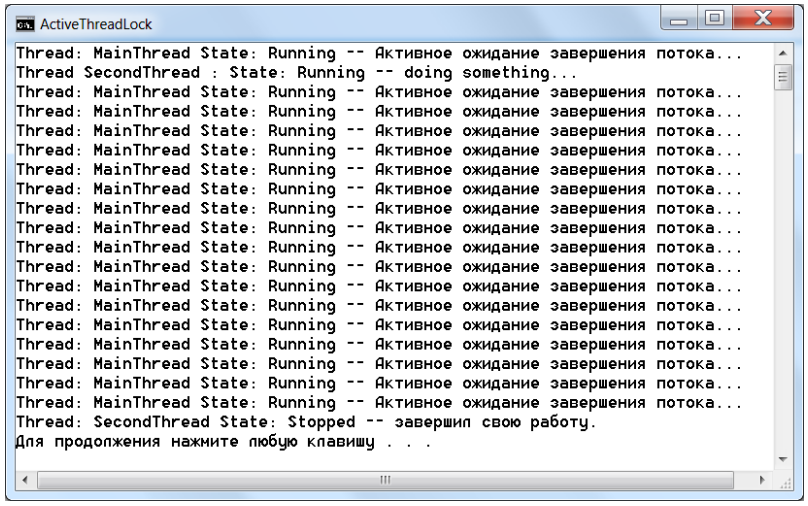


Рис. 4.1. Иллюстация активной блокировки

Вместо использования свойства *IsAlive* можно было использовать другие флаги или события, которые могут устанвливаться вторичным потоком. Такой приём обычно используется, если ожидание основного потока связано не с завершением вторичного потока в целом, а достижением этим потоком какого-либо состояния (например, освобождения ресурса или завершения какого-нибудь действия).

4.1.2.Пассивная блокировка

Пассивное ожидание реализуется с помощью операционной системы. Для потока, который переводится в пассивное ожидание, операционная система сохраняет контекст потока и выгружает его, предоставляя возможность выполняться другим потокам. При наступлении ожидаемого события операционная система «будит» поток – загружает контекст потока и выделяет ему процессорное время.

Пассивное ожидание требует времени на сохранение контекста потока при блокировке и загрузку контекста при возобновлении работы потока, но позволяет использовать вычислительные ресурсы во время пассивного ожидания для выполнения других потоков. Перевод потока в пассивное

ожидание называется ***пассивной блокировкой***. Следующий текст реализует пассивную блокировку потока (результат на рис. 4.2).

```
using System;
using System.Threading;
namespace passiveTreadLock
{
   class Program
   {
      static void SomeWork(object par)
      {
         Thread mainthread = (Thread)par;

         Console.WriteLine("Thread: {0}  State: {1} -- doing some-
thing...", Thread.CurrentThread.Name,
           Thread.CurrentThread.ThreadState);
         Console.WriteLine("Thread: {0}  State: {1} -- passive wait-
ing...", mainthread.Name, mainthread.ThreadState);
         Thread.Sleep(1);
      }
      static void Main(string[] args)
      {
         Thread.CurrentThread.Name = "MainThread";

         Thread thr = new
                        Thread((ParameterizedThreadStart)SomeWork);
         thr.Name = "SecondThread";
         thr.Start(Thread.CurrentThread);

                     // пассивная блокировка
         thr.Join();  //пассивное ожидание с выгрузкой контекста

         Console.WriteLine("Thread: {0} State: {1} -- end passive wait-
ing.", Thread.CurrentThread.Name,
        Thread.CurrentThread.ThreadState);

         Console.WriteLine("Thread: {0} State: {1} -- завершил свою
работу.", thr.Name,
             thr.ThreadState);
      }
   }
}
```

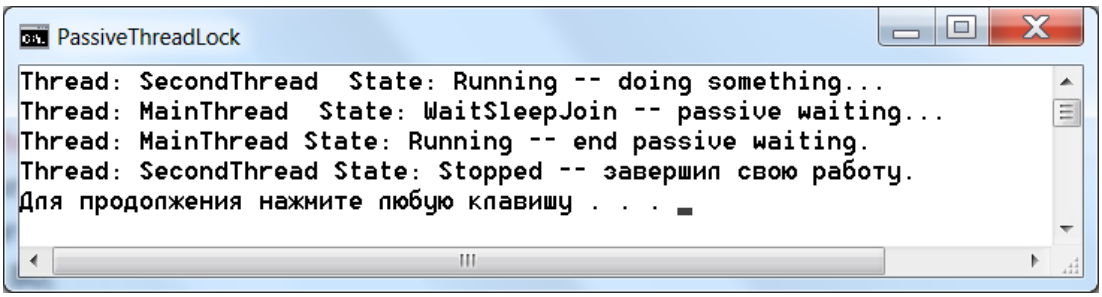


Рис. 4.2. Иллюстрация пассивной блокировки

Для перевода в состояние пассивного ожидания мы использовали объектный метод *Join(),* который не возвращает управление, пока не завершится вторичный поток, по отношению к которому вызван этот метод.

Если вызов *Join()* выполняется без аргумента, то вызывающий этот метод поток будет заблокирован до момента завершения потока, для которого был вызван метод *Join()*.

Если при запуске метода *Join()* ему указан аргумент (длительность блокировки), то метод возвращает управление по истечение этого времени, даже если вторичный поток не завершился. Для того чтобы узнать завершился ли вторичный поток необходимо анализировать возвращаемое методом *Join()* значение: true – говорит, что вторичный поток завершился, *false* – не завершился.

Чтобы показать, что основной поток действительно переводится в состояние пасивного ожидания (состояние *WaitSleepJoin*) мы вывели состояние основного потока во время выполнения вторичного потока, для чего передали во вторичный поток ссылку на основной поток.

Обратите внимание, что использование статического метода *Sleep()* также переводит поток, в котором применяется этот метод в пассивное ожидание. Отличие применения метода *Sleep()* от метода *Join()* состоит лишь в том, что ожидание не связывается с состоянием завершения другого потока, а определяется заданной для метода *Sleep()* задержкой времени.

4.2. Исключительный доступ

Средства блокировки решают задачу организации последовательного во времени порядка исполнения двух или более потоков. Но они не решают проблему гонки даных.

4.2.1.Иллюстрация проблемы гонки данных

Расмотрим простой пример. Пусть два параллельно работающих потока выполняют некоторый фрагмент кода. Первый поток выполняет оператор присваивания: Sum += vklad1; а второй: Sum += vklad2;

Если бы эти действия выполнялись в одном потоке или в потоках, выполняемых строго последовательно, то ожидаемым результатом должно было быть увеличение суммы Sum, как на величину первого, так и второго вклада.

Но при параллельном выполнении этих действий в потоках в ряде случаев из-за возникшей гонки данных сумма увеличится только на величину одного вклада, и нельзя сказать какого именно.

Дело в том, что оба оператора выполняются над общим ресурсом – переменной Sum. Оператор присваивания, рассматриваемый на уровне языка программирования, как одна операция, на уровне компьютера после трансляции превратится в группу команд. Вот как могут выглядеть два параллельно выполняемых потока команд компьютера:

Sum => R1	Sum => R2
Vklad1 => R3	Vklad2 => R4
R1 + R3 => R5	R2 + R4 => R6
R5 => Sum	R6 => Sum

Оба потока могут одновременно прочесть из памяти, отводимой переменной Sum, ее текущее значение и занести его на соответствующие регистры. Затем одновременно выполнить сложение в регистровой памяти. Но записать полученный результат в ячейку Sum придется последовательно.

Результат, который будет записан потоком, выполнившим действие первым, будет перезаписан вторым потоком. В результате переменная Sum будет увеличена только на величину одного вклада.

Если эти потоки (команды потоков) будут смещены по времени, т.е. сначала один поток, а потом другой, то в сумме будут учтены оба вклада.

Но чтобы добиться такого результата, необходимо предпринять определенные меры по синхронизации работы параллельно работающих потоков.

Для предметного анализа рассмотрим следующий тест, в котором имеем:

переменную sum, к которой будем прибавлять или вычитать удвоенное значение исходного числа. Прибавление или вычитание производится соответственно методами *Work1()* и *Work2()* с использованием промежуточных переменных *R1, R2* с внесением задержки (это необходимо, чтобы смоделировать ситуацию гонки данных). Методы выполняются в параллельных потоках, количество которых задаётся. Достоверные значения прибавленных и вычтенных значений фиксируются в переменных *was_add* и *was_sub*. По завершении всех вычислений сопоставим полученную сумму и ту, которая должна была получиться. Предложенный тест реализуется следующей программой:

```
class Race
   {
      public int R1, R2;
      public int sum=0;
      public int was_add=0;
      public int was_sub=0;

      public void Work1(object op)
      {
         was_add += (int)op * 2;
         R1 = (int)op;
         Thread.Sleep(0);
         R2 = R1 + (int)op;
         sum += R2;
         Console.Write("Thread {0}: add({1})\n",
                        Thread.CurrentThread.Name, 2 * (int)op);
```

```
        }
        public void Work2(object op)
        {
            was_sub -= (int)op * 2;
            R1 = (int)op;
            Thread.Sleep(0);
            R2 = -R1 - (int)op;
            sum += R2;
            Console.Write("Thread {0}: sub({1})\n",
                                Thread.CurrentThread.Name, 2 * (int)op);
        }
}

private void button1_Click(object sender, EventArgs e)
   {
       Race race_test = new Race();

       int num = (int)numericUpDown1.Value;

       Random rand = new Random();

       Thread[] t = new Thread[num];

      for (int i = 0; i < num; i++)
       if (rand.Next(-num, num) > 0)
        {
          t[i] = new Thread((ParameterizedThreadStart)race_test.Work1);
          t[i].Name = "поток" + i;
       }
         else
         {
           t[i] = new
Thread((ParameterizedThreadStart)race_test.Work2);
           t[i].Name = "поток" + i;
         }

       foreach (Thread p in t)
          p.Start(rand.Next(100,10000));

       foreach (Thread p in t)
          p.Join();
```

```
        Console.WriteLine("was_add( {0} ) + was_sub( {1} ) = {2}\n ,
    sum={3}", race_test.was_add,
                  race_test.was_sub,
                  race_test.was_add + race_test.was_sub,
                  race_test.sum);
        Console.WriteLine("Результат " +
              ((race_test.was_add + race_test.was_sub ==
    race_test.sum) ?
                  "правильный " : "неправильный"));
      }
```

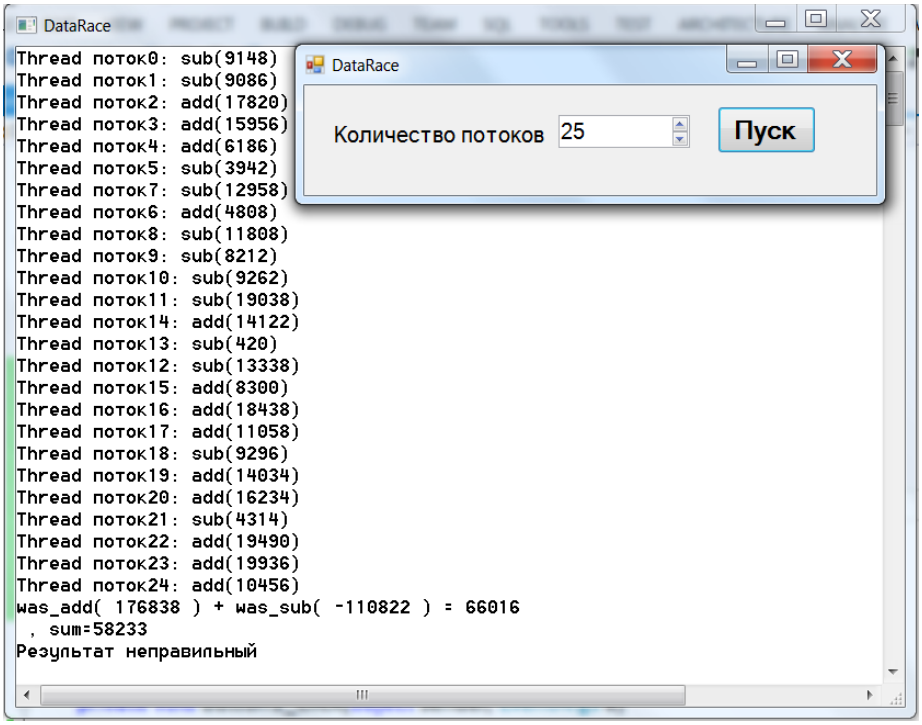


Рис. 4.3. Иллюстрация гонки данных

Как видно из рис.4.1 на 25 потоках тест даёт ошибку, которая является следствием гонки данных.

4.2.2.Обеспечение исключительного доступа к ресурсу

Одно из основных назначений средств синхронизации заключается в организации взаимно исключающего доступа к разделяемому ресурсу (ресурсам).

Исключительный доступ обеспечивает изменение общих данных, хранимых разделяемым ресурсом в одно и тоже время, только одним потоком, вследствие чего обеспечивается семантика логики решения задачи.

Фрагмент кода метода, в котором осуществляется работа с разделяемым ресурсом и который должен выполниться за один раз от начала и до конца без возможности использования этого ресурса другими потоками (чтобы исключить гонку данных), называется ***критической секцией***.

Решение проблемы гонки данных сводится к блокированию критической секции. Для блокирования критической секции можно использовать простую синтаксическую конструкцию *lock* или специальный класс *Monitor*.

4.2.3.Блокирование критической секции конструкцией lock

Выделение и блокировка критической секции с помощью конструкции *lock* позволяет решить проблему гонки данных.

Чтобы использовать конструкцию *lock* необходимо:

1) Для всех методов, которые будут выполняться в параллельных потоках и могут конкурировать за общий ресурс (ресурсы), выделить критические секции.
2) Для выделенных критических секций создать объект-флаг, видимый (доступный) для всех этих секций. Обычно это объект универсального типа *object* с именем, например, *locker*.
3) Закрыть каждую выделенную критическую секцию оператором *lock* с ключом *locker*.

Синтаксически конструкция блокировки критической секции с ключом *locker* выглядит так:

```
lock ( locker )
{
  < операторы критической секции>
}
```

а механизм блокировки критической секции определяет такие правила взаимодействия потоков с критическими секциями, блокируемыми одним и тем же ключом:

1) Блокировка критической секции с ключом *locker* не позволит ни одному другому потоку войти в свою критическую секцию с ключом *locker* пока какой-нибудь поток находится в своей критической секции, заблокированной этим ключом.
2) Доступ в критическую секцию потокам будет разблокирован после того как поток, установивший блокировку, выйдет из своей критической секции.
3) Если другой поток должен войти в свою критическую секцию, которая заблокирована, он будет ожидать снятия блокировки.

Благодаря такому механизму в одно и то же время будет выполняться только критическая секция одного потока, который первым установил блокировку ключа, что позволит справиться с проблемой гонки данных и разработать потокобезопасное приложение.

Для иллюстрации рассмотренного механизма блокировки критической секции вернёмся к примеру программы, приведённом в разделе 4.2.1 . Проанализируем этот код.

В програмной реализации параллельные потоки образуются для выполнения методов *Work1()* и *Work2(),* которые используют разделяемые ресурсы – переменные R1, R2 и sum. Конкуренция на этих ресурсах и является причиной не правильного результата в переменной sum. Отметим, что переменные *was_add* и *was_sub* не являются разделяемыми.

