# Многопоточное программирование

Многопоточная программа состоит из двух или больше частей, которые могут выполняться одновременно. Каждая часть такой программы называется **потоком** (thread), и каждый поток определяет собственный путь выполнения инструкций.

Различают два вида многозадачности:

* с ориентацией на процессы и
* с ориентацией на потоки.

**Процесс**по сути представляет собой выполняемую программу. Следовательно, многозадачность, ориентированная на процессы, — это средство, позволяющее компьютеру выполнять две или больше программ одновременно. Например, именно благодаря многозадачности, ориентированной на процессы, мы можем работать с текстовым редактором (или с электронными таблицами) и в то же самое время искать нужную информацию в Internet. В процессно-ориентированной многозадачности программа является наименьшим элементом кода, которым может манипулировать планировщик задач.

**Поток**— это управляемая единица выполняемого кода. В многозадачной среде, ориентированной на потоки, все процессы имеют по крайней мере один поток, но возможно и большее их количество. Это означает, что одна программа может выполнять сразу две и более задач. Например, текстовый редактор может форматировать текст и в то же время выводить что-либо на печать, поскольку эти два действия выполняются двумя отдельными потоками.

Таким образом, процессно-ориентированная многозадачность обеспечивает одновременное выполнение программ, а поточно-ориентированная —одновременное выполнение частей одной и той же программы.

Большинство устройств ввода-вывода (сетевые порты, дисководы или клавиатура) работают гораздо медленнее, чем центральный процессор (ЦП). Поэтому в программах львиная доля времени выполнения зачастую тратится на ожидание окончания отправки информации устройству (или получения от него). Используя многопоточность, можно построить программу так, чтобы она в такие периоды ожидания выполняла другую задачу.

Поток может находиться в одном из нескольких возможных состояний.

* Поток может **выполняться (Running)**.
* Поток может **быть готовым к выполнению (Unstarted)** (как только получит время ЦП).
* Выполняющийся поток может быть **приостановлен (WaitSleepJoin)**, т.е. его выполнение временно прекращается.
* Поток может быть **возобновлен**.
* Поток может быть **заблокирован** (**AbortRequested**) в ожидании необходимого ресурса.
* Поток может **завершиться (Aborted)** , и в этом случае его выполнение окончено и продолжению (возобновлению) не подлежит.

В .NET Framework определено два типа потоков:

* **высокоприоритетный**(foreground) и
* **низкоприоритетный**, или фоновый (background).

По умолчанию поток создается высокоприоритетным, но его тип можно изменить, т.е. сделать его фоновым.

Единственное различие между высоко- и низкоприоритетным потоками состоит в том, что низкоприоритетный поток будет автоматически завершен, если все высокоприоритетные

потоки в его процессе остановились.

**Синхронизация** позволяет координировать выполнение потоков.

Все процессы имеют, по крайней мере, один поток управления, который обычно называется основным (main thread), поскольку именно с этого потока начинается выполнение программы.

Используя С#, можно создавать как процессы, так и потоки, а затем ими эффективно управлять. При этом, чтобы создать новый процесс, потребуется написать небольшую программку, поскольку каждый процесс в значительной степени отделен от следующего.

Классы, которые поддерживают многопоточное программирование, определены в пространстве имен System.Threading. Поэтому в начало любой многопоточной программы необходимо включить следующую инструкцию:

using System.Threading;

Многопоточная система С# встроена в класс **Thread**, который инкапсулирует поток управления. Класс Thread является **sealed**-классом, т.е. он не может иметь наследников. В классе Thread определен ряд методов и свойств для управления потоками.

## Класс Thread

Класс Thread определяет набор методов (как статических, так и уровня объекта), которые позволяют создавать новые потоки внутри текущего, а также приостанавливать, останавливать и уничтожать определенный поток.

**Список основных статических элементов класса Thread приведен ниже:**

* CurrentContext – свойство только для чтения возвращает контекст, в котором в данный момент выполняется поток.
* CurrentThread – свойство только для чтения возвращает ссылку на текущий выполняемый поток.
* Sleep() – метод приостанавливает текущий поток на заданное время.

**Класс Thread также поддерживает несколько методов уровня объекта.**

* IsAlive – возвращает булевское значение, указывающее на то, запущен ли поток (и еще не прерван и не отменен)
* IsBackground – получает или устанавливает значение, указывающее, является ли данный поток "фоновым" (подробнее объясняется далее)
* Name – позволяет вам установить дружественное текстовое имя потока
* Priority – получает или устанавливает приоритет потока, который может принимать значение из перечисления ThreadPriority
* ThreadState – получает состояние данного потока, которому может быть присвоено значение из перечисления ThreadState
* Abort() – прерывание потока, как только это будет возможно
* Interrupt() – прерывает (т.е. приостанавливает) текущий поток на заданный период ожидания
* Join() – блокирует вызывающий поток до тех пор, пока указанный поток (тот, в котором вызван Join()) не завершится
* Resume() – возобновляет ранее приостановленный поток
* Start() – запуск потока
* Suspend() – приостанавливает поток. Если поток уже приостановлен, вызов Suspend() не дает эффекта.

Во время сеанса отладки в Visual Studio 2010 можно открыть окно Threads (Потоки), выбрав пункт меню Debug --> Windows --> Threads (Отладка --> Окна --> Потоки).

При программном создании дополнительных потоков для выполнения некоторой единицы работы необходимо следовать строго регламентированному процессу:

1. Создать метод, который будет точкой входа для нового потока.
2. Создать новый делегат ParametrizedThreadStart (или ThreadStart), передав конструктору адрес метода, определенного на предыдущем шаге.
3. Создать объект Thread, передав в качестве аргумента конструктора ParametrizedThreadStart/ThreadStart.
4. Установить начальные характеристики потока (имя, приоритет и т.п.).
5. Вызвать метод Thread.Start(). Это запустит поток на методе, который указан делегатом, созданным на втором шаге, как только это будет возможно.