Для иллюстрации борьбы с гонкой данных на рассмотренных ресурсах создадим новый класс *LockedRace,* который унаследуем от класса *Race:*

```
class LockedRace : Race
  {
    object used = new object();

    public new void Work1(object op)
    {
      was_add += (int)op * 2;

      lock (used)
      {
```

```
            R1 = (int)op;
            Thread.Sleep(0);
            R2 = R1 + (int)op;
            sum += R2;
        }
        Console.Write("Thread {0}: add({1})\n",
                        Thread.CurrentThread.Name, 2 * (int)op);
    }

    public new void Work2(object op)
    {
        was_sub -= (int)op * 2;

        lock (used)
        {
            R1 = (int)op;
            Thread.Sleep(0);
            R2 = -R1 - (int)op;
            sum += R2;
        }

        Console.Write("Thread {0}: sub({1})\n",
                        Thread.CurrentThread.Name, 2 * (int)op);
    }
}
```

В новом классе мы используем объект used типа object, который выполняет роль ключа при блокировании критических секций, выделенные синтаксической конструкцией *lock*. Для завершения эксперимента необходимо соответственно изменить метод обработки нажатия кнопки пуск, заменив строку

```
Race race_test = new Race();
```

на строку

```
LockedRace race_test = new LockedRace();
```

На рис. 4.4 представлен результат работы новой версии приложения. Как видно из листинга в этом случае ошибка исключается.

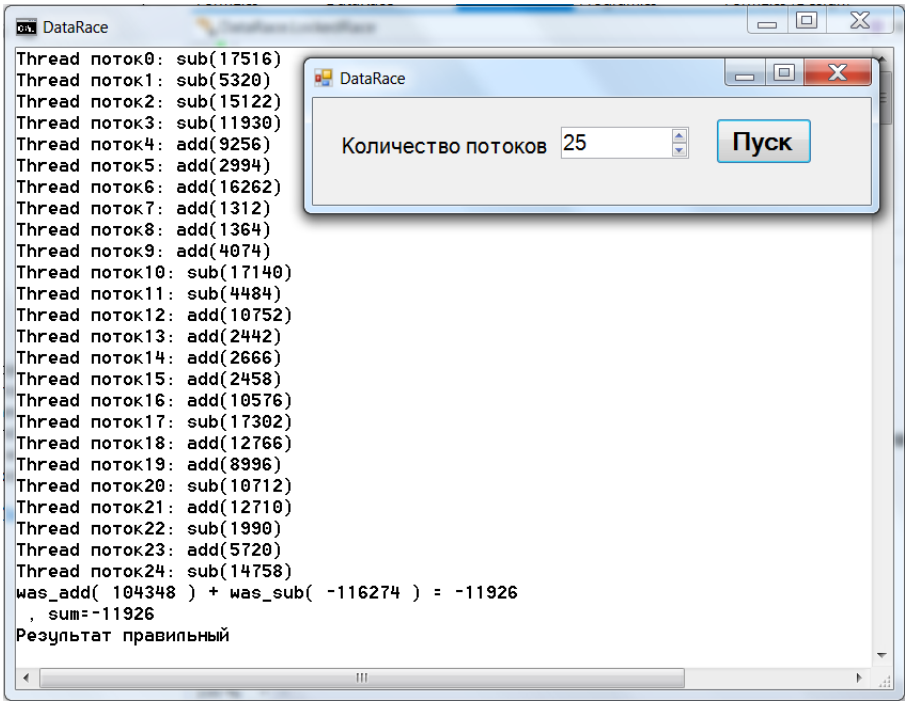


Рис. 4.4. Иллюстрация результата борьбы с гонкой данных

4.2.4.Использование класса Monitor

Синтаксическая конструкция *lock* является упрощённым аналогом применения класса *Monitor* из пространства *System.Threading*.

Класс *Monitor* поддерживает рассмотренный ранее механизм блокирования критических секций конкурирующих на общих ресурсах потоков через объекты синхронизации (ключи в конструкции *lock*). Его использование может показаться сложнее, чем использование конструкции *lock*, но взамен он обеспечивает более гибкие средства для организации взаимодействия потоков на общих ресурсах.

Для синхронизации потоков класс *Monitor* использует следующие основные методы:

Методы *Enter()* и *TryEnter()*, которые используются для блокировки объекта (критической секции). Вызов этих методов начинает критическуй секцию кода. Поток не войдёт в критическую секцию, связанную с объектом синхронизации пока какой-либо поток находится в критической секции, определяемой этим объектом синхронизации, т.е. блокирует этот объект.

Например, метод

public static void Enter(Object obj)

устанавливает исключительную блокировку заданного объекта *obj* и возвращает управление, если объект не заблокирован. Иначе этот метод ожидает, пока объект будет разблокирован. В качестве объекта блокировки *obj* можно использовать только объект ссылочного типа.

Метод

public static void Exit(Object obj)

снимает исключительную блокировку объекта и используется как последний оператор критической секции, с которой связан этот объект.

Кроме рассмотренных методов класс *Monitor* содержит другие методы: для уведомления о снятии блокировки (методы *Pulse()* и *PulseAll()*) и передачи права блокировки другому потоку с ожиданием уведомления о снятии блокировки (*Wait()*). Эти методы позволяют тонко управлять конкурирующими потоками, они будут рассмотрены при изложении проблемы клинча и способа борьбы с ним.

Для илюстрации создадим класс *MonitoredRace* по аналогии с классом *LockedRace*.

```
class MonitoredRace : Race
   {
      object used = new object();

      public new void Work1(object op)
      {
         was_add += (int)op * 2;

         Monitor.Enter(used); // начало критической секции
         R1 = (int)op;
         Thread.Sleep(0);
         R2 = R1 + (int)op;
         sum += R2;
         Monitor.Exit(used); // конец критической секции

         Console.Write("Thread {0}: add({1})\n",
                        Thread.CurrentThread.Name, 2 * (int)op);
      }
```

```
        public new void Work2(object op)
        {
            was_sub -= (int)op * 2;

            Monitor.Enter(used); // начало критической секции
            R1 = (int)op;
            Thread.Sleep(0);
            R2 = -R1 - (int)op;
            sum += R2;
            Monitor.Exit(used); // конец критической секции

            Console.Write("Thread {0}: sub({1})\n",
                          Thread.CurrentThread.Name, 2 * (int)op);
        }
    }
```

Эффект от применения класса *Monitor* в данной ситуации будет такой же как и в случае применения конструкции *lock*.

В справочнике по классу *Monitor* можно найти рекомендацию выделения критической секции в try блок таким образом:

```
try
{
  Monitor.Enter(sync_obj);
  < операторы критической секции>
}
finally
{
  Monitor.Exit(sync_obj);
}
```

Обоснованием является необходимость гарантирования освобождения блокировки критической секции. Кроме этого в таком виде критическая секция имеет более красивое структурное оформление.

Для илюстрации ниже приводится новый класс *tryMonitoredRace:*

```
class tryMonitoredRace : Race
    {
        object used = new object();

        public new void Work1(object op)
        {
```

```
        was_add += (int)op * 2;

        try
        {
            Monitor.Enter(used); // начало критической секции
            R1 = (int)op;
            Thread.Sleep(0);
            R2 = R1 + (int)op;
            sum += R2;
        }
        finally
        {
            Monitor.Exit(used); // конец критической секции
        }

        Console.Write("Thread {0}: add({1})\n",
                        Thread.CurrentThread.Name, 2 * (int)op);
    }

    public new void Work2(object op)
    {
        was_sub -= (int)op * 2;

        try
        {
            Monitor.Enter(used); // начало критической секции
            R1 = (int)op;
            Thread.Sleep(0);
            R2 = -R1 - (int)op;
            sum += R2;
        }
        finally
        {
            Monitor.Exit(used); // конец критической секции
        }

        Console.Write("Thread {0}: sub({1})\n",
                        Thread.CurrentThread.Name, 2 * (int)op);
    }
}
```

4.2.5.Иллюстрация проблемы клинча

Клинч или дедлок (deadlock) одна из самых серьезных проблем, возникающих при параллельном программировании.

Клинч может возникнуть в ситуации, когда два или более параллельно выполняемых потока конкурируют за обладание двумя или более общими ресурсами.

При клинче каждый из потоков успевает захватить один из общих ресурсов. Но для окончания работы каждому потоку необходимы другие ресурсы, уже захваченные другими потоками. В результате, никто из потоков не может завершить свою работу, все стоят в очередях, которые не двигаются и приложение «зависает».

Модель клинча можно описать следующим образом.

Пусть имеем два потока P1 и P2 и два ресурса R1 и R2. Пусть поток P1, входя в критическую секцию, захватывает ресурс R1, блокируя его для использования потоком P2. Аналогично, работая параллельно, поток P2, входя в критическую секцию, захватывает ресурс R2, блокируя его для использования потоком P1.

Потоку P1 в какой-то момент работы в критической секции становится необходимым ресурс R2, но этот ресурс заблокирован и поток становится в очередь, прерывая свое выполнение. Симметричная ситуация возникает с потоком P2.

Из описанной ситуации можно сделать вывод. Блокировка критической секции позволяет решить проблему гонки данных, но может привести к проблеме клинча.

Для анализа проблемы клинча используем пример программы, на котором была проиллюстрирована проблема гонки данных. Решением проблемы гонки данных было выделение и бокирование критических секций, например:

```
lock (used)
  {
     R1 = (int)op;
     Thread.Sleep(0);
     R2 = R1 + (int)op;
     sum += R2;
  }
```

Блокирование критической секции целикрм является не совсем эффективным в плане организации вычислений. Ведь сначала мы используем регистр R2, а затем и R1, но блокируем совместно.

Если в критических секциях работа с ресурсами ведется последовательно, а не одновременно, то ресурс следует освобождать, как только работа с ним закончена. Это правило работы во многих случаях позволяет избавиться от клинча и экономит время.

В следующем примере выполним блокирование критической секции по-другому:

```
class DeadLock : Race
   {
      object R1_sync = new object();
      object R2_sync = new object();

      public new void Work1(object op)
      {
         was_add += (int)op * 2;

         lock (R1_sync)
         {

            R1 = (int)op;
            Thread.Sleep(0);

            Console.Write("Thread {0}: ожидает ресурс R2\n",
                          Thread.CurrentThread.Name);
            lock (R2_sync)
            {
               Console.Write("Thread {0}: получил ресурс R2\n",
                             Thread.CurrentThread.Name);
               R2 = R1 + (int)op;
               sum += R2;
               Thread.Sleep((new Random()).Next(0, 100));
            }
         }

         Console.Write("Thread {0}: add({1})\n",
                       Thread.CurrentThread.Name, 2 * (int)op);
      }

      public new void Work2(object op)
      {
         was_sub -= (int)op * 2;
```

```
            lock(R2_sync)
            {
                R2 = (int)op;
                Thread.Sleep(10);

                Console.Write("Thread {0}: ожидает ресурс R1\n",
                                Thread.CurrentThread.Name);
                lock (R1_sync)
                {
                    Console.Write("Thread {0}: получил ресурс R1\n",
                                    Thread.CurrentThread.Name);
                    R1 = -R2 - (int)op;
                    sum += R1;
                    Thread.Sleep((new Random()).Next(500, 1000));
                }
            }
            Console.Write("Thread {0}: sub({1})\n",
                            Thread.CurrentThread.Name, 2 * (int)op);
        }
    }
```

Как видно из текста мы ввели два объекта синхронизации *R1_sync* и *R2_sync* для блокирования использования ресурсов *R1* и *R2*.

Метод *Work1* сначала блокирует *R1_sync*, получая ресурс R1, а затем пытается получить ресурс *R2*. Но метод *Work2* делает то же в точности на оборот и если метод *Work1* не успеет раньше метода *Work2* захватить ресурс *R2*, то потоки впадают в клинч (см. рис. 4.5).

Мы привели пример, используя конструкцию *lock*, такая же ситуации будет и при использовании класса *Monitor*.

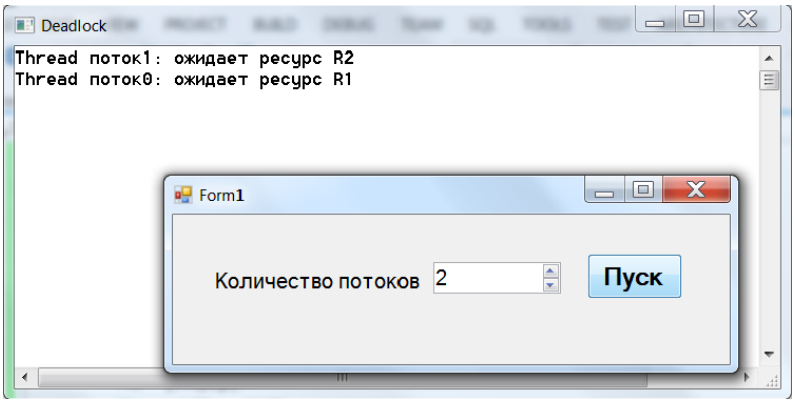


Рис. 4.5. Иллюстрация проблемы клинча

Блокировка критической секции позволяет избавиться от гонки данных, но может приводить к клинчу. Для борьбы с клинчем используют разные способы.

4.2.6.Борьба с клинчем закрытием критических секций одним ключом

Самый простой способ борьбы с клинчем – это закрытие критических секций одним ключом.

Если каждая критическая секция закрывается одним ключом, это гарантирует, что в каждый момент времени исполняться будет только одна критическая секция и клинча не будет. Такое решение мы использовали, когда боролись с гонкой данных.

Недостатком такого подхода является увеличение общего времени ожидания, что не всегда разумно, когда все ресурсы принадлежат одному владельцу, а он не пользуется ими одновременно.

И, как кажется сейчас, в нашем примере это единственное правильное решение. Но кроме этого способа существуют и другие, которые позволяют не потерять эффект от распараллеливаня.

4.2.7.Борьба с клинчем использованием сигнальной синхронизации класса Monitor

Класс Monitor позволяет более тонко синхронизировать конкурирующие потоки. Объект Monitor предоставляет методы для обмена сигналами: Pulse и Wait, которые могут быть полезны для предотвращения взаимоблокировки (клинча) в случае одновременной работы с несколькими разделяемыми ресурсами.

Рассмотрим схематично модель борьбы с клинчем этими средствами, которая в точности воспроизводит наш предыдущий пример:

Пусть имеем два потока, которые пытаются захватить ресурсы P и Q. Первый поток захватывает сначала ресурс P, а затем пытается захватить ресурс Q. Второй поток сначала захватывает ресурс Q, а затем пытается захватить ресурс P – типичная ситуация для возможного клинча.

Применение обычной конструкции lock, как мы выяснили, в некоторых случаях приводит к взаимоблокировке потоков – потоки успевают захватить по одному ресурсу и пытаются получить доступ к недостающему ресурсу.