**Пример 1 (Шилдт).**

class MyThread

{

string threadName;

public int Count

{set;get;}

public MyThread(string name)

{

Count = 0;

threadName = name;

}

public void run() //входная точка потока

{

Console.WriteLine("Старт потока "+threadName);

do

{

Thread.Sleep(500);

Console.WriteLine("В потоке " + threadName + ", count = " + Count);

Count++;

}while(Count < 10);

Console.WriteLine("Конец потока "+threadName);

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Основной поток стартовал.");

MyThread mt = new MyThread("Потомок #1");

Thread newThrd = new Thread(new ThreadStart(mt.run));// из первого потока создаем еще один поток

newThrd.Start(); //запускаем поток

do

{

Console.Write(".");

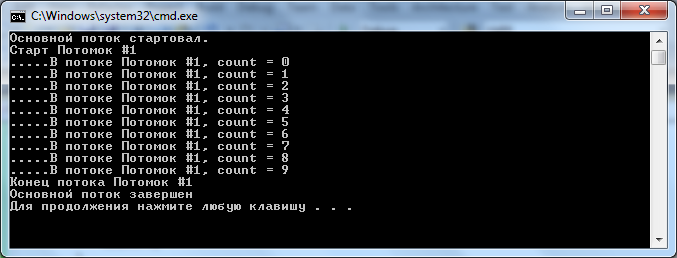
Thread.Sleep(100);

} while (mt.Count != 10);

Console.WriteLine ("Основной поток завершен") ;

}

}



1. Создается объект mt класса MyThread.
2. Класс MyThread используется для создания второго потока управления newThrd. Объект класса MyThread используется для создания объекта класса Thread путем передачи значения mt. run в качестве имени стартового метода, или метода входной точки.
3. Вызов метода start () запускает новый поток newThrd на выполнение. Это приводит к вызову метода run () дочернего потока.
4. В методе run() объекта newThrd организован цикл, который повторяется 10 раз. Метод Sleep () заставляет поток, из которого он был вызван, приостановить выполнение на период времени, заданный в миллисекундах. Метод Sleep() является статическим и определен в классе Thread. Если параметр метода Sleep() равен нулю, вызывающий поток приостанавливается только для того, чтобы ожидающему потоку разрешить выполнение.
5. После вызова метода start () выполнение основного потока возвращается в метод Main (), в цикл do. Теперь выполняются оба потока, разделяя время ЦП до тех пор, пока не закончатся их циклы.

Внесем небольшие изменения в программу:

class MyThread

{

**public Thread thrd;//потоковый объект**

public int Count

{set;get;}

public MyThread(string name)

{

Count = 0;

//создаем поток

**thrd = new Thread(this.run);**

thrd.Name = name;

//запускаем поток

**thrd.Start();**

}

public void run()

{

Console.WriteLine("Старт "+thrd.Name);

do

{

Thread.Sleep(500);

Console.WriteLine("В потоке " + thrd.Name + ", count = " + Count);

Count++;

}

while(Count < 10);

Console.WriteLine("Конец потока " + thrd.Name);

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Основной поток стартовал.");

MyThread mt = new MyThread("Потомок #1");

do

{

Console.Write(".");

Thread.Sleep(100);

} while (mt.Count != 10);

Console.WriteLine ("Основной поток завершен");

}

}

}

Эта версия программы генерирует те же результаты, что и предыдущая. Потоковый объект хранится теперь в переменной thrd внутри класса MyThread.

## Создание нескольких потоков

Программа способна порождать столько потоков, сколько потребуется в конкретной ситуации.

class MyThread

{. . . }

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Основной поток стартовал.");

MyThread mt1 = new MyThread("Потомок #1");

MyThread mt2 = new MyThread("Потомок #2");

MyThread mt3 = new MyThread("Потомок #3");

do

{

Console.Write(".");

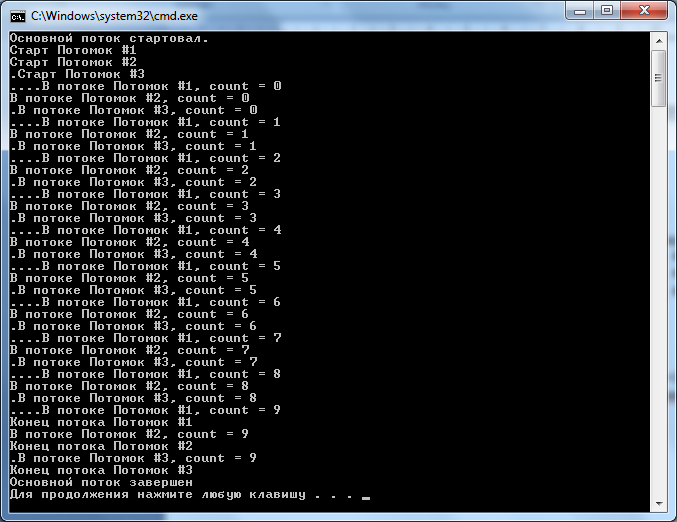
Thread.Sleep(100);

} while (mt1.Count != 10||mt2.Count!=10||mt3.Count!=10);

Console.WriteLine ("Основной поток завершен") ;

}

}



Таким образом, как видно из результатов программы сразу после старта все три потока разделяют время ЦП.

## Управление потоком

Поток создается за счет вызова метода Start() объекта Thread. Однако после вызова метода Start() новый поток все еще пребывает не в состоянии **Running**, а в состоянии **Unstarted**. В состояние **Running** поток переходит сразу после того, как планировщик потоков операционной системы выберет его для выполнения. Информация о текущем состоянии потока доступна через свойство **Thread.ThreadState**.

С помощью метода Thread.Sleep() поток можно перевести в состояние **WaitSleepJoin** и при этом указать, через какой промежуток времени поток должен возобновить работу.

Чтобы остановить поток, необходимо вызвать метод Thread.Abort(). При вызове этого метода в соответствующем потоке генерируется исключение типа **ThreadAbortException**. В случае если для этого исключения предусмотрен обработчик, перед завершением поток сможет выполнить необходимые операции по очистке. Чтобы продолжить выполнение потока после выдачи исключения **ThreadAbortException**, следует вызвать метод Thread.ResetAbort(). Состояние потока, получающего запрос на немедленное прекращение, изменяется с **AbortRequested** на **Aborted**, если поток не производит сброс.

Если необходимо дожидаться завершения работы потока, можно вызвать метод Thread.Join(). Этот метод блокирует текущий поток и переводит его в состояние **WaitSleepJoin** до тех пор, пока не будет завершен присоединенный к нему поток.