Рассмотрим способ решения проблемы клинча, используя сигнальную синхронизацию класса *Monitor*:

```
void ThreadOne()
{
// Получаем доступ к ресурсу P
Monitor.Enter(P);

// Пытаемся захватить ресурс Q
if(!Monitor.TryEnter(Q))
{
  // Если Q занят другим потоком, освобождаем P и
  // ожидаем завершения работы потока – сигнала Pulse
  Monitor.Wait(P);

 // Освободился ресурс P, смело захватываем ресурс Q
 Monitor.Enter(Q);
}
// Теперь у потока есть и P, и Q, выполняем работу:

<операторы, решающие задачу с использованием P и Q>..

// Освобождаем ресурсы в обратной последовательности
Monitor.Exit(Q);
Monitor.Exit(P);
}

void ThreadTwo()
{
 // Получаем доступ к ресурсам P и Q
   Monitor.Enter(Q);
   Monitor.Enter(P);

   // Выполняем необходимую работу:

  <операторы, решающие задачу с использованием P и Q>

   // посылка сигнала о завершении использования ресурса P
  // для потока, заблокированного вызовом Monitor.Wait(P)
```

```
        Monitor.Pulse(P);

    // Освобождаем ресурсы в обратной последовательности
        Monitor.Exit(P);
        Monitor.Exit(Q);
    }
```

Первый поток после захвата ресурса P пытается захватить ресурс Q. Если ресурс Q уже занят, то первый поток, зная, что второму нужен еще и ресурс P, освобождает его и ждёт освобождения обоих ресурсов. Вызов Wait блокирует первый поток и позволяет другому потоку (одному из ожидающих) войти в критическую секцию для работы с ресурсом P.

Работа заблокированного первого потока может быть продолжена после того как выполняющийся второй поток вызовет метод Pulse и покинет критическую секцию.

Опишем подробно методы класса Monitor, используемые в рассмотренной схеме.

Метод

public static bool TryEnter(Object obj)

пытается получить исключительную блокировку указанного объекта и возвращает значение true, если это удётся, в противном случае возвращает значение false.

Метод

public static bool Wait(Object obj)

Снимает блокировку объекта и не возвращает управление, пока не получит блокировку этого объекта снова. Метод возвращает значение true, если снова получена блокировка.

Метод

public static void Pulse(object obj)

уведомляет о предстоящем изменении состояния заблокированного объекта синхронизации (о снятии блокировки). Этот метод может вызываться только потоком, который установил свою блокировку над объектом.

При получении импульса (сигнала), посланного в результате выполнения этого метода, ожидающий поток (выполняющий метод *Wait()*) перемещается в очередь готовности. Когда поток, вызвавший метод Pulse освобождает блокировку (запуском метода *Exit()*), следующий поток в очереди готовности (который не обязательно является потоком, получившим импульс) получает блокировку.

Ниже приводится программный код класса, написанный в соответствии с рассмотренной моделью борьбы с клинчем, а на рис.4.6-7 результаты работы программы.

```
class MonitoredDeadLock : Race
   {
      object R1_sync = new object();
      object R2_sync = new object();

      public new void Work1(object op)
      {
         was_add += (int)op * 2;

         // Получаем доступ к ресурсу R1_sync
         Monitor.Enter(R1_sync);
         Console.Write("Thread {0}: получил ресурс R1\n",
                           Thread.CurrentThread.Name);

         R1 = (int)op;
         Thread.Sleep(0);

         // Пытаемся захватить ресурс R2_sync
         Console.Write("Thread {0}: пытается получить ресурс R2\n",
                           Thread.CurrentThread.Name);
         if (!Monitor.TryEnter(R2_sync))
         {
            Console.Write("Thread {0}: ресурс R2 занят другим потоком, освобождаем R1 переходим к Wait(R1_sync)\n",
                              Thread.CurrentThread.Name);
            Monitor.Wait(R1_sync);
```

```
            Console.Write("Thread {0}: получен сигнал Pulse для
Wait(R1_sync)\n",
                                Thread.CurrentThread.Name);
            // Освободился R1_sync, смело захватываем ресурс
R2_sync
            Monitor.Enter(R2_sync);
        }

        // Теперь у потока есть оба ресурса, выполняем работу:

        Console.Write("Thread {0}: получил ресурс R2\n",
                        Thread.CurrentThread.Name);
        R2 = R1 + (int)op;
        sum += R2;
        Thread.Sleep((new Random()).Next(0, 100));

        Console.Write("Thread {0}: выполнил действие add({1})\n",
                        Thread.CurrentThread.Name, 2 * (int)op);

        // Освобождаем ресурсы в обратной последовательности
        Monitor.Exit(R2_sync);
        Console.Write("Thread {0}: освободил ресурс R2\n",
                        Thread.CurrentThread.Name);
        Monitor.Exit(R1_sync);
        Console.Write("Thread {0}: освободил ресурс R1\n",
                        Thread.CurrentThread.Name);
    }

    public new void Work2(object op)
    {
        was_sub -= (int)op * 2;

        // Получаем доступ к ресурсам
        Monitor.Enter(R2_sync);
        Console.Write("Thread {0}: получил ресурс R2\n",
                        Thread.CurrentThread.Name);

        R2 = (int)op;
        Thread.Sleep(10);

        Console.Write("Thread {0}: пытается получить ресурс R1\n",
```

```
                    Thread.CurrentThread.Name);
        Monitor.Enter(R1_sync);

        // Выполняем необходимую работу:

        Console.Write("Thread {0}: получил ресурс R1\n",
                    Thread.CurrentThread.Name);
        R1 = -R2 - (int)op;
        sum += R1;
        Thread.Sleep((new Random()).Next(500, 1000));

        Console.Write("Thread {0}: выполнил действие sub({1})\n",
                    Thread.CurrentThread.Name, 2 * (int)op);

        // посылка сигнала о завершении использования ресурса R1
        // для потока, заблокированного вызовом
Monitor.Wait(R1_sync)

        Monitor.Pulse(R1_sync);

        // Освобождаем ресурсы в обратной последовательности
        Monitor.Exit(R1_sync);
        Console.Write("Thread {0}: освободил ресурс R1\n",
                    Thread.CurrentThread.Name);
        Monitor.Exit(R2_sync);
        Console.Write("Thread {0}: освободил ресурс R2\n",
                    Thread.CurrentThread.Name);
    }
}
```

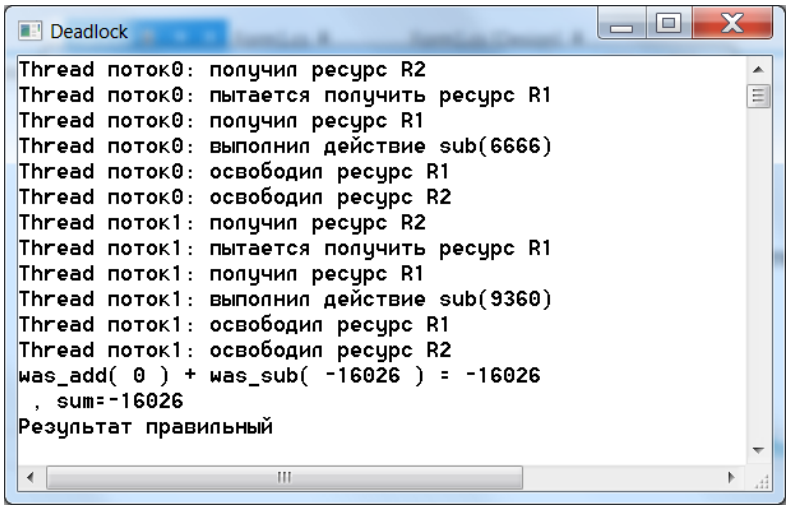


Рис. 4.6. Ситуация нет клинча

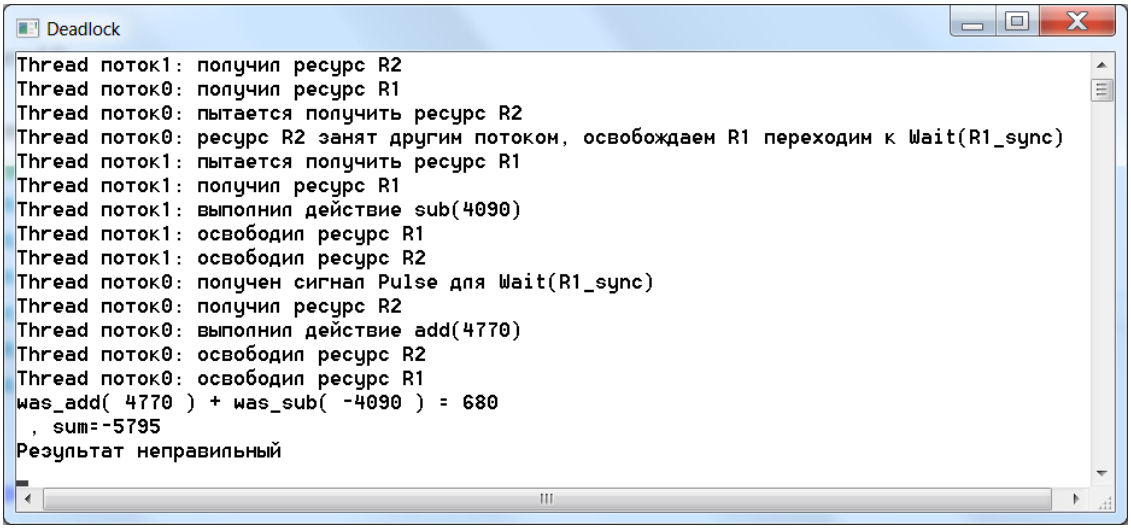


Рис. 4.7. Ситуация ухода от клинча

Отметим, что приведенное решение позволяет бороться с клинчем в случае двух потоков. Но в случае большего чем два числа потоков клинч по-прежнему возможен. Так на рис. 4.8 приведен результат работы рассмотренной выше программы при четырёх потоках, который привёл к клинчу. Разберитесь и ответьте почему и предложите решение проблемы.

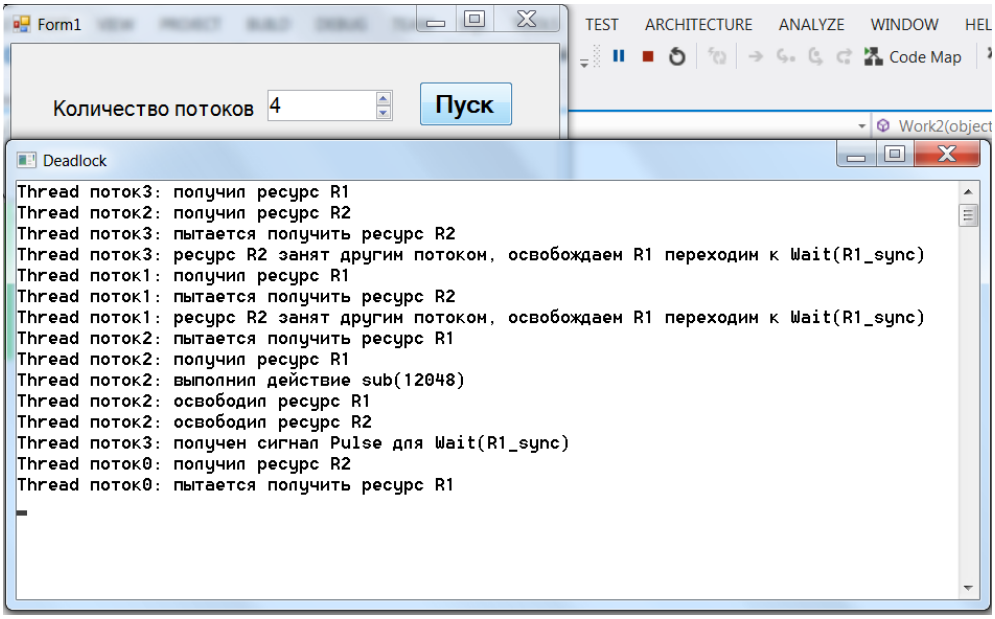


Рис. 4.8. Ситуация клинча на четырёх потоках

4.2.8.Использование класса Mutex

Объект Mutex используется, как и *Monitor*, для обеспечения взаимно-исключительного доступа к критической секции кода. В основе объекта *Mutex* лежит вызов функции ядра операционной системы, и поэтому блокировка с помощью Mutex является менее эффективной по сравнению с классом Monitor и конструкцией lock.

Отличие от Monitor заключается в том, что *Mutex* обеспечивает глобальную блокировку и с его помощью можно организовать синхронизацию нескольких разных приложений.

Для этого в каждом приложении необхоимо создать объет класса *Mutex* с одним и тем же именем, например:

Mutex mut = new Mutex(false, "MyMutex");

Перед входом в критическую секцию необходимо вызвать метод WaitOne() для объекта класса *Mutex,* который используется для синхронизации. Метод

bool WaitOne(int msTimeout)

блокирует текущий поток на время не более *msTimeout* до снятия блокировки критической секции используемым объектом класса *Mutex.* Если блокировка снята, метод возвращает значение true, иначе – false.

Класс *Mutex* можно использовать для разных приложений. Доступ к необходимому объекту класса *Mutex* задаётся именем, заданным при создании объекта этого класса.

В следующем примере мы создаём два разных приложения, но которые имеют одинаковую реализацию класса *Program* с точностью до имени окна консоли и имени объекта *Mutex.* Эти приложения запускаются на исполнение и как видно из рис.4.9 используемый объект класса *Mutex* успешно организует исключительную блокировку секций на уровне операционной системы.

Текст приложения *Mutex:*

```
using System;
using System.Threading;
using System.Windows.Forms;
```

```
namespace Mutex
{
    public partial class Form1 : Form
    {
        private static Mutex mut = new Mutex(false, "MyMutex");

        public Form1()
        {
            InitializeComponent();
            // настройка консоли
            Console.Title = "Mutex2";
            Console.BackgroundColor = ConsoleColor.White;
            Console.ForegroundColor = ConsoleColor.Black;
            Console.WindowWidth = 90;
            Console.WindowHeight = 20;
            Console.Clear();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Console.Clear();
            while (!mut.WaitOne(1000))
            {
                Console.WriteLine("Поток {0} не может получить
мьютекс",
                                  Thread.CurrentThread.Name);
            }
            // начало критической секции
            Console.WriteLine("Поток {0} зашёл в критическую секцию
{1}", Thread.CurrentThread.Name, mut.ToString());

            // Обработка:   выполнение каких-либо действий

            Thread.Sleep(10000);
            Console.WriteLine("Поток {0} покидает критическую сек-
цию", Thread.CurrentThread.Name);

            // освобождаем мьютекс
            mut.ReleaseMutex();
```

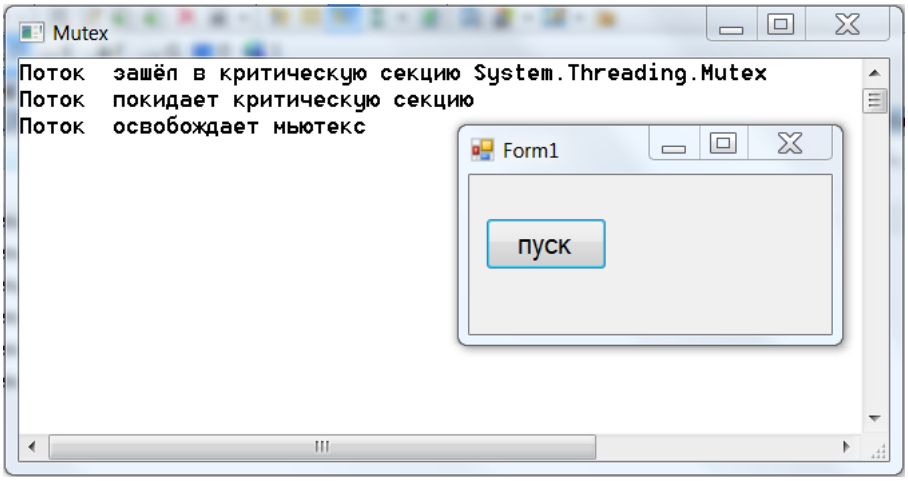
```
            // конец критической секции
            Console.WriteLine("Поток {0} освобождает мьютекс",
                                    Thread.CurrentThread.Name);
        }
    }
}
```