## Проверка завершения потока

В классе Thread предусмотрено два средства, которые позволяют установить факт завершения выполнения потока.

1. Свойство **isAlive**. возвращает значение true, если поток, для которого оно опрашивается, еще выполняется. В противном случае оно возвращает значение false.

Его можно использовать для проверки выхода из цикла :

**do**

**{**

Console.Write(".");

Thread.Sleep(100);

**} while (mt1.thrd.IsAlive||mt2.thrd.IsAlive||mt3.thrd.IsAlive);**

1. Метод Join () ожидает, пока поток, для которого он был вызван, не завершится. Если заданный поток даже не стартовал, будет сгенерировано исключение типа ThreadStateException. Другие форматы метода Join() позволяют указать максимальный объем времени, в течение которого будет ожидаться завершение выполнения потока.

static void Main(string[] args)

{

Console.WriteLine("Основной поток стартовал.");

MyThread mt1 = new MyThread("Потомок #1");

MyThread mt2 = new MyThread("Потомок #2");

MyThread mt3 = new MyThread("Потомок #3");

**mt1.thrd.Join();**

Console.WriteLine("Поток №1 присоединен");

**mt2.thrd.Join();**

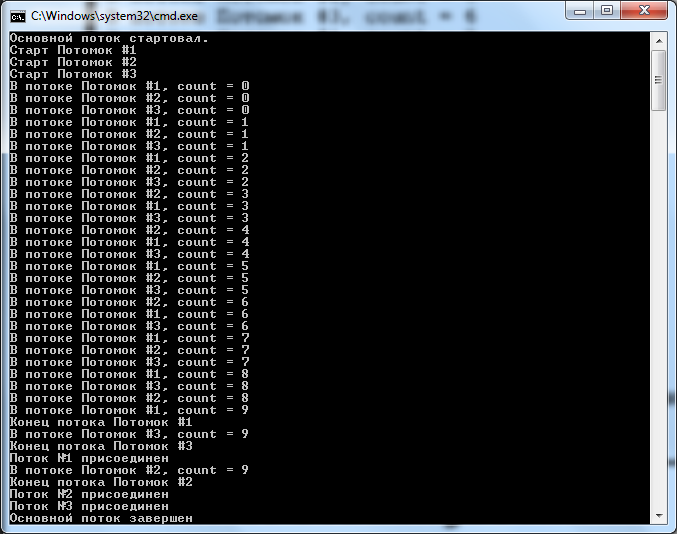
Console.WriteLine("Поток №2 присоединен");

**mt3.thrd.Join();**

Console.WriteLine("Поток №3 присоединен");

Console.WriteLine("Основной поток завершен");

}



## Приоритеты потоков

Среда .NET Framework определяет два типа потоков: высокоприоритетные и фоновые. Процесс не завершится до тех пор, пока не завершится выполнение всех его высокоприоритетных потоков, при этом фоновые потоки заканчиваются автоматически после завершения всех высокоприоритетных потоков. По умолчанию любой поток создается как высокоприоритетный. При необходимости его можно сделать фоновым с помощью свойства IsBackground, которое определено в классе Thread.Чтобы перевести поток в категорию фоновых, достаточно присвоить свойству IsBackground значение true. Значение false означает, что соответствующий поток является высокоприоритетным.

Кроме того, каждый поток имеет определенный приоритет. Приоритет потока, в частности, определяет, какой объем процессорного времени получает поток.После старта дочерний поток получает стандартное значение приоритета. Его можно изменить с помощью свойства Priority, которое может принимать следующие значения:

* ThreadPriority.Highest
* ThreadPriority.AboveNormal
* ThreadPriority.Normal
* ThreadPriority. BelowNoritial
* ThreadPriority.Lowest

По умолчанию потоку присваивается значение приоритета ThreadPriority.Normal.

**Пример 2**

Метод run () содержит цикл, в котором подсчитывается количество итераций. Цикл останавливается либо на счете 1 000 000, либо в тот момент, когда статическая переменная stop примет значение true. До входа в этот цикл переменная stop установлена равной false. Первый поток, "досчитав" до 1 000 000, устанавливает переменную stop равной значению true. Это заставляет второй поток отказаться от следующего кванта времени (объем процессорного времени, выделяемого потоку). На каждом проходе цикла содержимое строковой переменной currentName сравнивается с именем выполняющегося потока. Если они не совпадают, значит, имело место переключение задач. При переключении задач отображается имя нового потока, а переменной currentName присваивается имя текущего потока. Это позволяет наблюдать за тем, как часто каждый из потоков получает доступ к ЦП.

class MyThread

{

public Thread thrd;

public int Count

{set;get;}

**static bool stop = false;**

**static string currentName;**

public MyThread(string name)

{

Count = 0;

//используется делегат ThreadStart для передачи метода, но можно без него

**thrd = new Thread(new ThreadStart(this.run));**

thrd.Name = name;

// thrd.Start(); не запускается на выполнение!

currentName = name;

}

public void run()

{

Console.WriteLine("Старт "+thrd.Name);

do

{

Count++;

**if (currentName != thrd.Name)**

**{**

**currentName = thrd.Name;**

**Console.WriteLine("В потоке " + currentName);**

**}**

**}**

**while (stop == false && Count < 1000000);**

**stop = true;**

Console.WriteLine("Конец потока " + thrd.Name);

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Основной поток стартовал.");

MyThread mt1 = new MyThread("Потомок #1 c высоким приоритетом");

MyThread mt2 = new MyThread("Потомок #2 с низким приоритетом");

MyThread mt3 = new MyThread("Потомок #3 нормальным приоритетом");

mt1.thrd.Priority = ThreadPriority.AboveNormal;

mt2.thrd.Priority = ThreadPriority.BelowNormal;

mt1.thrd.Start();

mt2.thrd.Start();

mt3.thrd.Start();

mt1.thrd.Join();

Console.WriteLine("Поток №1 присоединен");

mt2.thrd.Join();

Console.WriteLine("Поток №2 присоединен");

mt3.thrd.Join();

Console.WriteLine("Поток №3 присоединен");

Console.WriteLine();

Console.WriteLine("Поток " + mt1.thrd.Name +" досчитал до " + mt1.Count);

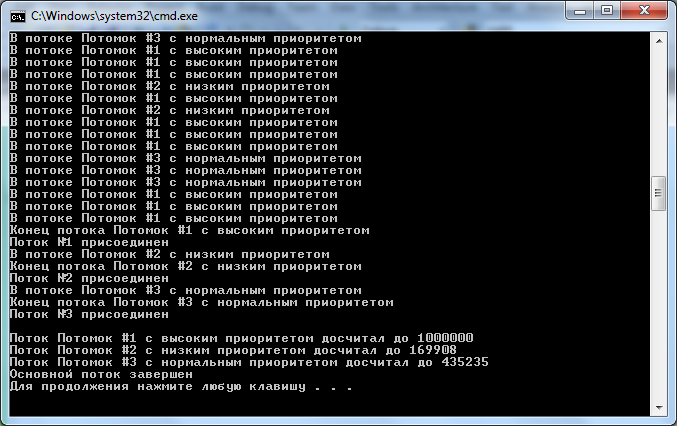
Console.WriteLine("Поток " + mt2.thrd.Name +" досчитал до " + mt2.Count);

Console.WriteLine("Поток " + mt3.thrd.Name + " досчитал до " + mt3.Count);

Console.WriteLine("Основной поток завершен");

}

}



## Асинхронные делегаты

Наиболее простым способом для создания потока является определение делегата и его вызов асинхронным образом. Например, делегат BinaryOp может указывать на любой метод, принимающий два целых числа (по значению) в качестве аргументов и возвращающий целое число.

public delegate int BinaryOp(int x, int y);

После компиляции сборка с определением делегата будет содержать полноценное определение класса, сгенерированного динамически при построении проекта, на основе объявления делегата. В случае BinaryOp этот класс более или менее похож на следующий (записан в псевдокоде):

public sealed class BinaryOp : System.MulticastDelegate

{

//конструктор

public BinaryOp(object target, uint functionAddress);

// вызов метода в синхронном режиме

public void Invoke(int x, int y);

// вызов метода асинхронным образом, в отдельном потоке выполнения

public IAsyncResult Beginlnvoke(int x, int y, AsyncCallback cb, object state);

public int Endlnvoke(IAsyncResult result);

}

**Invoke()** – используется для синхронного вызова каждого из методов, поддерживаемых объектом делегата; это означает, что вызывающий код должен ожидать завершения вызова, прежде чем продолжить свою работу. Его не надо вызывать явно.

Методы **BeginInvoke()** и **EndInvoke()** предлагают возможность вызова текущего метода асинхронным образом, в отдельном потоке выполнения, используется если есть необходимость вызова методов, которые требуют определенного времени для завершения.

Метод **BeginInvoke()** возвращает результат типа IAsyncResult (интерфейс). С помощью IAsyncResult можно извлекать информацию о делегате.

public delegate int BinaryOp(int data, int time);

class Program

{

static int DelegateThread(int data, int time)

{

Console.WriteLine("DelegateThread запущен");

// Делаем задержку, для эмуляции длительной операции

Thread.Sleep(time);

Console.WriteLine("DelegateThread завершен");

return ++data;

}

static void Main(string[] args)

{

Console.WriteLine("Синхронный вызов метода: \n");

// Синхронный вызов метода

DelegateThread(1, 5000);

Console.WriteLine("\nАсинхронный вызов метода с двумя потоками: \n");

// Асинхронный вызов метода с применением делегата

BinaryOp op = DelegateThread;

IAsyncResult ar = op.BeginInvoke(1, 5000, null, null);

while (!ar.IsCompleted)

{

// Выполнение операций в главном потоке

Console.Write(".");

Thread.Sleep(50);

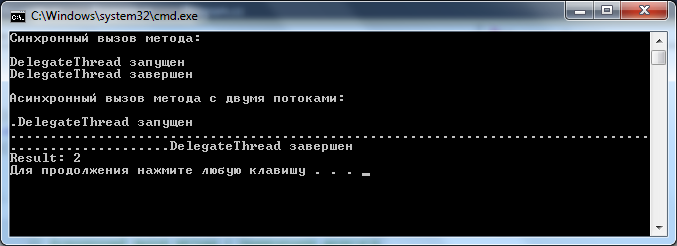
}

int result = op.EndInvoke(ar);

Console.WriteLine("Result: " + result);

}

}



Сгенерированный метод Invoke() используется для вызова метода, поддерживаемого объектом делегата в синхронном режиме. Поэтому вызывающий поток (такой как первичный поток приложения) должен будет ждать, пока не завершится вызов делегата.

Созданный класс delegate предоставляет метод BeginInvoke**()**, в котором могут передаваться входные параметры, определенные вместе с типом делегата. Метод BeginInvoke() всегда имеет два дополнительных параметра типа AsyncCallback и object, которые будут рассматриваться позже. С помощью IAsyncResult можно извлекать информацию о делегате и проверять, завершил ли он свою работу, что здесь и делается с применением свойства IsCompleted. Цикл while продолжает выполняться в главном потоке программы до тех пор, пока делегат не завершит работу:

Вместо выполнения проверки на предмет того, завершил ли делегат работу, после завершения работы главным потоком можно вызвать метод **EndInvoke()** типа делегата. Метод EndInvoke() сам ожидает, когда делегат завершит свою работу. Если главный поток завершает выполнение, не дожидаясь завершения работы делегата, поток делегата останавливается.

Метод BeginInvoke() всегда возвращает объект, реализующий **интерфейс IAsyncResult**, в то время как EndInvoke() ожидает параметр совместимого с IAsyncResult типа. Совместимый с IAsyncResult объект, возвращаемый из BeginInvoke() позволяет вызывающему потоку получить результат вызова асинхронного метода через EndInvoke().

Еще один способ для ожидания результатов от делегата заключается в применении так называемого **асинхронного обратного вызова (asynchronous callback)**. С помощью третьего параметра в методе BeginInvoke можно передать метод, удовлетворяющий требованиям **делегата AsyncCallback**. Делегат AsyncCallback требует определять параметр IAsnycResult и возвращаемый тип void.