Текст приложения *Mutex2:*

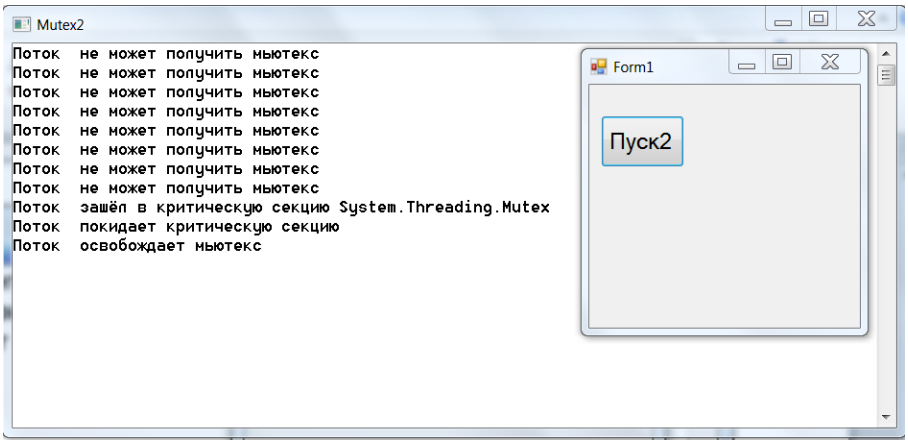
```
using System;
using System.Threading;
using System.Windows.Forms;

namespace Mutex2
{
    public partial class Form1 : Form
    {
        private static Mutex mut2 = new Mutex(false, "MyMutex");

        public Form1()
        {
            InitializeComponent();
            // настройка консоли
            Console.Title = "Mutex2";
            Console.BackgroundColor = ConsoleColor.White;
            Console.ForegroundColor = ConsoleColor.Black;
            Console.WindowWidth = 90;
            Console.WindowHeight = 20;
            Console.Clear();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Console.Clear();

            while (!mut2.WaitOne(1000))
            {
                Console.WriteLine("Поток {0} не может получить
мьютекс",  Thread.CurrentThread.Name);
            }
```

```
            // начало критической секции
            Console.WriteLine("Поток {0} зашёл в критическую секцию
{1}", Thread.CurrentThread.Name, mut2.ToString());

            // Обработка:   выполнение каких-либо действий

            Thread.Sleep(10000);
            Console.WriteLine("Поток {0} покидает критическую сек-
цию",
                                        Thread.CurrentThread.Name);
            // освобождаем мьютекс
            mut2.ReleaseMutex();

            // конец критической секции
            Console.WriteLine("Поток {0} освобождает мьютекс",
                                    Thread.CurrentThread.Name);
        }
    }
}
```

В каждом из приведённых приложений имеется цикл ожидания выхода другого приложения из критической секции, который длится пока приложение не получит блокировку используемого объекта *Mutex*. Как видно из текста не важно какое используется имя объекта, а важно используемое имя (в нашем примере *MyMutex*) при создании этого объекта.



а) окно приложения *Mutex*



б) окно приложения *Mutex2*

Рис. 4.9. Использование класса *Mutex*

4.2.9.Использование класса SpinLock

Убрать

Структура *SpinLock* позволяет организовать взаимно исключающую блокировку, при которой поток, пытающийся получить блокировку, ожидает в состоянии цикла, проверяя доступность блокировки. Использование такой блокировки может быть лучше по сравнению с блокировкой *Monitor*, но однозначной аргументации этого нет.

SpinLock является типом значения для повышения производительности. По этой причине необходимо быть очень осторожным, чтобы случайно не скопировать экземпляр *SpinLock*, поскольку два экземпляра (оригинал и копия) будут полностью независимы друг от друга, что может привести к ошибочному поведению приложения. Если экземпляр *SpinLock* должен передаваться, он должен передаваться по ссылке, а не по значению.

4.3. Сигнальные сообщения

Сигнальные сообщения могут использоваться как для обеспечения взаимно исключительного доступа, так и для условной синхронизации. При условной синхронизации поток блокируется в ожидании события, которое генерируется в другом потоке.

Сигнальную синхронизацию между потоками можно организовать самостоятельно или использовать уже готовые средства .NET.

К имеющимся в .NET средствам относятся три типа сигнальных сообщений, обеспечиваемые классами AutoResetEvent, ManualReset-Event и ManualResetEventSlim, а также шаблоны синхронизации, построенные на сигнальных сообщениях (CountdownEvent, Barrier).

4.3.1.Использование собственного события синхронизации потоков

Частным случаем проблемы синхронизации двух потоков является определение момента завершения работы потока. Для ожидания завершения работы потока можно использовать объектный метод Join() класса Thread. Иначе информировать ожидающий поток можно с помощью события.

Пусть имеем два потока: основной и вторичный. Для того чтобы информировать основной поток о завершении вторичного потока последним оператором вторичного потока будем выбрасывать событие о завершении обработки.

Для того чтобы основной поток реагировал на это событие необходимо определить обработчик этого события и организовать его прослушивание. Заметим, что обработку события можно выполнить в самом обработчике, а можно установить флаг о том, что событие произошло. К этому флагу основной поток может обращаться, когда это необходимо.

Рассмотренный принцип синхронизации резализуется в следующем примере программы. Программный модуль *ProgrModule.cs* включает определение типов, необходимых для организации вторичного потока:

```
using System;
using System.Threading;

namespace ThreadSynchroEvent
{
   // тип операция
   public enum Operation {nul, add, sub, mult, div, mod };

   // тип аргументы
   class Arguments
```

```
{
    Operation   oper;  // операция
    int             arg1;  //аргумент 1
    int             arg2;  // аргумент 2
    int             result; //результат

    public Arguments( Operation op, int a1, int a2)
    {
        if(op!=Operation.nul)
            oper = op;
        else
            throw (new Exception("Не задана операция!"));

        arg1 = a1;
        arg2 = a2;
    }

    // свойства доступа к аргументам
    public Operation Oper
    {
        get { return oper; }
    }

    public int Arg1
    {
        get { return arg1; }
    }

    public int Arg2
    {
        get { return arg2; }
    }

    public int Result
    {
        set { result = value; }
        get { return result; }
    }
}
 // класс программный модуль
class ProgrModule
```

```
{
    #region // методы операций
    // сложение
    public int Add(int par1, int par2)
    {
        return par1 + par2;
    }

    // вычитание
    public int Sub(int par1, int par2)
    {
        return par1 - par2;
    }

    // умножение
    public int Mult(int par1, int par2)
    {
        return par1 * par2;
    }

    // целая часть деления
    public int Div(int par1, int par2)
    {
        return par1 / par2;
    }
    // остаток деления
    public int Mod(int par1, int par2)
    {
        return par1 % par2;
    }
    #endregion
    // метод выполнить операцию
    public void Execute(object par)
    {
        Arguments args = (Arguments)par;
        switch (args.Oper)
        {
            case Operation.add: args.Result = Add(args.Arg1, args.Arg2);
                break;
            case Operation.sub: args.Result = Sub(args.Arg1, args.Arg2);
                break;
```

```
            case Operation.mult: args.Result = Mult(args.Arg1,
args.Arg2);
                break;
            case Operation.div: args.Result = Div(args.Arg1, args.Arg2);
                break;
            case Operation.mod: args.Result = Mod(args.Arg1,
args.Arg2);
                break;
        }
        Thread.Sleep(2000);

        // выбрасываем событие завершения вычислений
        if (EventEndExecution != null)
        EventEndExecution();
    }
    // организация события "завершение исполнения"

    // делегат для события EventEndExecution
    public delegate void EndExecution();

    // тип событие EventEndExecution
    public event EndExecution EventEndExecution;

    // свойство, которое определяет обработчик (слушателя)
события
    public EndExecution OnEndExecution
    {
        set { EventEndExecution += new EndExecution(value);  }
    }
  }
}
```

Программный модуль, реализующий класс формы формирует основной поток:

```
using System;
using System.Windows.Forms;
using System.Threading;

namespace ThreadSynchroEvent
{
  public partial class Form1 : Form
```

```
{
    public Form1()
    {
        InitializeComponent();

        // настройка консоли
        Console.Title = "ThreadSynchroEvent";
        Console.BackgroundColor = ConsoleColor.White;
        Console.ForegroundColor = ConsoleColor.Black;
        Console.WindowWidth = 90;
        Console.WindowHeight = 20;
        Console.Clear();

        // первый поток
        Thread firstThread = Thread.CurrentThread;
        firstThread.Name = "Поток1";
    }

    // флаг – обработка втоичным потоком завершена
    private bool isDone = false;

    // обработчики события завершенр вычисление

    // устанавливает флаг завершения обработки
    public void EndExecutionHandler()
    {
        isDone = true;
    }

    // выдаёт сообщение о завешении обработки на консоль
    public void EndExecutionHandler2()
    {
        Console.WriteLine("\n{0} : завершил исполнение",
                Thread.CurrentThread.Name);
    }

    // запускает вторичный поток на выполнение
    private void button1_Click(object sender, EventArgs e)
    {
        Console.WriteLine("\n{0} выполняет Main() метод",
Thread.CurrentThread.Name);
```

```
        #region  // шаг 0. Готовим аргументы для метода

        Arguments args = null;  // объект для аргументов метода

        bool error;

        string str_operator;    // исходный оператор
        do
        {
            error = false;
            Console.WriteLine("\nЗадайте оператор для выполнения
арг1 опер арг2 : ");
            str_operator = Console.ReadLine();
            try
            {
                string[] str_args = str_operator.Split(' ');

                Operation oper = Operation.nul;
                switch (str_args[1])
                {
                    case "+": oper = Operation.add;
                        break;
                    case "-": oper = Operation.sub;
                        break;
                    case "*": oper = Operation.mult;
                        break;
                    case "/": oper = Operation.div;
                        break;
                    case "%": oper = Operation.mod;
                        break;
                }

                args = new Arguments(oper, int.Parse(str_args[0]),
                                           int.Parse(str_args[2]));
            }
            catch
            {
                Console.WriteLine("Error: Недопустимый оператор! По-
пробуйте заново...");
                error=true;
```

```
            }
        }
        while (error);
        #endregion

        ProgrModule myProgrModule = new ProgrModule();

        // установка обработчика события EndExecution в ProgrModule
        myProgrModule.OnEndExecution = new ProgrMod-
ule.EndExecution(EndExecutionHandler);
        myProgrModule.OnEndExecution = new ProgrMod-
ule.EndExecution(EndExecutionHandler2);

        isDone = false;

        // шаг 2. Создать делегат-метод  для выполнения в отдель-
        // ном потоке
        ParameterizedThreadStart method_for_thread = new Parameter-
izedThreadStart(myProgrModule.Execute);

        // шаг 3. Создать объект-поток для выполнения метода
        Thread secondThread = new Thread(method_for_thread);

        // шаг 4. Определить характеристики потока
        secondThread.Name = "Поток2";

        // шаг 5. Запуск потока на выполнение с аргументами
        secondThread.Start(args);

        while (!isDone)
        {
            Console.WriteLine("Ожидаем завершения вычислений...");
            Thread.Sleep(1000);
        }

        Console.WriteLine("Вычисление завершено: {0} = {1}",
str_operator, args.Result);
      }
   }
}
```

Результат выполнения представлен на рис. 4.10.

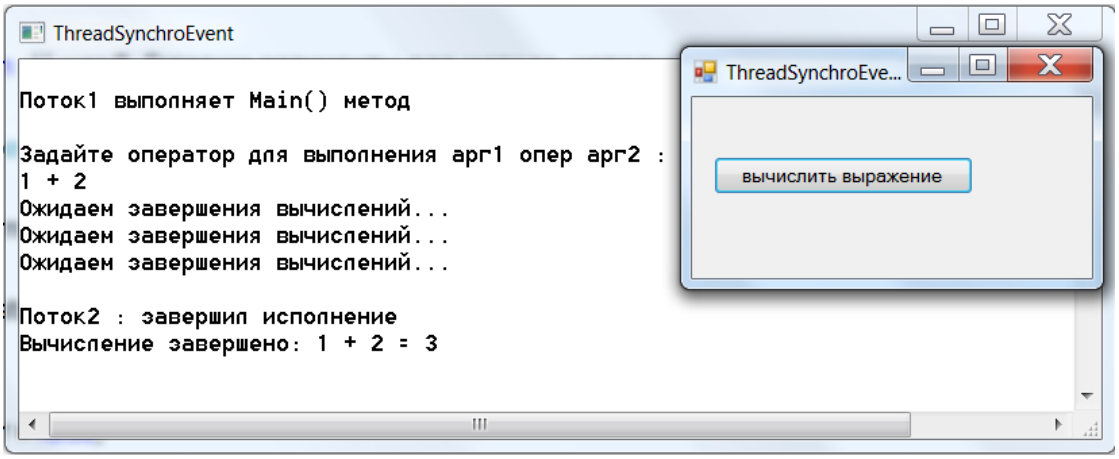


Рис. 4.10. Использование события для сигнализации о завершении потока

Как видно из листинга использованое события позволяет сигнализировать о завершении работы потока.

Обратите внимание так же на то, что обработчики события выполняются во вторичном потоке, о чём свидетельствует строка на экране: «Поток2: завершил исполнение». Т.е. налицо ассинхронное взаимодействие потоков – Поток2 выполняет вычисления параллельно основному потоку и меняет состояние флага *isDone*, а Поток1 ожидает его завершения проверяя состояние этого флага.

4.3.2.Использование собственных сигнальных сообщений

Предположим, что нам необходимо вычислить болше одного выражения и каждое вычисление должно производиться в своём потоке. В этом случае проблему синхронизации потоков можно было бы решить уже рассмотренными средствами.

Для этой задачи предложим другой способ, основанный на принципе сигнальных сообщений.