Вместо опроса делегата о том, завершился ли асинхронно вызванный метод, было бы более эффективно заставить вторичный поток информировать вызывающий поток о завершении выполнения задания. Чтобы включить такое поведение, необходимо передать экземпляр делегата System.AsyncCallback в качестве параметра методу BeginInvoke(); до сих пор этот параметр был равен null. Если передается объект AsyncCallback, делегат автоматически вызовет указанный метод по завершении асинхронного вызова.

Метод обратного вызова будет вызван во вторичном потоке, а не в первичном. Это имеет важное последствие для потоков с графическим интерфейсом пользователя (WPF или Windows Forms), поскольку элементы управления привязаны к потоку, который их создал, и могут управляться только им.

AsyncCallback может вызывать только методы, соответствующие определенному шаблону, который в данном случае требует единственного параметра IAsyncResult и ничего не возвращает:

void MyAsyncCallbackMethod(IAsyncResult res)

Пример

static void TakesAWhileCompleted(IAsyncResult ar)

{

if (ar == null) throw new ArgumentNullException("ar");

BinaryOp bo = ar.AsyncState as BinaryOp;

int result = bo.EndInvoke(ar);

Console.WriteLine("Результат: " + result);

}

// Асинхронный вызов метода с применением делегата

BinaryOp op = DelegateThread;

bo.BeginInvoke(1, 5000, TakesAWhileCompleted, op);

for (int i = 0; i < 100; i++)

{

// Выполнение операций в главном потоке

Console.Write(".");

Thread.Sleep(50);

}

## Синхронизация

Процесс координации потоков называется **синхронизацией**. К синхронизации прибегают в тех случаях, когда двум или большему числу потоков необходимо получить доступ к общему ресурсу, который в каждый момент времени может использовать только один поток. Или один поток ожидает, пока не произойдет событие, которое зависит от другого потока. В этом случае необходимо иметь средство, которое бы удерживало первый поток в состоянии приостановки до тех пор, пока не произойдет ожидаемое им событие.

В основе синхронизации лежит понятие **блокировки**, т.е. управление доступом к некоторому блоку кода в объекте. На то время, когда объект заблокирован одним потоком, никакой другой поток не может получить доступ к заблокированному блоку кода. Когда поток снимет блокировку, объект станет доступным для использования другим потоком.

Средство блокировки встроено в язык С#, поэтому доступ ко всем объектам может быть синхронизирован. Синхронизация поддерживается ключевым словом lock.

lock(object)

{

// Инструкции, подлежащие синхронизации.

}

где object - ссылка на синхронизируемый объект.

Инструкция lock гарантирует, что указанный блок кода, защищенный блокировкой для данного объекта, может быть использован только потоком, который получает эту блокировку. Все другие потоки остаются заблокированными до тех пор, пока блокировка не будет снята. А снята она будет лишь при выходе из этого блока.

**Способ 1.**

В классе, MyThread, используется static-объект sa типа SumArray. Таким образом, все объекты типа MyThread совместно используют только один объект типа SumArray. Этот объект служит для получения суммы целочисленных элементов массива. Промежуточная сумма хранится в поле sum класса SumArray. Следовательно, если два потока будут использовать метод Summa () одновременно, каждый из них попытается хранить в поле sum "свою" промежуточную сумму. Поскольку это приведет к ошибочным результатам, доступ к методу Summa() необходимо синхронизировать.

SumArray

sum

MyThread mtl

answer = sa.Summa(a);

MyThread mt2

answer = sa.Summa(a);

Summa  
(int[]mas)

class SumArray

{

int sum;

public int Summa (int[] nums)

{

**lock(this)**

**{ // Блокировка всего метода,**

sum = 0 ; // Начальное значение суммы.

for(int i=0; i < nums.Length; i++)

{

sum += nums[i];

Console.WriteLine("Промежуточная сумма для потока " +Thread.CurrentThread.Name +" равна " + sum);

// приостанавливает выполнение текущего потока на 10 мс

Thread.Sleep(10); //т.е. разрешает переключение потоков

}

**}**

return sum;

}

class MyThread

{

public Thread thrd;

int[] a;

int answer;

static SumArray sa = new SumArray () ;

// Создаем новый поток.

public MyThread(string name, int[] nums) //конструктор

{

a = nums;

thrd = new Thread(new ThreadStart(this.run));

thrd.Name = name;

thrd.Start(); // Запускаем поток на выполнение.

}

void run()

{

Console.WriteLine(thrd.Name + " стартовал.");

answer = sa. Summa (a);

Console.WriteLine("Сумма для потока " + thrd.Name + " равна " + answer);

Console.WriteLine(thrd.Name + " завершен.");

}

}

class Program

{

static void Main(string[] args)

{

int[] mas = { 1, 2, 3, 4, 5 };

MyThread mtl = new MyThread("Потомок #1", mas);

MyThread mt2 = new MyThread("Потомок #2", mas);

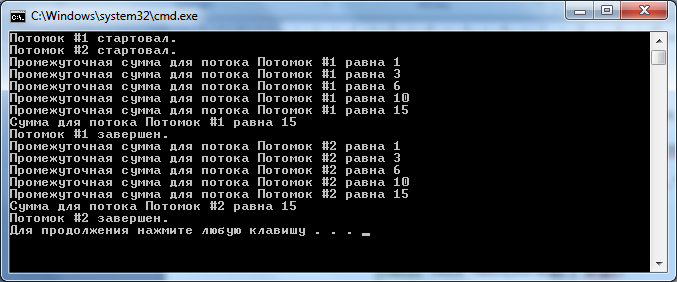
mtl.thrd.Join();

mt2.thrd.Join();

}

}

}



В методе Summa() инструкция lock предотвращает одновременное его использование различными потоками. В этой инструкции блокировки в качестве синхронизируемого объекта служит ссылка this. Обычно именно так выполнятся инструкция lock, если блокируется вызывающий объект. Метод Sleep () вызывается специально для того, чтобы разрешить переключение задач (если оно возможно), но в данном случае это невозможно. Поскольку код метода Summa() заблокирован, он доступен одновременно только для одного потока. Следовательно, после того, как начнет выполняться второй поток, он не войдет в метод Summa() до тех пор, пока его (метод) полностью не отработает первый дочерний поток. Тем самым гарантируется получение корректного результата.