Ниже приводится текст приложения с использованием сигнальной синхронизации, а на рис. 4.11 Листинг его работы для 4 потоков:

1) Модуль *ProgrModule.cs:*

```
using System;
using System.Threading;

namespace ThreadSynchroEvent
```

```
{
   // тип операция
   public enum Operation {nul, add, sub, mult, div, mod };

   // тип аргументы
   class Arguments
   {
      //public AutoResetEvent waitHandle = new AutoResetEvent(false);

      Operation   oper;  // операция
      int              arg1;  //аргумент 1
      int              arg2;  // аргумент 2
      int              result; //результат

      MySynchroEvent freeEvent;   // объект синхронизации

      public Arguments( Operation op, int a1, int a2,
                        MySynchroEvent synObject)
      {
         if(op!=Operation.nul)
            oper = op;
         else
            throw (new Exception("Не задана операция!"));

         arg1 = a1;
         arg2 = a2;

         freeEvent = synObject;
      }

      // свойства доступа к аргументам
      public Operation Oper
      {
         get { return oper; }
      }

      public int Arg1
      {
         get { return arg1; }
      }
```

```
    public int Arg2
    {
        get { return arg2; }
    }

    public int Result
    {
        set { result = value; }
        get { return result; }
    }

    public MySynchroEvent FreeEvent
    {
        get { return freeEvent; }
    }
}

// класс программный модуль
class ProgrModule
{
    #region // методы операций
    // сложение
    public int Add(int par1, int par2)
    {
        return par1 + par2;
    }

    // вычитание
    public int Sub(int par1, int par2)
    {
        return par1 - par2;
    }

    // умножение
    public int Mult(int par1, int par2)
    {
        return par1 * par2;
    }

    // целая часть деления
    public int Div(int par1, int par2)
```

```
        {
            return par1 / par2;
        }

        // остаток деления
        public int Mod(int par1, int par2)
        {
            return par1 % par2;
        }
        #endregion

        // метод выполнить операцию
        public void Execute(object par)
        {
            // получаем аргументы, выполняем , записываем рез-тат
            Arguments args = (Arguments)par;

            Console.WriteLine("\n{0} ожидает освобождения вычисли-
тельного ресурса", Thread.CurrentThread.Name);

            args.FreeEvent.WaitOne();

            Console.WriteLine("\n{0} получил вычислительный ресурс",
Thread.CurrentThread.Name);

            switch (args.Oper)
            {
                case Operation.add: args.Result = Add(args.Arg1, args.Arg2);
                    break;
                case Operation.sub: args.Result = Sub(args.Arg1, args.Arg2);
                    break;
                case Operation.mult: args.Result = Mult(args.Arg1,
args.Arg2);
                    break;
                case Operation.div: args.Result = Div(args.Arg1, args.Arg2);
                    break;
                case Operation.mod: args.Result = Mod(args.Arg1,
args.Arg2);
                    break;
            }
            Thread.Sleep(1*(new Random()).Next(3));
```

```
            // выбрасываем событие завершения вычислений
            if (EventEndExecution != null)
            EventEndExecution();

            // установить состояние освобождения вычислительного
ресурса
            args.FreeEvent.Set();
        }

        // организация события "завершение исполнения"
        // делегат для события EventEndExecution
        public delegate void EndExecution();

        // тип событие EventEndExecution
        public event EndExecution EventEndExecution;

        // свойство
        public EndExecution OnEndExecution
        {
            set { EventEndExecution += new EndExecution(value);  }
        }
    }
}
```

2) Модуль *Form1.cs:*

```
using System;
using System.Windows.Forms;
using System.Threading;

namespace ThreadSynchroEvent
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            // настройка консоли
            Console.Title = "ThreadSynchroEvent";
```

```
            Console.BackgroundColor = ConsoleColor.White;
            Console.ForegroundColor = ConsoleColor.Black;
            Console.WindowWidth = 90;
            Console.WindowHeight = 20;
            Console.Clear();

            // первый поток
            Thread firstThread = Thread.CurrentThread;
            firstThread.Name = "Основной_поток";
        }

        private int thread_counter;

        // обработчики события завершенр вычисление
        public void EndExecutionHandler()
        {
            thread_counter --;
        }

        public void EndExecutionHandler2()
        {
            Console.WriteLine("\n{0} : завершил исполнение",
Thread.CurrentThread.Name);
        }

        // запускает поток
        private void button1_Click(object sender, EventArgs e)
        {
            Console.WriteLine("\n{0} выполняет Main() метод",
Thread.CurrentThread.Name);

            int n = (int)numericUpDown1.Value;

            Thread[] t = new Thread[n];    // массив потоков

            // объекты для аргументов методов потока
            Arguments[] args = new Arguments[n];
            MySynchroEvent  synObject=new MySynchroEvent(true);

            ProgrModule myProgrModule = new ProgrModule();
```

```
        for (int i = 0; i < n; i++)
        {

            #region  // шаг 0. Готовим аргументы для метода

            args[i] = null;  // объект для аргументов метода

            bool error;

            string str_operator;    // исходный оператор
            do
            {
                error = false;
                Console.WriteLine("\nЗадайте оператор для выполне-
ния арг1 опер арг2 : ");
                str_operator = Console.ReadLine();
                try
                {
                    string[] str_args = str_operator.Split(' ');

                    Operation oper = Operation.nul;
                    switch (str_args[1])
                    {
                        case "+": oper = Operation.add;
                            break;
                        case "-": oper = Operation.sub;
                            break;
                        case "*": oper = Operation.mult;
                            break;
                        case "/": oper = Operation.div;
                            break;
                        case "%": oper = Operation.mod;
                            break;
                    }

                    args[i] = new Arguments(oper, int.Parse(str_args[0]),
                                            int.Parse(str_args[2]),synObject);
                }
                catch
                {
```

```
                    Console.WriteLine("Error: Недопустимый оператор!
Попробуйте заново...");
                    error = true;
                }
            }
            while (error);
            #endregion

            // шаг 2. Создать делегат-метод
            ParameterizedThreadStart method_for_thread =
                new ParameterizedThreadStart(myProgrModule.Execute);

            // шаг 3. Создать объект-поток для выполнения метода
            t[i] = new Thread(method_for_thread);

            // шаг 4. Определить характеристики потока
            t[i].Name = "Поток"+i.ToString();
        }
        // установка обработчика события EndExecution в  ProgrModule
        myProgrModule.OnEndExecution =
                new ProgrModule.EndExecution(EndExecutionHandler);
        myProgrModule.OnEndExecution =
                new ProgrModule.EndExecution(EndExecutionHandler2);

        thread_counter = n;

        // шаг 5. Запуск потока на выполнение с аргументами
        for(int i=0; i<n; i++)
            t[i].Start(args[i]);

        Thread.Sleep(n*5);
        while (thread_counter!=0)
            Console.WriteLine("Ожидаем завершения вычислений...");

        Console.WriteLine("{0} : Вычисление завершено! Результа-
ты: ", Thread.CurrentThread.Name);

        for (int i = 0; i < n; i++)
            Console.WriteLine("Поток{0} : {1} {2} {3} = {4}", i, args[i].Arg1
, args[i].Oper, args[i].Arg2, args[i].Result);
```

```
        }
    }
}
```

3) Модуль *MySynchroEvent.cs*:

```
using System;
using System.Threading;

namespace ThreadSynchroEvent
{
    class MySynchroEvent
    {
        private bool opened;

        public MySynchroEvent(bool state=false)
        {
            opened = state;
        }

        public void Set()
        {
            opened = true;
        }

        public void WaitOne()
        {
            while (!opened) ;
            opened = false;
        }
    }
}
```

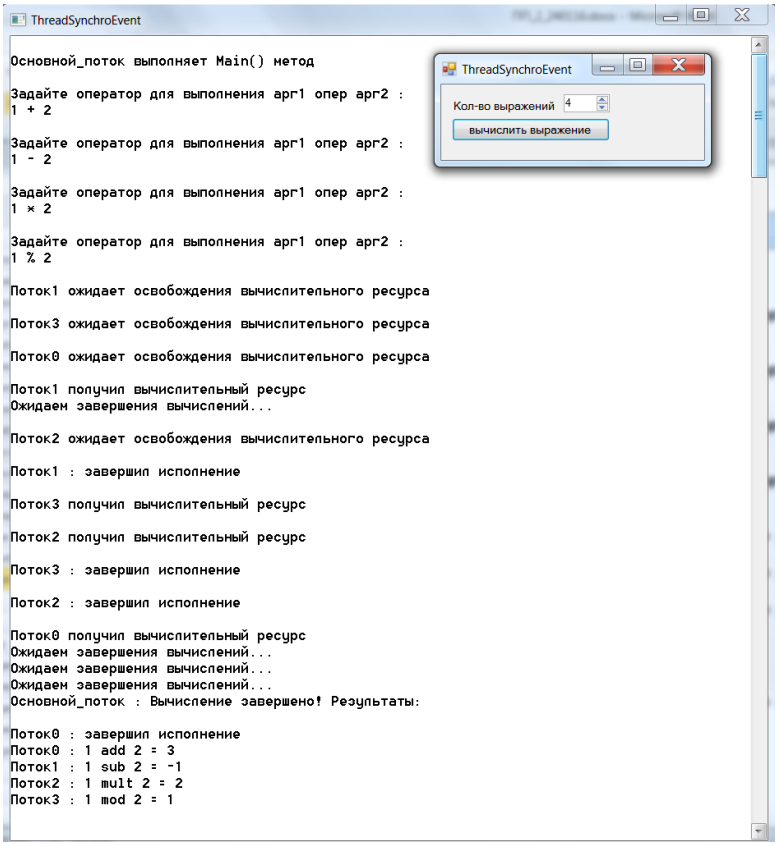


Рис. 4.11. Использование сигнальных сообщений

В классе *MySynchroEvent* имеется переменная-флаг *opened* значение true, которой говорит, что разделяемый ресурс свободен, а false – занят. В нём также имеются два метода:

void WaitOne() – ожидает освобождение разделяемого ресурса и возвращает управление, если освободился;

void Set() – устанавливает флаг, что разделяемый ресурс освобождён;

void Close() – освобождает все ресурсы, удерживаемые объектом синхронизации.

Смысл такой сигнализации состоит в том, что конкурирующие за обладание разделяемым ресурсом потоки ождидают и получают его методом *WaitOne()*, а когда он не нужен, сигнализируют установкой сигнала методом *Set()*.

Как видно из рис.4.11 использованный приём сигнальной синхронизации решил возложенную на него проблему.

Также заслуживающим внимание в этом примере является механизм ожидания завершения потоков. Для его реализации используется счетчик незавршенных потоков *thread_counter*, который в началь-

ном состоянии хранит число запущенных потоков и уменьшается на 1 с каждым завершившимся потоком. Нулевое значение этого счётчика говорит о том, что все потоки завершены.

4.3.3.Средства сигнальных сообщений .NET

К имеющимся в .NET средствам относятся три типа сигнальных сообщений, обеспечиваемые классами *AutoResetEvent*, *ManualReset-Event* и *ManualResetEventSlim*, а также шаблоны синхронизации, построенные на сигнальных сообщениях (*CountdownEvent*, *Barrier*).

Первые два типа построены на объекте ядра операционной системы. Третий тип *ManualResetEventSlim* является облегченной версией объекта *ManualResetEvent* и более производительный.

Классы *AutoResetEvent* и *ManualResetEvent* являются производными от класса *EventWaitHandle* и не имеют никакого отношения к делегатам и событиям C#. Обоим классам доступны все функциональные возможности базового класса, единственное отличие состоит в вызове конструктора базового класса с разными параметрами.

Класс *EventWaitHandle* позволяет потокам взаимодействовать друг с другом путем передачи сигналов подобно рассмотренному способу в п.4.3.2. Обычно один или несколько потоков блокируются на *EventWaitHandle* до тех пор пока незаблокированные потоки не вызывают метод *Set()* для освобождения одного или нескольких заблокированных потоков.

Если в примере в п.4.3.2 наш класс *MySynchroEvent можно* заменить на класс *AutoResetEvent,* то результат работы приложения не изменится. Но *AutoResetEvent* имеет больше возможностей. В его состав входят такие методы:

bool WaitOne() – блокирует текущий поток до получения сигнала синхронизации. Возвращает значение true, если сигнал получен. Метод WaitOne не возвращает управление, пока не будет получен сигнал синхронизации;

bool WaitOne(int millisecondsTimeout) – блокирует текущий поток не больше чем на время *millisecondsTimeout* до получения сигнала синхронизации. Возвращает значение true, если сигнал получен, иначе false;

bool Set() – устанавливает (отправляет) сигнал синхронниззации;

bool Reset() – сбрасывает сигнал синхронизации.

Отличия инструментов *AutoResetEvent* и *ManualResetEvent* заключаются в режиме сброса сигнала синхронизации: автоматический (auto) или ручной (manual).

Сигнал с автоматическим сбросом сбрасывается сразу же после того, как *WaitOne()* возвращает управление в заблокированный поток.

Сигнал с ручным сбросом не снимается до тех пор, пока какой-либо поток не вызовет метод *Reset().*

Поэтому *AutoResetEvent* реализует взаимоисключительный доступ, а *ManualResetEvent* управляемый пользователем обмен сигналами.

В этом можно убедиться, если в примере из п.4.3.3 заменить класс *MySynchroEvent* на *ManualResetEvent.* В этом случае не будет обеспечен взаимоисключительный доступ к вычислительному ресурсу.

Или убедитесь в этом на следующем примере:

```
using System;
using System.Threading;

namespace AutoResetEv
{
   class Program
   {
      public Program()
      {
         Console.Title = "AutoManual";
         Console.BackgroundColor = ConsoleColor.White;
         Console.ForegroundColor = ConsoleColor.Black;
         Console.WindowWidth = 90;
         Console.WindowHeight = 20;
         Console.Clear();
      }
      void Worker(object initWorker)
      {
         string name = ((object[])initWorker)[0] as string;
         //ManualResetEvent mre = ((object[])initWorker)[1] as Manu-
alResetEvent;

         AutoResetEvent mre = ((object[])initWorker)[1] as AutoRe-
setEvent;

         // Waiting to start work
```

```
            mre.WaitOne();


            Console.WriteLine("Worker {0} starts ..", name);

            // useful work

        }

        public void Manager()
        {
            int nWorkers = 5;
            Thread[] worker = new Thread[nWorkers];
            //ManualResetEvent mre = new ManualResetEvent(false);

            AutoResetEvent mre = new AutoResetEvent(false);

            for (int i = 0; i < nWorkers; i++)
            {
                worker[i] = new Thread(Worker);


                worker[i].Start(new object[] { "#" + i, mre });
            }
            // preparing data in shared variables for workers
            // let start work
            mre.Set();
        }

        static void Main(string[] args)
        {
            Program p = new Program();
            p.Manager();
        }
    }
}
```

Результат работы при использовании класса *AutoResetEvent* приведен на рис.4.12. Как видно завершился только один поток, первый захвативший ресурс. Другим потокам это не удалось сделать, поскольку метод *WaitOne()* класса *AutoResetEvent* автоматически сбросил сигнал синхронизации.

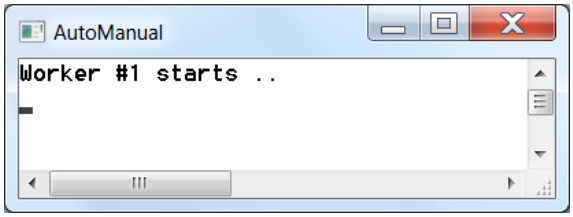


Рис. 4.12. Результат при использовании класса *AutoResetEvent*

Результат использования класса *ManualResetEvent* представлен на рис. 4.13. Как видно, поскольку метод *WaitOne()* класса *ManualResetEvent* автоматически не сбросил сигнал синхронизации, то все потоки завершились.

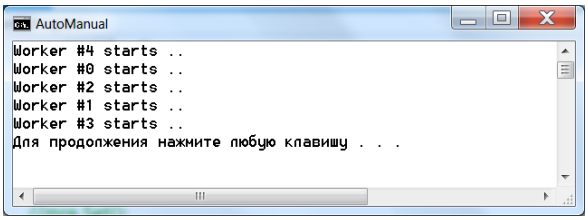


Рис. 4.13. Результат при использовании класса *ManualResetEvent*

Объект *ManualResetEventSlim* функционально соответствует сигнальному событию с ручным сбросом. Применение гибридной блокировки повышает производительность в сценариях с малым временем ожидания. Вызов метода *Wait()* в течение ограниченного промежутка времени сохраняет поток в активном состоянии (циклическая проверка статуса сигнала), если сигнал не поступил, то осуществляется вызов дескриптора ожидания ядра операционной системы.