Таким образом:

Если для заданного объекта инструкция lock размещена в некотором блоке кода, этот объект блокируется, и никакой другой поток не сможет запросить блокировку.

1. Другие потоки, пытающиеся запросить блокировку для того же объекта, перейдут в состояние ожидания и будут находиться в нем до тех пор, пока код не будет разблокирован.
2. Объект разблокируется, когда поток выходит из заблокированного блока.

Необходимо также отметить, что инструкция lock должна использоваться только для объектов, которые определены либо как private-, либо как internal-объекты. В противном случае внешние по отношению к вашей программе потоки смогут получать блокировку и не снимать ее.

Если необходимо выполнить блокировку метода, находящегося в классе, написанном другим программистом, то придется использовать другой способ блокировки, т.к. поставить инструкцию lock в код такого метода не получится. В этом случае можно заблокировать доступ к объекту из внешнего (по отношению к объекту) кода, указав этот объект в инструкции lock.

**Способ 2**

class SumArray

{

int sum;

public int Summa(int[] nums)

{

**//lock(this)**

**//{ // Блокировка всего метода**

sum = 0 ; // Начальное значение суммы.

for(int i=0; i < nums.Length; i++)

{

sum += nums[i];

Console.WriteLine("Промежуточная сумма для потока " +Thread.CurrentThread.Name +" равна " + sum);

Thread.Sleep(10); // приостанавливает выполнение текущего потока на 10 мс

}

**//}**

return sum;

}

class MyThread

{

public Thread thrd;

int[] a;

int answer;

static SumArray sa = new SumArray () ;

// Создаем новый поток.

public MyThread(string name, int[] nums) //конструктор

{

a = nums;

thrd = new Thread(new ThreadStart(this.run));

thrd.Name = name;

thrd.Start(); // Запускаем поток на выполнение.

}

void run()

{

Console.WriteLine(thrd.Name + " стартовал.");

**lock (sa) answer = sa.Summa(a);//блокируется обращение к методу**

Console.WriteLine("Сумма для потока " + thrd.Name + " равна " + answer);

Console.WriteLine(thrd.Name + " завершен.");

}

}

class Program

{

static void Main(string[] args)

{

int[] mas = { 1, 2, 3, 4, 5 };

MyThread mtl = new MyThread("Потомок #1", mas);

MyThread mt2 = new MyThread("Потомок #2", mas);

mtl.thrd.Join();

mt2.thrd.Join();

}

}

Чтобы заблокировать static-метод, достаточно использовать инструкцию lock в следующем формате:

lock(typeof (class) )

{

// Блокируемый код.

}

Здесь class представляет собой имя класса, в котором содержится static-метод, подлежащий блокировке.

## Методы класса Monitor

Другие средства синхронизации определенны в классе Monitor, принадлежащем пространству имен System. Threading. В классе Monitor определено несколько методов синхронизации.

* public static void Enter(object syncOb); - предоставляет возможность блокировки для объекта, syncOb.
* public static void Exit(object syncOb); – снимает блокировку для объекта, syncOb.

Здесь syncOb— синхронизируемый объект. Если при вызове метода Enter() заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока объект не станет доступным. Разработчики из компании Microsoft утверждают, что lock-блок "совершенно эквивалентен" вызову метода Enter () с последующим вызовом метода Exit(). Но поскольку lock — это встроенная инструкция языка С#, то для получения блокировки в С#-программировании предпочтительнее использовать именно ее.

* public static bool TryEnter(object syncOb);— метод возвращает значение true, если вызывающий поток получает блокировку для объекта syncOb, и значение false в противном случае. Если заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока он не станет доступным.

В классе Monitor также определены методы Wait(), Pulse() и PulseAll(). Эти методы можно вызывать только внутри lock-блока кода.

**Пример использования.** Поток Т выполняет инструкции lock-блока и при этом обращается к ресурсу, который оказывается занят. Тогда Т переходит в режим ожидания и ждет освобождения ресурса. В это время никакой другой поток не сможет выполнять свою работу до тех пор, пока Т не получит ресурс и не закончит lock-блок.

Удобнее организовать вычисления таким образом, чтобы Т не ждал освобождения ресурса, а позволял другому потоку выполнять свои действия. Когда же ресурс освободится Т должен получить уведомление об этом и возобновить свое выполнение.

Для такого взаимодействия потоков и можно применять методы Wait(), Pulse() и PulseAll().

Когда выполнение потока временно блокируется, вызывается метод Wait(), т.е. поток переходит в режим ожидания ("засыпает") и снимает блокировку с объекта, позволяя другому потоку использовать этот объект. Позже, когда другой поток входит в аналогичное состояние блокирования, то вызывается метод Pulse() или PulseAll() и"спящий" поток "просыпается". Обращение к методу Pulse() возобновляет выполнение потока, стоящего первым в очереди потоков, пребывающих в режиме ожидания. Обращение к методу PulseAll() сообщает о снятии блокировки всем ожидающим потокам.

**Форматы использования методов:**

* public static bool Wait(object waitOb) - ожидание до уведомления;
* public static bool Wait(object waitOb, int milliseconds) - ожидание до уведомления или до истечения периода времени, заданного в миллисекундах;
* public static void Pulse(object waitOb) - waitOb - объект, освобождаемый от блокировки;
* public static void PulseAll(object waitOb) - waitOb - означает объект, освобождаемый от блокировки.

**Пример (Шилдт)**

class Clock

{

public void tick(bool running)

{

lock(this)

{

if(!running) { // Останов часов

**Monitor.Pulse(this);** // Освобождаем ожидающие потоки

return;

}

Console.Write("тик-");

**Monitor.Pulse(this);** // Разрешает выполнение метода tock().

**Monitor.Wait(this);** // Ожидаем завершения метода tock().

}

}

public void tack(bool running)

{

lock(this)

{

if(!running) { // Останов часов.

Monitor.Pulse(this); // Освобождаем ожидающие потоки

return;

}

Console.Write("тaк-");

Monitor.Pulse(this); // Разрешает выполнение метода tick().

Monitor.Wait(this); // Ожидаем завершения метода tick().