*4.3.4.*Класс *CountdownEvent*

Класс *CountdownEvent* обеспечивает примитив синхронизации, разблокирующий ожидающие потоки после получения определенного числа сигналов.

Например, в схеме (см. рис.4.14) с разветвлением (точка A) и соединением (точка B) можно просто создать CountdownEvent со счетчиком сигналов, равным количеству параллельных потоков n, в каждом из которых вызывать после завершения работы метод Signal этого класса. Каждый вызов метода Signal уменьшает значение счетчика на 1. В главном потоке вызов метода Wait приведет к блокировке, ко-

торая будет длиться до тех пор, пока значение счетчика не будет равно нулю.

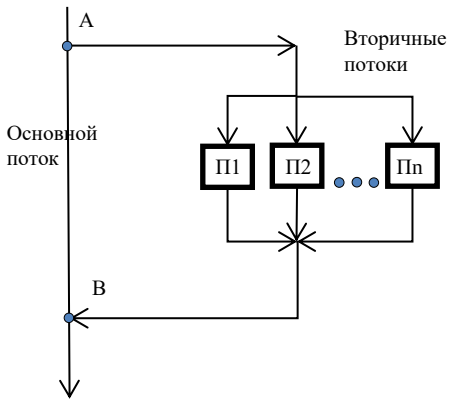


Рис. 4.14. Запуск и ожидания параллельно исполняемых потоков

Похожий приём мы использовали в примере в п.4.3.2, где использовали считчик *thread_counter,* который уменьшался по событию – поток завершился. При этом основной поток находился в цикле ожидания, пока этот счетчик не станет равным 0.

С помощью класса *CountdownEvent* всё это можно сделать проще, при этом этот класс не требует использования событий.

В табл. 4.2 приведены свойства этого класса, а в табл. 4.3 описаны основные методы необходимые для решения рассматриваемой проблемы синхронизации потоков.

В качестве иллюстрации приводим текст приложения, в котором основной поток запускает пять вторичных потоков, на рис.4.15 представлен результат работы.

```
using System;
using System.Threading;

namespace CountDownEv
{
   class Program
   {
      static CountdownEvent allEnd;// событие все потоки завершены

      public static void Met(object ob)
      {
         int num = (int)ob;

          Console.WriteLine("Поток{0} : allEnd.CurrentCount={1} ",
```

```
                                num, allEnd.CurrentCount);

            Thread.Sleep(num*(new Random()).Next(5));
            allEnd.Signal();        // сигнализируем о завершении потока
        }

        static void Main(string[] args)
        {
            int n = 5;  // количество потоков

            allEnd = new CountdownEvent(n); // создание объекта события

            for (int i = 0; i < n; i++)
                (new Thread(Met)).Start(i);     // запуск вторичных потоков

            allEnd.Wait(); // ожидание события

            Console.WriteLine("Все потоки завершены!");
        }
    }
}
```

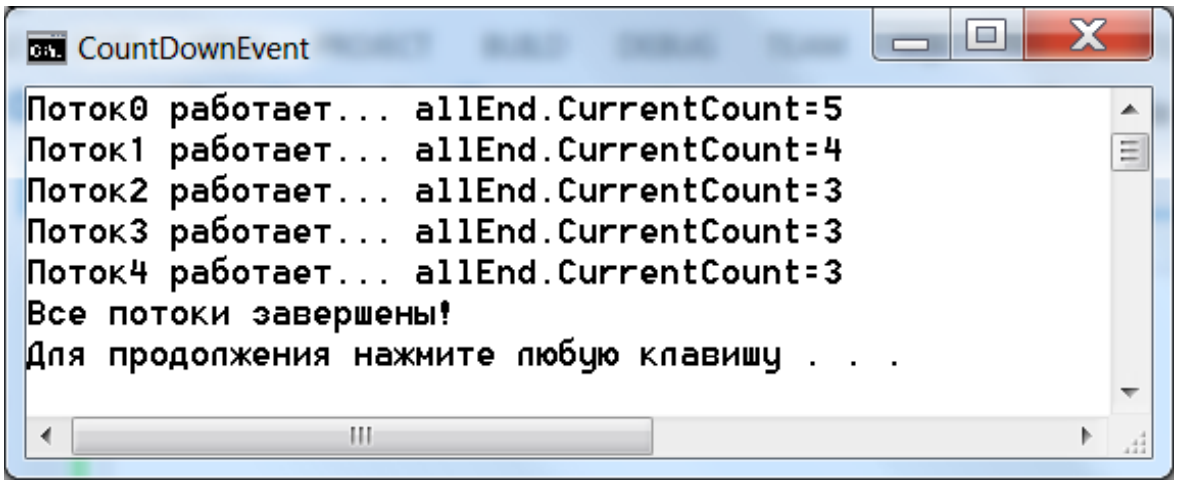


Рис. 4.15. Результат работы

Таблица 4.2 – Свойства класса *CountdownEvent*

Свойство	Описание
CurrentCount	*public int CurrentCount { get; }*
	Получает количество сигналов, оставшееся до установки события
InitialCount	*public int InitialCount { get; }*
	Получает количество сигналов, изначально нужное для установки события.
IsSet	*public bool IsSet { get; }*
	Определяет, установлено ли событие (значение true).
WaitHandle	*public WaitHandle WaitHandle { get; }*
	Получает дескриптор WaitHandle, используемый для ожидания установки события.

Таблица 4.3 – Методы класса *CountdownEvent*

Метод	Описание
CountdownEvent()	*public CountdownEvent(int initialCount)*
	Инициализирует объект класса *CountdownEvent* значением *initialCount*, соответствующим количеству ожидаемых сигналов (потоков).
bool Signal()	*public bool Signal()*
	регистрирует сигнал, уменьшая значение свойства *CurrentCount*. Возвращает true, если *CurrentCount* стал равен нулю, т.е. было создано событие; в противном случае – значение false.
AddCount()	*public void AddCount()*
	Увеличивает *CurrentCount* на один.
void Reset()	*public void Reset()*
	Сбрасывает свойство *CurrentCount* на значение свойства *InitialCount*
void Wait()	*public void Wait()*
	Блокирует текущий поток до установки события *CountdownEvent* (*CurrentCount=0*).
	public bool Wait(int millisecondsTimeout)
	Ожидает события *CountdownEvent* не более *millisecondsTimeout*. Возвращает значение true, если установлено событие, иначе – false.

4.4. Семафоры

Семафоры бывают двух типов: локальные семафоры и именованные системные семафоры. Семафоры первого типа используются в рамках одного приложения. Семафоры второго типа видимы для операционной системы и пригодны для синхронизации процессов.

SemaphoreSlim это упрощенная альтернатива *Semaphore* класс, который не использует ядро Windows. В отличие от *Semaphore SemaphoreSlim* не поддерживает именованные системные семафоры.

Семафоры используется для ограничения число потоков, которые могут одновременно получать доступ к ресурсу. Объект синхронизации *Semaphore* (*SemaphoreSlim*) отличается от сигнальных событий наличием внутреннего счетчика, устанавливаемого заданным максимальным значением, который ограничивает количество потоков для использования ресурса. Поэтому, объект *AutoResetEvent* можно интерпретировать как семафор с максимальным счетчиком равным 1 (двоичный семафор).

Счетчик семафора уменьшается на единицу каждый раз поток входит в семафор и увеличивается на единицу, когда поток освобождает семафор. Если счетчик равен нулю, последующие запросы блокируются, пока другие потоки не освободят семафор. Когда семафор освобожден всеми потоками, счетчик, имеет максимальное значение, заданное при создании семафора.

В следующем примере рассматривается применение семафоров. В коде используется объект *SemaphoreSlim* (результат на рис. 4.16). Вместо него можно использовать объект *Semaphore*.

```
using System;
using System.Threading;

namespace Semaphor
{
   class Program
   {

      private static SemaphoreSlim sem;

      private static void Met(object num)
      {
```

```
            sem.Wait(); // ждём сигнала от семафора

            // Начинаем работу
            Console.WriteLine("Поток{0} : работает  sem={1}",
                              num, sem.CurrentCount);

            Thread.Sleep((int)num * (new Random()).Next(1000));

            sem.Release();  // освобождаем ресурс

            Console.WriteLine("Поток{0} : закончил работу sem={1}",
                              num, sem.CurrentCount);
        }

        static void Main(string[] args)
        {
          // Максимальная емкость семафора: 5
          // Начальное состояние: 0 (все блокируются)
            sem = new SemaphoreSlim(0, 5);
            for (int i = 0; i < 10; i++)
                (new Thread(Met)).Start(i);      // запуск вторичных потоков

            Thread.Sleep(50);

            Console.WriteLine("Разрешаем работу двум потокам");
            sem.Release(2);

            Thread.Sleep(50);

            Console.WriteLine("Разрешаем работу еще двум потокам");
            sem.Release(2);
        }
    }
}
```

При создании объекта *SemaphoreSlim* ему указываются начальное и максимальное количество разблокированных сигналов (потоков).

Начальное значение определяет количество потоков *CurrentCount*, которым разрешено входить в объект SemaphoreSlim.

Увеличить это количество можно действием разблокирования методом *Release()*.

Поток, который использует ограниченный ресурс, должен вызывать метод *Wait()*. Этот метод блокирует текущий поток, пока не сможет войти в *SemaphoreSlim* (пока количество разблокированных сигналов равно 0). Если поток может войти в *SemaphoreSlim*, то метод *Wait()* уменьшает количество разблокированных сигналов на 1.

При выходе из потока необходимо разблокировать сигнал методом *Release()*.

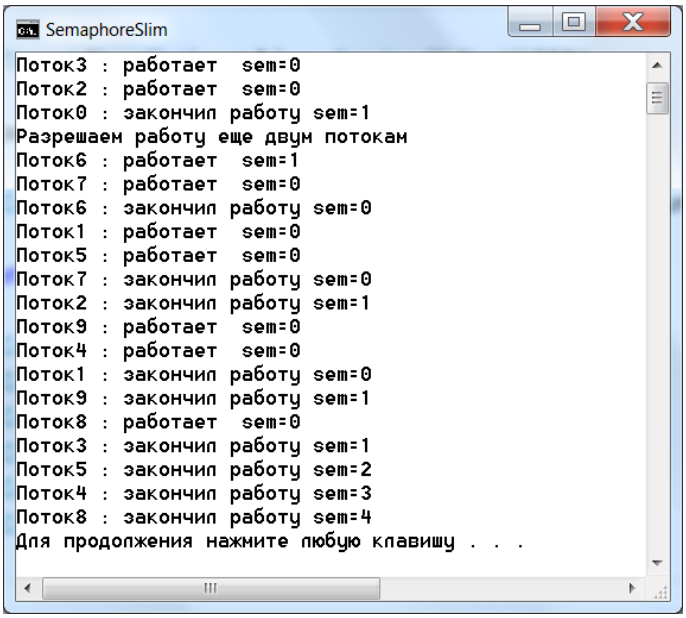


Рис. 4.16. Результат работы

4.5. Атомарные операторы

Простые арифметические операции в .Net не являются атомарными, т.е. при их выполнении над общими данными в разных потоках может наблюдаться проблема гонки данных.

Решать эту проблему средствами lock или Monitor является накладным по затратам. Для решения этой проблемы предусмотрен специальный статический класс *Interlocked* (см. табл.4.4), который содержит атомарные операторы, предназначеные для потокобезопасного неблокирующего выполнения операций над данными, преимущественно целочисленного типа. Атомарность операторов класса *Interlocked* означает, что при выполнении оператора никто не сможет использовать задействованные оператором данные.

Функционально, атомарные операторы равносильны критической секции, выделенной с помощью lock, *Monitor* или других средств синхронизации.

В отличие от указанных средства атомарные операторы являются неблокирующими , это значит, что – поток не выгружается и не ожидает, поэтому обеспечивают высокую эффективность. Выполнение оператора *Interlocked* занимает вдвое меньшее время, чем выполнение критической секции с lock-блокировкой.

Таблица 4.4 – Операторы класса *Interlocked*

Метод \(оператор)	Описание
Increment	*int Increment(ref int location)* *long Increment(ref long location)*
	Увеличивает значение заданной переменной *location* на 1
Decrement	*int Decrement(ref int location)* *long Decrement(ref long location)*
	Уменьшает значение заданной переменной *location* на 1
Add	*int Add(ref int location,int value)* *long Add(ref long location,long value)*
	Увеличивает значение заданной переменной *location* на величину *value*
Exchange	*int Exchange(ref int location, int value)* *double Exchange(ref double location1,double value)* *long Exchange(ref long location1, long value)* *object Exchange(ref object location1,object value)* *float Exchange(ref float location1,float value)* *T Exchange<T>(ref T location1,T value)where T: class*
	Присваивает заданной переменной *location* значение *value* и возвращает исходное значение
CompareExchange	*double CompareExchange(ref double location, double value, double comparand)* (поддерживает все варианты перегрузки как и метод *Exchange*)
	Присваивает заданной переменной *location* значение *value* и возвращает исходное значение, если выполняется отношение: *location == comparand*
Read	*long Read(ref long location)*
	Возвращает значение заданной переменной *location*

Например, вызов метода:

```
Interlocked.Increment(ref counter);
```

эквивалентен:

```
lock (sync_obj)
{
  counter++;
}
```

4.6. Блокировки чтения-записи

Классы *ReaderWriterLock* и *ReaderWriterLockSlim* обеспечивают блокировку потоков для управления доступом к ресурсу, которая позволяет нескольким потокам производить считывание или получать монопольный доступ на запись. Эту модель блокировки называют «читатели-писатели».

Из двух указанных классов рекомендуется использовать класс *ReaderWriterLockSlim*, который позволяет выделить три группы пользователей ресурса – читателей (Read), писателей (Write) и редакторов (Upgradeable), соответственно, читающих, пишущих и редактирующих ресурс.

Для каждой группы класс предлагает свой метод блокировки:

1) void EnterReadLock()

Если при выполнении этого метода есть очередь потоков со статусом Write, то поток становится в очередь потоков со статусом Read. Потоки из этой очереди смогут начать выполняться, только когда нет ждущих "писателей". Если очередь состоит только из потоков со статусом Upgradeable, то поток входит в критическую секцию, поскольку разрешается в критической секции одновременное присутствие потоков со статусом Read и одного потока со статусом Upgradeable.

2) void EnterWriteLock()

Если при выполнении этого метода есть очередь потоков со статусом Write, то поток становится в конец этой очереди. Если же очереди нет, то поток дожидается завершения работы потоков, находящихся в критиче-

ской секции. После чего поток входит в критическую секцию и единолично работает с ресурсом.

3) void EnterUpgradeableReadLock()

Если при выполнении этого метода есть очередь потоков со статусом Write, то поток становится в очередь потоков со статусом Upgradeable. Если последняя очередь пуста, то поток будет первым в этой очереди. Первый поток из этой очереди может начать свою работу, когда очередь потоков со статусом Write пуста.

Для разблокирования критической секции применяют три соответствующих метода:

1) void ExitReadLock(),

2) void ExitWriteLock(),

3) void ExitUpgradeableReadLock().

Общая схема организации критической секции выглядит так:

```
Enter-метод
try
{
   <операторы критической секции, работающие с ресурсом>
}
finally
{
   Exit-метод
}
```

Блок finally всегда будет выполняться, что гарантирует освобождение ресурса.