}

}

}

class MyThread

{

public Thread thrd;

Clock clk;

public MyThread(string name, Clock c)

{

thrd = new Thread(new ThreadStart(this.run));

clk = c;

thrd.Name = name;

thrd.Start();

}

void run()

{

if (thrd.Name == "тик")

{

for (int i = 0; i < 5; i++) clk.tick(true);

clk.tick(false);

}

else

{

for (int i = 0; i < 5; i++) clk.tack(true);

clk.tack(false);

}

}

}

class Program

{

static void Main(string[] args)

{

Clock clc = new Clock();

MyThread mtl = new MyThread("тик", clc);

MyThread mt2 = new MyThread("так", clc);

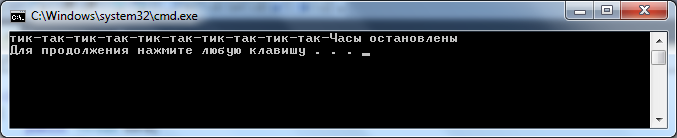
mtl.thrd.Join();

mt2.thrd.Join();

Console.WriteLine("Часы остановлены");

}

}



## Взаимоблокировка

При **взаимоблокировке** (deadlock) один поток ожидает, пока другой не выполнит некоторое действие, но в то же время второй поток ожидает действия первого.

public void tick(bool running)

{

lock(this)

{

if(!running) { // Останов часов.

**//Monitor.Pulse(this);**

return;

}

Console.Write("тик-");

Monitor.Pulse(this); // Разрешает выполнение метода tаck().

Monitor.Wait(this); // Ожидаем завершения метода tаck().

}

}

public void tack(bool running)

{

lock (this)

{

if(!running) { // Останов часов.

**//Monitor.Pulse(this);**

return**;**

}

Console.Write("тaк-");

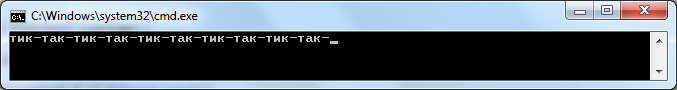
Monitor.Pulse(this); // Разрешает выполнение метода tick().

Monitor.Wait(this); // Ожидаем завершения метода tick().

}

}

}



Произошла взаимоблокировка, т.к для завершающего вызова метода Wait() из последнего выполнения метода tаck () нет соответствующего обращения к методу Pulse(), который бы позволил бы методу завершиться. Поэтому метод tаck() находится в бесконечном ожидании.

## Класс Mutex

Класс Mutex (mutual exclusion — взаимное исключение или мьютекс) позволяет обеспечить синхронизацию среди множества процессов.

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

Он также как и класс Monitor допускает наличие только одного владельца. Только один поток может получить блокировку и иметь доступ к защищаемым **мьютексом** синхронизированным областям кода.

Кроме того, конструктор позволяет получить информацию о том, существует ли уже такой класс.

Конструкторы класса Mutex

public Mutex()//создается мьютекс, которым первоначально никто не владеет

public Mutex(bool initiallyOwned) // указывается, должен ли мьютексом изначально владеть вызывающий поток

Во второй форме исходным состоянием мьютекса завладевает вызывающий поток, если параметр initiallyOwned имеет логическое значение true. В противном случае мьютексом никто не владеет.

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

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

Пример

// В этом классе содержится общий ресурс в виде переменной Count, а так же //мьютекс mtx

class SharedRes

{

public static int Count;

public static Mutex mtx = new Mutex();

}

// В этом классе Count инкременируется

class IncThread

{

int num;

public Thread Thrd;

public IncThread(string name, int n)

{

Thrd = new Thread(this.Run);

num = n;

Thrd.Name = name;

Thrd.Start();

}

// Точка входа в поток

void Run()

{

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

// Получить мьютекс

SharedRes.mtx.WaitOne();

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

do

{

Thread.Sleep(500);

SharedRes.Count++;

Console.WriteLine("в потоке {0}, Count={1}", Thrd.Name, SharedRes.Count);

num--;

} while (num > 0);

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

SharedRes.mtx.ReleaseMutex();

}

}

class DecThread

{

int num;

public Thread Thrd;

public DecThread(string name, int n)

{

Thrd = new Thread(new ThreadStart(this.Run));

num = n;

Thrd.Name = name;

Thrd.Start();

}

// Точка входа в поток

void Run()

{

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

// Получить мьютекс

SharedRes.mtx.WaitOne();

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

do

{

Thread.Sleep(500);

SharedRes.Count--;

Console.WriteLine("в потоке {0}, Count={1}", Thrd.Name, SharedRes.Count);

num--;

} while (num > 0);

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

SharedRes.mtx.ReleaseMutex();

}

}

class Program

{

static void Main(string[] args)

{

IncThread mt1 = new IncThread("Inc thread", 5);

// разрешить инкременирующему потоку начаться

Thread.Sleep(1);

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

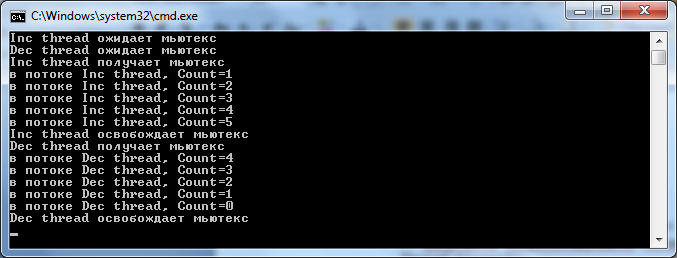
mt1.Thrd.Join();

mt2.Thrd.Join();

Console.ReadLine();

}

}



Доступ к общему ресурсу (переменной SharedRes.Count) синхронизирован, и поэтому значение данной переменной может быть одновременно изменено только в одном потоке.

## Класс Semaphore

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

Когда же потоку больше не требуется доступ к общему ресурсу, он высвобождает разрешение, а счетчик семафора инкрементируется. Если разрешения ожидает другой поток, то он получает его в этот момент. Количество одновременно разрешаемых доступов указывается при создании семафора. Так, если создать семафор, одновременно разрешающий только один доступ, то такой семафор будет действовать как мьютекс.