Поскольку из-за блокировки совместно используемых ресурсов можно войти в ситуацию клинча, то часто применяют более мягкую схему блокировки, заменяя Enter-методы их аналогами:

1) bool TryEnterReadLock(int timewait),.

2) bool TryEnterWriteLock(int timewait),

3) *bool EnterUpgradeableReadLock(int timewait).*

При использовании таких методов поток либо войдет в критическую секцию и начнет выполняться, возвращая в качестве результата значение true, либо выйдет из очереди по истечении времени ожидания wait, возвращая false. В любом случае зависание потока исключается.

Кроме рассмотренных возможностей класс *ReaderWriterLockSlim* имеет много полезных свойств, позволяющих, например, анализировать состояние очередей. В частности, есть группа свойств Waiting, позволяющих узнать число потоков, ожидающих блокировку в очередях каждого типа:

1) int WaitingReadCount { get; },

2) int WaitingWriteCount { get; },

3) int WaitingUpgradeCount { get; }.

Класс *ReaderWriterLockSlim* позволяет использовать рекурсивную блокировку, но для новых разработок использовать рекурсии не рекомендуется, поскольку она приводит к ненужным усложнениям и повышает вероятность возникновения взаимоблокировок в коде.

Использование рассмотренной модели и использование класса *ReaderWriterLockSlim* рассмотрим на примере организации хранения паролей приложения.

Пусть пароли изначально храняться в файле на диске с именем, например, *"d:\psw.txt"* (рис. 4.17) так, что пароль pas1 соответствует пользователю 1, pas1 – пользователю 2 и т.д. и могут меняться за пределами приложения.

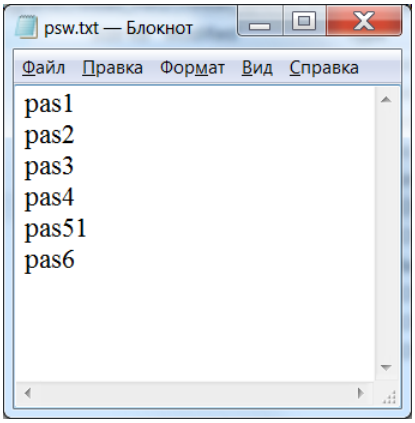


Рис. 4.17. Начальноре состояние файла паролей

Промоделируем механизм хранения паролей в памяти и контроля паролей при ассинхронном доступе к приложению пользователей на примере разработки специального кэша хранения паролей в памяти, построенного с использованием модели чтения-записи.

Приложение может одновременно обслуживать разное количество пользователей, которые могут входить (если уже зарегистрированы) или регистрироваться в системе. Пароли храняться в специальном кэше-словаре как пара <ключ, пароль>, в которой ключ – это номер пользователя, а пароль – это некоторая последовательность символов. Назначение механизма хранения и контроля паролей – обеспечить контроль ассинхронного доступа пользователей к приложению с возможность динамического изменени паролей извне приложения (изменение файла паролей) и со стороны приложения (при регистрации новых пользователей или изменения паролей).

Базовым классом рассматриваемого механизма является класс *SynchronizedCache,* который организует хранилище паролей в объекте словаре *innerCache* типа *Dictionary<int, string>*.

Этот класс обеспечивает такие методы:

public string Read(int key) – получения пароля прользователя с номером *key;*

public void Add(int key, string value) – добавления нового пользователя с номером *key* и паролем *value;*

public bool AddWithTimeout(int key, string value, int timeout) – добавления нового пользователя с номером *key* и паролем *value* с интервалом ожидания *timeout.* Если пользователь добавлен в кэш, то метод возвращает true*;*

public AddOrUpdateStatus AddOrUpdate(int key, string value) – добавления нового пользователя с номером *key* и паролем *value* или обновления пароля пользователя (если пользователь существует). Метод возвращает статус выполненной операции как значение типа *AddOrUpdateStatus;*

public void Delete(int key) – удаления пользователя с номером *key.*

Работой приложения допускается, что действия входа пользователей, регистрации или изменения паролей могут выполняться ассинхронно, а значит объект класса *SynchronizedCache* является объектом, за который конкурируют параллельные потоки при выполнении этих операций.

Для синхронизации доступа к объекту *innerCache* использована модель блокировки чтения-записи, обеспечиваемая объектом *cacheLock* класса *ReaderWriterLockSlim,* что видно из реализации методов.

Класс *SynchronizedCache* организует само хранилище паролей и методы чтения и изменения его состояния. На его основе построен класс *PSWCache*, который под-

держивает работу с файлом паролей и организует вход и регистрацию пользователей методами:

public bool LoadPSW() – выполняет загрузку паролей из файла паролей в хранилище паролей;

public void SavePSW() – выполняет сохранение паролей из хранилища в файле паролей;

public int AddNewPersonWithTimeOut(string value, int timeout) - добавления нового пользователя с паролем *value* с интервалом ожидания *timeout.* Если пользователь добавлен, то метод возвращает номер пользователя.

Для возможности реагирования на изменение файла паролей извне приложения в состав класса *PSWCache* включен объект *psw_watcher* класса *FileSystemWatcher,* с использованием которого организовано прослушивание события изменения состояния файла паролей. Обработка этого события осуществляется методом

private void OnPSW_Changed(object source, FileSystemEventArgs e),

который изменяет пароли в хранилище паролей в соответствии с состоянием файла паролей.

Тестирование разработанного хранилища паролей организовано в методе *Main()* класса *Program,* в котором запускаются на исполнение параллельные потоки моделирующие вход и регистрацию пользователей в приложение на основе методов:

public static void Coming(object numpsw) – имитирует вход пользователя в приложение. Предполагается, что в метод через объект массив передаётся описание пользователя: *numpsw[0]* хранит номер пользователя, а *numpsw[1]* – пароль*;*

public static void Register(object newpsw) – имитирует регистрацию нового пользователя с паролем newpsw.

Ниже приводится полный код описанного приложения, а результат его тестирования представлен скриншотом на рис. 4.18.

```
// Модуль SynchronizedCache.cs

using System;
using System.Collections.Generic;
using System.Threading;

namespace RWLSlim
{
   class SynchronizedCache
   {
      protected ReaderWriterLockSlim cacheLock =
```

```
        new ReaderWriterLockSlim(); // объект синхронизации кэша
protected Dictionary<int, string> innerCache =
        new Dictionary<int, string>();  // кэш

public int Count    // число записей в кэше
{
   get { return innerCache.Count; }
}

// метод чтения записи по ключу
public string Read(int key)
{
   // получаем блокировку кэша для чтения
   cacheLock.EnterReadLock();
   try
   {
      return innerCache[key];
   }
   finally
   {
      // снимаем блокировку кэша для чтения
      cacheLock.ExitReadLock();
   }
}

// метод добавления записи в кэше
public void Add(int key, string value)
{
   // получаем блокировку кэша для записи
   cacheLock.EnterWriteLock();
   try
   {
      innerCache.Add(key, value);     // добавили запись
   }
   finally
   {
      // снимаем блокировку кэша для записи
      cacheLock.ExitWriteLock();
   }
}
```

```
// метод добавления записи в кэш с ожиданием
public bool AddWithTimeout(int key, string value, int timeout)
{
    // пытаемся получить блокировку кэша для записи
    if (cacheLock.TryEnterWriteLock(timeout))
    {// получили
        try
        {
            innerCache.Add(key, value); // добавили запись
        }
        finally
        {
            // сняли блокировку кэша для записи
            cacheLock.ExitWriteLock();
        }
        return true;
    }
    else
    {// не получили
        return false;
    }
}

// метод обновления или добавления записи в кэш
public AddOrUpdateStatus AddOrUpdate(int key, string value)
{
    // получаем блокировку кэша для обновления
    cacheLock.EnterUpgradeableReadLock();
    try
    {
        string result = null;
        if (innerCache.TryGetValue(key, out result))
        {// если запись с ключом key существует
            if (result == value)
            {// если значение в  записи равно  value
                return AddOrUpdateStatus.Unchanged;
            }
            else
            {// если значение в  записи не равно  value
                cacheLock.EnterWriteLock();
```

```
                try
                {
                    innerCache[key] = value; // обновляем запись
                }
                finally
                {
                    cacheLock.ExitWriteLock();
                }
                return AddOrUpdateStatus.Updated;
            }
        }
        else
        {   // если записи с ключом key не существует
            // получаем блокировку кэша для записи
            cacheLock.EnterWriteLock();
            try
            {
                innerCache.Add(key, value); // добавляем запись
            }
            finally
            {
                cacheLock.ExitWriteLock();
            }
            return AddOrUpdateStatus.Added;
        }
    }
    finally
    {
        // снимаем блокировку кэша для обновления
        cacheLock.ExitUpgradeableReadLock();
    }
}

// удаление записи из кэша
public void Delete(int key)
{
    // получаем блокировку кэша для записи
    cacheLock.EnterWriteLock();
    try
    {
        innerCache.Remove(key); // удаляем запись
```

```
            }
            finally
            {
                // снимаем блокировку кэша для записи
                cacheLock.ExitWriteLock();
            }
        }

        // состояние выполнения операции над кэшем
        public enum AddOrUpdateStatus
        {
            Added,
            Updated,
            Unchanged
        };

        // деструктор
        ~SynchronizedCache()
        {
            if (cacheLock != null) cacheLock.Dispose();
        }
    }
}

// Модуль PSWCache.cs

using System;
using System.Collections.Generic;
using System.Threading;
using System.IO;

namespace RWLSlim
{
    class PSWCache : SynchronizedCache
    {
        string path;
        FileSystemWatcher psw_watcher;

        public PSWCache(string filepath)
        {
            path = filepath;
```

```
    psw_watcher =
            new FileSystemWatcher(Path.GetDirectoryName(path),
                                  Path.GetFileName(path));
    psw_watcher.Changed +=
            new FileSystemEventHandler(OnPSW_Changed);

    LoadPSW();

    psw_watcher.EnableRaisingEvents = true;
}

// загружает пароли из файла
private  bool LoadPSW()
{
    if (File.Exists(path))
    {
        StreamReader reader = new StreamReader(path);

        try
        {
            string psw_i;
            int i = 0;
            while ((psw_i = reader.ReadLine()) != null)
                Add(++i, psw_i);
            return true;
        }
        catch
        {
            return false;
        }
        finally
        {
            reader.Close();
        }
    }
    else
        return false;
}

// сохранение паролей
```

```
public void SavePSW()
{
    psw_watcher.EnableRaisingEvents = false;
    StreamWriter writer = new StreamWriter(path);
    for (int j = 1; j < Count; j++)
        writer.WriteLine(Read(j));

    writer.Close();
}

public int AddNewPersonWithTimeOut(string value, int timeout)
{
    int person_num=Count+1;
    if (AddWithTimeout( person_num, value, timeout))
      return person_num;
    else
        return -1;
}

// обработчик события изменение файла паролей
private void OnPSW_Changed(object source, FileSystemEven-
tArgs e)
{
    // Specify what is done when a file is changed, created, or delet-
ed.
    Console.WriteLine("Внесены изменения в файл паролей: " +
                          e.FullPath + " " + e.ChangeType);

    (new Thread(() =>
        {
            StreamReader reader = new StreamReader(path);
            string psw_i;
            int i = 0;
            while ((psw_i = reader.ReadLine()) != null)
            {
                string record = "пароль для person"+  (i+1).ToString()+
                                    " ("+psw_i+")";
                switch (AddOrUpdate(++i, psw_i))
                {
                    case AddOrUpdateStatus.Updated:
                        Console.WriteLine(record + " обновлён");
```

```
                    break;
                case AddOrUpdateStatus.Unchanged:
                    break;
                case AddOrUpdateStatus.Added:
                    Console.WriteLine(record + " добавлен");
                    break;
            }
        }
        reader.Close();
    })).Start();
  }
 }
}

// Модуль Program.cs

using System;
using System.Threading;
using System.IO;

namespace RWLSlim
{
  class Program
  {
    public static void Coming(object numpsw)
    {
      int person_num = (int) ((object[])numpsw)[0];
      string person_psw = (string)((object[])numpsw)[1];

      Console.WriteLine("Вход пользователя {0} с паролем {1}",
                                          person_num, person_psw);
      string psw=passwords.Read(person_num);

      if (psw == person_psw)
        Console.WriteLine("Вход пользователя {0} с паролем {1}
разрешен!", person_num, person_psw);
      else
        Console.WriteLine("Вход пользователя {0} с паролем {1} не
разрешен!", person_num, psw);
    }
```

```
        public static void Register(object newpsw)
        {
            string person_psw = (string)newpsw;
            Console.WriteLine("Регистрация нового пользователя с па-
ролем {0}", person_psw);

            int person_num = pass-
words.AddNewPersonWithTimeOut(person_psw, 1000);
            if (person_num!=-1)
                Console.WriteLine("добавлен пользователь {0} с паролем
{1}",
                                        person_num, pass-
words.Read(person_num));
            else
                Console.WriteLine("не добавлен пользователь с паролем
{1}",  person_psw);
        }

        static PSWCache passwords;
        static string pswfile_path;

        static void Main(string[] args)
        {
            pswfile_path=@"d:\psw.txt";
            passwords = new PSWCache(pswfile_path);

            for (int j = 1; j < 10; j++)
            {
                (new Thread(Register)).Start("pas"+
                                            (new Random()).Next(555).ToString() );
                Thread.Sleep(1000);
                (new Thread(Coming)).Start(new object[]{ j, "pas" +
j.ToString()});
                Thread.Sleep(1000);
            }
            Thread.Sleep(30000);
            passwords.SavePSW();
            Console.WriteLine("System ShutDown!");
        }
    }
}
```

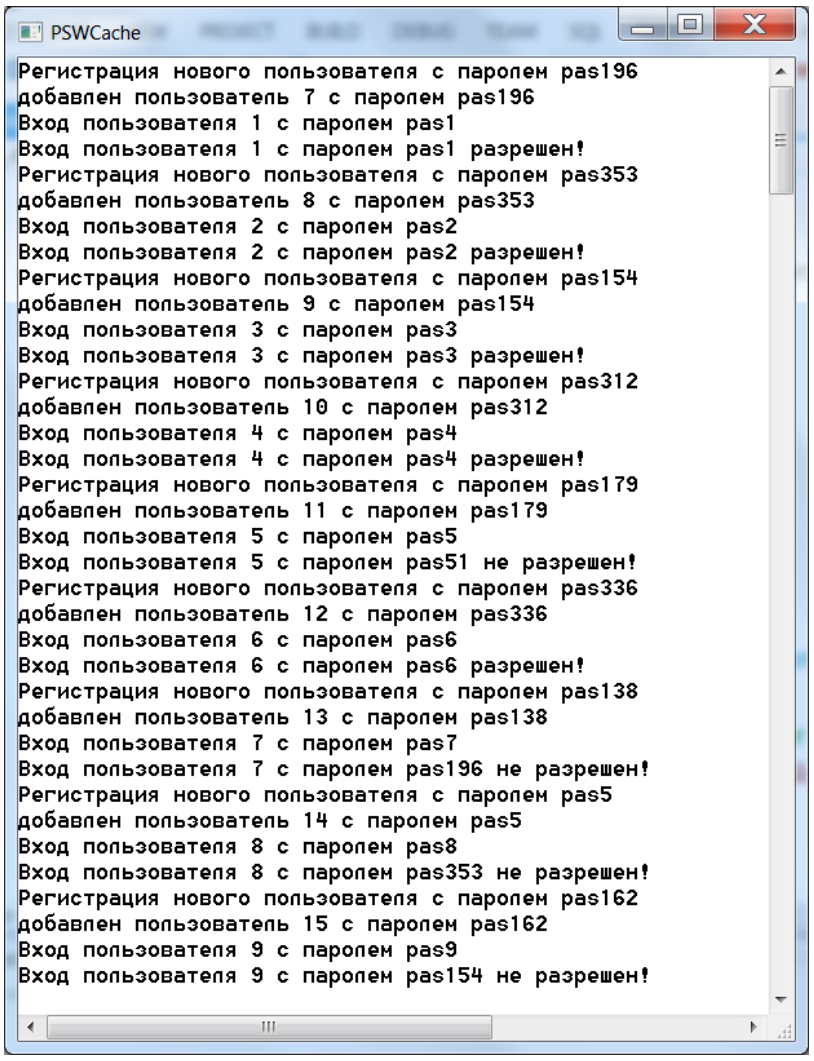


Рис. 4.18. Результат тестирования

5.ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

Для выполнения работ могут быть использованы системы программирования MS Visual Studio .Net версий 2012 года и выше.