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

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

public Semaphore(int initialCount, int maximumCount)

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

Семафор применяется таким же образом, как и описанный ранее мьютекс. В целях получения доступа к ресурсу в коде программы вызывается метод WaitOne() для семафора. Метод WaitOne() ожидает до тех пор, пока не будет получен семафор, для которого он вызывается. Таким образом, он блокирует выполнение вызывающего потока до тех пор, пока указанный семафор не предоставит разрешение на доступ к ресурсу.

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

public int Release()

public int Release(int releaseCount)

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

## Использование событий для синхронизации

События представляют собой еще один ресурс для обеспечения синхронизации в масштабе всей системы.

Для использования системных событий из управляемого кода .NET Framework предлагает классы ManualResetEvent, AutoResetEvent, ManualResetEventSlim и CountdownEvent, которые находятся в пространстве имен System.Threading. Классы ManualResetEventSlim и CountdownEvent появились в версии .NET 4.

Эти классы являются производными от класса EventWaitHandle, находящегося на верхнем уровне иерархии классов, и применяются в тех случаях, когда один поток ожидает появления некоторого события в другом потоке. Как только такое событие появляется, второй поток уведомляет о нем первый поток, позволяя тем самым возобновить его выполнение.

Ниже приведены конструкторы классов ManualResetEvent и AutoResetEvent:

public ManualResetEvent(bool initialState)

public AutoResetEvent(bool initialState)

Если в обеих формах параметр initialState имеет логическое значение true, то о событии первоначально уведомляется. А если он имеет логическое значение false, то о событии первоначально не уведомляется.

Применение событий:

* Для события типа **ManualResetEvent** порядок применения следующий. Поток, ожидающий некоторое событие, вызывает метод WaitOne() для событийного объекта, представляющего данное событие. Если событийный объект находится в сигнальном состоянии, то происходит немедленный возврат из метода WaitOne(). В противном случае выполнение вызывающего потока приостанавливается до тех пор, пока не будет получено уведомление о событии. Как только событие произойдет в другом потоке, этот поток установит событийный объект в сигнальное состояние, вызвав метод Set(). Поэтому метод Set() следует рассматривать как уведомляющий о том, что событие произошло. После установки событийного объекта в сигнальное состояние произойдет немедленный возврат из метода WaitOne(), и первый поток возобновит свое выполнение. А в результате вызова метода Reset() событийный объект возвращается в несигнальное состояние.
* **Событие AutoResetEvent** отличается от события типа ManualResetEvent лишь способом установки в исходное состояние. Если для события типа ManualResetEvent событийный объект остается в сигнальном состоянии до тех пор, пока не будет вызван метод Reset(), то для события типа AutoResetEvent событийный объект автоматически переходит в несигнальное состояние, как только поток, ожидающий это событие, получит уведомление о нем и возобновит свое выполнение. Поэтому если применяется событие типа AutoResetEvent, то вызывать метод Reset() необязательно.
* **Событие ManualResetEventSlim** переводится в сигнальное состояние вызовом метода Set(), а с помощью Reset() возвращается обратно в несигнальное состояние. В случае вызова метода Set() при наличии множества потоков, ждущих перехода события в сигнальное состояние, ожидание всех этих потоков немедленно прекращается. В случае, если поток просто вызывает метод WaitOne(), а событие уже находится в сигнальном состоянии, ожидавший поток может сразу же продолжить работу.

class MyThread

{

public Thread Thrd;

ManualResetEvent mre;

public MyThread(string name, ManualResetEvent evt)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

mre = evt;

Thrd.Start();

}

void Run()

{

Console.WriteLine("Внутри потока " + Thrd.Name);

for (int i = 0; i < 5; i++)

{

Console.WriteLine(Thrd.Name);

Thread.Sleep(500);

}

Console.WriteLine(Thrd.Name + " завершен!");

// Уведомление о событии

mre.Set();

}

}

class Program

{

static void Main(string[] args)

{

ManualResetEvent evtObj = new ManualResetEvent(false);

MyThread mt1 = new MyThread("Событийный поток 1",evtObj);

Console.WriteLine("Основной поток ожидает событие");

evtObj.WaitOne();

Console.WriteLine("Основной поток получил уведомление о событии от первого потока");

evtObj.Reset();

mt1 = new MyThread("Событийный поток 2",evtObj);

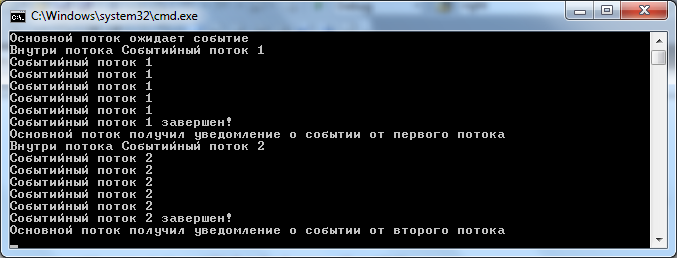
evtObj.WaitOne();

Console.WriteLine("Основной поток получил уведомление о событии от второго потока");

Console.ReadLine();

}

}



Событие типа ManualResetEvent передается непосредственно конструктору класса MyThread. Когда завершается метод Run() из класса MyThread, он вызывает для событийного объекта метод Set(), устанавливающий этот объект в сигнальное состояние. В методе Main() формируется событийный объект evtObj типа ManualResetEvent, первоначально устанавливаемый в исходное, несигнальное состояние. Затем создается экземпляр объекта типа MyThread, которому передается событийный объект evtObj.

После этого основной поток ожидает уведомления о событии. А поскольку событийный объект evtObj первоначально находится в несигнальном состоянии, то основной поток вынужден ожидать до тех пор, пока для экземпляра объекта типа MyThread не будет вызван метод Set(), устанавливающий событийный объект evtObj в сигнальное состояние. Это дает возможность основному потоку возобновить свое выполнение.

Затем событийный объект устанавливается в исходное состояние, и весь процесс повторяется, но на этот раз для второго потока. Если бы не событийный объект, то все потоки выполнялись бы одновременно, а результаты их выполнения оказались бы окончательно запутанными.

Если бы в рассматриваемой здесь программе событийный объект типа AutoResetEvent использовался вместо событийного объекта типа ManualResetEvent, то вызывать метод Reset() в методе Main() не пришлось бы. Ведь в этом случае событийный объект автоматически устанавливается в несигнальное состояние, когда поток, ожидающий данное событие, возобновляет свое выполнение.