По результатам выполнения каждой работы студент должен подготовить и представить преподавателю отчет, который включает разделы:

– постановка задачи;
– метод решения задачи;
– описание разработанных классов (структура данных и алгоритмы);
– описание программы;
– тестовый пример;
– результаты, выдаваемые программой;
– выводы.

ЛАБОРАТОРНАЯ РАБОТА 1
Распараллеливание решения задачи посредством пула потоков

Цель работы – ознакомление с возможностями класса *ThreadPool* выполнения методов приложения через пул потоков.

Задание

1) Изучить назначение и функциональные возможности класса *ThreadPool.*
2) Разработать параллельное решение предложенной задачи используя пул потоков.

Порядок выполнения работы

1. Пункты задания 1 изучить по соответствующим разделам данных методических указаний и справочной системе msdn (https://msdn.microsoft.com).

2. Пункт 2 задания выполнить на примере решения задачи нахождение простых чисел на интервале от 2 до n:

1) разработать последовательное приложение поиска простых чисел, реализующее классический метод «Решето Эратосфена»;

2) разработать параллельное приложение, реализующее модифицированный алгоритм «Решето Эратосфена», допускающий распараллеливание, используя пул потоков.

Число называется простым, если оно делится нацело только на 1 и на само себя. Для проверки, является ли число k простым, можно следовать определению простого числа и построить проверку делимости нацело начиная с 2 и до k-1. Если при делении на один из делителей заданного диапазона получим остаток от деления равный нулю, то число k – непростое.

На самом деле не надо проверять все делители указанного диапазона, а в качестве делителей использовать простые числа до $\sqrt{k}$. Но, такая схема неэффективная, если нам необходимо найти все простые числа на интервале от 2 до n. Для такой задачи лучше подходит классический алгоритм «Решето Эратосфена». Он заключается в последовательном переборе уже известных простых чисел, начиная с m=2, и проверки делимости всех чисел диапазона $(\mathrm{m}, \mathrm{n}]$ на это число m.

Если проверяемое число $p \in (m, n]$ делится на m, то оно не простое и вычеркивается из списка возможных простых чисел. После проверки делимости всех чисел $p \in (m, n]$ на m переходим к следующему делителю после m.

Недостатком классического алгоритма «Решето Эратосфена» является его последовательная схема – нельзя определить является ли число k простым не определив все простые числа до числа k, а значит он не пригоден для распараллеливания.

Для распараллеливания удобнее использовать модифицированный алгоритм «Решето Эратосфена».

Для проверки на простое всех чисел в диапазоне (2,n] необходимо знать простые числа в диапазоне $(2, \sqrt{\mathrm{n}}]$. Поэтому, на первом этапе выполним нахождение простых чисел в диапазоне $(2, \sqrt{\mathrm{n}}\ ]$ классическим последовательным алгоритмом «Решето Эратосфена». На втором этапе будем использовать найденные простые числа для поиска простых чисел в интервале $(\sqrt{\mathrm{n}}, \mathrm{n}]$.

Для каждого простого числа из $(2, \sqrt{\mathrm{n}}\ ]$ создадим поток, в котором все числа из $(\sqrt{n}, n]$ будем делить почередно на это простое число. Если текущее проверяемое число из $(\sqrt{\mathrm{n}}, \mathrm{n}]$ делится, то оно не простое.

В этой лабараторной работе создаваемые потоки будем выполнять через пул потоков. Для получения информации о звершении потока использовать событие, выбрасываемое в методе потока.

ЛАБОРАТОРНАЯ РАБОТА 2
Распараллеливание решения задачи созданием параллельных потоков

Цель работы – освоение приёмов распараллеливания решения задачи с применениям потоков.

Задание

1) Изучить способы распараллелливания решения задач с применением потоков.
2) Разработать параллельное решение предложенной задачи используя явное создание потоков.

Порядок выполнения работы

1. Пункт задания 1 изучить по соответствующим разделам данных методических указаний.
2. Пункт 2 задания выполнить на примере решения задачи нахождение простых чисел на интервале от 2 до n, разработав параллельное приложение с явным созданием потоков, реализующее модифицированный алгоритм «Решето Эратосфена» (описан в лаб.раб.№1

ЛАБОРАТОРНАЯ РАБОТА 3
Исследование и борьба с проблемой гонки данных

Цель работы – исследование проблемы гонки данных в многопоточном приложении и освоение средств борьбы с ней.

Задание

1) Изучить проблему гонки данных в многопоточном приложении.
2) Изучить способы борьбы с проблемой гонки данных: оператор lock и класс Monitor.
3) Разработать приложения для решения предлагаемых задач.

Порядок выполнения работы

1. Пункты задания 1-2 изучить по соответствующим разделам данных методических указаний и справочной системе msdn (https://msdn.microsoft.com).

2. Пункт 3 задания выполнить на примере решения задачи нахождение простых чисел на интервале от 2 до n, разработав параллельное приложение с явным созданием потоков, реализующее модифицированный алгоритм «Решето Эратосфена», использованный в лаб. раб. №2 со следующими изменениями:

Для определения простых чисел в диапазоне $(\sqrt{n}, n]$ приложение создаёт s потоков (s – вводится). Каждый поток исполняет один и тот же метод – проверки чисел из диапазона $(\sqrt{n}, n]$ на делимость следующим проверяемым делителем из множества простых чисел P, найденных на первом этапе. Все потоки создаются одновременно.

Для указания на следующее проверяемое простое число из P как делителя необходимо ввести в рассмотрение переменную i, хранящую индекс следующего проверяемого простого числа в P, который сначала равен 0. Каждый поток для получения очередного делителя обращается к P[i] и после изменяет i=i+1. После проверки делителя поток не завершает свою работу, а снова обращается к следующему необработанному делителю из P. Потоки заканчивают свою работу, когда все простые числа из P обработаны.

Таким образом, каждый поток работает, пока есть необработанные простые числа из P. И когда он берет очередное число на обработку, он должен изменять индекс следующего необработанного числа в списке. Т.е. в этом случае возникает проблема гонки данных на ресурсе i, с которой надо бороться.

4) Исследовать проблему гонки данных при выполнении п.3.
5) Устранить проблему гонки данных используя оператор lock и Monitor.

ЛАБОРАТОРНАЯ РАБОТА 4
Исследование и борьба с проблемой клинча

Цель работы – исследование проблемы клинча в многопоточном приложении и освоение средств борьбы с ней.

Задание

1) Изучить проблему гонки данных в многопоточном приложении.
2) Изучить способы борьбы с проблемой клинча: оператор lock и класс Monitor.
3) Разработать приложения для решения предлагаемых задач.

Порядок выполнения работы

1. Пункты задания 1-2 изучить по соответствующим разделам данных методических указаний и справочной системе msdn (https://msdn.microsoft.com).
2. Пункт 3 выполнить при решении следующих упражнений:

1) Смоделировать проблему клинча на двух потоках;
2) Устранить проблему клинча на двух потоках средствами lock;
3) Устранить проблему клинча на двух потоках средствами класса Monitor.
4) Смоделировать проблему клинча на К потоках - в цикле случайным образом запускаются К потоков двух типов: первый, прибавляет к общей переменной суммы заданное число, а второй – вычитает. Создаём ситуацию, когда более двух потоков впадают в клинч на двух ресурсах.
5) Устранить проблему клинча на К потоках средствами класса Monitor.

ЛАБОРАТОРНАЯ РАБОТА 5
Использование сигнальных сообщений для синхронизации потоков приложения

Цель работы – изучение механизма сигнальных сообщений для синхронизации потоков приложения и возможностей классов *AutoResetEvent, ManualResetEvent, CountdownEvent*.

Задание

1) Изучить механизм сигнальных сообщений для синхронизации потоков приложения.
2) Изучить возможности классов *AutoResetEvent, ManualResetEvent, CountdownEvent*.
3) Разработать приложение для решения предложенной задачи.

Порядок выполнения работы

1. Пункты задания 1-2 изучить по соответствующим разделам данных методических указаний и справочной системе msdn (https://msdn.microsoft.com).
2. Пункт 3 выполнить для решения задачи из лаб. раб. № 3, используя классы *AutoResetEvent (ManualResetEvent)* и *CountdownEvent.*

ЛАБОРАТОРНАЯ РАБОТА 6

Использование модели чтения-записи для синхронизации доступа к разделяемому ресурсу

Цель работы – изучение модели чтения-записи для синхронизации доступа к разделяемому ресурсу.

Задание

1) Изучить модель блокировки чтения-записи.
2) Изучить возможности классов *ReaderWriterLock* и *ReaderWriterLockSlim*.
3) Разработать приложение для решения предложенной задачи.

Порядок выполнения работы

1. Пункты задания 1-2 изучить по соответствующим разделам данных методических указаний и справочной системе msdn (https://msdn.microsoft.com).
2. Пункт 3 выполнить для решения задачи, используя класс *ReaderWriterLockSlim*:

Для клиент-серверного приложения необходимо разработать подсистему контроля доступа к сервису. Контроль доступа строится на учетных записях. Каждая учетная запись включает информацию о пользователе: фамилию, имя, логин, пароль, почта, состояние. Информация о пользователях хранится во внешенем файле, который может меняться. Для пользователя предусмотрены такие режимы:

- регистрация, обеспечивает создание учетной записи пользователя (учетная запись определяется логином);
- вход, обеспечивает доступ к сервису (вход разрешен, если указаны правильный логин и пароль);
- изменение пароля, обеспечивает изменение пароля;
- деактивация учётной записи, обеспечивает создание для пользователя состояния не активный, но учётная запись не уничстожается. Вход для пользователя с неактивной записью невозможен.

Разрабатываемая система контроля доступа должна строится на модели блокировки чтения-записи (*ReaderWriterLockSlim)*. Для блокировки доступа к файлу учетных записей использовать класс *Mutex*.

Промоделировать работу разработанной системы на основе формирования запросов к ней на регистрацию, вход, изменения пароля, деактивации, выполняемых в параллельных потоках.

Вопросы для контроля

1. Пул потоков, его назначение и использование.
2. Получение информацию о завершении выполнения метода в пуле потоков.
3. Преимущества недостатки выполнения методов приложения через пул потоков.
4. Способы распараллеливания задач.
5. Потоко безопасный метод.
6. Реентерабельный метод.
7. Что понимается под синхронизацией потоков?
8. Проблема гонки данных.
9. Клинч в многопоточном приложении.
10. Задачи синхронизации потоков.
11. Что понимается под блокировкой потока?
12. Активная блокировка потока.
13. Пассивная блокировка потока.
14. Чем отличается активная блокировка потока от пассивной?
15. Дайте классификацию средств потоковой синхронизации.
16. Приведите пример кода с активной блокировкой потока.
17. Приведите пример кода с пассивной блокировкой потока.
18. Как можно реализовать исключительный доступ к ресурсу?
19. Назначение конструкции *lock*, пример кода.
20. Что понимается под критической секцией?
21. Назначение и возможности класса *Monitor*.
22. Отличие методов класса *Monitor*: *Enter()* и *TryEnter()*.
23. Приведите пример кода, который приводит к клинчу.
24. Приведите пример кода, в котором устраняется проблема клинча.
25. Методы класса *Monitor: Wait()* и *Pulse()*.
26. Отличие класса *Mutex* от класса *Monitor*.
27. Охарактеризуйте механизм сигнальных сообщений для синхронизации потоков.
28. Назначение и возможности класса *ManualResetEventSlim*.
29. Назначение и возможности класса *CountdownEvent*.
30. Назначение семафоров, укажите классы, реализующие семафоры.
31. Назначение атомарных операторов.
32. Назначение блокировки чтения-записи.
33. Назначение класса *ReaderWriterLockSlim*.

СПИСОК ЛИТЕРАТУРЫ

1. Рейли Д. Создание приложений Microsoft ASP.Net / Д.Рейли : пер.с англ.– М.: Изд.-торговый дом «Русская редакция», 2002.–480с., ил.

2. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.1. / Ч. Петцольд : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002.– 576 с., ил.

3. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.2. / Ч. Петцольд» : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002.– 624 с., ил.

4. Лабор В. В. Си Шарп : Создание приложений для Windows / В. В. Лабор.– Мн.: Харвест, 2003.– 384 с.

5. Рихтер Дж. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд. – СПб.: Питер, 2013. – 896 с. , ил.

6. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# -NET : Учеб.курс MCAD/MCSD : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2003.– 512 стр., ил.

7. Троелсен Э. Язык программирования C# 5.0 и платформа .NET 4.5, 6-е изд.: Пер. с англ. М.: «И.Д. Вильямс», 2013 – 1312с.,ил.

8. Анализ требований и создание архитектуры решений на основе Microsoft .NET : учеб.курс MCSD/пер. с англ.–М.: Издательско-торговый дом «Русская Редакция», 2004.– 416 стр., ил.

9. Шилдт Г. Полный справочник по C# / Шилдт Г. : пер. с англ. – М.: Издательский дом "Вильямс", 2004. – 752 с., ил.

10. Бишоп Дж. C# в кратком изложении / Дж. Бишоп, Н.Хорспул : пер. с англ.– М.: Бином, Лаборатория знаний, 2005. – 472с., ил.

СОДЕРЖАНИЕ

Навчальне видання

Методичні вказівки
до лабораторних робіт “Синхронизация потоков в параллельном программирования на языке C#” з дисципліни “Технології програмування”
для студентів спеціальностей
122 – Компютерні науки та інформаційні технології,
124 – Системний аналіз, у тому числі для іноземних студентів

Російською мовою

Укладачі:
КОЖИН Юрій Миколайович
МАЛИХ Олег Миколайович
ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск О. С. Куценко

Роботу до друку рекомендував О. В. Горілий

Редактор О.І. Шпільова

План 2017 , поз. 114/

Підп. до друку	Формат 60x84 1/16	Папір офсетний.
Riso-друк.	Гарнітура Таймс.	Ум.друк.арк. 4,8
	Наклад 100 прим.	Зам. №

Ціна договірна.

Видавець
Видавничий центр НТУ “ХПІ” м. Харків, 61002, вул. Кирпичова, 2
Свідоцтво про державну реєстрацію ДК №4064 від 21.08.2017 р